What Are the Enduring Innovations of Lisp?

Saturday, December 31, 2022 Programming

Historically, Lisp was a behemoth in the world of language design innovation. The list of Lisp's innovations is so long that I'd have to list most of the features associated with modern languages. (Do you like `if` statements? You can thank Lisp!)

But the language landscape has changed a lot since then, and realistically no programmer today cares about what made a language stand out 50 years ago. Clearly, the good ideas have been copied into other languages. Paul Graham even suggests this convergence towards Lisp is inevitable. I wouldn't go so far. But this begs the question: *Is there anything left?* Are there any features that couldn't be copied so easily into the various descendants of Algol?

I'd say these features, particularly in combination, continue to be distinctive:

1. An "everything is available all the time" approach to system design. Lisp allows you to run code at compile time, compile code at runtime, run and compile code while debugging, iteratively compile and profile different sets of code, etc. Everything blurs together so that there are no obvious boundaries between different parts of the system. The way that common Lisp systems produce executable binaries to be used as application deliverables is by literally dumping the contents of memory into a file with a little header to start things back up again.

2. Pervasive interactivity, i.e. the REPL. This was more of a contrast to e.g. Fortran and other ahead-of-time compiled languages, but is still notable today. For example, the Common Lisp package manager is executed via the Lisp shell. Nearly all comparable alternatives I'm aware of are executed out of process via the system shell (`/bin/bash`, or what have you). Same with the debugger, profiler, IDE (if that's your thing), even in some cases the OS.

3. A canonical representation of programs in terms of the literal syntax of the language's core data structures, permitting a design where the program text *is the literal representation of the program AST*.

   Let me unpack that.

   Python has literal syntax for various core language data structures: `[]` for lists, `{}` for dictionaries, `()` for tuples, etc. Python programs are not represented in terms of those literals; the language itself has different syntax (`def`, `class`, etc.). But Lisp programs are.

   Lisp programs are expressed using Lisp's list core data structure. Therefore, the text of a program in Lisp consists simply of the serialization of this data structure into text via the literal syntax for lists (i.e. the infamous `()`). Furthermore, this permits an implementation where the language AST (at least through to the first stage of the compiler) also uses the same representation. This is what people mean when they say the language is *homeoiconic*.

   Homeoiconicity has a surprising advantage. Now it is possible to manipulate, using only core language data structures, program ASTs. It is possible, in other words, to write code that produces code. And since the compiler is available at all times, you can not just build

ASTs, but actually compile and run them. This is the basis of metaprogramming in Lisp.

It's worth noting that macros are not on my list above. Macros are simply syntax sugar around the capability you already have as a consequence of (1) and (3). Macros allow you to hide the fact that you're doing code generation, but they don't fundamentally give you any new capabilities when the compiler was already available at runtime anyway.

It's also worth noting that code generation (and therefore metaprogramming itself) are also not fundamentally innovations of Lisp. For example, in C++, it is entirely possible to link your application to `libclang` and build Clang ASTs inside your application C++ code, and use the Clang compiler to emit and run that code. Before you laugh, know that people can and have done it. But unsurprisingly, it's hard—hard enough that you wouldn't bother unless you had a problem you couldn't solve any other way, and so big that you couldn't ignore it.

Lisp is easier. In fact, Lisp is so much easier that people do it in the course of their day-to-day programming tasks. This is the real secret sauce of macros. Macros, together with the key innovations above, make it possible to do metaprogramming easily enough that it's actually a regular occurrence.

In case you missed it, I did a minor sleight of hand two paragraphs back: I used the word "metaprogramming", but the links I included were all to domain-specific languages (DSLs). This was not a mistake. As metaprograms become more sophisticated, they become increasingly difficult to distinguish from full-on languages. Furthermore, in a language that makes metaprogramming easy, an application that consists of a series of layered libraries begins to look increasingly like a series of layered *languages*. To put it another way, libraries and languages are both

abstraction layers, but languages are the more powerful one. This is, by the way, not a new idea, but it is one that the broader software community has yet to internalize.

As someone who makes a living by creating languages and compilers, it's easy to downplay the significance that all of this has on writing code. Even in the programming languages community, Lisp probably gets less credit than it deserves. But with the recent resurgence of DSLs, we should really be taking a hard look at what Lisp provides. If the goal is to make it easier to build DSLs, Lisp already provides a lot of the starting blocks.

(I originally wrote this on February 27, 2017, but it remains as true today as it was at the time. I'm finally getting around to publishing it.)