Being Popular

Paul Graham

This article was written as a kind of business plan for a new Lisp. So it is missing (because it takes for granted) the most important feature of a good programming language: very powerful abstractions. *Written March 2001, Revised April 17, 2001.*

A friend of mine once told an eminent operating systems expert that he wanted to design a really good programming language. The expert told him that it would be a waste of time, that programming languages don't become popular or unpopular based on their merits, and so no matter how good his language was, no one would use it. At least, that was what had happened to the language *he* had designed.

What does make a language popular? Do popular languages deserve their popularity? Is it worth trying to define a good programming language? How would you do it?

I think the answers to these questions can be found by looking at hackers, and learning what they want. Programming languages are *for* hackers, and a programming language is good as a programming language (rather than, say, an exercise in denotational semantics or compiler design) if and only if hackers like it.

1 The Mechanics of Popularity

It's true, certainly, that most people don't choose programming languages simply based on their merits. Most programmers are told what language to use by someone else. And yet I think the effect of such external factors on the popularity of programming languages is not as great as it's sometimes thought to be. I think a bigger problem is that a hacker's idea of a good programming language is not the same as most language designers'.

Between the two, the hacker's opinion is the one that matters. Programming languages are not theorems. They're tools, designed for people, and they have to be designed to suit human strengths and weaknesses as much as shoes have to be designed for human feet. If a shoe pinches when you put it on, it's a bad shoe, however elegant it may be as a piece of sculpture.

It may be that the majority of programmers can't tell a good language from a bad one. But that's no different with any other tool. It doesn't mean that it's a waste of time to try designing a good language. Expert hackers can tell a good language when they see one, and they'll use it. Expert hackers are a tiny minority, admittedly, but that tiny minority write all the good software, and their influence is such that the rest of the programmers will tend to use whatever language they use. Often, indeed, it is not merely influence but command: often the expert hackers are the very people who, as their bosses or faculty advisors, tell the other programmers what language to use.

The opinion of expert hackers is not the only force that determines the relative popularity of programming languages—legacy software (Cobol) and hype (Ada, Java) also play a role—but I think it is the most powerful force over the long term. Given an initial critical mass and enough time, a programming language probably becomes about as popular as it deserves to be. And popularity further separates good languages from bad ones, because feedback from real live users always leads to improvements. Look at how much any popular language has changed during its life. Perl and Fortran are extreme cases, but even Lisp has changed a lot. Lisp 1.5 didn't have macros, for example; these evolved later, after hackers at MIT had spent a couple years using Lisp to write real programs.°

So whether or not a language has to be good to be popular, I think a language has to be popular to be good. And it has to stay popular to stay good. The state of the art in programming languages doesn't stand still. And yet the Lisps we have today are still pretty much what they had at MIT in the mid-1980s, because that's the last time Lisp had a sufficiently large and demanding user base.

Of course, hackers have to know about a language before they can use it. How are they to hear? From other hackers. But there has to be some initial group of hackers using the language for others even to hear about it. I wonder how large this group has to be; how many users make a critical mass? Off the top of my head, I'd say twenty. If a language had twenty separate users, meaning twenty users who decided on their own to use it, I'd consider it to be real.

Getting there can't be easy. I would not be surprised if it is harder to get from zero to twenty than from twenty to a thousand. The best way to get those initial twenty users is probably to use a trojan horse: to give people an application they want, which happens to be written in the new language.

2 External Factors

Let's start by acknowledging one external factor that does affect the popularity of a programming language. To become popular, a programming language has to be the scripting language of a popular system. Fortran and Cobol were the scripting languages of early IBM mainframes. C was the scripting language of Unix, and so, later, was Perl. Tcl is the scripting language of Tk. Java and Javascript are intended to be the scripting languages of web browsers.

Lisp is not a massively popular language because it is not the scripting language of a massively popular system. What popularity it retains dates back to the 1960s and 1970s, when it was the scripting language of MIT. A lot of the great programmers of the day were associated with MIT at some point. And in the early 1970s, before C, MIT's dialect of Lisp, called MacLisp, was one of the only programming languages a serious hacker would want to use.

Today Lisp is the scripting language of two moderately popular systems, Emacs and Autocad, and for that reason I suspect that most of the Lisp programming done today is done in Emacs Lisp or AutoLisp.

Programming languages don't exist in isolation. To hack is a transitive verb—hackers are usually hacking *something*—and in practice languages are judged relative to whatever they're used to hack. So if you want to design a popular language, you either have to supply more than a language, or you have to design your language to replace the scripting language of some existing system.

Common Lisp is unpopular partly because it's an orphan. It did originally come with a system to hack: the Lisp Machine. But Lisp Machines (along with parallel computers) were steamrollered by the increasing power of general purpose processors in the 1980s. Common Lisp might have remained popular if it had been a good scripting language for Unix. It is, alas, an atrociously bad one.

One way to describe this situation is to say that a language isn't judged on its own merits. Another view is that a programming language really isn't a programming language unless it's also the scripting language of something. This only seems unfair if it comes as a surprise. I think it's no more unfair than expecting a programming language to have, say, an implementation. It's just part of what a programming language is.

A programming language does need a good implementation, of course, and this must be free. Companies will pay for software, but individual hackers won't, and it's the hackers you need to attract.

A language also needs to have a book about it. The book should be thin, well-written, and full of good examples. K&R is the ideal here. At the moment I'd almost say that a language has to have a book published by O'Reilly. That's becoming the test of mattering to hackers.

There should be online documentation as well. In fact, the book can start as online documentation. But I don't think that physical books are outmoded yet. Their format is convenient, and the *de facto* censorship imposed by publishers is a useful if imperfect filter. Bookstores are one of the most important places for learning about new languages.

3 Brevity

Given that you can supply the three things any language needs—a free implementation, a book, and something to hack—how do you make a language that hackers will like?

One thing hackers like is brevity. Hackers are lazy, in the same way that mathematicians and modernist architects are lazy: they hate anything extraneous. It would not be far from the truth to say that a hacker about to write a program decides what language to use, at least subconsciously, based on the total number of characters he'll have to type. If this isn't precisely how hackers think, a language designer would do well to act as if it were.

It is a mistake to try to baby the user with long-winded expressions that are meant to resemble English. Cobol is notorious for this flaw. A hacker would consider being asked to write

```
add x to y giving z
```

instead of

```
z = x+y
```

as something between an insult to his intelligence and a sin against God. Syntax counts.

It has sometimes been said that Lisp should use `first` and `rest` instead of `car` and `cdr`, because it would make programs easier to read. Maybe for the first couple hours. But a hacker can learn quickly enough that `car` means the first element of a list and `cdr` means the rest. Using `first` and `rest` means 50% more typing. And they are also different lengths, meaning that the arguments won't line up when they're called, as `car` and `cdr` often are, in successive lines. I've found that it matters a lot how code lines up on the page. I can barely read Lisp code when it is set in a variable-width font, and friends say this is true for other languages too.

Brevity is one place where strongly typed languages lose. All other things being equal, no one wants to begin a program with a bunch of declarations. Anything that can be implicit, should be.

The individual tokens should be short as well. Perl and Common Lisp occupy opposite poles on this question. Perl programs can be almost cryptically dense, while the names of built-in Common Lisp operators are comically long. The designers of Common Lisp probably expected users to have text editors that would type these long names for them. But the cost of a long name is not just the cost of typing it. There is also the cost of reading it, and the cost of the space it takes up on your screen.

4 Hackability

There is one thing more important than brevity to a hacker: being able to do what you want. In the history of programming languages a surprising amount of effort has gone into preventing programmers from doing things considered to be improper. This is a dangerously presumptuous plan. How can the language designer know what the programmer is going to need to do? I think language designers would do better to consider their target user to be a genius who will need to do things they never anticipated, rather than a bumbler who needs to be protected from himself. The bumbler will shoot himself in the foot anyway. You may save him from referring to variables in another package, but you can't save him from writing a badly designed program to solve the wrong problem, and taking forever to do it.

Good programmers often want to do dangerous and unsavory things. By unsavory I mean things that go behind whatever semantic facade the language is trying to present: getting hold of the internal representation of some high-level abstraction, for example. Hackers like to hack, and hacking means getting inside things and second guessing the original designer.

Let yourself be second guessed. When you make any tool, people use it in ways you didn't intend, and this is especially true of a highly articulated tool like a programming language. Many a hacker will want to tweak your semantic model in a way that you never imagined. I say, let them; give the programmer access to as much internal stuff as you can without endangering runtime systems like the garbage collector.

In Common Lisp I have often wanted to iterate through the fields of a struct—to comb out references to a deleted object, for example, or find fields that are uninitialized. I know the structs are just vectors underneath. And yet I can't write a general purpose function that I can call on any struct. I can only access the fields by name, because that's what a struct is supposed to mean.

A hacker may only want to subvert the intended model of things once or twice in a big program. But what a difference it makes to be able to. And it may be more than a question of just solving a problem. There is a kind of pleasure here too. Hackers share the surgeon's secret pleasure in poking about in gross innards, the teenager's secret pleasure in popping zits.° For boys, at least, certain kinds of horrors are fascinating. *Maxim* magazine publishes an annual volume of photographs, containing a mix of pin-ups and grisly accidents. They know their audience.

Historically, Lisp has been good at letting hackers have their way. The political correctness of Common Lisp is an aberration. Early Lisps let you get your hands on everything. A good deal of that spirit is, fortunately, preserved in macros. What a wonderful thing, to be able to make arbitrary transformations on the source code.

Classic macros are a real hacker's tool—simple, powerful, and dangerous. It's so easy to understand what they do: you call a function on the macro's arguments, and whatever it returns gets inserted in place of the macro call. Hygienic macros embody the opposite principle. They try to protect you from understanding what they're doing. I have never heard hygienic macros explained in one sentence. And they are a classic example of the dangers of deciding what programmers are allowed to want. Hygienic macros are intended to protect me from variable capture, among other things, but variable capture is exactly what I want in some macros.

A really good language should be both clean and dirty: cleanly designed, with a small core of well understood and highly orthogonal operators, but dirty in the sense that it lets hackers have their way with it. C is like this. So were the early Lisps. A real hacker's language will always have a slightly raffish character.

A good programming language should have features that make the kind of people who use the phrase "software engineering" shake their heads disapprovingly. At the other end of the continuum are languages like Ada and Pascal, models of propriety that are good for teaching and not much else.

5 Throwaway Programs

To be attractive to hackers, a language must be good for writing the kinds of programs they want to write. And that means, perhaps surprisingly, that it has to be good for writing throwaway programs.

A throwaway program is a program you write quickly for some limited task: a program to automate some system administration task, or generate test data for a simulation, or convert data from one format to another. The surprising thing about throwaway programs is that, like the "temporary" buildings built at so many American universities during World War II, they often don't get thrown away. Many evolve into real programs, with real features and real users.

I have a hunch that the best big programs begin life this way, rather than being designed big from the start, like the Hoover Dam. It's terrifying to build something big from scratch. When people take on a project that's too big, they become overwhelmed. The project either gets bogged down, or the result is sterile and wooden: a shopping mall rather than a real downtown, Brasilia rather than Rome, Ada rather than C.

Another way to get a big program is to start with a throwaway program and keep improving it. This approach is less daunting, and the design of the program benefits from evolution. I think, if one looked, that this would turn out to be the way most big programs were developed. And those that did evolve this way are probably still written in whatever language they were first written in, because it's rare for a program to be ported, except for political reasons. And so, paradoxically, if you want to make a language that is used for big systems, you have to make it good for writing throwaway programs, because that's where big systems come from.

Perl is a striking example of this idea. It was not only designed for writing throwaway programs, but was pretty much a throwaway program itself. Perl began life as a collection of utilities for generating reports, and only evolved into a programming language as the throwaway programs people wrote in it grew larger. It was not until Perl 5 (if then) that the language was suitable for writing serious programs, and yet it was already massively popular.

What makes a language good for throwaway programs? To start with, it must be readily available. A throwaway program is something that you expect to write in an hour. So the language probably must already be installed on the computer you're using. It can't be something you have to install before you use it. It has to be there. C was there because it came with the operating system. Perl was there because it was originally a tool for system administrators, and yours had already installed it.

Being available means more than being installed, though. An interactive language, with a command-line interface, is more available than one that you have to compile and run separately. A popular programming language should be interactive, and start up fast.

Another thing you want in a throwaway program is brevity. Brevity is always attractive to hackers, and never more so than in a program they expect to turn out in an hour.

6 Libraries

Of course the ultimate in brevity is to have the program already written for you, and merely to call it. And this brings us to what I think will be an increasingly important feature of programming languages: library functions. Perl wins because it has large libraries for manipulating strings. This class of library functions are especially important for throwaway programs, which are often originally written for converting or extracting data. Many Perl programs probably begin as just a couple library calls stuck together.

I think a lot of the advances that happen in programming languages in the next fifty years will have to do with library functions. I think future programming languages will have libraries that are as carefully designed as the core language. Programming language design will not be about whether to make your language strongly or weakly typed, or object oriented, or functional, or whatever, but about how to design great libraries. The kind of language designers who like to think about how to design type systems may shudder at this. It's almost like writing applications! Too bad. Languages are for programmers, and libraries are what programmers need.

It's hard to design good libraries. It's not simply a matter of writing a lot of code. Once the libraries get too big, it can sometimes take longer to find the function you need than to write the code yourself. Libraries need to be designed using a small set of orthogonal operators, just like the core language. It ought to be possible for the programmer to *guess* what library call will do what he needs.

Libraries are one place Common Lisp falls short. There are only rudimentary libraries for manipulating strings, and almost none for talking to the operating system. For historical reasons, Common Lisp tries to pretend that the OS doesn't exist. And because you can't talk to the OS, you're unlikely to be able to write a serious program using only the built-in operators in Common Lisp. You have to use some implementation-specific hacks as well, and in practice these tend not to give you everything you want. Hackers would think a lot more highly of Lisp if Common Lisp had powerful string libraries and good OS support.

7 Syntax

Could a language with Lisp's syntax, or more precisely, lack of syntax, ever become popular? I don't know the answer to this question. I do think that syntax is not the main reason Lisp isn't currently popular. Common Lisp has worse problems than unfamiliar syntax. I know several programmers who are comfortable with prefix syntax and yet use Perl by default, because it has powerful string libraries and can talk to the OS.

There are two possible problems with prefix notation: that it is unfamiliar to programmers, and that it is not dense enough. The conventional wisdom in the Lisp world is that the first problem is the real one. I'm not so sure. Yes, prefix notation makes ordinary programmers panic. But I don't think ordinary programmers' opinions matter. Languages become popular or unpopular based on what expert hackers think of them, and I think expert hackers might be able to deal with prefix notation. Perl syntax can be pretty incomprehensible, but that has not stood in the way of Perl's popularity. If anything it may have helped foster a Perl cult.

A more serious problem is the diffuseness of prefix notation. For expert hackers, that really is a problem. No one wants to write `(aref a x y)` when they could write `a[x,y]`.

In this particular case there is a way to finesse our way out of the problem. If we treat data structures as if they were functions on indexes, we could write `(a x y)` instead, which is even shorter than the Perl form. Similar tricks may shorten other types of expressions.

We can get rid of (or make optional) a lot of parentheses by making indentation significant. That's how programmers read code anyway: when indentation says one thing and delimiters say another, we go by the indentation. Treating indentation as significant would eliminate this common source of bugs as well as making programs shorter.

Sometimes infix syntax is easier to read. This is especially true for math expressions. I've used Lisp my whole programming life and I still don't find prefix math expressions natural. And yet it is convenient, especially when you're generating code, to have operators that take any number of arguments. So if we do have infix syntax, it should probably be implemented as some kind of read-macro.

I don't think we should be religiously opposed to introducing syntax into Lisp, as long as it translates in a well-understood way into underlying s-expressions. There is already a good deal of syntax in Lisp. It's not necessarily bad to introduce more, as long as no one is forced to use it. In Common Lisp, some delimiters are reserved for the language, suggesting that at least some of the designers intended to have more syntax in the future.

One of the most egregiously unlispy pieces of syntax in Common Lisp occurs in `format` strings; `format` is a language in its own right, and that language is not Lisp. If there were a plan for introducing more syntax into Lisp, `format` specifiers might be able to be included in it. It would be a good thing if macros could generate `format` specifiers the way they generate any other kind of code.

An eminent Lisp hacker told me that his copy of CLTL falls open to the part about `format`. Mine too. This probably indicates room for improvement. It may also mean that programs do a lot of I/O.

8 Efficiency

A good language, as everyone knows, should generate fast code. But in practice I don't think fast code comes primarily from things you do in the design of the language. As Knuth pointed out long ago, speed only matters in certain critical bottlenecks. And as many programmers have observed since, one is very often mistaken about where these bottlenecks are.

So, in practice, the way to get fast code is to have a *very* good profiler, rather than by, say, making the language strongly typed. You don't need to know the type of every argument in every call in the program. You do need to be able to declare the types of arguments in the bottlenecks. And even more, you need to be able to find out where the bottlenecks are.

One complaint people have had with Lisp is that it's hard to tell what's expensive. This might be true. It might also be inevitable, if you want to have a very abstract language. And in any case I think good profiling would go a long way toward fixing the problem: you'd soon *learn* what was expensive.

Part of the problem here is social. Language designers like to write fast compilers. That's how they measure their skill. They think of the profiler as an add-on, at best. But in practice a good profiler may do more to improve the speed of actual programs written in the language than a compiler that generates fast code. Here, again, language designers are somewhat out of touch with their users. They do a really good job of solving slightly the wrong problem.

It might be a good idea to have an active profiler—to push performance data to the programmer instead of waiting for him to come asking for it. For example, the editor could display bottlenecks in red when the programmer edits the source code. Another approach would be to somehow represent what's happening in running programs. This would be an especially big win in server-based applications, where you have lots of running programs to look at. An active profiler could show graphically what's happening in memory as a program's running, or even make sounds that tell what's happening.

Sound is a good cue to problems. In one place I worked, we had a big board of dials showing what was happening to our web servers. The hands were moved by little servomotors that made a slight noise when they turned. I couldn't see the board from my desk, but I found that I could tell immediately, by the sound, when there was a problem with a server.

It might even be possible to write a profiler that would automatically detect inefficient algorithms. I would not be surprised if certain patterns of memory access turned out to be sure signs of bad algorithms. If there were a little guy running around inside the computer executing our programs, he would probably have as long and plaintive a tale to tell about his job as a federal government employee. I often have a feeling that I'm sending the processor on a lot of wild goose chases, but I've never had a good way to look at what it's doing.

A number of Lisps now compile into byte code, which is then executed by an interpreter. This is usually done to make the implementation easier to port, but it could be a useful language feature. It might be a good idea to to make the byte code an official part of the language, and to allow programmers to use inline byte code in bottlenecks. Then such optimizations would be portable too.

The nature of speed, as perceived by the end-user, may be changing. With the rise of server-based applications, more and more programs may turn out to be I/O-bound. It will be worth making I/O fast. The language can help with straightforward measures like simple, fast, formatted output functions, and also with deep structural changes like caching and persistent objects.

Users are interested in response time. But another kind of efficiency will be increasingly important: the number of simultaneous users you can support per processor. Many of the interesting applications written in the near future will be server-based, and the number of users per server is the critical question for anyone hosting such applications. In the capital cost of a business offering a server-based application, this is the divisor.

For years, efficiency hasn't mattered much in most end-user applications. Developers have been able to assume that each user would have an increasingly powerful processor sitting on their desk. And by Parkinson's Law, software has expanded to use the resources available. That will change with server-based applications. In that world, the hardware and software will be supplied together. For companies that offer server-based applications, it will make a very big difference to the bottom line how many users they can support per server.

In some applications, the processor will be the limiting factor, and execution speed will be the most important thing to optimize. But often memory will be the limit; the number of simultaneous users will be determined by the amount of memory you need for each user's data. The language can help here too. Good support for threads will enable all the users to share a single heap. It may also help to have persistent objects and/or language level support for lazy loading.

9 Time

The last ingredient a popular language needs is time. No one wants to write programs in a language that might go away, as so many programming languages do. So most hackers will tend to wait until a language has been around for a couple years before even considering using it.

Inventors of wonderful new things are often surprised to discover this, but you need time to get any message through to people. A friend of mine rarely does anything the first time someone asks him. He knows that people sometimes ask for things that they turn out not to want. To avoid wasting his time, he waits till the third or fourth time he's asked to do something; by then, whoever's asking him may be fairly annoyed, but at least they probably really do want whatever they're asking for.

Most people have learned to do a similar sort of filtering on new things they hear about. They don't even start paying attention until they've heard about something ten times. They're perfectly justified: the majority of hot new whatevers do turn out to be a waste of time, and eventually go away. By delaying learning VRML, I avoided having to learn it at all.

So anyone who invents something new has to expect to keep repeating their message for years before people will start to get it. We wrote what was, as far as I know, the first web-server based application, and it took us years to get it through to people that it didn't have to be downloaded. It wasn't that they were stupid. They just had us tuned out.

The good news is, simple repetition solves the problem. All you have to do is keep telling your story, and eventually people will start to hear. It's not when people notice you're there that they pay attention; it's when they notice you're *still* there.

It's just as well that it usually takes a while to gain momentum. Most technologies evolve a good deal even after they're first launched—programming languages especially. Nothing could be better, for a new techology, than a few years of being used only by a small number of early adopters. Early adopters are sophisticated and demanding, and quickly flush out whatever flaws remain in your technology. When you only have a few users you can be in close contact with all of them. And early adopters are forgiving when you improve your system, even if this causes some breakage.

There are two ways new technology gets introduced: the organic growth method, and the big bang method. The organic growth method is exemplified by the classic seat-of-the-pants underfunded garage startup. A couple guys, working in obscurity, develop some new technology. They launch it with no marketing and initially have only a few (fanatically devoted) users. They continue to improve the technology, and meanwhile their user base grows by word of mouth. Before they know it, they're big.

The other approach, the big bang method, is exemplified by the VC-backed, heavily marketed startup. They rush to develop a product, launch it with great publicity, and immediately (they hope) have a large user base.

Generally, the garage guys envy the big bang guys. The big bang guys are smooth and confident and respected by the VCs. They can afford the best of everything, and the PR campaign surrounding the launch has the side effect of making them celebrities. The organic growth guys, sitting in their garage, feel poor and unloved. And yet I think they are often mistaken to feel sorry for themselves. Organic growth seems to yield better technology and richer founders than the big bang method. If you look at the dominant technologies today, you'll find that most of them grew organically.

This pattern doesn't only apply to companies. You see it in sponsored research too. Multics and Common Lisp were big-bang projects, and Unix and MacLisp were organic growth projects.

10 Redesign

"The best writing is rewriting," wrote E. B. White. Every good writer knows this, and it's true for software too. The most important part of design is redesign. Programming languages, especially, don't get redesigned enough.

To write good software you must simultaneously keep two opposing ideas in your head. You need the young hacker's naive faith in in his abilities, and at the same time the veteran's skepticism. You have to be able to think "how hard can it be?" with one half of your brain while thinking "it will never work" with the other.

The trick is to realize that there's no real contradiction here. You want to be optimistic and skeptical about two different things. You have to be optimistic about the possibility of solving the problem, but skeptical about the value of whatever solution you've got so far.

People who do good work often think that whatever they're working on is no good. Others see what they've done and are full of wonder, but the creator is full of worry. This pattern is no coincidence: it is the worry that made the work good.

If you can keep hope and worry balanced, they will drive a project forward the same way your two legs drive a bicycle forward. In the first phase of the two-cycle innovation engine, you work furiously on some problem, inspired by your confidence that you'll be able to solve it. In the second phase, you look at what you've done in the cold light of morning, and see all its flaws very clearly. But as long as your critical spirit doesn't outweigh your hope, you'll be able to look at your admittedly incomplete system, and think, how hard can it be to get the rest of the way?, thereby continuing the cycle.

It's tricky to keep the two forces balanced. In young hackers, optimism predominates. They produce something, are convinced it's great, and never improve it. In old hackers, skepticism predominates, and they won't even dare to take on ambitious projects.

Anything you can do to keep the redesign cycle going is good. Prose can be rewritten over and over until you're happy with it. But software, as a rule, doesn't get redesigned enough. Prose has readers, but software has *users.* If a writer rewrites an essay, people who read the old version are unlikely to complain that their thoughts have been broken by some newly introduced incompatibility.

Users are a double-edged sword. They can help you improve your language, but they can also deter you from improving it. So choose your users carefully, and be slow to grow their number. Having users is like optimization: the wise course is to delay it. Also, as a general rule, you can at any given time get away with changing more than you think. Introducing change is like pulling off a bandage: the pain is a memory almost as soon as you feel it.

Everyone knows that it's not a good idea to have a language designed by a committee. Committees yield bad design. But I think the worst danger of committees is that they interfere with *redesign.* It is so much work to introduce changes that no one wants to bother. Whatever a committee decides tends to stay that way, even if most of the members don't like it.

Even a committee of two gets in the way of redesign. This happens particularly in the interfaces between pieces of software written by two different people. To change the interface both have to agree to change it at once. And so interfaces tend not to change at all, which is a problem because they tend to be one of the most ad hoc parts of any system.

One solution here might be to design systems so that interfaces are horizontal instead of vertical—so that modules are always vertically stacked strata of abstraction. Then the interface will tend to be owned by one of them. The lower of two levels will either be a language in which the upper is written, in which case the lower level will own the interface, or it will be a slave, in which case the interface can be dictated by the upper level.

11 Lisp

What all this implies is that there is hope for a new Lisp. There is hope for any language that gives hackers what they want, including Lisp. I think we may have made a mistake in thinking that hackers are turned off by Lisp's strangeness. This comforting illusion may have prevented us from seeing the real problem with Lisp, or at least Common Lisp, which is that it sucks for doing what hackers want to do. A hacker's language needs powerful libraries and something to hack. Common Lisp has neither. A hacker's language is terse and hackable. Common Lisp is not.

The good news is, it's not Lisp that sucks, but Common Lisp. If we can develop a new Lisp that is a real hacker's language, I think hackers will use it. They will use whatever language does the job. All we have to do is make sure this new Lisp does some important job better than other languages.

History offers some encouragement. Over time, successive new programming languages have taken more and more features from Lisp. There is no longer much left to copy before the language you've made is Lisp. The latest hot language, Python, is a watered-down Lisp with infix syntax and no macros. A new Lisp would be a natural step in this progression.

I sometimes think that it would be a good marketing trick to call it an improved version of Python. That sounds hipper than Lisp. To many people, Lisp is a slow AI language with a lot of parentheses. Fritz Kunze's official biography carefully avoids mentioning the L-word.° But my guess is that we shouldn't be afraid to call the new Lisp Lisp. Lisp still has a lot of latent respect among the very best hackers—the ones who took 6.001 and understood it, for example. And those are the users you need to win.

In "How to be a Hacker," Eric Raymond describes Lisp as something like Latin or Greek—a language you should learn as an intellectual exercise, even though you won't actually use it:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

If I didn't know Lisp, reading this would set me asking questions. A language that will make me a better programmer? (For the rest of my days?) Languages are tools. A language that would make me a better programmer, if it means anything at all, means a language that would be better for programming. And that is in fact the implication of what Raymond is saying, though he himself may not realize it.

As long as that idea is still floating around, I think hackers will be receptive enough to a new Lisp, even if it is called Lisp. But this Lisp must be a hacker's language, like the classic Lisps of the 1970s. It must be terse, simple, and hackable. And it must have powerful libraries for doing what hackers want to do now.

In the matter of libraries I think there is room to beat languages like Perl and Python at their own game. A lot of the new applications that will need to be written in the coming years will be server-based applications. There's no reason a new Lisp shouldn't have string libraries as good as Perl, and if this new Lisp also had powerful libraries for server-based applications, it could be very popular. Real hackers won't turn up their noses at a new tool that will let them solve hard problems with a few library calls. Remember, hackers are lazy.

It could be an even bigger win to have core language support for server-based applications. For example, explicit support for programs with multiple users, or data ownership at the level of type tags.

Server-based applications also give us the answer to the question of what this new Lisp will be used to hack. It would not hurt to make Lisp better as a scripting language for Unix. (It would be hard to make it worse.) But I think there are areas where existing languages would be easier to beat. I think it might be better to follow the model of Tcl, and supply the Lisp together with a complete system for supporting server-based applications. Lisp is a natural fit for server-based applications. Lexical closures provide a way to get the effect of subroutines when the UI is just a series of web pages. S-expressions map nicely onto HTML, and macros are good at generating it. There need to be better tools for writing server-based applications, and there needs to be a new Lisp, and the two would work very well together.

12 The Dream Language

By way of summary, let's try describing the hacker's dream language. The dream language is beautiful, clean, and terse. It has an interactive toplevel that starts up fast. You can write programs to solve common problems with very little code. Nearly all the code in any program you write is code that's specific to your application. Everything else has been done for you.

The syntax of the language is brief to a fault. You never have to type an unnecessary character, or even to use the shift key much.

Using big abstractions you can write the first version of a program very quickly. Later, when you want to optimize, there's a really good profiler that tells you where to focus your attention. You can make inner loops blindingly fast, even writing inline byte code if you need to.

There are lots of good examples to learn from, and the language is intuitive enough that you can learn how to use it from examples in a couple minutes. You don't need to look in the manual much. The manual is thin, and has few warnings and qualifications.

The language has a small core, and powerful, highly orthogonal libraries that are as carefully designed as the core language. The libraries all work well together; everything in the language fits together like the parts in a fine camera. Nothing is deprecated, or retained for compatibility. The source code of all the libraries is readily available. It's easy to talk to the operating system and to applications written in other languages.

The language is built in layers. The higher-level abstractions are built in a very transparent way out of lower-level abstractions, which you can get hold of if you want.

Nothing is hidden from you that doesn't absolutely have to be. The language offers abstractions only as a way of saving you work, rather than as a way of telling you what to do. In fact, the language encourages you to be an equal participant in its design. You can change everything about it, including even its syntax, and anything you write has, as much as possible, the same status as what comes predefined.

Notes

2 Macros very close to the modern idea were proposed by Timothy Hart in 1964, two years after Lisp 1.5 was released. What was missing, initially, were ways to avoid variable capture and multiple evaluation; Hart's examples are subject to both.

5 In *When the Air Hits Your Brain,* neurosurgeon Frank Vertosick recounts a conversation in which his chief resident, Gary, talks about the difference between surgeons and internists ("fleas"):

> Gary and I ordered a large pizza and found an open booth. The chief lit a cigarette. "Look at those goddamn fleas, jabbering about some disease they'll see once in their lifetimes. That's the trouble with fleas, they only like the bizarre stuff. They hate their bread and butter cases. That's the difference between us and the fucking fleas. See, we love big juicy lumbar disc herniations, but they hate hypertension...."

It's hard to think of a lumbar disc herniation as juicy (except literally). And yet I think I know what they mean. I've often had a juicy bug to track down. Someone who's not a programmer would find it hard to imagine that there could be pleasure in a bug. Surely it's better if everything just works. In one way, it is. And yet there is undeniably a grim satisfaction in hunting down certain sorts of bugs.

13 Fritz Kunze is the president of the leading commercial Lisp vendor. His biography reads:

> Fritz Kunze, President, co-founded Franz Inc. in 1984. Franz is the world-wide market leader supplying CLOS Systems and CLOS persistent storage systems. In addition Franz defined and popularized the category Dynamic Object Oriented Programming Languages. Prior to Franz, Fritz was a graduate student in mathematics and computer science at the University of California at Berkeley. He holds an M.S. in Mathematics from the University of California at Berkeley.

Fritz evidently believes that he'll do better by saying that Franz offers something people have never heard of than by saying that they offer Lisp. He is probably right, for Franz's target market.