The Ciel Manifesto

Erann Gat

29 April 2003

WORK IN PROGRESS

After spending the last fifteen years of my life patiently waiting for the rest of the world to realize what a wonderful language Lisp is, I have reluctantly come to the following conclusion: it ain't gonna happen. Every now and then we'll get a convert (like Pascal Costanza) but that is all we can hope for. The rest of the world will continue to to prove Greenspun's tenth law again and again and again and again. After all, the world didn't flock to Lisp ten years ago, when its advantages over the competition were truly overwhelming. Why should the world move to Lisp now when ten years of stagnation have significantly eroded its lead?

Nonetheless, I am loath to give up on Lisp. I am by nature a lazy person; I don't like to work harder than I have to. Having used Lisp for most of my life, programming in C++ or Java now feels to me akin to going jogging with a 100 lb backpack on. Some people get off on that sort of macho thing. I don't. I'd rather leave the extra baggage behind.

What to do? The answer is obvious: if you can't beat 'em, join 'em. If the rest of the world is reinventing Lisp, why not us? After all, who better to reinvent Lisp than the people who invented it to begin with and their intellectual descendants?

I find myself with some free time on my hands nowadays so I have been thinking about a strategy for the successful reinvention of Lisp. My model is Python, which is an existence proof that a new language can succeed even today. Following that model, I see three necessary conditions for success: first, it has to be perceived as something new. That is why I don't think fixing Common Lisp, or extending it, or whatever, is going to work. The world has already made up its mind about Lisp. What's more, it has done so in a state of almost total ignorance, which means that no changes that are made to Lisp will change the situation. Reality is almost totally irrelevant. If you doubt this, look at the current hype about XML.

So the first thing we have to do to save Lisp is, sadly, to stop calling it Lisp.

The second necessary condition for success is a *hook*. A hook is a feature that is immediately obvious, different, and thought-provoking. Python's hook was syntactically significant white space. The hook doesn't have to be a particularly good idea (Python's wasn't), it just has to be something that looks cool enough to get people's attention and makes them willing to invest some time looking into the new language to see what else is there.

The third condition is that the language has to be useful. Nowadays that means (among other things) that it has to have libraries.

How to get from here to there? Here's my strategy.

First, we need a new name. I'm calling my new language Ciel, which is Italian for sky. End of story.

Second, we need a hook. I have two ideas for a hook.

First, the idea of having code as a first-class data structure is a cool idea, but there's no reason why the data structure that represents the code has to be a linked list. In fact, using linked lists has some serious drawbacks, not least of which is that you end up with what the world at large perceives as a pretty screwy syntax. There are alternatives: vectors with fill pointers, for example, or associative maps. More about this later.

The second idea is to discard the idea that what you do when you produce code is editing text in a file. The programming world has been enamored of this idea for far too long. If code is going to be a first-class data structure, why not take the next obvious step and provide structured facilities for producing those data structures directly? And for rendering them non-volatile? The right way to code in a language where code is data is not by editing files, but by editing records in a database.

Third, we need to make it useful. This is the painful part. (The worst part of any endeavor is dealing with those damn customers!) Nowadays being useful means providing an enormous amount of functionality. The only reasonable hope of being able to accomplish this is to leverage some other effort. In a perfect world we would start from scratch and build a Ciel-OS that did the Right Thing from the ground up. In the real world we have to compromise. The best compromise, it seems to me, is to implement Ciel using Linux and C++, though I am open to suggestions. The advantages of this approach are 1) the cost of entry is low (free), 2) C++, despite its horrific cruftiness, has the best library support of any language. In fact, using the Boehm GC and Bruno Haible's CLN numeric library I have produced a prototype implementation of an interpreter for a Scheme-like language (which I hope will eventually morph into version 0 of Ciel) in about 1000 lines of code. This implementation includes a complete numeric tower (courtesy of CLN), exceptions (courtesy of C++), vectors with fill pointers, hash tables, and full lexical closures (all courtesy of the STL and the fact that a naïve i.e. slow implementation of lexical closures is really easy to do). I hope to have regular expressions and a database interface done very soon.

But the unvarnished truth is that Ciel is vaporware at the moment.

Here are some random additional thoughts.

1. Just because you're using Linux and C++ doesn't mean you have to use it all. If you identify a subset from which you get the most leverage (e.g. the device drivers, parts of the STL, gcc) and don't use the rest then at some point in the future you might be able to make a more-or-less clean break with the past simply be excising all the cruft that you didn't use. It is possible to build a nice system on top of an awful one. Existence proofs include emacs, MCL and OSX.
2. All data structures are special cases of associative maps. A vector is an associative map whose keys are all integers within a certain range. A string is a vector whose elements are all characters. A structure is an associative map whose keys are all members of an enumerated type. Even a function (in the functional programming sense) is an associative map (typically an infinite one) whose keys are (again typically) tuples.

(Paul Graham makes basically the same observation, except that he says that everything is a list. This is not quite true. Everything can be *implemented* using a list, but I think this is the wrong way to think about things. Everything can be implemented with a Turing Machine too; that doesn't mean it's a good idea. A corollary: just because you can implement numbers as lists doesn't mean it's a good idea. I believe that numbers have enough inherent structure that they ought to be their own data types. The system should not allow you to do obviously stupid things like take the CDR of an integer. So when I say that everything is an associative map what I mean is not that everything can be implemented as an associative map (though that is also true) but that (nearly) everything can be given an API that looks like (a special case of) the API for an associative map.)

3. The code you write to specify behavior should be strictly separate from the code you write to make things run fast. The right way to write software is IMO first to write a prototype, and then to optimize the parts that need optimizing. But a properly designed language should allow optimization by *adding* code (and only adding code), not *changing* code. This is because you want to be able to use your prototype code as a *specification* of correct behavior. It should be possible to optimize without having to change that specification. (Another way to say the same thing: there should be a strict separation from the code that specifies *what it does* from the code that specifies *how it does it*. And the system should be smart enough to come up with some reasonable (meaning correct but possible slow) default behavior if the former is all you give it.)
4. A corollary to #3: the process of insuring that the code you write to make things run fast is consistent with the specification (i.e. the code you wrote to make things work) should be automated as much as possible. I suspect that type inference will play a major role in this.
5. Another corollary: The way to manage complexity is to hide complex implementations underneath simple interfaces. Nearly every computational system ever built gets this wrong by hiding complex implementations under complex interfaces.
6. I like the idea of having types *defined* by their interfaces.
7. The biggest thing wrong with Scheme is the lack of opaque types. It should not be possible to take the CDR of a structure, and I should not have to redefine CDR in order to prevent this.
8. Variables with dynamic binding are a really useful feature, but Common Lisp's design (special variables) is badly broken. The proof that it is badly broken is that you have to use a typographical convention in order to keep yourself out of trouble. If you have to use a typographical convention to keep yourself out of trouble then you have placed a burden on the programmer that really ought to be taken care of by the system. Anything that places an unnecessary burden on the programmer is a bug. The right design would have been for all variables starting and ending with asterisks to be automatically declared special. (Nowadays a better typographical convention to use is to precede special variables with dollar signs. This is more in keeping with convention, and requires fewer keystrokes.)
9. I like Dylan and Goo. (Goo is what Common Lisp could have been if it had not been designed by committee.) The main thing wrong with Dylan is the lack of a

user-accessible S-expression layer. The main thing wrong with Goo is that it does not have a non-S-expression syntax layer. (Personally I love S-expressions, but it is clear that they are an anti-hook. Any language that does not offer an alternative syntax is doomed to obscurity.)

10. I like syntactically significant white space, but only as an adjunct for sanity checking, not as the primary source of semantic information. You'd think people would have learned their lesson from the "make" nightmare. Python is not quite as bad as make, but I predict that it is only a matter of time before some disaster is caused by someone cutting and pasting some Python code and not realizing that emacs reindented it incorrectly. (The really ironic thing is that Python actually does have open-braces in the form of colons, and it would be really easy to extend the syntax in a backwards-compatible way to give it (possibly optional) close braces as well. But the Python community resists this idea violently.)
11. Code must be readable (and understandable) by humans. On this metric, Python is probably the best design, and C++ and Perl are tied for last place.