Lexicons: First-Class Global Lexical Environments for Common Lisp

Ron Garret

Version 2.0

December 2008

1. Overview

Lexicons are first-class global lexical environments, what are sometimes called "modules" or "namespaces" in other languages. They are designed as an adjunct (or replacement) for Common Lisp packages.

Lexicons were designed to address two shortcomings of the Common Lisp package system:

1. The need to manually manage list of exported symbols, and
2. The fact that common mistakes have read-time side-effects that can be difficult to recover from, especially for beginners.

In addition, it was an explicit design goal for lexicons to solve all of the problems that packages solve so that they could at least in principle be a complete replacement for packages. Lexicons were intended to be one component in an effort to design a dialect of Lisp embedded in Common Lisp that had a better "mental impedance match" to new programmers and programmers used to coding in other languages, specifically the new breed of dynamic languages like Python and Ruby. This effort is on-going [1].

This document is targeted primarily at Common Lisp programmers. It is assumed that the reader is already familiar with lexical scoping and the behavior of the Common Lisp package system as described in [2,3,4].

2. Operational description

2.1 Basics

Lexicons are similar to packages, except that instead of mapping strings to symbols, lexicons map symbols to (global) bindings. This best illustrated by way of an example:

```
? (make-lexicon :l1)
#<lexicon L1>
? (in-lexicon :l1)
#<lexicon L1>
? (ldefun foo () 'l1-foo)
FOO
? (foo)
```

```
L1-FOO
? (make-lexicon :l2)
#<lexicon L2>
? (in-lexicon :l2)
#<lexicon L2>
? (ldefun foo () 'l2-foo)
FOO
? (foo)
L2-FOO
? (in-lexicon :l1)
#<lexicon L1>
? (foo)
L1-FOO
?
```

Note that there are now two different global functions associated with the symbol FOO at the same time. These are not two different symbols named FOO (as would be the case when using packages), these two functions are associated with (and in particular they are globally lexically bound to) a single symbol object.

There are analogous forms for defining global lexical variables (LDEFVAR), macros (LDEFMACRO), classes (LDEFCLASS), and methods (LDEFMTHOD). These work in the intuitively obvious way, with a few exceptions which will be described shortly.

2.2 Libraries and deferred bindings

Lexicons, like packages, are intended for building units of functionality to be used by code in other lexicons. There are several ways to use the bindings in one lexicon in another lexicon. The most straightforward is the USE-LEXICON function:

```
? (in-lexicon :l2)
#<lexicon L2>
? (ldefun library-function () 'l2-lib-result)
LIBRARY-FUNCTION
? (in-lexicon :l1)
#<lexicon L1>
? (ldefun foo () (library-function))
; Warning: Defering lexical binding of LIBRARY-FUNCTION
; While executing: *REF, in process Listener(111).
FOO
? (use-lexicon :l2)
(#<lexicon L2>)
? (foo)
Resolving lexical binding of LIBRARY-FUNCTION
L2-LIB-RESULT
? (foo)
L2-LIB-RESULT
?
```

There are three things to note here. First, we didn't have to explicitly export anything from L2 in order to use LIBRARY-FUNCTION in L1.

Second, there were no name conflicts despite the fact that FOO is bound in both L1 and L2 (from the previous example).

Third, we didn't invoke use-lexicon until *after* we had already written a reference to LIBRARY-FUNCTION. Since there was no visible lexical binding for LIBRARY-FUNCTION at the time, the binding was *deferred* and not resolved until the function was actually called, but that this only happened once.

Contrast this with what happens when we try something analogous using packages:

```
? (make-package :p1)
#<Package "P1">
? (in-package :p1)
#<Package "P1">
? (defun library-function () 'p1-lib-result)
LIBRARY-FUNCTION
? (export 'library-function)
T
? (make-package :p2)
#<Package "P2">
? (in-package :p2)
#<Package "P2">
? (defun foo () (library-function))
;Compiler warnings :
;   Undefined function LIBRARY-FUNCTION, in FOO.
FOO
? (use-package :p1)
> Error: Using #<Package "P1"> in #<Package "P2">
>        would cause name conflicts with symbols already present in that
package:
>        LIBRARY-FUNCTION  P1:LIBRARY-FUNCTION
```

It gets worse. Consider:

```
? (in-package :cl-user)
#<Package "COMMON-LISP-USER">
? (in-lexicon :l1)
#<lexicon L1>
? (library-function)
L2-LIB-RESULT
? (ldefun foo () (if (eq (library-function) 'l2-lib-result) 'woohoo 'bummer))
FOO
? (foo)
WOOHOO
? (in-package :p2)
#<Package "P2">
? (unintern 'library-function)
T
? (use-package :p1)
T
? (defun foo () (if (eq (library-function) 'l2-lib-result) 'woohoo 'bummer))
```

```
FOO
? (foo)
BUMMER
?
```

To make the P1 package do the Right Thing we have to export not only the symbols bound to all the functions we want to use, but also all the symbols we intend to use as external data. But alas, it's not so simple:

```
? (export 'p1::l2-lib-result :p1)
> Error: Name conflict detected by EXPORT :
>        EXPORT'ing P1::L2-LIB-RESULT from #1=#<Package "P1"> would cause a
name conflict with
>        the present symbol L2-LIB-RESULT in the package #<Package "P2">,
which uses #1#.
```

Fans of packages would argue that this is a feature, not a bug, that explicit exportation of symbols from packages is necessary when using symbols as data (otherwise there might be conflicts between, say, two functions wanting to manipulate a symbol's property list). This is a matter of taste. Personally, I find that I use symbols as identifiers (and therefore care only about their names) vastly more often than I use them as actual first-class data objects, but YMMV. Since lexicons and packages co-exist you do not have to choose one or the other (though in my experience trying to use both at the same time can get mightily confusing).

2.3 Bindings, not values

It is worth pointing out that these really are lexical bindings, not just values:

```
? (in-package :cl-user)
#<Package "COMMON-LISP-USER">
? (in-lexicon :l1)
#<lexicon L1>
? (ldefun foo () (library-function))
FOO
? (foo)
L2-LIB-RESULT
? (in-lexicon :l2)
#<lexicon L2>
? (ldefun library-function () 'new-l2-lib-result)
LIBRARY-FUNCTION
? (in-lexicon :l1)
#<lexicon L1>
? (foo)
NEW-L2-LIB-RESULT
?
```

Because these are lexical bindings (which is to say, they are resolved at compile-time) this can sometimes have unexpected consequences. The LDEF* forms are

designed to work in the “intuitive” way, which is to say, the first time they are invoked they establish a new lexical binding in the current (at compile time) lexicon. On subsequent invocations they mutate the already existing binding, which generally makes them behave like their non-lexical DEF* counterparts. But resolved binding references are NOT re-established when the lexicon structure changes, e.g. through a call to USE-LEXICON. For example:

```
? (make-lexicon :l3)
#<lexicon L3>
? (in-lexicon :l3)
#<lexicon L3>
? (ldefun library-function () 'l3-lib-result)
LIBRARY-FUNCTION
? (in-lexicon :l1)
#<lexicon L1>
? (library-function)
NEW-L2-LIB-RESULT
? (foo)
NEW-L2-LIB-RESULT
? (use-lexicon :l3)
(#<lexicon L3> #<lexicon L2>)
? (library-function)
L3-LIB-RESULT       ; More recent libraries shadow less recent ones
? (foo)
NEW-L2-LIB-RESULT   ; But existing bindings are not re-established
?
```

2.4 Semi-hygienic lexical macros

One of the problems that packages solve is unintentional name capture in macro expansions. For example:

```
? (defun confrabulate (arg) (list 'confrabulated arg))
CONFRABULATE
? (defmacro mymacro (arg) `(confrabulate ',arg))
MYMACRO
? (mymacro foo)
(CONFRABULATED FOO)
?
```

MYMACRO will fail if it is in a context where CONFRABULATE is fbound:

```
? (flet ((confrabulate (arg) (list 'hornswiggled arg))) (mymacro foo))
(HORNSWIGGLED FOO)
?
```

(Believe me, getting your foo hornswiggled when you wanted it confrabulated can be disastrous.)

Packages solve this problem by putting MYMACRO and CONFRABULATE in their own package, and exporting MYMACRO but not exporting CONFRABULATE. Name capture is still possible, but it is much less likely to happen by accident since you'd have to deliberately type in the package name in order to shadow the unexported symbol.

Note that sometimes name capture is actually a feature, not a bug. For example:

```
(defmacro with-alternate-confrabulation (confrabulator &body body)
  `(flet ((confrabulate (arg) (,confrabulator arg))) ,@body))
```

This is a real problem for lexicon-based macros because we can't use multiple symbols with the same name to distinguish between capturable and non-capturable references in macroexpansions (since the whole point of lexicons is to have a one-to-one correspondence between names and symbols, and use lexical environments at compile time to establish the symbol's semantics). But we have to distinguish between them somehow, otherwise lexicon-based macros would be strictly less powerful than regular macros, and no one would want to use them[1].

The "traditional" solution to this problem is hygienic macro systems, but personally I find them horribly confusing, and they tend to come with a lot of baggage (like little mini-languages for defining macros) that I am not fond of. Fortunately, there is a sneaky middle-ground solution that is not fully hygienic, but which (I believe) solves the problem at least to the same extent that packages solve it, which is probably Good Enough.

The solution, in a nutshell, is:

> Within a backquote, precede any function call or global variable reference that you want to be non-shadowable with a carat (^).

For example:

```
? (ldefun confrabulate (arg) (list 'confrabulated arg))
; Warning: The function CONFRABULATE is being redefined as a macro.
; While executing: (SETF MACRO-FUNCTION), in process Listener(111).
CONFRABULATE
? (ldefmacro mymacro (arg) `(list (confrabulate ',arg) (^confrabulate
',arg)))
MYMACRO
? (flet ((confrabulate (arg) (list 'alternate-confrabulation arg))) (mymacro
foo))
((ALTERNATE-CONFRABULATION FOO) (CONFRABULATED FOO))
?
```

---

[1] Historical note: solving this one problem delayed the release of lexicons for five years because I really didn't want to use a fully hygienic macro system. Everything else in lexicons is relatively straightforward. I am indebted to Pascal Costanza for providing the crucial hint that broke the logjam.

(The warning gives you a little clue about what's going on under the hood.)

The carat is intended to be a mnemonic for "top level binding".

Note that while it is possible to implement the carat syntax using a reader macro, it's not actually done that way in the current implementation. Instead, the LDEF* macros simply define two symbols — the one being defined and a symbol with the same name with a prepended carat — as referring to the same binding. So "top-level" references *can* be shadowed. It is up to the programmer not to abuse this loophole.

3. How it works

3.1 Basic design

Conceptually, lexicons are associative maps (i.e. dictionaries) that map symbols onto storage locations. The original lexicon implementation actually used an abstract associative map layer that made the bindings themselves first-class, but that turned out to have several serious problems, mostly arising from the fact that the design of Common Lisp is heavily biased towards the use of symbols as designators for objects rather than the objects themselves. For example, the DEFCLASS macro requires that the list of superclasses be specified as a list of class names (which is to say, symbols). Specifying an actual class object is not allowed. This bias towards the use of symbols and names manifest itself even in the most fundamental construct there is, the compound form, which must begin with a function or macro name. Function objects are not allowed.

Rather than try to fight this, I eventually (after a lot of kicking and screaming) decided to leverage this aspect of Common Lisp's design in order to simplify the implementation, which has the nice side-effect of making lexicons integrate more smoothly into the CL package system. Lexicons are actually implemented *using* packages. Every lexicon has an associated package with the same name as the lexicon. A symbol's binding in a lexicon is a (different) symbol with the same name interned that lexicon's associated package. See section 3.2 ("lexification") for more details on how this works.

In addition to the bindings that actually reside within them, lexicons also contain a list of *library* lexicons, and a *parent* lexicon from which it inherits bindings. Bindings are resolved by searching first the bindings that are in the lexicon proper, then the library lexicons, and then up the chain of parent lexicons, which normally terminate in the *root* lexicon. Note that library lexicons are searched for their resident bindings only. Libraries are not searched recursively. This allows libraries to emulate non-exported symbols by putting bindings that are to remain hidden from client lexicons in a library of their own and then using that library. For example:

```
? (make-lexicon :hidden)
#<lexicon HIDDEN>
? (in-lexicon :hidden)
#<lexicon HIDDEN>
? (ldefun hidden-helper () 'result)
```

```
HIDDEN-HELPER
? (in-lexicon :l2)
#<lexicon L2>
? (use-lexicon :hidden)
(#<lexicon HIDDEN>)
? (ldefun libfn () (hidden-helper))
LIBFN
? (in-lexicon :l1)
#<lexicon L1>
? (libfn)
RESULT
? (hidden-helper)
; Warning: Defering lexical binding of HIDDEN-HELPER
; While executing: *REF, in process Listener(111).
Resolving lexical binding of HIDDEN-HELPER
> Error: HIDDEN-HELPER is not bound in the FUNCTION namespace of #<lexicon
L1>
```

(The utility of parent lexicons is not altogether clear, and they may go away some day unless someone comes up with a good use for them. The only reason I included them is that it was easy to do, and it reflects the structure of lexical environments created in the "usual" way, with nested lambda bindings. Whether this structure is useful for global lexicals is not clear.)

In the previous implementation of lexicons, the bindings in each lexicon were divided up into first-class namespaces, but this feature has gone away now that bindings are symbols. Most CL fans will probably not count this as a great loss, but there is one feature of the old implementation that I personally will miss, namely, the ability to merge the function and value namespace and create a "Lisp-1 lexicon". For now, with lexicons more seamlessly integrated into CL, all lexicons are Lisp-N environments.

3.2 Lexification

"Lexification" is the process of turning a symbol into a global lexical. It consists of defining both a macro and a symbol macro for the symbol, both of which expand into a derefencing of the symbol's binding in the current (at compile time) lexicon.

It is important to lexify all symbols before they are used. This is because if the compiler gets a hold of an undefined symbol before it is lexified then it will be compiled as a regular CL function call (or a regular CL special variable reference). There is no way to retroactively change this without recompiling the function.

The Really Right Way (IMO) to solve this problem is to modify Common Lisp to add a compile-time "undefined function call hook" and "undefined global variable reference hook" that allows one to intercept the compilation of undefined function calls and undefined global variable references and do some user-defined thing (which in our case would be to lexify the symbol before returning control back to the compiler). But this is unlikely to happen. The next best thing is to hack in this functionality on an implementation-by-implementation bases. The next best thing is to lexify all symbols in the bodies of

LDEFUN and LDEFMACRO forms whether they need it or not. The least attractive option is to leave it up to the user to make sure everything is lexified that needs to be.

The current implementation takes that last approach simply because it's the easiest to implement and the least anti-social. But it is high on my priority list to come up with something better.

3.3 Deferred bindings

Lexification does not actually establish a binding for the symbol, it merely establishes the symbol as a global lexical. Because of this, it is possible to dereference a global lexical before it has a binding. Naively, this would cause a compile-time error (because lexical rererences are normally resolved at compile time). But this is unacceptable because it would make it cumbersome (though not impossible) to define mutually-recursive functions. One function would need to be bound -- but not defined -- before the other function could be defined. The first function would then need to be redefined.

To avoid saddling the user with this annoyance, lexicons provide a mechanism to defer the resolution of a lexical binding until run-time. The macros produced by the lexification process do not return the binding directly, but rather return a *form* that, when evaluated, returns the binding. If the binding is known at compile-time then this form returns the symbol's binding directly, i.e. the form is (QUOTE <binding>). If the binding is not known, then the form is a closure (actually, it is a list whose second element is a closure and whose first element is the symbol FUNCALL) that, when called, makes another attempt to look up the binding of the symbol *in the same lexicon that was used for the original (failed) lookup*. If that lookup succeeds, the resulting cell is stored in the closure and is returned directly on subsequent calls to the closure. Thus, deferred bindings are not quite as efficient as non-deferred bindings, but the overhead is very small.

3.4 The semi-hygienic macro hack

Because lexicons are intended to be (at least potentially) a replacement for packages they need to provide a solution for unintentional name capture in macros. For "downward" capture (i.e. the macro defines a symbol that is used in code passed to the macro as an argument) the solution remains the same as in Common Lisp: use gensyms where appropriate.

For "upward capture" (i.e. the macro makes what is intended to be a top-level reference that is shadowed by enclosing user code) lexicons provide an ad hoc solution in the form of new syntax: a symbol preceded by a carat (e.g. ^FOO) is unconditionally a global reference to that symbol in the current (at compile time) lexicon. See section 2.4 for an example.

The conceptually pure way to implement this would have been to use a reader macro for the carat character that would read ^FOO as something like (TOP-LEVEL-LEXICAL-REFERENCE FOO). That is not how it's actually done. Instead, this is taken care of in the lexification process. When a symbol FOO is lexified, both FOO and ^FOO are lexified *with the same macro expansions*, that is, with the macro expansions that dereference the symbol FOO. Thus, ^FOO is not "really" a

global reference, but relies on constraining the programmer not to bind variables whose names start with a carat.

3.5 LDEFMETHOD and lexical classes

There is one feature that "normal" Common Lisp has that is not reflected in lexicons. In normal CL you can define a class that inherits from a class that does not exist. It is not possible to implement lexicons in portable Common Lisp and preserve this feature. This is because the symbols that name the parent classes in the DEFCLASS macro are not symbol-macroexpanded; they are simply used verbatim. So the class names of all parent classes for a lexical class must be known at compile time, at least if you want LDEFCLASS to expand into DEFCLASS. It might be possible to implement the equivalent of deferred bindings for parent classes using the MOP, but this is probably more trouble than it is worth. Inheritance graphs are DAGs; they cannot form loops. So it is probably not a great burden on the programmer to require them to define their classes according to some total ordering of the inheritance graph.

4. Final thoughts

Lexicons are still a work in progress. Please send suggestions for improvement to me at ron at flownet dot com.

References

[1] The Ciel Manifesto. http://www.flownet.com/gat/ciel.pdf

[2] The Idiot's Guide to Special Variables and Lexical Closures. http://www.flownet.com/ron/specials.pdf

[3] The Idiot's Guide to Common Lisp Packages. http://www.flownet.com/ron/packages.pdf

[4] The Common Lisp Hyperspec. http://www.lisp.org/HyperSpec/FrontMatter/index.html