Maintaining Portable Lisp Programs

It's a bug, not a feature

Christophe Rhodes*

February 5, 2004

Abstract

We examine the use of read-time feature conditionals, with particular emphasis on writing portable Common Lisp code which aspires to both forwards- and backwards-compatibility. We examine cases from real libraries which demonstrate the various pitfalls, and propose a scheme for robust treatment of implementation-specific functionality by performing programmatic tests at the appropriate time. We discuss the additional difficulties in dealing with differences in the operating environment, and suggest that agreements between Lisp distributors would alleviate these difficulties.

1 Introduction

Common Lisp is enjoying a growth in interest, driven by many factors: the steady improvement of free and low-cost implementations (see e.g. [1–3]); high-quality libraries for common programming tasks are more likely to be available; and the suitability of Common Lisp environments for solving complex problems [4–6] to name but three. With this growth in interest comes a fresh perspective on the language and its idioms, and it is in this light that we examine one problem that historically has had multiple solutions, some more successful than others.

Common Lisp is by design a specified, multiply-implemented language. The first specification [7] was a design by consensus of a new dialect of Lisp attempting to unify the various development communities of MacLisp, Zetalisp, Spice Lisp, NIL and S-1 Lisp, among others. As such, it was deemed necessary from the start to provide a means for querying the implementation for its characteristics, to allow the programmer to take advantage of system-specific features when they are available, yet preserve the portability of the program.

The mechanism for achieving this is the use of read-time conditionals, which causes the program text seen by the Lisp compiler to be dependent on the contents of the variable `*features*`; if a given feature keyword `:feature` is present in the `*features*` list, then

```
(#+feature a #-feature b)
```

is read in as the singleton list `(a)`; if it is not present, the reader returns `(b)`.

The Common Lisp standard underwent a decade-long revision process under the auspices of ANSI, being finalised in 1994 as the ANSI standard for Common Lisp [8]. As part of this process, several keyword features were given standard meanings; among them were `:cltl2`, `:ansi-common-lisp` and `:ieee-floating-point`. Thus, Common Lisp programs can completely portably query the implementation for whether it purports to conform to the ANSI Common Lisp standard, or to IEEE 754 standard for floating point.

*C.S.Rhodes@damtp.cam.ac.uk

> The `#+` construction must be used judiciously if unreadable code is not to result. The user should make a careful choice between read-time conditionalisation and run-time conditionalisation.
>
> — *Common Lisp the Language*, section 22.1.4

This admonishment in the defining document of the language has been sadly, if not ignored, at least treated as less important than it should. The scope of the standardised features is limited, and with the advent of distributed development on publically-accessible Lisp programs, well-written read-time conditionals take on more importance to prevent a combinatorial explosion of maintenance headaches. We could expect that maintainablilty would be assisted by the ability to use the whole of the Common Lisp environment at read-time and at compile-time; however, as of today, some Lisp libraries cause unnecessary headaches for their maintainers and users. The aim of this article is to characterise the problems caused by injudicious use of `#+`, and to propose a recipe for robust read-time conditional treatment.

We shall begin in section 2 by looking at some historical attempts at rationalising the use of read-time conditionals by implementors and by users. In section 3, we examine some past and present uses of read-time conditionals in libraries, and discuss the problems found; we propose a method for robust use of read-time conditionals in section 4, and draw conclusions in section 5.

2 Historical Perspective

> It is generally considered wise for an implementation to include one or more features identifying the specific implementation, so that conditional expressions can be written which distinguish idiosyncrasies of one implementation from those of another.
>
> — *Common Lisp the Language*, section 22.1.4

There has been at several points in the past a strong perception that knowing about implementation-defined features would be a good thing; one such example is the existence on the community-driven CLiki [6] site of a page detailing the features present in a number of different implementations; before this, the Association of Lisp Users has attempted to maintain a central registry of implementation-defined features.

As we shall see, this perception of a need for centrally-defined features is largely misguided. We shall observe exceptions, but in general, the library writer or end-user of a Lisp implementation need not, and indeed should not, interact with the implementation-defined features present in `*features*` on the startup of the environment.

Recently, on the CMUCL development mailing list, a proposal was made [9] to standardise the treatment of version-indicating features, along the lines of the Genera software product; a scheme with a similar effect is present in Franz Inc.'s Allegro Common Lisp, using the `:version>=` extension to read-time conditionals. This proposal, providing for consistent treatment of major and minor software release numbers, is perhaps best illustrated by an example: consider version 3.2 of a product Foolisp; then the following keywords would be present on `*features*`:

```
:foolisp-3.2, :foolisp-3.1, :foolisp-3.0,
:foolisp-3,   :foolisp-2,   :foolisp-1
```

This allows for read-time conditionals which trigger on precise criteria (see table 1). However, there are problems with this. Firstly, this mapping disavows the existence of interim, unreleased versions of a software product.

Criterion	Conditional expression
Exactly version 3.2	`(and foolisp-3.2 (not foolisp-3.3))`
Version 3.2 or later	`(or foolisp-3.2 foolisp-4)`
Version 3, 3.2 or later	`foolisp-3.2`

Table 1: Conditional expressions for version discrimination in a hypothetical Foolisp implementation.

A system as complex as a Common Lisp compiler will typically undergo substantial revision between releases, whether these releases are time-boxed or scheduled when some set of criteria have been met. However, at least the core developers of a compiler, explicit beta-testers, and possibly many more people if development is happening in a distributed manner associated with Free Software, will attempt to test or even deploy with a system that is not a released version of the software. In such cases, the above, supposedly 'precise' criteria break down, because the assumption of discrete instances is not valid.

Secondly, these numerous revisions prove problematic in another respect: they strongly indicate that product evolution is likely to be non-monotonic. An underlying assumption behind the above scheme for managing `*features*` is that minor revisions of a product do not break interface compatibility, and to an extent this assumption is borne out in the real world. However, it is not too hard to envisage exceptions to this rule; they are typically found in the region where a bug fix blurs into an interface change. For instance, a common extension to the Common Lisp standard is the Gray interface for user stream redefinition [10], which did not specify `read-sequence` and `write-sequence` extensions; as a consequence, incompatible extensions coexisted for a while, before the interface being agreed across implementations, which necessitated breaking of backward compatibility in some cases.

The most problematic objection to software authors using this feature scheme, however, revolve around the implicit assumption that the software author has future knowledge that is, in fact, impossible to have. A request for conditional compilation for 'exactly version 3.2', for instance, assumes knowledge of the state of version 3.3: knowledge that the developer cannot have until close to the release of version 3.3. In other words, while this class of keyword feature provision by implementors is useful to the developer by providing the ability to express such a request during the course of maintenance of a piece of software, it is of no help in building software that requires less maintenance.

3 Uses of Read-Time Conditionals

> In effect, using `#+` or `#-` in a conforming program means that the variable `*features*` becomes just one more piece of input data to that program. Like any other data coming into a program, the programmer is responsible for assuring that the program does not make unwarranted assumptions on the basis of input data.
>
> — *Common Lisp the Language*, section 22.1.4

Let us examine some of the attempts that have been made in the past to write portable code, in order to evaluate what techniques are likely to cause problems and how we can avoid maintenance headaches.

Listing 1 From defsystem.lisp, revision 1.61 in CLOCC CVS.

```
(eval-when (compile load eval)
  #+(or (and allegro-version>= (version>= 4 0)) :mcl :sbcl)
  (pushnew :cltl2 *features*))
```

Listing 2 From defsystem.lisp, revision 1.61 in CLOCC CVS.

```
#+(and :cltl2 (not (or :cmu :clisp :sbcl
                       (and :excl (or :allegro-v4.0 :allegro-v4.1))
                       :mcl)))
(eval-when (compile load eval)
  (unless (find-package "MAKE")
    (make-package "MAKE" :nicknames '("MK") :use '("COMMON-LISP"))))
```

3.1 MK:DEFSYSTEM

We start our examination of previous treatments of read-time conditionals with excerpts from the MK:DEFSYSTEM utility for dealing with software compilation, originally by Mark Kantrowitz but now maintained by volunteers in the CLOCC project. This is a project with a venerable history; the revision logs go back to 1990 – well before the ANSI standard was finalised. Given this, it is perhaps not too surprising that the program has accreted some features undesirable to maintainers.

The intent in Listing 1 is to indicate compatibility with the second edition of Common Lisp the Language [11] "or above", but there are two problems with this. Firstly, the language as standardised by ANSI is not 'backward-compatible' with the standardisation snapshot of the second edition of Common Lisp the Language; therefore, depending on the rest of the code in the MK:DEFSYSTEM utility, this implicit assumption that sufficiently-conforming (with ANSI) implementations are good for programming to the CLtL2 'standard' is not safe. Secondly, the `:cltl2` feature is reserved to indicate *exactly* CLtL2 compatibility – the features `:cltl2` and `:ansi-common-lisp` are, for a conforming Common Lisp implementation, mutually exclusive, and other libraries should be able to depend on this. A better means of achieving the effect of the above would be to replace read-time conditional tests for `:cltl2` with tests for `(or :cltl2 :ansi-common-lisp)` in the remainder of the utility.

In listing 2 we see another headache for maintainers: arising from the fact that earlier Lisps did not support `defpackage`. This fragment attempts to emulate the package creation effect of `defpackage` in those lisps which do not have it. However, this is a perfect example for the non-maintainability of this style of porting; each time the MK:DEFSYSTEM library is ported to a new, fully standards-conforming Common Lisp, an entry for the implementation must be made in the above feature conditional. A better method for achieving the same end is shown in Listing 3.

The code in listing 4 is compensating for a CMUCL deficiency in versions earlier than version 17. However, it will fail to work in CMUCL 19a, when the `:cmu18` feature (which has been present throughout the released versions of the CMUCL 18 series) is removed. Although this is no longer an issue, as this workaround for a long-gone CMUCL deficiency was removed from MK:DEFSYSTEM in revision 1.61, preserving the support of ancient Lisp versions while behaving gracefully in the future could be achieved by an idiom of the form in listing 5.

The code in listing 6 deals with Franz Inc.'s Allegro Lisp's "modern" mode. However, be-

Listing 3 An improved version of listing 2.

```
(eval-when (compile eval)
  (unless (find-symbol "DEFPACKAGE" "COMMON-LISP")
    (pushnew :no-defpackage *features*)))
#+no-defpackage
(eval-when (compile load eval)
  (unless (find-package "MAKE")
    (make-package "MAKE" :nicknames '("MK") :use '("COMMON-LISP"))))
```

Listing 4 From defsystem.lisp, revisions before 1.61 in CLOCC CVS.

```
   :directory
   #-(and :cmu (not (or :cmu17 :cmu18)))
   directory
   #+(and :cmu (not (or :cmu17 :cmu18)))
   (coerce directory 'simple-vector)
```

Listing 5 An improved version of listing 4.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (when (typep (pathname-directory #p"") 'simple-vector)
    (pushnew 'directory-is-simple-vector *features*)))

[...]

  :directory
  #-mk::directory-is-simple-vector directory
  #+mk::directory-is-simple-vector (coerce directory 'simple-vector)
```

Listing 6 From defsystem.lisp, revision 1.61 in CLOCC CVS.

```
#+allegro
(eval-when (:compile-toplevel :load-toplevel :execute)
  (unless (or (find :case-sensitive common-lisp:*features*)
              (find :case-insensitive common-lisp:*features*))
    (if (or (eq excl:*current-case-mode* :case-sensitive-lower)
            (eq excl:*current-case-mode* :case-sensitive-upper))
        (push :case-sensitive common-lisp:*features*)
      (push :case-insensitive common-lisp:*features*))))
```

Listing 7 From cl-ilisp.lisp, revision 1.21 in ILISP CVS.

```
#+:clisp
(setq system::*command-index* (max 0 (- system::*command-index* 2)))
;; [do the same gross hack for ACL.  -- rgr, 27-Sep-02.]
#+allegro (setq tpl::*this-command-number*
                (max 0 (- tpl::*this-command-number* 2)))
```

Listing 8 From cl-ilisp.lisp, revision 1.21 in ILISP CVS.

```
#-(or sbcl :cormanlisp)
(eval-when (load eval)
  (when
    #+(and :CMU (or :CMU17 :CMU18))
    (eval:interpreted-function-p #'ilisp-matching-symbols)
    #-(and :CMU (or :CMU17 :CMU18))
    (not (compiled-function-p #'ilisp-matching-symbols))
    (ilisp-message *standard-output*
                   "File is not compiled, use M-x ilisp-compile-inits")))
```

cause it is pushing keywords onto `*features*` at run-time, not only at build time, it runs the risk of collision with other pieces of software which may not have the same understanding of `:case-sensitive` and `:case-insensitive` (for instance, another piece of software may use those keywords for choices of its own user interface). Also, there is a certain amount of scope for confusion in the use of these read-time conditionals, as `#+case-insensitive` presumably means the same as `#-case-sensitive` and *vice versa*.

3.2 ILISP

ILISP is an add-on package for GNU Emacs and XEmacs to facilitate interaction with so-called "inferior" lisps – Common Lisp (or Scheme) processes running as children of the Emacs editor session; it provides a means for requesting evaluation of specific forms, compilation of files or entire systems, querying the running Lisp for argument list or cross-referencing information, and other such functionality. Also, it provides a buffer for direct interaction with the underlying Lisp; see [12] for a discussion of its features and a comparison with other Emacs packages.

Since manipulating user interface internals (as in listing 7, for instance) is a common issue in the interaction between ILISP and a Lisp which displays a history number in its prompt, it should be factored into a function of its own. Otherwise, adding the implementation for SBCL's (optional) sb-aclrepl, or other customised read-eval-print loops, will cause this to degenerate over time into a mess of read-time conditionals.

Whether the implementation of this function is in one place, or once per interface, is largely (though see [13] for a stronger view) a matter of taste; we suggest the latter as this localises the effort of porting the relevant protocol for a new implementation, and the logic for finding the relevant interface definitions is trivially automatable using a system definition tool.

Listing 8 shows a more problematic case. It will fail to do what is intended in a number of cases: not only if CMUCL releases version 19 (with a similar failure cause as in listing 4), but also if cmucl changes the internal implementation of its minimally-compiled but not compiled-to-machine-code functions, or if SBCL or Corman Lisp add an interpreter to their environments

Listing 9 From Lisp-Dep/fix-sbcl.lisp, revision 1.7 in McCLIM CVS.

```
(eval-when (:compile-toplevel :execute)
  (when (find-package "SB-MOP")
    (pushnew :sb-mop *features*)))

(defpackage #:clim-mop
  (:use #+sb-mop #:sb-mop #-sb-mop #:sb-pcl)
  #-sb-mop
  (:shadowing-import-from #:sb-pcl #:eql-specializer-object))

(defmacro clim-lisp-patch:defconstant (symbol value &optional docu)
  `(defvar ,symbol ,value ,@(and docu (list docu))))
```

(these implementations are, at the time of writing, essentially compiler-only).

This form of introspection is difficult to arrange in a robust manner, however; there is no requirement for an implementation to advertise its evaluation model. Given this, it is probably best to punt on the possibility of a radical change in an implementation's evaluator, and to treat the inner feature conditionals as being a test for the existence of the `eval:interpreted-function-p`, better expressed in a manner similar to listing 3 above.

3.3 McCLIM

McCLIM [14] is an implementation from scratch of the Common Lisp Interface Manager (version 2.0) standard: a library for writing powerful user interfaces. By necessity, it uses functionality that is not completely standardised (such as in interfacing to graphical backends: at present X, OpenGL and PostScript implementations are extant), and for convenience it also makes strong use of the Metaobject Protocol [15]. Since these features do not have standard or even usual locations in Lisp implementations, the library must conditionalise to find and use them.

McCLIM has one file per supported implementation to execute the necessary conditionalisation to bring the implementation up to a common base of functionality. Listing 9 contains an excerpt from the file for the SBCL implementation, with some noteworthy features. Firstly, the test for finding the location of the MOP functionality is programmatic, not depending on a specific meaning of a given implementation-provided keyword; this is therefore a low-maintenance strategy for the library implementors. The use of the `:sb-mop` keyword is suboptimal, as it is, in some sense, part of the Lisp implementation's namespace (SBCL uses keywords starting with `:sb-` to customise its own build); the effect on this code of the SBCL implementors deciding to define a meaning for the `:sb-mop` keyword themselves is not necessarily disastrous, but it could certainly cause problems; it would probably have been better to use a symbol such as `clim-lisp-patch::sb-mop` for later discrimination instead (see also listing 5 for another example of this technique). It is perhaps also worth noting the use of a patch package to transparently and conformingly alter the behaviour of the Lisp implementation.

4 Recommendations

By now, it should be clear how to design robust, maintainable feature conditional expressions. The recipe has several elements, in decreasing order of importance:

- test programmatically for the existence of the functionality or interface required;
- avoid exporting features that are in a public package (`keyword`, `cl-user`), using instead a package that is under the library's control. If you must, choose a name that is unlikely to clash with any other use;
- where possible, define a protocol for implementation-dependent support, rather than using ad-hoc feature tests within the library or application logic;
- avoid using two features where one (and its absence) will do;
- try to avoid pushing features on at load-time.

4.1 Developing under IDEs

This last ingredient merits more discussion. The principle behind it is largely aesthetic: namely, that a large `*features*` list obscures programmer comprehension. However, it does raise an issue with the above recipe: not adding features to the `*features*` list at load-time has an unfortunate interaction with rapid, interactive development.

Typically, in Lisp development sessions, one makes changes to individual functions and asks the running lisp to recompile them 'on the fly'. It then behoves the programmer to ensure that he has a consistent state from which to make the modifications; if an application has been deployed with the modifications to `*features*` not being evaluated at `:load-toplevel` time, and if changes need to be made to those portions of the code base that are implementation-specific, the relevant feature-examining clauses must be evaluated prior to incremental modification. This provides a further argument for localising feature-conditional code inside implementation-specific files providing an (internal) functional protocol; this localisation allows the desired incremental modifications to take effect automatically by simply compiling and loading the file in question.

It should also be mentioned that a possible effect of different `*features*`-manipulation policy at compile- and load-times is a degradation in the quality of bug reports; if a user finds the need to report a problem, `*features*` can no longer be taken as an authoritative description of the user's Lisp environment.

4.2 Environmental discrimination

There is another, *a priori* valid, use of `*features*` that would in principle benefit from a central registry of cross-implementation defined keyword meanings; namely, querying the Lisp about the environment in which it is running. Certain libraries may require interaction with the operating system, and as such may wish to compile conditionally implementations of functionality specific to a given OS, but portable across Lisp implementations.

Rigorously defining and enforcing keyword features with given meanings would allow this. However, an ultimately more useful set of definitions would be a standard set of responses to the built-in functions `software-type` and `software-version`; in particular, the ability to query these functions reliably both at compile-time and at run-time would allow for robust diagnostics within a library, with a finer granularity in detecting the version of the operating system than could be provided by any scheme based on `*features*`.

5 Conclusions

> There are now many implementations of Common Lisp, some programmed by research groups in universities and some by companies that sell them commercially,

> and a number of useful programming environments have indeed grown up around these implementations. What is more, all the goals stated above have been achieved, most notably that of portability. Moving large bodies of Lisp code from one computer to another is now routine.
>
> — *Common Lisp the Language*, 2nd edition, section 1.1

We have examined the historical use of `*features*` in the quest for developing low-maintenance, portable library code, examining problems that have surfaced over the course of decade-long maintenance. While the essential portability referred to in the above quotation is certainly possible, we have seen that there are idioms common in existing code that lead to suboptimal maintainability. We have presented a list of instructions aiming to guide the development of maintainable conditional compilation, and identified the area of interaction with the outside environment where support from an agency external to the implementation itself is necessary.

Acknowledgements

Thanks to Brian Mastenbrook, Rudi Schlatte and Robert Strandh for reading early drafts of this paper and providing useful feedback.

References

[1] Robert A. Maclachlan. CMU Common Lisp User's Manual. Technical report, School of Computer Science, Carnegie-Mellon University, 1992.

[2] Christophe S. Rhodes et al. Steel Bank Common Lisp User Manual. in preparation.

[3] Roger Corman. Corman Lisp User Guide. Technical report, Corman Technologies, 2003.

[4] Marco Antoniotti. Two Bioinformatics Applications of Common Lisp. In *International Lisp Conference Proceedings*, 2002.

[5] Marc Battyani. Automatic Web Application Generation. In *International Lisp Conference Proceedings*, 2002.

[6] Daniel Barlow. CLiki: Collaborative Content Management for Community Web Sites. In *International Lisp Conference Proceedings*, 2002. `http://www.cliki.net/`.

[7] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, 1984.

[8] Kent Pitman and Kathy Chapman, editors. *Information Technology – Programming Language – Common Lisp*. Number 226–1994 in INCITS. ANSI, 1994.

[9] Robert G. Rogers, Jr. Re: adding `version>=`. CMUCL Implementors mailing list, September 2003. Message-ID: `<16244.60579.258491.103143@rgrjr.dyndns.org>`.

[10] David N. Gray. Issue STREAM-DEFINITION-BY-USER. Technical report, ANSI, 1989. `http://www.nhplace.com/kent/CL/Issues/stream-definition-by-user.html`.

[11] Guy L. Steele, Jr. *Common Lisp: The Language*. Digital Press, second edition, 1990.

[12] Bill Clementson. Using Emacs as a Lisp IDE. In *International Lisp Conference Proceedings*, 2003.

[13] Henry Spencer and Geoff Collyer. `#ifdef` Considered Harmful, or Portability Experience with C News. In *Proceedings of the Summer USENIX Conference*, 1992.

[14] Robert Strandh and Timothy Moore. A Free Implementation of CLIM. In *International Lisp Conference Proceedings*, 2002. `http://www.cliki.net/`.

[15] Gregor Kizkales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.