Quickref: Common Lisp reference documentation as a stress test for Texinfo

Didier Verna

Abstract

Quickref is a global documentation project for the Common Lisp ecosystem. It creates reference manuals automatically by introspecting libraries and generating corresponding documentation in Texinfo format. The Texinfo files may subsequently be converted into PDF or HTML. Quickref is non-intrusive: software developers do not have anything to do to get their libraries documented by the system.

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Common Lisp ecosystem at once. The result is a website containing almost two thousand reference manuals. Quickref provides a Docker image for an easy recreation of this website, but a public version is also available and actively maintained.

Quickref constitutes an enormous and successful stress test for Texinfo. In this paper, we give an overview of the design and architecture of the system, describe the challenges and difficulties in generating valid Texinfo code automatically, and put some emphasis on the currently remaining problems and deficiencies.

1 Introduction

Lisp is a high level, general purpose, multi-paradigm programming language created in 1958 by John McCarthy [2]. We owe to Lisp many of the programming concepts that are still considered fundamental today (functional programming, garbage collection, interactive development, *etc.*). Over the years, Lisp has evolved as a family of dialects (including Scheme, Racket, and Clojure, to name a few) rather than as a single language. Another Lisp descendant of notable importance is Common Lisp, a language targeting the industry, which was standardized in 1994 [5].

The Lisp family of languages is mostly known for two of its most prominent (and correlated) characteristics: a minimalist syntax and a very high level of expressiveness and extensibility. The root of the latter, right from the early days, is the fact that code and data are represented in the same way (a property known as *homoiconicity* [1, 3]). This makes meta-programming not only possible but also trivial. Being a Lisp, Common Lisp not only maintains this property, but also provides an unprecedented arsenal of paradigms making it much more expressive and extensible than its industrial competitors such as C++ or Java.

Interestingly enough, the technical strengths of the language bring serious drawbacks to its community of programmers (a phenomenon affecting all the dialects). These problems are known and have been discussed many times [4, 7]. They may explain, at least partly, why in spite of its technical potential, the Lisp family of languages never really took over, and probably never will. The situation can be summarized as follows: Lisp usually makes it so easy to "hack" things away that every Lisper ends up developing his or her own solution, inevitably leading to a *paradox of choice*. The result is a plethora of solutions for every single problem that every single programmer faces. Most of the time, these solutions work, but they are either half-baked or targeted to the author's specific needs and thus not general enough. Furthermore, it is difficult to assert their quality, and they are usually not (well) documented.

As this situation is well known, the community has been attempting to "consolidate" itself in various ways. Several websites aggregate resources related to the language or its usage (books, tutorials, implementations, development environments, applications, *etc.*). The Common Lisp Foundation (`cl-foundation.org`) provides technical, sometimes even financial, support and infrastructure for project authors. Once a year, the European Lisp Symposium (`european-lisp-symposium.org`) gathers the international community, open equally to researchers and practitioners, newcomers and experts.

From a more technical standpoint, solving the paradox of choice, that is, deciding on official solutions for doing this or that, is much more problematic — there is no such thing as an official authority in the community. On the other hand, some libraries do impose themselves as *de facto* standards. Two of them are worth mentioning here. Most non-trivial Common Lisp packages today use ASDF for structuring themselves (fig.1 has an example). ASDF allows you to define your package architecture in terms of source files and directories, dependencies and other metadata. It automates the process of compiling and loading (dependencies included). The second one is Quicklisp (`quicklisp.org`). Quicklisp is both a central repository for Common Lisp libraries (not unlike CTAN) and a programmatic interface for it. With Quicklisp, downloading, installing, compiling and loading a specific package on your machine (again, dependencies included) essentially becomes a one-liner.

One remaining problem is that of documentation. Of course, it is impossible to force a library author to properly document his or her work. One

```
(asdf:defsystem :net.didierverna.declt
  :long-name "Documentation Extractor from Common Lisp to Texinfo"
  :description "A reference manual generator for Common Lisp libraries"
  :author "Didier Verna"
  :mailto "didier@didierverna.net"
  :homepage "http://www.lrde.epita.fr/~didier/software/lisp/"
  :source-control "https://github.com/didierverna/declt"
  :license "BSD"
  ...)
```

Figure 1: ASDF system definition excerpt

could consider writing the manuals they miss for the third-party libraries they use, but this never happens in practice. There is still something that we can do to mitigate the issue, however. Because Common Lisp is highly reflexive, it is relatively straightforward to retrieve the information necessary to automatically create and typeset *reference* manuals (as opposed to *user* manuals). Several such projects exist already (remember the paradox of choice). In this paper we present our own, probably the most complete Common Lisp documentation generator to date.

Enter Quickref…

2 Overview

Quickref is a global documentation project for the Common Lisp ecosystem. It generates reference manuals for libraries available in Quicklisp automatically. Quickref is non-intrusive, in the sense that software developers do not have anything to do to get their libraries documented by the system: mere availability in Quicklisp is the only requirement. In this section, we provide a general overview of the system's features, design, and implementation.

2.1 Features

Quickref may be used to create a local website documenting your current, partial, working environment, but it is also able to document the whole Quicklisp world at once, which means that almost two thousand reference manuals are generated. Creating a local documentation website can be done in two different ways: either by using the provided Docker image (the most convenient solution for an exhaustive website), or directly via the programmatic interface, from within a running Lisp environment (when only the documentation for the local, partial, installation is required). If you don't want to run Quickref yourself, a public website is also provided and actively maintained at `quickref.common-lisp.net`. It always contains the result of a full run of the system on the latest Quicklisp distribution.

2.2 Documentation items

Reference manuals generated by Quickref contain information collected from various sources. First of all, many libraries provide a README file of some sort, which can make for a nice introductory chapter. In addition to source files and dependencies, ASDF offers ways to specify project-related metadata in the so-called *system definition* form. Figure 1 illustrates this. Such information can be easily (programmatically) retrieved and used. Next, Lisp itself has some built-in support for documentation, in the form of so-called *docstrings*. As their name suggests, docstrings are (optional) documentation strings that may be attached to various language constructs such as functions, variables, methods and so on. Figure 2 has an example. When available, docstrings greatly contribute to the completeness of reference manuals, and again, may be retrieved programmatically through a simple standard function call.

```
(defmacro @defconstant (name &body body)
  "Execute BODY within a @defvr Constant.
NAME is escaped for Texinfo prior to rendering.
BODY should render on *standard-output*."
  `(@defvr "Constant" ,name ,@body))
```

Figure 2: Common Lisp docstring example

As for the rest, the solution is less straightforward. We want our reference manuals to advertise as many software components as possible (functions, variables, classes, packages, *etc.*). In general there are two main strategies for collecting this information.

Code walking. The first one, known as *code walking*, consists of statically analyzing the source code. A code walker is usually at least as complicated as the syntax of the target language, because it requires a parser for it. Because of Lisp's minimalist syntax, using a code walker is a very tempting solution. On the other hand, Lisp is extremely dynamic in nature, meaning that many of the final program's components may not be directly visible in the source

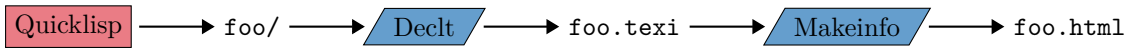

Figure 3: Quickref pipeline (■ Main thread, ▱ External Process)

code. On top of that, programs making syntactic extensions to the language would not be directly parsable. In short, it is practically impossible to collect all the required information by code walking alone. Therefore, we do not use that approach.

Introspection. Our preferred approach is by *introspection*. Here, the idea is to actually compile and load the libraries, and then collect the relevant information by inspecting memory. As mentioned before, the high level of reflexivity of Lisp makes introspection rather straightforward. This approach is not without its own drawbacks however. First, actually compiling and loading the libraries requires that all the necessary (possibly foreign) components and dependencies are available. This can turn out to be quite heavy, especially when the two thousand or so Quicklisp libraries are involved. Secondly, some libraries have platform, system, compiler, or configuration-specific components that may or may not be compiled and loaded, depending on the exact conditions. If such a component is skipped by our system, we won't see it and hence we won't document it. We think that the simplicity of the approach by introspection greatly compensates for the risk of missing a software component here and there. That is why introspection is our preferred approach.

2.3 Toolchain

Figure 3 depicts the typical manual production pipeline used by Quickref, for a library named `foo`.

1. Quicklisp is used first, to make sure the library is installed, which results in the presence of a local directory for that library.
2. Declt (`lrde.epita.fr/~didier/software/lisp/misc.php#declt`) is then run on that library to generate the documentation. Declt is another library of ours, written five years before Quickref, but with that kind of application in mind right from the start. In particular, it is for that reason that the documentation generated by Declt is in Texinfo intermediate format.
3. The Texinfo file is processed into HTML. Texinfo (`gnu.org/software/texinfo`) is the GNU official documentation format. There are three main reasons why this format was chosen when Declt was originally written. First, it is particularly well suited to technical documentation. More importantly, it is designed as an abstract, intermediate format from which human-readable documentation can in turn be generated in many different forms (notably PDF and HTML). Finally, it includes very convenient built-in anchoring, cross-referencing, and indexing capabilities.

Quickref essentially runs this pipeline on the required libraries. Some important remarks need to be made about this process.

1. Because Declt works by introspection, it would be unreasonable to load almost two thousand libraries in a single Lisp image. For that reason, Quickref doesn't actually run Declt directly, but instead forks it as an external process.
2. Similarly, `makeinfo` (`texi2any` in fact), the program used to convert the Texinfo files to HTML, is an external program written in Perl (with some parts in C), not a Lisp library. Thus, here again, we fork a `makeinfo` process out of the Quickref Lisp instance in order to run it.

2.4 Performance

Experimental studies have been conducted on the performance of the system. There are different scenarios in which Quickref may run, depending on the exact number of libraries involved, their current state, and the level of required "isolation" between them. All the details are provided in [6], but in short, there is a compromise to be made between the execution time and the reliability of the result. We found that for a complete sequential run of the system on the totality of Quicklisp, the most frequent scenario takes around two hours on our test machine, whereas the safest one requires around seven hours.

In order to improve the situation, we recently added support for parallelism to the system. The upgraded architecture is depicted in Figure 4. In this new processing scheme, an adjustable number of threads is devoted to generating the Texinfo files in parallel. In a second stage, a likewise adjustable number of threads is in charge of taking the Texinfo files as they come, and creating the corresponding HTML versions. A specific scheduling algorithm (not unlike that of the `make` program) delivers libraries in an order, and at a time suitable to parallel processing by the Declt threads, avoiding any concurrency problems. With this new architecture in place, we were able to cut the processing time by a factor of four, reducing the worst case scenario to 1h45 and the most frequent one to half an hour. These numbers

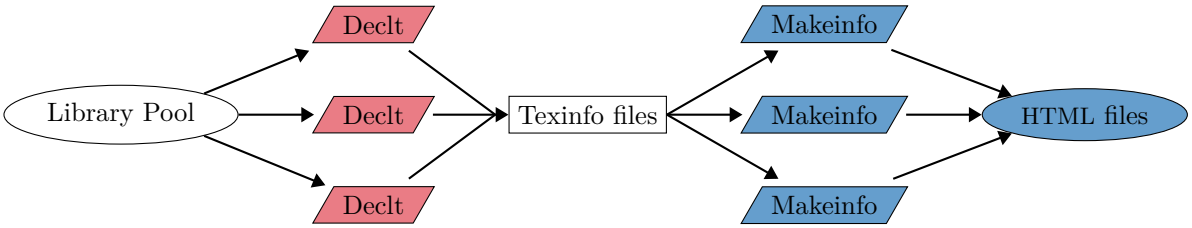

Figure 4: Quickref parallel processing (Declt thread, Makeinfo thread)

make it reasonable to run Quickref on one's local machine again.

3 Challenges

Quickref is a challenging project in many regards. Two thousand libraries is *a lot* to process. Setting up the environment necessary to properly compile and run those libraries is not trivial, especially because many of them have platform or system-specific code and require foreign dependencies. Finally, Quickref constitutes a considerable (and successful) stress test for Texinfo. The Texinfo file sizes range from 7KB to 15MB (double that for the generated HTML ones). The number of lines of Texinfo code in those files extends from 364 to 285 020, the indexes may contain between 14 and 44 500 entries, and the processing times vary from 0.3s to 1m 38s per file.

Challenges related to the project scalability and performance have been described previously [6]. This section focuses on more general or typesetting/Texinfo issues.

3.1 Metadata format underspecification

One difficulty in collecting metadata is that their format is often underspecified, or not specified at all, as is the case with ASDF system items. To give just one example, Figure 5 lists several of the possible values we found for the author metadata. As you can see, most programmers use strings, but the actual contents vary greatly (single or multiple names, email addresses, middle letter, nicknames, *etc.*), and so does the formatting. For the anecdote, we found one attempt at pretty printing the contents of the string with a history of authors, and one developer even went as far as concealing his email address by inserting Lisp code into the string itself…

It would be unreasonable to even try to understand all these formats (what others will we discover in the future?), so we remain somewhat strict in what we recognize — in this particular case, strings either in the form `"author"` or `"author <email>"` (as in either:

```
"Didier Verna"
"Didier Verna <didier@lrde.epita.fr>"
```

respectively), or a list of these. The Declt user manual has a Guidelines section with some advice for library authors that would like to be friendlier with our tool. We cannot force anyone to honor our guidelines however.

```
"Didier Verna"
"Didier Verna <didier@lrde.epita.fr>"
"Didier Verna didier@lrde.epita.fr"
"didier@lrde.epita.fr"
"<didier@lrde.epita.fr>"
"Didier Verna and Antoine Martin"
"Didier Verna, Antoine Martin"
"Didier Verna Antoine Martin"
"D. Verna Antoine E Martin"
"D. Verna Antoine \"Joe Cool\" Martin"
("Didier Verna" "Antoine Martin")
"
     Original Authors:
       Salvi Péter,
       Naganuma Shigeta,
       Tada Masashi,
       Abe Yusuke,
       Jianshi Huang,
       Fujii Ryo,
       Abe Seika,
       Kuroda Hisao
     Author Post MSI CLML Contribution:
       Mike Maul  <maul.mike@gmail.com>"
"(let ((n \"Christoph-Simon Senjak\"))
          (format nil \"~A <~C~C~C~C~A>\"
            n (elt n 0) (elt n 10) (elt n 16)
            #\\@ \"uxul.de\"))"
```

Figure 5: ASDF author metadata variations

On the other hand, Quickref has an interesting social effect that we particularly noticed the first time the public website was released. In general, people don't like *our* documentation for *their* work to look bad, especially when it is publicly available. In the first few days following the initial release and announcement of Quickref, we literally got dozens of reports related to typesetting glitches. Programmers *rushed* to the website in order to see what *their* library looked like. If the bugs were not on our side, many of the concerned authors were hence willing

to slightly bend their own coding style, in order for *our* documentation to look better. We still count on that social effect.

3.2 Definitions grouping

Rather than just providing a somewhat boring list of functions, variables, and other definitions, as reference manuals do, Declt attempts to improve the presentation in different ways. In particular, it tries to group related definitions together when possible.

A typical example of this is when we need to document accessors (readers and writers to the same information). It makes sense to group these definitions together, provided that their respective docstrings are either nonexistent, or exactly the same (this is one of the incentives given to library authors in the Declt guidelines). This is exemplified in Figure 6. Another typical example consists in listing methods (in the object-oriented sense) within the corresponding generic function's entry.

`context-hyperlinksp` *CONTEXT* [Function]
`(setf context-hyperlinksp)` *BOOL CONTEXT* [Function]
Access *CONTEXT*'s hyperlinksp flag.

Package [`net.didierverna.declt`], page 29,
Source [`doc.lisp`], page 24, (file)

Figure 6: Accessors definitions grouping

Texinfo provides convenient macros for defining usual programming language constructs (`@defun`, `@defvar`, *etc.*), and "extended" versions for adding sub-definitions (`@defunx`, `@defvarx`, *etc.*). Unfortunately, definitions grouping prevents us from using them, for several reasons.

1. Nesting `@def...` calls would lead to undesirable indentation.
2. Heterogeneous nesting is prohibited. For example, it is not possible use `@defvarx` within a call to `@defun` (surprising as it may sound, this kind of heterogeneous grouping makes sense in Lisp).

On the other hand, that kind of thing is possible with the lower-level (more generic) macros, as heterogeneous *categories* become simple macro arguments. One can, for example use the following (which we frequently do):

```
@deffn {Function} ...
@deffnx {Compiler Macro} ...
...
@end deffn
```

This is why we stick to those lower-level macros, at the expense of re-inventing some of the higher-level built-in functionality.

Even with this workaround, some remaining limitations still get in our way.

1. There are only nine canonical categories and it is not possible to add new ones (at least not without hacking Texinfo's internals).
2. Although we understand the technical reasons for it (parsing problems, probably), some of the canonical categories are arguable. For example, the distinction between typed and untyped functions makes little sense in Common Lisp which has optional static typing. We would prefer to have a single function definition entry point handling optional types.
3. Heterogeneous mixing of the lower-level macros is still prohibited. For example, it remains impossible to write the following (still making sense in Lisp):

```
@deffn {Function} ...
@defvrx {Symbol Macro} ...
...
@end deffn
```

3.3 Pretty printing

Pretty printing is probably the biggest challenge in typesetting Lisp code, because of the language's flexibility. In particular, it is very difficult to find the right balance between readability and precision.

Identifiers. In Lisp, identifiers can be basically *anything*. When identifiers contain characters that are normally not usable (*e.g.* blanks or parentheses), the identifier must be escaped with pipes. In order to improve the display of such identifiers, we use several heuristics.

- A symbol containing blank characters is normally escaped like this: `|my identifier|`. Because the escaping syntax doesn't look very nice in documentation, we replace blank characters with more explicit Unicode ones, for instance `my␣identifier`. We call this technique "revealing". Of course, if one identifier happens to contain one of our revealing characters already, the typesetting will be ambiguous. This case is essentially nonexistent in practice, however.
- On the other hand, in some situations it is better to *not* reveal the blank characters. The so-called *setf* (setter / writer) functions are such an example. Here, the identifier is in fact composed of several symbols, such as in `(setf this)`. Revealing the whitespace character would only clutter the output, so we leave it alone.
- Finally, some unusual identifiers that are normally escaped in Lisp, such as `|argument(s)|`,

do not pose any readability problems in documentation, so we just typeset them without the escaping syntax.

Qualification. Another issue is symbol *qualification*. With one exception, symbols in Lisp belong to a *package* (more or less the equivalent of a namespace). Many Lispers use Java-style package names, which can end up being quite long. Typesetting a fully qualified symbol would give something like this: `my.long.package.name:symbol`. Lisp libraries usually come with their own very few packages, so typesetting a reference manual with thousands of symbols fully qualified with the same package name would look pretty bad. Because of that, we avoid typesetting the package names in general. Unfortunately, if different packages contain eponymous symbols, this leads to confusing output. Currently, we don't have a satisfactory answer to this problem.

Docstrings. The question of how to typeset docstrings is also not trivial. People tend to use varying degrees of plain-text formatting in them, with all kinds of line lengths, *etc.* Currently, we use only a very basic heuristic to determine whether an end of line in a docstring is really wanted here, or just a consequence of reaching the "right margin". We are also considering providing an option to simply display the docstrings verbatim. In the long term, we plan to support markup languages such as Markdown.

References. A Texinfo-related problem we have is that links are displayed differently, depending on the output format, and with some rather undesirable DWIM behavior. Table 1 shows the output of a call to `@ref{anchor, , label}` in various formats (`anchor` is the link's internal name, `label` is the desired output).

Table 1: Texinfo links formatting in various output formats

HTML	label
PDF	[label], page 12,
Info	*note label: anchor.
Emacs Info mode	See label.

In PDF, the presence of the trailing comma is context dependent. In Info, both the label and the actual anchor name are typeset, which is very problematic for us (see Section 3.4). In Emacs Info mode, the casing of "See" seems to vary. In general, we would prefer to have more consistent output across the different formats, or at least, more control over it.

3.4 Anchoring

The final Texinfo challenge we want to address here is that of anchoring. In Texinfo, anchor names have severe limitations: dots, commas, colons, and parentheses are explicitly forbidden (due to the final display syntax in Info). This is very unfortunate because those characters are extremely common in Lisp (parentheses of course, but also dots and colons in the package qualification syntax).

Note that formats other than Info are not affected by this problem. There is an Info-specific workaround documented in Appendix G.1 of the Texinfo user manual. In short, a sufficiently recent version can automatically "protect" problematic node names by surrounding them with a special marker in the resulting Info files. Unfortunately, neither older Info readers, nor the current Emacs mode are aware of this feature. Besides, the latest stable release of Texinfo still has problems with it (menus do not work correctly). Consequently, this workaround is not a viable solution for us (yet).

Our original (and still current) solution is to replace those characters by a sequence such as `<dot>`. Of course, this makes anchor names particularly ugly, but we didn't think that was a problem because we have nicer *labels* to point to them in the output (in fact, labels have a less limited syntax, although this is not well documented). However, we later realized that anchor names still appear in the HTML output and also in pure Info. Consequently, we are now considering changing our escaping policy, perhaps by using Unicode characters as replacements, just as we already do on identifiers (see Section 3.3).

The second anchoring problem we have is that of Texinfo nodes, the fundamental document structuring construct. In addition to the aforementioned restrictions related to anchoring, nodes have two very strong limitations: their names must be unique and there is no control over the way they are displayed in the output. This is a serious problem for us because Lisp has a lot of different namespaces. A symbol may refer to a variable, a function, a class, and many other things at the same time. Consequently, when nodes are associated with Lisp symbols, we need to mangle their names in a way that makes them barely human readable. Because of that, our use of nodes remains rather limited, which is somewhat paradoxical, given the importance of nodes in Texinfo. Apart from general, high level sectioning, the only nodes associated with Lisp symbols are for ASDF components and packages, probably already a bit too much. It is our hope that one day, the node names uniqueness constraint in Texinfo might be relaxed, perhaps disambiguating by using their hierarchical organization.

4 Conclusion and perspectives

Although a relatively young project, Quickref is already quite successful. It is able to document almost two thousand Common Lisp libraries without any showstoppers. Less than 2% of the Quicklisp libraries still pose problems and some of the related bugs have already been identified. The Common Lisp community seems generally grateful for this project.

Quickref also constitutes an enormous, and successful, stress test for Texinfo. Given the figures involved, it was not obvious how `makeinfo` would handle the workload, but it turned out to be very reliable and scalable. Although the design of Texinfo sometimes gets in our way, we still consider it a good choice for this project, in particular given the diversity of its output formats and its built-in indexing capabilities.

In addition to solving the problems described in this paper, the project also has much room for improvement left. In particular, the following are at the top level of our TODO list.

1. The casing problem needs to be addressed. Traditional Lisp is case-insensitive but internally upcases every symbol name (except for escaped ones). Several modern Lisps offer alternative policies with respect to casing. Quickref doesn't currently address casing problems at all (not even that of escaped symbols).
2. Our indexing policy could be improved. Currently, we only use the built-in Texinfo indexes (Functions, Variables, *etc.*) but we also provide one level of sub-indexing. For instance, macros appear in the function index, but they are listed twice: once as top level entries, and once under a Macro sub-category. The question of which amount of sub-indexing we want, and whether to create and use new kinds of indexes is under consideration.
3. Although our reference manuals are already stuffed with cross-references, we plan to add more. Because Declt was originally designed to generate one reference manual at a time, only internal cross-references are available. The existence of Quickref now raises the need for external cross-references (that is, between different manuals).
4. Many aspects of the pretty printing could be improved, notably that of so-called "unreadable" objects and lambda lists.
5. In addition to HTML, we plan to provide PDF as well as Info files on the website, since they are readily available.
6. We intend to integrate Quickref with Emacs and Slime (a *de facto* standard Emacs-based development environment for Common Lisp). In particular, we want to give Emacs the ability to browse the Info reference manuals online or locally if possible, and provide Slime with commands for opening the Quickref documentation directly from Lisp source code displayed in Emacs buffers.
7. Finally, we are working on providing new index pages for the website. Currently, we have a library index and an author index. We are working on providing keyword and category indexes as well.

References

[1] A. C. Kay. *The Reactive Engine*. PhD thesis, University of Utah, 1969.

[2] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* 3(4):184–195, Apr. 1960. `doi:10.1145/367177.367199`

[3] M. D. McIlroy. Macro instruction extensions of compiler languages. *Communications of the ACM* 3:214–220, Apr. 1960. `doi:10.1145/367177.367223`

[4] M. Tarver. The bipolar Lisp programmer. `marktarver.com/bipolar.html`, 2007.

[5] ANSI. American National Standard: Programming Language — Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[6] D. Verna. Parallelizing Quickref. In *12th European Lisp Symposium*, pp. 89–96, Genova, Italy, Apr. 2019. `doi:10.5281/zenodo.2632534`

[7] R. Winestock. The Lisp curse, Apr. 2011. `winestockwebdesign.com/Essays/Lisp_Curse.html`

◇ Didier Verna
14-16 rue Voltaire
94276 Le Kremlin-Bicêtre
France
`didier (at) lrde dot epita dot fr`
`http://www.didierverna.info`