Lisp in Web-Based Applications

Paul Graham

(This is an excerpt of a talk given at BBN Labs in Cambridge, MA, in April 2001.)

Any Language You Want

One of the reasons to use Lisp in writing Web-based applications is that you *can* use Lisp. When you're writing software that is only going to run on your own servers, you can use whatever language you want.

For a long time programmers didn't have a lot of choice about what language to use for writing application programs. Until recently, writing application programs meant writing software to run on desktop computers. In desktop software there was a strong bias toward writing the application in the same language as the operating system. Ten years ago, for all practical purposes, applications were written in C.

With Web-based applications, that changes. You control the servers, and you can write your software in any language you want. You can take it for granted now that you have the source code of both your operating system and your compilers. If there does turn out to be any kind of problem between the language and the OS, you can fix it yourself.

This new freedom is a double-edged sword, however. Having more choices means that you now have to think about which choice to make. It was easier in the old days. If you were in charge of a software project, and some troublesome person suggested writing the software in a different language from whatever you usually used, you could just tell them that it would be impractical, and that would be the end of it.

Now, with server-based applications, everything is changed. You're now subject to market forces in what language you choose. If you try to pretend that nothing has changed, and just use C and C++, like most of our competitors did, you are setting yourself up for a fall. A little startup using a more powerful language will eat your lunch.

Incremental Development

There is a certain style of software development associated with Lisp. One of its traditions is incremental development: you start by writing, as quickly as possible, a program that does almost nothing. Then you gradually add features to it, but at every step you have working code.

I think this way you get better software, written faster. Everything about Lisp is tuned to this style of programming, because Lisp programmers have worked this way for at least thirty years.

The Viaweb editor must be one of the most extreme cases of incremental development. It began with a 120-line program for generating Web sites that I had used in an example in a book that I finished just before we started Viaweb. The Viaweb editor, which eventually grew to be about 25,000 lines of code, grew incrementally from this program. I never once sat down and rewrote the whole thing. I don't think I was ever more than a day or two without running code.

The whole development process was one long series of gradual changes.

This style of development fits well with the rolling releases that are possible with Web-based software. It's also a faster way to get software written generally.

Interactive Toplevel

Lisp's interactive toplevel is a great help in developing software rapidly. But the biggest advantage for us was probably in finding bugs. As I mentioned before, with Web-based applications you have the users' data on your servers and can usually reproduce bugs.

When one of the customer support people came to me with a report of a bug in the editor, I would load the code into the Lisp interpreter and log into the user's account. If I was able to reproduce the bug I'd get an actual break loop, telling me exactly what was going wrong. Often I could fix the code and release a fix right away. And when I say right away, I mean while the user was still on the phone.

Such fast turnaround on bug fixes put us into an impossibly tempting position. If we could catch and fix a bug while the user was still on the phone, it was very tempting for us to give the user the impression that they were imagining it. And so we sometimes (to their delight) had the customer support people tell the user to just try logging in again and see if they still had the problem. And of course when the user logged back in they'd get the newly released version of the software with the bug fixed, and everything would work fine. I realize this was a bit sneaky of us, but it was also a lot of fun.

Macros for Html

Lisp macros were another big win for us. We used them very extensively in the Viaweb editor. It could accurately be described as one big macro. And that gives you an idea of how much we depended on Lisp, because no other language has macros in the sense that Lisp does.

One way we used macros was to generate Html. There is a very natural fit between macros and Html, because Html is a prefix notation like Lisp, and Html is recursive like Lisp. So we had macro calls within macro calls, generating the most complicated Html, and it was all still very manageable.

Embedded Languages

Another big use for macros was the embedded language we had for describing pages, called Rtml. (We made up various explanations for what Rtml was supposed to stand for, but actually I named it after Robert Morris, the other founder of Viaweb, whose username is Rtm.)

Every page made by our software was generated by a program written in Rtml. We called these programs templates to make them less frightening, but they were real programs. In fact, they were Lisp programs. Rtml was a combination of macros and the built-in Lisp operators.

Users could write their own Rtml templates to describe what they wanted their pages to look like. We had a structure editor for manipulating these templates, a lot like the structure editor they

had in Interlisp. Instead of typing free-form text, you cut and pasted bits of code together. This meant that it was impossible to get syntax errors. It also meant that we didn't have to display the parentheses in the underlying s-expressions: we could show structure by indentation. By this means we made the language look a lot less threatening.

We also designed Rtml so that there could be no errors at runtime: every Rtml program yielded some kind of Web page, and you could debug it by hacking it until it produced the page you meant it to.

Initially we expected our users to be Web consultants, and we expected them to use Rtml a lot. We provided some default templates for section pages and item pages and so on, and the idea was that the users could take them and modify them to make whatever pages they wanted.

In fact it turned out that Web consultants didn't like Viaweb. Consultants, as a general rule, like to use products that are too hard for their clients to use, because it guarantees them ongoing employment. Consultants would come to our Web site, which said all over it that our software was so easy to use that it would let anyone make an online store in five minutes, and they'd say, there's no way we're using that. So we didn't get a lot of interest from Web consultants. Instead the users all tended to be end-users, the actual merchants themselves. They loved the idea of being in control of their own Web sites. And this kind of user did not want to do any kind of programming. They just used the default templates.

So Rtml didn't end up being the main interface to the program. It ended up playing two roles. First of all, it was an escape valve for the really sophisticated users, who wanted something our built-in templates couldn't provide. Somewhere in the course of doing Viaweb, someone gave me a very useful piece of advice: users always want an upgrade path, even though as a rule they'll never take it. Rtml was our upgrade path. If you wanted to, you could get absolute control over everything on your pages.

Only one out of every couple hundred users actually wrote their own templates. And this led to the second advantage of Rtml. By looking at the way these users modified our built-in templates, we knew what we needed to add to them. Eventually we made it our goal that no one should ever have to use Rtml. Our built-in templates should do everything people wanted. In this new approach, Rtml served us as a warning sign that something was missing in our software.

The third and biggest win from using Rtml was the advantage we ourselves got from it. Even if we had been the only people who used Rtml, it would have been very much worth while writing the software that way. Having that extra layer of abstraction in our software gave us a big advantage over competitors. It made the design of our software much cleaner, for one thing. Instead of just having bits of actual C or Perl code that generated our Web pages, like our competitors, we had a very high-level language for generating Web pages, and our page styles specified in that. It made the code much cleaner and easier to modify. I've already mentioned that Web-based applications get released as a series of many small modifications. When you do that you want to be able to know how serious any given modification is. By dividing your code into layers, you get a better handle on this. Modifying stuff in lower layers (Rtml itself) was a serious matter to be done rarely, and after much thought. Whereas modifying the top layers (template code) was something you could do quickly without worrying too much about the consequences.

Rtml was a very Lispy proposition. It was mostly Lisp macros, to start with. The online editor was, behind the scenes, manipulating s-expressions. And when people ran templates, they got compiled into Lisp functions by calling compile at runtime.

Rtml even depended heavily on keyword parameters, which up to that time I had always considered one of the more dubious features of Common Lisp. Because of the way Web-based software gets released, you have to design the software so that it's easy to change. And Rtml itself had to be easy to change, just like any other part of the software. Most of the operators in Rtml were designed to take keyword parameters, and what a help that turned out to be. If I wanted to add another dimension to the behavior of one of the operators, I could just add a new keyword parameter, and everyone's existing templates would continue to work. A few of the Rtml operators didn't take keyword parameters, because I didn't think I'd ever need to change them, and almost every one I ended up kicking myself about later. If I could go back and start over from scratch, one of the things I'd change would be that I'd make every Rtml operator take keyword parameters.

We had a couple embedded languages within the editor, in fact. Another one, which we didn't expose directly to the users, was for describing images. Viaweb included an image generator, written in C, that could take a description of an image, create that image, and return its url. We used s-expressions to describe these images as well.

Closures Simulate Subroutines

One of the problems with using Web pages as a UI is the inherent statelessness of Web sessions. We got around this by using lexical closures to simulate subroutine-like behavior. If you understand about continuations, one way to explain what we did would be to say that we wrote our software in continuation-passing style.

When most web-based software generates a link on a page, it tends to be thinking, if the user clicks on this link, I want to call this cgi script with these arguments. When our software generated a link, it could think, if the user clicks on this link, I want to run this piece of code. And the piece of code could an arbitrary piece of code, possibly (in fact, usually) containing free variables whose value came from the surrounding context.

The way we did this was to write a macro that took an initial argument expected to be a closure, followed by a body of code. The code would then get stored in a global hash table under a unique id, and whatever output was generated by the code in the body would appear within a link whose url contained that hash key. If that link was the next one clicked on, our software would find and call the corresponding bit of code, and the chain would continue. Effectively we were writing cgi scripts on the fly, except that they were closures that could refer to the surrounding context.

So far this sounds very theoretical, so let me give you an example of where this technique made an obvious difference. One of the things you often want to do in Web-based applications is edit an object with various types of properties. Many of the properties of an object can be represented as form fields or menus. If you're editing an object representing a person, for example, you might get a field, for their name, a menu choice for their title, and so on.

Now what happens when some object has a property that is a color? If you use ordinary cgi scripts, where everything has to happen on

one form, with an Update button at the bottom, you are going to have a hard time. You could use a text field and make the user type an rgb number into it, but end-users don't like that. Or you could have a menu of possible colors, but then you have to limit the possible colors, or otherwise even to offer just the standard Web colormap, you'd need 256 menu items with barely distinguishable names.

What we were able to do, in Viaweb, was display a color as a swatch representing the current value, followed by a button that said "Change." If the user clicked on the Change button they'd go to a page with an imagemap of colors to choose among. And after they chose a color, they'd be back on the page where they were editing the object's properties, with that color changed. This is what I mean about simulating subroutine-like behavior. The software could behave as if it were returning from having chosen a color. It wasn't, of course; it was making a new cgi call that looked like going back up a stack. But by using closures, we could make it look to the user, and to ourselves, as if we were just doing a subroutine call. We could write the code to say, if the user clicks on this link, go to the color selection page, and then come back here. This was just one of the places were we took advantage of this possibility. It made our software visibly more sophisticated than that of our competitors.