



Why We Love Lisp

For opposing viewpoints, see: [WhyWeHateLisp](#)

In what follows, "Lisp" refers to [CommonLisp](#), as defined by the ANSI X3J13 standard, unless stated otherwise.

Nihilism

Lisp is nihilist so you don't have to be. In contrast, some languages have certain ideologies: in [SchemeLanguage](#), functional programming is a priority; [PythonLanguage](#) has the goal of ideally providing the one right way to do anything. Of course, if you prefer a language to be ideological, Lisp may not be for you. Which is fine.

What Lisp provides is a substrate. You do have what's needed for functional programming, or OOP... unbounded size numbers, and so forth. Which you can use directly, or build more on top of.

While Lisp may itself be nihilistic, you don't have to be. You are free to be a religious nut.

[Caveat: Lisp has been expected to run on a number of devices, including megaliths with 65536 processors. So you won't find things like standard network and GUI APIs in the standard; it wouldn't make sense. But there is a use for limited substandards, which apply to already-standardized homogeneous platforms like Unix/Windows.]

Dangerous

David Noble's book *Forces of Production* argues that technology encountered one fork in the road - a decision between removing management in favor of skilled workers, or deskilling workers to empower management. It turned out in favor of management, though the other path could also have been taken. <http://www.amazon.com/exec/obidos/tg/detail/-/0195040465/002-3851349-3240830?v=glance>

Lisp doesn't have many "sharp edges" in the normal sense, like requiring use of pointer arithmetic. However, people have correctly pointed out Lisp offers enough power to both require and reward good judgment. Whether or not this is useful depends on what you mean by "technology," at least if you believe David Noble's claim.

Useful macros

In Lisp, the program you are working on is represented using data structures. (The fancy technical term is "symbolic expressions" or sexps, but it's just data like numbers, symbols, strings, lists of data, etc.) This means that, for example at compile time, simple code can be converted to less human-readable code:

```
(loop for i from 0 below 10)
```

is just the same as the `for(i=0;i<10;i++)` loop in other languages. If you "macroexpand" it, you see it probably

turns into a messy bunch of gotos and who knows what else.

So, Lisp users view code as something you can operate on with muscles well trained for data manipulation. It is the main basis for supporting new paradigms which languages aimed at "mainstream programmers" can't easily do, since they are not easily customizable for special requirements.

Macros imply a transfer of power from programming language implementors to users. From a social perspective, the arguments start appearing like those used in grassroots democracy vs. totalitarianism debates, concerning things like mob rule and expertise. At very best, other languages might achieve some sort of top-down democracy, which Java seems to be attempting with BugParade² voting on new features. Or a "Love it or fork it" attitude.

Some might argue that this level of customization allows one to speak more fluently in terms of the program's domain. After all, domains have their own paradigms. For example, business programmers might want a closer trust relationship with clients by thinking and speaking in terms of the client's domain.

Proper numbers

In Lisp, $3/5$ doesn't mean 0 (as in C and Python*) or the inexact floating-point number 0.6 (as in Fortran and Perl**). It means the rational number $3/5$, which gives the exact integer 3 (not 2.99999999) when you multiply it by 5.

In Lisp, $1000000000 * 1000000000$ doesn't silently overflow and give the wrong answer (as it is liable to do in C), nor does it fall over with an exception (as it is liable to do in Python*), nor does it produce a floating-point number of doubtful exactitude (as it does in Perl). It's just the number 1000000000000000000.

All this has drawbacks too, of course. If you need efficiency then you need to be careful that your program doesn't start working with 100-digit bignums when floating-point is enough. That's not difficult to do. But when you forget, the only result is inefficiency, not nonsense.

(both points have been addressed in newer versions of Python, AFAIK)*

Not really, the decimal.Decimal type tries to do better (provides support for "true" decimal floating-point arithmetics) but the regular int type still evaluates $3/5$ to 0. $\text{Decimal}(3)/\text{Decimal}(5)$ evaluates to $\text{Decimal}("0.6")$.

About the overflow/exception, on the other hand, overflowing integers get silently promoted to Longs (which are arbitrary length integers in python). Yields a noticeable drop in performances though.

***[PerlSix](#) has/will have $3/5$ as the literal for a Rational object, which is stored as a 64-bit signed numerator and a 64-bit unsigned denominator.*

[MultipleDispatch](#)

Ever been puzzled by the [VisitorPattern](#)? Bid a glad farewell to it with the [CommonLispObjectSystem](#), in which methods can dispatch dynamically on the types of all their arguments.

The [CommonLispObjectSystem](#) has many other wonders, too. :-)

I agree that Visitor stinks, but there are many approaches to a replacement outside of Lisp, I would note.

Interactivity and introspection

If your Lisp program isn't working right, you can play with its bits on the fly. If it isn't working *yet*, you can develop it in fragments and experiment with them as you go. Things like class names persist at run time. You can interrogate and manipulate your program while it's running Safely.

Good built in types

Container types: Vectors, arrays, linked lists, hash tables. (As well as structures and classes, of course.)

Numeric types: Integers (unbounded in length), rationals (unbounded in size), floating-point numbers (of up to 4 different sizes according to need), complex numbers.

Stringy things: Characters, strings (which are, by the way, just a kind of vector). Strings (as well as vectors of other types) with "fill pointers" and grow-on-demand, which are SemanticSugar² to make it simple to append to a string without scanning through it.

Symbols: A specialized type, not normally found in any other language (because they lack the concept of *READ time*, used to hold variable bindings, functions, properties, or for their identities. Symbols are largely what makes the lisp macro system so effective. Symbols are segregated into their own namespaces using the *Package* system.

Functions: In lisp, *functions* are a first class type, i.e. they can be created at runtime and passed as arguments. This makes lisp suitable as a *functional* programming language (although it is by no means restricted to that style, having operators for side effects, such as SETQ).

- *Too many built-in types takes away from the simplicity of a language. If you want to strive for simplicity, then find a more morphable/adaptable single data structure.*
 - That's a very amusing comment, considering the traditional complaint (usually heard from people who know very little about contemporary Lisp) that Lisp deals with lists and only lists. Too few built-in types just causes them to be recreated in terms of the ones that are built-in.
 - I'm guessing that the author of the "too many types" comment above is probably [TopMind](#), who thinks that everything should be a relational table, and that anyone who uses any other data structure is guilty of the mortal sin of [ReinventingTheDatabaseInApplication](#). If I ever met someone who claimed, for example, that there were too many types in the Java Collections Framework, and that all higher-order data structures should be reimplemented in terms of arrays, I would laugh that person out of the room.

Dynamic typing

Of course, the jury is still out (and likely will be for a long time) about whether this is a good thing or a bad thing. See the pages called [DynamicTyping](#) and [StaticTyping](#), and their myriad progeny. However, Lisp users are generally strongly in favour, so this counts as a reason [WhyWeLoveLisp](#) even if it shouldn't. :-)

Lisp has *optional static typing*, in the sense that you can tell the compiler "this thing is always supposed to be of that type". The compiler can use this to provide helpful compile-time complaints, or better speed, or both.

[AnswerMe](#): Do any implementations of [CommonLisp](#) actually do either of the above? Or are type annotations treated as comments by most Lisp implementations; and the typechecking and performance improvements implied are only found on the ever-mythical [SufficientlySmartCompiler](#)?

Yes, Lisp implementations actually do this. All commercial implementations do this. CMUCL and SBCL both do extensive type-inference even without declarations.

Some lisps do things better than others. CMUCL and SBCL use a type inferencing engine that makes numerical computation phenomenally fast. We're talking [AsFastAsCee](#) here. On the order of OCaml results if set up properly. However, that setting -up-properly process isn't always intuitive.

Commercial Lisps tend to focus less on generating tight ASM for computations and more on handling memory very intelligently, since this is a major slow point for most modern computers. Don't underestimate Allegro or [LispWorks](#) because it doesn't bench well in, say, Fibonacci or Factorial benchmarks. That's not the way they're tuned. Watch though as they run webapps like nobody's business.

Decent performance

There are other languages with a rather Lispy feel to them (good range of built in data types, interactive, highly dynamic, and so on); the most popular at the moment seems to be Python. However, these languages tend to have only interpreted implementations (sometimes via a byte-compiler) and be awfully slow. Lisp is designed to be compilable to code that runs pretty fast. (Say, within a factor of 2 of optimized C.)

[There's no reason it can't run as fast or faster than optimized C.] *[Is there some reason C can't run as fast or faster than optimized Lisp? It's simply that I think that there's a limit to how fast compiled code can be, and I would expect well-optimized C (and well-optimized Lisp) to be fairly close to this limit. -- [AlistairBayley](#)]* Of course not. The point is, very few languages with anything near the power of lisp generate native code with anything near the speed of C. The amazing thing is that while lisp is arguably the most powerful programming language existent, with a good compiler all this extra power doesn't slow you down much at all.

Aside: See [SufficientlySmartCompiler](#).

Powerful Error Handling

[CommonLisp](#) has one of the most complete and flexible environments for error handling, including the ability to establish *restarts*, search up the stack for handlers, possibly invoking more than one of them, and having code inspect which restarts are currently available to decide which to call. Overall, this makes it possible to write extremely robust programs.

You can also have *warnings*, which are vaguely like optional exceptions, which can be handled or "caught" if desired.

See [CommonLispConditionSystem](#)

Clear and precise standard

[CommonLisp](#) is blessed with an extremely precisely written standard, which leaves zero margin for error when determining if an implementation is conformant. This helps when complaining to your vendor... :-)

Alas, if only the standard really did leave zero margin for error. Some sections (such as section 19) are notoriously vague. --[PaulDietz](#)

- Which section is section 19?
 - Path names (<http://www.lisp.org/HyperSpec/Body/chap-19.html>), according to the [CommonLispHyperSpec](#). I don't actually see the ambiguity; I thought the main complaint about CL pathnames is that they're incredibly Byzantine in a world where Unix Won[?]. They were designed to provide CrossPlatform[?] standardization in the era where every minicomputer had a different file system.

Fast and productive development environment

While not unique to lisp, having a top-level [ReadEvalPrintLoop](#) where one may ask questions of a running program, and modify it as it runs, is certainly one of the reasons [WhyWeLoveLisp](#). Farewell to the [EditCompileDebugCycle?](#)!

Beauty

Build layer on layer of abstraction like a drawing.

Not very specific. Everyone thinks their pet language has the best abstraction abilities.

Wealth of outstanding literature

By learning some Lisp, you gain access to a lifetime supply of enlightening literature and breathtaking programs. Lisps have been used for incredible research and exposition over decades, and the fruits are readily available to anyone inclined to dig in. For example, [ParadigmsOfArtificialIntelligenceProgramming](#), [StructureAndInterpretationOfComputerPrograms](#), [EssentialsOfProgrammingLanguages](#), [GuySteele](#)'s *Rabbit* compiler and *Programming Language Based on Constraints*, [ConnectionMachineLisp?](#), MIT's AI Lab publications like [GuySteele](#) and [GeraldSussman](#)'s *LambdaTheUltimate* series, [HenryBaker](#)'s research archive (garbage collection, parsing, linearity), and oh-so-much more. The Lisp literature records the achievements of some of the world's best programmers.

Discussion

Re Scheme: Scheme not only doesn't make FP a priority, it was developed as an [ActorsModel](#) language, which is essentially OO. The usual complaint about Scheme is that *all* it provides is a substrate, which must then be built up with libraries. SLIB appears to have emerged as a sort of de facto standard for scheme, so much as you wouldn't make comparisons of C++ to anything else without its standard library, any "fair" comparisons to CL really should be with Scheme+SLIB.

The reason I wrote that is because through looking at all the standard textbooks which teach Scheme, it is clear that they teach a style of programming which is "functional." I take this from [SiCp](#), Little/Seasoned Schemer books, and HTDP. In fact, when people complain that their experience of Lisp is it's too "functional," further questions almost inevitably show it was Scheme, not Common Lisp, which they experienced. I do not recall seeing one obviously iterative construct in those books. [Caveats: I haven't read HTDP through, I remember [SiCp](#)'s discussion of tail recursion being iterative in some sense, and I don't mean functional in the sense of "purity".]

How might I go about reconciling your explanation with this? Or could you help me pinpoint the flaw in my observations?

- [If I might interject: Lisp proponents often complain that most Lisp texts (and college courses) teach the language as if lists were the only data type, leaving an impression of inefficiency, and as if iterative constructs didn't exist. I imagine that Scheme suffers similar problems. Basing an understanding of a language on tutorial textbooks seems likely to be a bad idea. -- [AnonymousDonor](#)]

[SiCp](#) is a highly academic book that doesn't really even teach scheme, but a bit of functional programming. [SiCp](#) is about FP, not scheme. The Schemer books are written by FP advocates, and again there's nothing in scheme that requires you to use an FP style. If you want to see a language that locks you into FP, try Haskell. HTDP is also fairly FP-oriented - anything out of MIT is going to be - and TeachYourselfSchemeInFixnumDays? lingers on lots of academic subjects (like amb). Perhaps "Scheme for Perl/Python/Ruby/Vb/Tcl hackers" might be in order. It'd have to be heavy on libraries for sure.

IIRC, HTDP and [SiCp](#) where written for the same course, and Little/Seasoned Schemer is a FP-book that seems to be usable as a companion piece to [SiCp](#). TYSifd isn't very FP.

Perhaps what we also need is a Scheme book that covers the [SchemeRequestsForImplementation](#) libraries, as well. [HowToDesignPrograms](#) uses some SRFI tools, but only certain ones, and it also leans heavily on the PLT implementation. -- [JayOsako](#)

The complaint about FP comes from beginners struggling to solve all their problems using a handful of basic functional building blocks such as lists, map, and foldl. Textbooks intentionally force students to solve programming exercises under this constraint to help them learn functional ways of thinking, but an unfortunate side effect is that students begin to think that simple tasks are impractical or disproportionately intellectually taxing in Scheme. Beginners who want to do practical programming should be guided to essential practical tools such as vectors, hash tables, sets, and iterations and comprehensions (<http://docs.racket-lang.org/guide/for.html>) instead of being left to discover them on their own. Beginners who are not experienced programmers may even have a hard time finding the docs for (e.g.) file I/O without help; introductory books should keep that in mind. -- DavidHuebel?

When comparing the size of [WhyWeHateLisp](#) to the size of [WhyWeLoveLisp](#), it becomes clear that the number of people angry because of losing their job to a lisp guy is far bigger than the number of people angry because of losing their job to a non-lisp guy.


It's interesting what sort of conclusions people come to based on a non scientific wiki with about 5-15 people participating at a time. One could call a sample of 10 people from a phone book (yes 10 people out of a few billion) and come to strong rigorous scientific conclusions also!

Hmm. That is rather odd. In [EmbeddedSystems](#) I've never seen Lisp used for anything at all in 35 years of experience. Perhaps the dozens of firms I've been exposed to are simply not representative of the industry?

So it's not often used in [EmbeddedSystems](#). That doesn't mean it's not usable for those purposes - I would point at, for example, ecl. "Nobody uses it" is not ever a valid reason to criticise a tool, though the reasons that they don't might be. Consider suggesting why you've never seen it used, rather than saying you haven't as if that demonstrates that it shouldn't be.

See: [WhyWeHateLisp](#), [HowToSellGoldenHammers](#)

[CategoryRant](#), [CategoryLisp](#)

 [EditText](#) of this page (last edited [June 24, 2013](#)) or [FindPage](#) with title or text search