PAUL GRAHAM

# The Hundred-Year Language

April 2003

*(This essay is derived from a keynote talk at PyCon 2003.)*

It's hard to predict what life will be like in a hundred years. There are only a few things we can say with certainty. We know that everyone will drive flying cars, that zoning laws will be relaxed to allow buildings hundreds of stories tall, that it will be dark most of the time, and that women will all be trained in the martial arts. Here I want to zoom in on one detail of this picture. What kind of programming language will they use to write the software controlling those flying cars?

This is worth thinking about not so much because we'll actually get to use these languages as because, if we're lucky, we'll use languages on the path from this point to that.

I think that, like species, languages will form evolutionary trees, with dead-ends branching off all over. We can see this happening already. Cobol, for all its sometime popularity, does not seem to have any intellectual descendants. It is an evolutionary dead-end-- a Neanderthal language.

I predict a similar fate for Java. People sometimes send me mail saying, "How can you say that Java won't turn out to be a successful language? It's already a successful language." And I admit that it is, if you measure success by shelf space taken up by books on it (particularly individual books on it), or by the number of undergrads who believe they have to learn it to get a job. When I say Java won't turn out to be a successful language, I mean something more specific: that Java will turn out to be an evolutionary dead-end, like Cobol.





This is just a guess. I may be wrong. My point here is not to dis Java, but to raise the issue of evolutionary trees and get people asking, where on the tree is language X? The reason to ask this question isn't just so that our ghosts can say, in a hundred years, I told you so. It's because staying close to the main branches is a useful heuristic for finding languages that will be good to program in now.

At any given time, you're probably happiest on the main branches of an evolutionary tree. Even when there were still plenty of Neanderthals, it must have sucked to be one. The Cro-Magnons would have been constantly coming over and beating you up and stealing your food.

The reason I want to know what languages will be like in a hundred years is so that I know what branch of the tree to bet on now.

The evolution of languages differs from the evolution of species because branches can converge. The Fortran branch, for example, seems to be merging with the descendants of Algol. In theory this is possible for species too, but it's not likely to have happened to any bigger than a cell.

Convergence is more likely for languages partly because the space of possibilities is smaller, and partly because mutations are not random. Language designers deliberately incorporate ideas from other languages.

It's especially useful for language designers to

think about where the evolution of programming languages is likely to lead, because they can steer accordingly. In that case, "stay on a main branch" becomes more than a way to choose a good language. It becomes a heuristic for making the right decisions about language design.

Any programming language can be divided into two parts: some set of fundamental operators that play the role of axioms, and the rest of the language, which could in principle be written in terms of these fundamental operators.

I think the fundamental operators are the most important factor in a language's long term survival. The rest you can change. It's like the rule that in buying a house you should consider location first of all. Everything else you can fix later, but you can't fix the location.

I think it's important not just that the axioms be well chosen, but that there be few of them. Mathematicians have always felt this way about axioms-- the fewer, the better-- and I think they're onto something.

At the very least, it has to be a useful exercise to look closely at the core of a language to see if there are any axioms that could be weeded out. I've found in my long career as a slob that cruft breeds cruft, and I've seen this happen in software as well as under beds and in the corners of rooms.

I have a hunch that the main branches of the evolutionary tree pass through the languages that have the smallest, cleanest cores. The more of a language you can write in itself, the better.

Of course, I'm making a big assumption in even asking what programming languages will be like in a hundred years. Will we even be writing programs in a hundred years? Won't we just tell computers what we want them to do?





There hasn't been a lot of progress in that department so far. My guess is that a hundred years from now people will still tell computers what to do using programs we would recognize as such. There may be tasks that we solve now by writing programs and which in a hundred years you won't have to write programs to solve, but I think there will still be a good deal of programming of the type that we do today.

It may seem presumptuous to think anyone can predict what any technology will look like in a hundred years. But remember that we already have almost fifty years of history behind us. Looking forward a hundred years is a graspable idea when we consider how slowly languages have evolved in the past fifty.

Languages evolve slowly because they're not really technologies. Languages are notation. A program is a formal description of the problem you want a computer to solve for you. So the rate of evolution in programming languages is more like the rate of evolution in mathematical notation than, say, transportation or communications. Mathematical notation does evolve, but not with the giant leaps you see in technology.

Whatever computers are made of in a hundred years, it seems safe to predict they will be much faster than they are now. If Moore's Law continues to put out, they will be 74 quintillion (73,786,976,294,838,206,464) times faster. That's kind of hard to imagine. And indeed, the most likely prediction in the speed department may be that Moore's Law will stop working. Anything that is supposed to double every eighteen months seems likely to run up against some kind of fundamental limit eventually. But I have no trouble believing that computers will be very much faster. Even if they only end up being a paltry million times faster, that should change the ground rules for programming languages substantially. Among other things, there will be

more room for what would now be considered slow languages, meaning languages that don't yield very efficient code.

And yet some applications will still demand speed. Some of the problems we want to solve with computers are created by computers; for example, the rate at which you have to process video images depends on the rate at which another computer can generate them. And there is another class of problems which inherently have an unlimited capacity to soak up cycles: image rendering, cryptography, simulations.

If some applications can be increasingly inefficient while others continue to demand all the speed the hardware can deliver, faster computers will mean that languages have to cover an ever wider range of efficiencies. We've seen this happening already. Current implementations of some popular new languages are shockingly wasteful by the standards of previous decades.

This isn't just something that happens with programming languages. It's a general historical trend. As technologies improve, each generation can do things that the previous generation would have considered wasteful. People thirty years ago would be astonished at how casually we make long distance phone calls. People a hundred years ago would be even more astonished that a package would one day travel from Boston to New York via Memphis.

I can already tell you what's going to happen to all those extra cycles that faster hardware is going to give us in the next hundred years. They're nearly all going to be wasted.

I learned to program when computer power was scarce. I can remember taking all the spaces out of my Basic programs so they would fit into the memory of a 4K TRS-80. The thought of all this stupendously inefficient software burning up cycles doing the same thing over and over seems kind of gross to me. But I think my intuitions





here are wrong. I'm like someone who grew up poor, and can't bear to spend money even for something important, like going to the doctor.

Some kinds of waste really are disgusting. SUVs, for example, would arguably be gross even if they ran on a fuel which would never run out and generated no pollution. SUVs are gross because they're the solution to a gross problem. (How to make minivans look more masculine.) But not all waste is bad. Now that we have the infrastructure to support it, counting the minutes of your long-distance calls starts to seem niggling. If you have the resources, it's more elegant to think of all phone calls as one kind of thing, no matter where the other person is.

There's good waste, and bad waste. I'm interested in good waste-- the kind where, by spending more, we can get simpler designs. How will we take advantage of the opportunities to waste cycles that we'll get from new, faster hardware?

The desire for speed is so deeply engrained in us, with our puny computers, that it will take a conscious effort to overcome it. In language design, we should be consciously seeking out situations where we can trade efficiency for even the smallest increase in convenience.

Most data structures exist because of speed. For example, many languages today have both strings and lists. Semantically, strings are more or less a subset of lists in which the elements are characters. So why do you need a separate data type? You don't, really. Strings only exist for efficiency. But it's lame to clutter up the semantics of the language with hacks to make programs run faster. Having strings in a language seems to be a case of premature optimization.

If we think of the core of a language as a set of axioms, surely it's gross to have additional axioms that add no expressive power, simply for the sake of efficiency. Efficiency is important, but

I don't think that's the right way to get it.

The right way to solve that problem, I think, is to separate the meaning of a program from the implementation details. Instead of having both lists and strings, have just lists, with some way to give the compiler optimization advice that will allow it to lay out strings as contiguous bytes if necessary.

Since speed doesn't matter in most of a program, you won't ordinarily need to bother with this sort of micromanagement. This will be more and more true as computers get faster.

Saying less about implementation should also make programs more flexible. Specifications change while a program is being written, and this is not only inevitable, but desirable.

The word "essay" comes from the French verb "essayer", which means "to try". An essay, in the original sense, is something you write to try to figure something out. This happens in software too. I think some of the best programs were essays, in the sense that the authors didn't know when they started exactly what they were trying to write.

Lisp hackers already know about the value of being flexible with data structures. We tend to write the first version of a program so that it does everything with lists. These initial versions can be so shockingly inefficient that it takes a conscious effort not to think about what they're doing, just as, for me at least, eating a steak requires a conscious effort not to think where it came from.

What programmers in a hundred years will be looking for, most of all, is a language where you can throw together an unbelievably inefficient version 1 of a program with the least possible effort. At least, that's how we'd describe it in present-day terms. What they'll say is that they want a language that's easy to program in.





Inefficient software isn't gross. What's gross is a language that makes programmers do needless work. Wasting programmer time is the true inefficiency, not wasting machine time. This will become ever more clear as computers get faster.

I think getting rid of strings is already something we could bear to think about. We did it in Arc, and it seems to be a win; some operations that would be awkward to describe as regular expressions can be described easily as recursive functions.

How far will this flattening of data structures go? I can think of possibilities that shock even me, with my conscientiously broadened mind. Will we get rid of arrays, for example? After all, they're just a subset of hash tables where the keys are vectors of integers. Will we replace hash tables themselves with lists?

There are more shocking prospects even than that. The Lisp that McCarthy described in 1960, for example, didn't have numbers. Logically, you don't need to have a separate notion of numbers, because you can represent them as lists: the integer n could be represented as a list of n elements. You can do math this way. It's just unbearably inefficient.

No one actually proposed implementing numbers as lists in practice. In fact, McCarthy's 1960 paper was not, at the time, intended to be implemented at all. It was a theoretical exercise, an attempt to create a more elegant alternative to the Turing Machine. When someone did, unexpectedly, take this paper and translate it into a working Lisp interpreter, numbers certainly weren't represented as lists; they were represented in binary, as in every other language.

Could a programming language go so far as to get rid of numbers as a fundamental data type? I ask this not so much as a serious question as as

a way to play chicken with the future. It's like the hypothetical case of an irresistible force meeting an immovable object-- here, an unimaginably inefficient implementation meeting unimaginably great resources. I don't see why not. The future is pretty long. If there's something we can do to decrease the number of axioms in the core language, that would seem to be the side to bet on as t approaches infinity. If the idea still seems unbearable in a hundred years, maybe it won't in a thousand.

Just to be clear about this, I'm not proposing that all numerical calculations would actually be carried out using lists. I'm proposing that the core language, prior to any additional notations about implementation, be defined this way. In practice any program that wanted to do any amount of math would probably represent numbers in binary, but this would be an optimization, not part of the core language semantics.

Another way to burn up cycles is to have many layers of software between the application and the hardware. This too is a trend we see happening already: many recent languages are compiled into byte code. Bill Woods once told me that, as a rule of thumb, each layer of interpretation costs a factor of 10 in speed. This extra cost buys you flexibility.

The very first version of Arc was an extreme case of this sort of multi-level slowness, with corresponding benefits. It was a classic "metacircular" interpreter written on top of Common Lisp, with a definite family resemblance to the eval function defined in McCarthy's original Lisp paper. The whole thing was only a couple hundred lines of code, so it was very easy to understand and change. The Common Lisp we used, CLisp, itself runs on top of a byte code interpreter. So here we had two levels of interpretation, one of them (the top one) shockingly inefficient, and the language was usable. Barely usable, I admit, but usable.





Writing software as multiple layers is a powerful technique even within applications. Bottom-up programming means writing a program as a series of layers, each of which serves as a language for the one above. This approach tends to yield smaller, more flexible programs. It's also the best route to that holy grail, reusability. A language is by definition reusable. The more of your application you can push down into a language for writing that type of application, the more of your software will be reusable.

Somehow the idea of reusability got attached to object-oriented programming in the 1980s, and no amount of evidence to the contrary seems to be able to shake it free. But although some object-oriented software is reusable, what makes it reusable is its bottom-upness, not its object-orientedness. Consider libraries: they're reusable because they're language, whether they're written in an object-oriented style or not.

I don't predict the demise of object-oriented programming, by the way. Though I don't think it has much to offer good programmers, except in certain specialized domains, it is irresistible to large organizations. Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches. Large organizations always tend to develop software this way, and I expect this to be as true in a hundred years as it is today.

As long as we're talking about the future, we had better talk about parallel computation, because that's where this idea seems to live. That is, no matter when you're talking, parallel computation seems to be something that is going to happen in the future.

Will the future ever catch up with it? People have been talking about parallel computation as something imminent for at least 20 years, and it hasn't affected programming practice much so far. Or hasn't it? Already chip designers have to

think about it, and so must people trying to write systems software on multi-cpu computers.

The real question is, how far up the ladder of abstraction will parallelism go? In a hundred years will it affect even application programmers? Or will it be something that compiler writers think about, but which is usually invisible in the source code of applications?

One thing that does seem likely is that most opportunities for parallelism will be wasted. This is a special case of my more general prediction that most of the extra computer power we're given will go to waste. I expect that, as with the stupendous speed of the underlying hardware, parallelism will be something that is available if you ask for it explicitly, but ordinarily not used. This implies that the kind of parallelism we have in a hundred years will not, except in special applications, be massive parallelism. I expect for ordinary programmers it will be more like being able to fork off processes that all end up running in parallel.

And this will, like asking for specific implementations of data structures, be something that you do fairly late in the life of a program, when you try to optimize it. Version 1s will ordinarily ignore any advantages to be got from parallel computation, just as they will ignore advantages to be got from specific representations of data.

Except in special kinds of applications, parallelism won't pervade the programs that are written in a hundred years. It would be premature optimization if it did.

How many programming languages will there be in a hundred years? There seem to be a huge number of new programming languages lately. Part of the reason is that faster hardware has allowed programmers to make different tradeoffs between speed and convenience, depending on the application. If this is a real trend, the





hardware we'll have in a hundred years should only increase it.

And yet there may be only a few widely-used languages in a hundred years. Part of the reason I say this is optimism: it seems that, if you did a really good job, you could make a language that was ideal for writing a slow version 1, and yet with the right optimization advice to the compiler, would also yield very fast code when necessary. So, since I'm optimistic, I'm going to predict that despite the huge gap they'll have between acceptable and maximal efficiency, programmers in a hundred years will have languages that can span most of it.

As this gap widens, profilers will become increasingly important. Little attention is paid to profiling now. Many people still seem to believe that the way to get fast applications is to write compilers that generate fast code. As the gap between acceptable and maximal performance widens, it will become increasingly clear that the way to get fast applications is to have a good guide from one to the other.

When I say there may only be a few languages, I'm not including domain-specific "little languages". I think such embedded languages are a great idea, and I expect them to proliferate. But I expect them to be written as thin enough skins that users can see the general-purpose language underneath.

Who will design the languages of the future? One of the most exciting trends in the last ten years has been the rise of open-source languages like Perl, Python, and Ruby. Language design is being taken over by hackers. The results so far are messy, but encouraging. There are some stunningly novel ideas in Perl, for example. Many are stunningly bad, but that's always true of ambitious efforts. At its current rate of mutation, God knows what Perl might evolve into in a hundred years.

It's not true that those who can't do, teach (some of the best hackers I know are professors), but it is true that there are a lot of things that those who teach can't do. Research imposes constraining caste restrictions. In any academic field there are topics that are ok to work on and others that aren't. Unfortunately the distinction between acceptable and forbidden topics is usually based on how intellectual the work sounds when described in research papers, rather than how important it is for getting good results. The extreme case is probably literature; people studying literature rarely say anything that would be of the slightest use to those producing it.

Though the situation is better in the sciences, the overlap between the kind of work you're allowed to do and the kind of work that yields good languages is distressingly small. (Olin Shivers has grumbled eloquently about this.) For example, types seem to be an inexhaustible source of research papers, despite the fact that static typing seems to preclude true macros-- without which, in my opinion, no language is worth using.

The trend is not merely toward languages being developed as open-source projects rather than "research", but toward languages being designed by the application programmers who need to use them, rather than by compiler writers. This seems a good trend and I expect it to continue.

Unlike physics in a hundred years, which is almost necessarily impossible to predict, I think it may be possible in principle to design a language now that would appeal to users in a hundred years.

One way to design a language is to just write down the program you'd like to be able to write, regardless of whether there is a compiler that can translate it or hardware that can run it. When you do this you can assume unlimited resources. It seems like we ought to be able to imagine unlimited resources as well today as in a





hundred years.

What program would one like to write? Whatever is least work. Except not quite: whatever *would be* least work if your ideas about programming weren't already influenced by the languages you're currently used to. Such influence can be so pervasive that it takes a great effort to overcome it. You'd think it would be obvious to creatures as lazy as us how to express a program with the least effort. In fact, our ideas about what's possible tend to be so limited by whatever language we think in that easier formulations of programs seem very surprising. They're something you have to discover, not something you naturally sink into.

One helpful trick here is to use the length of the program as an approximation for how much work it is to write. Not the length in characters, of course, but the length in distinct syntactic elements-- basically, the size of the parse tree. It may not be quite true that the shortest program is the least work to write, but it's close enough that you're better off aiming for the solid target of brevity than the fuzzy, nearby one of least work. Then the algorithm for language design becomes: look at a program and ask, is there any way to write this that's shorter?

In practice, writing programs in an imaginary hundred-year language will work to varying degrees depending on how close you are to the core. Sort routines you can write now. But it would be hard to predict now what kinds of libraries might be needed in a hundred years. Presumably many libraries will be for domains that don't even exist yet. If SETI@home works, for example, we'll need libraries for communicating with aliens. Unless of course they are sufficiently advanced that they already communicate in XML.

At the other extreme, I think you might be able to design the core language today. In fact, some might argue that it was already mostly designed in 1958.

If the hundred year language were available today, would we want to program in it? One way to answer this question is to look back. If present-day programming languages had been available in 1960, would anyone have wanted to use them?

In some ways, the answer is no. Languages today assume infrastructure that didn't exist in 1960. For example, a language in which indentation is significant, like Python, would not work very well on printer terminals. But putting such problems aside-- assuming, for example, that programs were all just written on paper-- would programmers of the 1960s have liked writing programs in the languages we use now?

I think so. Some of the less imaginative ones, who had artifacts of early languages built into their ideas of what a program was, might have had trouble. (How can you manipulate data without doing pointer arithmetic? How can you implement flow charts without gotos?) But I think the smartest programmers would have had no trouble making the most of present-day languages, if they'd had them.

If we had the hundred-year language now, it would at least make a great pseudocode. What about using it to write software? Since the hundred-year language will need to generate fast code for some applications, presumably it could generate code efficient enough to run acceptably well on our hardware. We might have to give more optimization advice than users in a hundred years, but it still might be a net win.

Now we have two ideas that, if you combine them, suggest interesting possibilities: (1) the hundred-year language could, in principle, be designed today, and (2) such a language, if it existed, might be good to program in today. When you see these ideas laid out like that, it's hard not to think, why not try writing the





hundred-year language now?

When you're working on language design, I think it is good to have such a target and to keep it consciously in mind. When you learn to drive, one of the principles they teach you is to align the car not by lining up the hood with the stripes painted on the road, but by aiming at some point in the distance. Even if all you care about is what happens in the next ten feet, this is the right answer. I think we can and should do the same thing with programming languages.

**Notes**

I believe Lisp Machine Lisp was the first language to embody the principle that declarations (except those of dynamic variables) were merely optimization advice, and would not change the meaning of a correct program. Common Lisp seems to have been the first to state this explicitly.

**Thanks** to Trevor Blackwell, Robert Morris, and Dan Giffin for reading drafts of this, and to Guido van Rossum, Jeremy Hylton, and the rest of the Python crew for inviting me to speak at PyCon.

- Japanese Translation

---