

## Re: What's so great about lisp?

---

*Source:* <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2005-10/msg00428.html>

---

- *From:* Pascal Costanza <[pc@xxxxxxxxx](mailto:pc@xxxxxxxxx)>
  - *Date:* Sat, 08 Oct 2005 14:56:44 +0200
- 

Jon Harrop wrote:

Joe Marshall wrote:

Now I want as much static checking as possible, but I'm going to get really irritated if it gets in the way.

If that were available then I'd want the final statically-checked code to be compiled efficiently, i.e. without unnecessary run-time checks and with optimised pattern matches and so on.

That doesn't make sense. The majority of a program is executed very rarely, typically only few parts are executed most of the time. It's one of those 80:20 things: 80% of the time only 20% of the code is executed. (These are, of course, not the exact numbers, but that's the idea.)

This means that the "efficiency" of the resulting code doesn't really pay off in the majority of the code - you wouldn't actually notice any difference. On the other hand, the removed run-time checks are areas where potential problems can occur. Think about security leaks or lack of information in case the program fails in unexpected ways.

The only reasonable way to get efficient programs is to try to identify the "hot spots" and optimize them and only them. The best way to optimize them is to completely get rid of them. This boils down to finding better algorithms / execution strategies rather than fine-tuning what are essentially bad algorithms, etc.

Any idea if such a language exists or if Lisp could be coerced into doing this? It would probably require two different run-times...

Common Lisp provides declarations with which you can declare your intentions. `(declaim (optimize speed))` would result in more efficient code, `(declaim (optimize debug))` would result in code with more debug information, `(declaim (optimize compilation-speed))` would focus on

## Re: What's so great about lisp?

generating the code as quickly as possible, (declaim (optimize safety)) would result in code that omits run-time safety checks. These are some of the standardized optimization qualities, and you get even more fine-grained control by assigning degrees with those qualities (0 = low to 3 = high).

Common Lisp implementations are allowed to ignore these declarations, but many do a pretty good job at interpreting them in useful ways. CMUCL and SBCL are especially interesting because they have a type inferencer that interacts with these settings (IIUC). (This doesn't mean that I generally recommend to focus just on those two implementations - their compiler is relatively slow because it apparently spends quite some time on optimizing code, and sometimes it is more important to have a fast compiler rather than have fast code.)

Pascal

--  
OOPSLA'05 tutorial on generic functions & the CLOS Metaobject Protocol  
++++ see <http://p-cos.net/oopsla05-tutorial.html> for more details +++++  
.