Five Questions about Language Design

paulgraham.com · Paul Graham · 2001-05 · source

May 2001
(These are some notes I made
for a panel discussion on programming language design
at MIT on May 10, 2001.)
1. Programming Languages Are for People.
Programming languages
are how people talk to computers. The computer would be just as
happy speaking any language that was unambiguous. The reason we
have high level languages is because people can't deal with
machine language. The point of programming
languages is to prevent our poor frail human brains from being
overwhelmed by a mass of detail.
Architects know that some kinds of design problems are more personal
than others. One of the cleanest, most abstract design problems
is designing bridges. There your job is largely a matter of spanning
a given distance with the least material. The other end of the
spectrum is designing chairs. Chair designers have to spend their
time thinking about human butts.
Software varies in the same way. Designing algorithms for routing
data through a network is a nice, abstract problem, like designing
bridges. Whereas designing programming languages is like designing
chairs: it's all about dealing with human weaknesses.
Most of us hate to acknowledge this. Designing systems of great
mathematical elegance sounds a lot more appealing to most of us
than pandering to human weaknesses. And there is a role for mathematical
elegance: some kinds of elegance make programs easier to understand.
But elegance is not an end in itself.
And when I say languages have to be designed to suit human weaknesses,
I don't mean that languages have to be designed for bad programmers.
In fact I think you ought to design for the
best programmers, but
even the best programmers have limitations. I don't think anyone
would like programming in a language where all the variables were
the letter x with integer subscripts.

2. Design for Yourself and Your Friends.

If you look at the history of programming languages, a lot of the best ones were languages designed for their own authors to use, and a lot of the worst ones were designed for other people to use.

When languages are designed for other people, it's always a specific group of other people: people not as smart as the language designer. So you get a language that talks down to you. Cobol is the most extreme case, but a lot of languages are pervaded by this spirit.

It has nothing to do with how abstract the language is. C is pretty low-level, but it was designed for its authors to use, and that's why hackers like it.

The argument for designing languages for bad programmers is that there are more bad programmers than good programmers. That may be so. But those few good programmers write a disproportionately large percentage of the software.

I'm interested in the question, how do you design a language that the very best hackers will like? I happen to think this is identical to the question, how do you design a good programming language?, but even if it isn't, it is at least an interesting question.

3. Give the Programmer as Much Control as Possible.

Many languages (especially the ones designed for other people) have the attitude of a governess: they try to prevent you from doing things that they think aren't good for you. I like the opposite approach: give the programmer as much control as you can.

When I first learned Lisp, what I liked most about it was that it considered me an equal partner. In the other languages I had learned up till then, there was the language and there was my program, written in the language, and the two were very separate. But in Lisp the functions and macros I wrote were just like those that made up the language itself. I could rewrite the language if I wanted. It had the same appeal as open-source software.

4. Aim for Brevity.

Brevity is underestimated and even scorned.

But if you look into the hearts of hackers, you'll see that they really love it. How many times have you heard hackers speak fondly of how in, say, APL, they could do amazing things with just a couple lines of code? I think anything that really smart people really

love is worth paying attention to.

I think almost anything
you can do to make programs shorter is good. There should be lots
of library functions; anything that can be implicit should be;
the syntax should be terse to a fault; even the names of things
should be short.

And it's not only programs that should be short. The manual should
be thin as well. A good part of manuals is taken up with clarifications
and reservations and warnings and special cases. If you force
yourself to shorten the manual, in the best case you do it by fixing
the things in the language that required so much explanation.

5. Admit What Hacking Is.

A lot of people wish that hacking was
mathematics, or at least something like a natural science. I think
hacking is more like architecture. Architecture is
related to physics, in the sense that architects have to design
buildings that don't fall down, but the actual goal of architects
is to make great buildings, not to make discoveries about statics.
What hackers like to do is make great programs.

And I think, at least in our own minds, we have to remember that it's
an admirable thing to write great programs, even when this work
doesn't translate easily into the conventional intellectual
currency of research papers. Intellectually, it is just as
worthwhile to design a language programmers will love as it is to design a
horrible one that embodies some idea you can publish a paper
about.

1. How to Organize Big Libraries?

Libraries are becoming an
increasingly important component of programming languages. They're
also getting bigger, and this can be dangerous. If it takes longer
to find the library function that will do what you want than it
would take to write it yourself, then all that code is doing nothing
but make your manual thick. (The Symbolics manuals were a case in
point.) So I think we will have to work on ways to organize
libraries. The ideal would be to design them so that the programmer
could guess what library call would do the right thing.

2. Are People Really Scared of Prefix Syntax?

This is an open
problem in the sense that I have wondered about it for years and
still don't know the answer. Prefix syntax seems perfectly natural

to me, except possibly for math. But it could be that a lot of Lisp's unpopularity is simply due to having an unfamiliar syntax. Whether to do anything about it, if it is true, is another question.

3. What Do You Need for Server-Based Software?

I think a lot of the most exciting new applications that get written in the next twenty years will be Web-based applications, meaning programs that sit on the server and talk to you through a Web browser. And to write these kinds of programs we may need some new things.

One thing we'll need is support for the new way that server-based apps get released. Instead of having one or two big releases a year, like desktop software, server-based apps get released as a series of small changes. You may have as many as five or ten releases a day. And as a rule everyone will always use the latest version.

You know how you can design programs to be debuggable? Well, server-based software likewise has to be designed to be changeable. You have to be able to change it easily, or at least to know what is a small change and what is a momentous one.

Another thing that might turn out to be useful for server based software, surprisingly, is continuations. In Web-based software you can use something like continuation-passing style to get the effect of subroutines in the inherently stateless world of a Web session. Maybe it would be worthwhile having actual continuations, if it was not too expensive.

4. What New Abstractions Are Left to Discover?

I'm not sure how reasonable a hope this is, but one thing I would really love to do, personally, is discover a new abstraction-- something that would make as much of a difference as having first class functions or recursion or even keyword parameters. This may be an impossible dream. These things don't get discovered that often. But I am always looking.

1. You Can Use Whatever Language You Want.

Writing application programs used to mean writing desktop software. And in desktop software there is a big bias toward writing the application in the same language as the operating system. And so ten years ago, writing software pretty much meant writing software in C.

Eventually a tradition evolved:
application programs must not be written in unusual languages.
And this tradition had so long to develop that nontechnical people
like managers and venture capitalists also learned it.
Server-based software blows away this whole model. With server-based
software you can use any language you want. Almost nobody understands
this yet (especially not managers and venture capitalists).
A few hackers understand it, and that's why we even hear
about new, indy languages like Perl and Python. We're not hearing
about Perl and Python because people are using them to write Windows
apps.
What this means for us, as people interested in designing programming
languages, is that there is now potentially an actual audience for
our work.
2. Speed Comes from Profilers.
Language designers, or at least
language implementors, like to write compilers that generate fast
code. But I don't think this is what makes languages fast for users.
Knuth pointed out long ago that speed only matters in a few critical
bottlenecks. And anyone who's tried it knows that you can't guess
where these bottlenecks are. Profilers are the answer.
Language designers are solving the wrong problem. Users don't need
benchmarks to run fast. What they need is a language that can show
them what parts of their own programs need to be rewritten. That's
where speed comes from in practice. So maybe it would be a net
win if language implementors took half the time they would
have spent doing compiler optimizations and spent it writing a
good profiler instead.
3. You Need an Application to Drive the Design of a Language.
This may not be an absolute rule, but it seems like the best languages
all evolved together with some application they were being used to
write. C was written by people who needed it for systems programming.
Lisp was developed partly to do symbolic differentiation, and
McCarthy was so eager to get started that he was writing differentiation
programs even in the first paper on Lisp, in 1960.
It's especially good if your application solves some new problem.
That will tend to drive your language to have new features that
programmers need. I personally am interested in writing
a language that will be good for writing server-based applications.
[During the panel, Guy Steele also made this point, with the

additional suggestion that the application should not consist of writing the compiler for your language, unless your language happens to be intended for writing compilers.]

4. A Language Has to Be Good for Writing Throwaway Programs.

You know what a throwaway program is: something you write quickly for some limited task. I think if you looked around you'd find that a lot of big, serious programs started as throwaway programs. I would not be surprised if most programs started as throwaway programs. And so if you want to make a language that's good for writing software in general, it has to be good for writing throwaway programs, because that is the larval stage of most software.

5. Syntax Is Connected to Semantics.

It's traditional to think of syntax and semantics as being completely separate. This will sound shocking, but it may be that they aren't.

I think that what you want in your language may be related to how you express it.

I was talking recently to Robert Morris, and he pointed out that operator overloading is a bigger win in languages with infix syntax. In a language with prefix syntax, any function you define is effectively an operator. If you want to define a plus for a new type of number you've made up, you can just define a new function to add them. If you do that in a language with infix syntax, there's a big difference in appearance between the use of an overloaded operator and a function call.

1. New Programming Languages.

Back in the 1970s it was fashionable to design new programming languages. Recently it hasn't been. But I think server-based software will make new languages fashionable again. With server-based software, you can use any language you want, so if someone does design a language that actually seems better than others that are available, there will be people who take a risk and use it.

2. Time-Sharing.

Richard Kelsey gave this as an idea whose time has come again in the last panel, and I completely agree with him. My guess (and Microsoft's guess, it seems) is that much computing will move from the desktop onto remote servers. In other words, time-sharing is back. And I think there will need to be support for it at the language level. For example, I know that Richard

and Jonathan Rees have done a lot of work implementing process scheduling within Scheme 48.

3. Efficiency.

Recently it was starting to seem that computers were finally fast enough. More and more we were starting to hear about byte code, which implies to me at least that we feel we have cycles to spare. But I don't think we will, with server-based software. Someone is going to have to pay for the servers that the software runs on, and the number of users they can support per machine will be the divisor of their capital cost.

So I think efficiency will matter, at least in computational bottlenecks. It will be especially important to do i/o fast, because server-based applications do a lot of i/o.

It may turn out that byte code is not a win, in the end. Sun and Microsoft seem to be facing off in a kind of a battle of the byte codes at the moment. But they're doing it because byte code is a convenient place to insert themselves into the process, not because byte code is in itself a good idea. It may turn out that this whole battleground gets bypassed. That would be kind of amusing.

1. Clients.

This is just a guess, but my guess is that the winning model for most applications will be purely server-based. Designing software that works on the assumption that everyone will have your client is like designing a society on the assumption that everyone will just be honest. It would certainly be convenient, but you have to assume it will never happen.

I think there will be a proliferation of devices that have some kind of Web access, and all you'll be able to assume about them is that they can support simple html and forms. Will you have a browser on your cell phone? Will there be a phone in your palm pilot? Will your blackberry get a bigger screen? Will you be able to browse the Web on your gameboy? Your watch? I don't know. And I don't have to know if I bet on everything just being on the server. It's just so much more robust to have all the brains on the server.

2. Object-Oriented Programming.

I realize this is a controversial one, but I don't think object-oriented programming is such a big deal. I think it is a fine model for certain kinds

of applications that need that specific kind of data structure, like window systems, simulations, and cad programs. But I don't see why it ought to be the model for all programming.

I think part of the reason people in big companies like object-oriented programming is because it yields a lot of what looks like work. Something that might naturally be represented as, say, a list of integers, can now be represented as a class with all kinds of scaffolding and hustle and bustle.

Another attraction of object-oriented programming is that methods give you some of the effect of first class functions. But this is old news to Lisp programmers. When you have actual first class functions, you can just use them in whatever way is appropriate to the task at hand, instead of forcing everything into a mold of classes and methods.

What this means for language design, I think, is that you shouldn't build object-oriented programming in too deeply. Maybe the answer is to offer more general, underlying stuff, and let people design whatever object systems they want as libraries.

3. Design by Committee.

Having your language designed by a committee is a big pitfall, and not just for the reasons everyone knows about. Everyone knows that committees tend to yield lumpy, inconsistent designs. But I think a greater danger is that they won't take risks. When one person is in charge he can take risks that a committee would never agree on.

Is it necessary to take risks to design a good language though? Many people might suspect that language design is something where you should stick fairly close to the conventional wisdom. I bet this isn't true. In everything else people do, reward is proportionate to risk. Why should language design be any different?