Parenthetically Speaking with Kent M. Pitman

dpANS Common Lisp

Don't Judge a Spec by its Cover!

Common Lisp recently became a draft proposed American National Standard (dpANS), and is now in Public Review.

I have been the project's Technical Editor for about two years now, and I wanted to take this occasion to respond to some common questions about the specification.

Textual Size

"It sure is big." I have made the rounds proudly showing off my copy of the final document to my friends and colleagues, and this is invariably their first comment. Counting cover sheets, tables of contents, etc.—it's about 1350 pages. It *is* big—no doubt about that.

"Maybe it's *too* big," some people add in a tone that varies from tentative or questioning to strongly assertive. Too big? "Yeah, too big. Look at how small C is," comes the inevitable challenge.

But that's not a fair comparison. The C language provides very little of the functionality that Common Lisp provides. Even with C and the C Library taken together, it's an apples and oranges comparison. With Common Lisp you get control abstractions like `CATCH` and `THROW`, `UNWIND-PROTECT`, and `LOOP`. You get arbitrary precision integer arithmetic. You get built-in aggregate datatypes like lists and hash-tables. You get support for dynamic program redefinition. I could go on and on. C doesn't attempt to deal with any of these issues in any kind of serious way. No wonder it's smaller.

Subsets for Teaching

"That's just my point. Common Lisp provides too much to learn all at once."

Who says people have to learn it all at once? It wasn't X3J13's goal to produce a classroom textbook. Is it the fault of the *language* that no one has yet written a suitably short introductory text? Could we at least wait until the language specification has been published and a few implementations become available before expecting such a feat? "It doesn't matter. You could never get that much information into a short enough book."

What about just teaching a subset? Imagine what would have happened if Mathematics were constrained such that mathemeticians could use only those concepts that could be taught in First Grade. "Common Lisp doesn't define a teaching subset."

(Does Mathematics?) During the design of Common Lisp, we (X3J13) asked a number of instructors if there was a possibility of defining a useful teaching subset. The general consensus was that courses

---

Parenthetically Speaking expresses opinions and analysis about the Lisp family of languages. Except as explicitly indicated otherwise, the opinions expressed are those of the author and do not necessarily reflect the official positions of any organization or company with which the author is affiliated. Kent M. Pitman can be reached via the Internet as KMP@Symbolics.COM, or by U.S. mail at Symbolics, Inc., 6 New England Tech Center, 555 Virginia Road, Concord, MA 01742-2727 U.S.A.

varied enough in scope that the specification of a universally acceptable teaching subset was not feasible. So instead we decided to just acknowledge the possibility of subsetting in general. In particular, the draft specification says the following in section 1.7 (Language Subsets) on page 1-32:

> The language described in this standard contains no subsets, though subsets are not forbidden.
>
> For a language to be considered a subset, it must have the property that any valid program in that language has equivalent semantics and will run directly (with no extralingual pre-processing, and no special compatibility packages) in any conforming implementation of the full language.
>
> A language that conforms to this requirement shall be described as being a 'subset of Common Lisp as specified by ANSI «*standard number*».'

At this point, there's usually a pause in the conversation. They're obviously still troubled and seek a new line of argument.

Program Size

"It's not just the spec. Your programs will be very large."

Does anyone have any proof of this? The people who say this generally haven't even opened the manual. They're still reasoning about the language by looking at the pages edgewise. "C applications are tiny, but Lisp applications are enormous."

Proof by example. Very rigorous. Well, in my experience C applications are small only if they don't do anything interesting. Common Lisp was designed specifically for writing large, powerful applications. Once a C working environment is configured with the runtime libraries needed to support similarly powerful applications, it's generally as big or bigger than a Common Lisp. "Yes, but with C you can also write small programs."

Am I missing something? I don't *think* the spec mandates a large image. There are implementation techniques like autoloading and tree-shaking that can be used to reduce image size if that's your primary concern. Or, if you can figure out what facilities people don't need, you can build an implementation that excludes them. The absence of subset implementations for small applications and teaching is a hindrance, but it's not clear that the language definition is to blame for this. The eyes roll, as if to say 'uh, oh—he's going to talk about subsets again.'

Libraries

"What about sharing? C provides for sharable libraries."

Isn't the question of how functions are arranged in libraries just an implementation issue? (C is full of implementation issues positioned in ways that it is impossible for programmers to ignore them.) Do programmers really *like* specifying these things explicitly? Personally, I hate it. And from what I know of natural languages, there are none that require the speaker to first identify that he plans to use a group of terms before he can use them. So it seems like an unnatural concept to me. Is it really necessary? "It makes programs share better."

But can't it be done automatically? C makes you explicitly deallocate unused storage. The process is clumsy and error-prone. Lisp does storage deallocation automatically. It's more reliable, and these days it's even relatively efficient. (I daresay no one would have bothered making it efficient if it hadn't been a required part of the language.) If automatic management of object-level storage is possible, why not the same for libraries. Why do users have to know? "But is that possible in Common Lisp?"

I see no reason why not. At the Lisp and Functional Programming Conference held recently in San Francisco, Wade Hennessey presented a paper [Hennessey 92] about how an implementation of Common Lisp called WCL is built using Unix shared libraries.

Alternate Libraries

"But what if I don't like the standard library and want to substitute another?"

Well, you could either seek a different vendor or you could work with your vendor to bring their implementation up to adequate quality. "No, I mean I want a different set of interfaces."

Well, I suppose you *could* just ignore the functionality that Common Lisp provides and use a separately defined variant. But why would you want to? The point of Common Lisp was not to be the best Lisp, but rather to be a common one—so we'd all be speaking the same language, not just so that we'd agree on an implementation that let us all speak different languages. And besides, if everyone is using a different shared library, then who will they be sharing with?

Efficiency

"But if it's that big, can it possibly be efficient?"

The size of the language is not correlated with efficiency. We've established that a large part of the language can be viewed as libraries. It's only the language core that affects efficiency. It was demonstrated over a decade ago [Steele 77] that Maclisp can be as efficient as other languages for numerical code. Recent work by MacLachlan on the Python compiler for CMU Common Lisp [MacLachlan 92] has continued this theme by showing that modern compiler techniques can be usefully applied in Common Lisp with very impressive results.

Issues of Presentation Style

Another pause. "But really—1350 pages?"

Well, there *are* 973 defined names—things like special operators, functions, macros, variables, and types. So it's averages only about a page and a half per name.

I'd estimate that about 40% of the document is taken up by examples, which, incidentally, are not formally part of the definition of the standard. We could remove all of that and it would be *smaller*, but I don't know if it would be *better*. Also, the organization and typography is chosen to make it visually easy to peruse as a reference document. I bet if took out some of the whitespace that makes it so readable, you'd find it was another 15% smaller. That would be less than a page per function.

"Couldn't it be said more concisely? Scheme [R4RS 91] seems to get away with a lot less space per entry."

The concise nature of the Scheme specification may be visually aesthetic, but it is often more practical for implementors than for users. Because so many details are left undefined, implementations vary widely in ways that become a barrier to the writing of portable programs.

By contrast, the Common Lisp specification caters to the developer of portable programs by providing a variety of functionality and by specifying (sometimes in considerable detail) how that functionality will manifest itself in implementations having widely varying characteristics. Experience with Guy Steele's original description of Common Lisp [Steele 84] showed up a number of specific portability problems that we have worked hard to address in the current specification. Unfortunately, addressing these areas in a way that is commercially useful often means adding text rather than removing it.

Modularity

"Do you really need 973 defined names?"

It may seem like a lot, but once it's broken down by chapter it's not as much. There are 26 chapters, dividing the language into functional areas. 23 of the chapters have a dictionary section, so that's an average of 42 names per chapter. Even that may seem unreasonably high if you're thinking it's just function names, but remember that it includes variable names, type names and names of accessor functions for each type. When you finally get down to raw functionality, what you find is pretty much the usual suspects for each functional area.

Complexity should be measured not by the aggregate size but by the way in which things are organized. By this metric, Common Lisp doesn't provide an unreasonable amount of complexity. It just solves a lot more problems than the average language tries to. It doesn't just go after types, control flow, and arithmetic—it goes beyond to address sophisticated issues like conditions, characters, hash tables, pathnames, streams and I/O, and as well as many issues related to debugging, inspection, and self-representation.

Relation to Common Lisp: The Language (First and Second Editions)

"Well, if I don't send for the draft, can I still use Guy Steele's second edition [Steele 90]?" I shudder, trying hard not to hear this question as 'haven't the last two years of your life been spent in vain?'

The first edition of Steele's book was the direct output of a committee. It was proofread by a committee and ultimately approved by the committee. We were fortunate to have Steele available to do that work, but from a technical standpoint it was only incidental that the document was written by an individual.

The second edition of Steele's book was *not* the output of a committee. Steele attended the committee meetings, but he produced the second edition as a personal project in order to give the community a snapshot of things to come. He was clear about this in the Second Edition Preface (pages xii-xiii) but these remarks have been all too often overlooked. By his own admission, the document is unsuitable for use as a standard both because of its presentation style and because it has been insufficiently proofread for errors and omissions.

Formally, the standard derives from the first edition, not the second. The second edition was never an official document of X3J13. Some things have been added, some have been removed, and some have changed since its publication. Enough records were kept such that someone could reconstruct precisely what, but it was beyond the scope of our work (and our available resources) to produce a summary. Sorry about that. But even if nothing had changed, the presentation style of the standard is very different than in Steele's books, making many things explicit that you previously had to dig around for. It may be worth getting a copy just for that.

Future Work

Let me be clear: I'm *not* saying the specification couldn't be trimmed here and there. There are definitely some inconsistencies that could be ironed out. There are still a few questionable features that could probably be removed. But even if all this were done and it were as pristine as Scheme, this level of specification and functionality would still take many hundreds of pages.

So don't judge the spec by its cover. Open it up and start reading. Read Chapter 1 (Introduction), particularly Chapter 1.4 (Definitions), and read Chapter 26 (Glossary). And read the details of the other sections that are your area of interest. I hope you'll find it all to be in pretty acceptable shape, but if you don't, now is the time for you to make your voice heard.

References

[R4RS 91] Clinger, W. and Rees, J. (editors), "Revised^4 Report on the Algorithmic Language Scheme," *Lisp Pointers*, Volume IV, Number 3, pp1-55.

[Hennessey 92] Hennessey, Wade, "WCL: Delivering Efficient Common Lisp Applications Under Unix," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, ACM Order No. 552920, pp260-269.

[MacLachlan 92] MacLachlan, Robert A., "The Python Compiler for CMU Common Lisp," *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, ACM Order #552920, pp235-246.

[Steele 77] Steele, Guy L., Jr., *Fast Arithmetic In Maclisp*, AI Memo 421, Massachusetts Institute of Technology, Cambridge, MA.

[Steele 84] Steele, Guy L., Jr., *Common Lisp: The Language*, Digital Press (Burlington, MA), 1984.

[Steele 90] Steele, Guy L., Jr., *Common Lisp: The Language* (Second Edition), Digital Press (Bedford, MA), 1990.

[X3J13 92] Pitman, K.M. and Chapman, K.C. (editors), *draft proposed American National Standard for Information Systems—Programming Language—Common Lisp*, 1992.