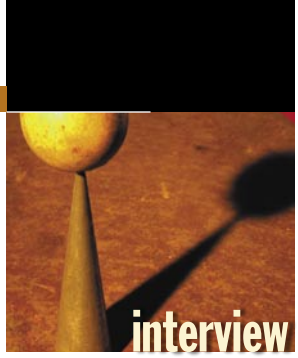


A Conversation with Alan Kay

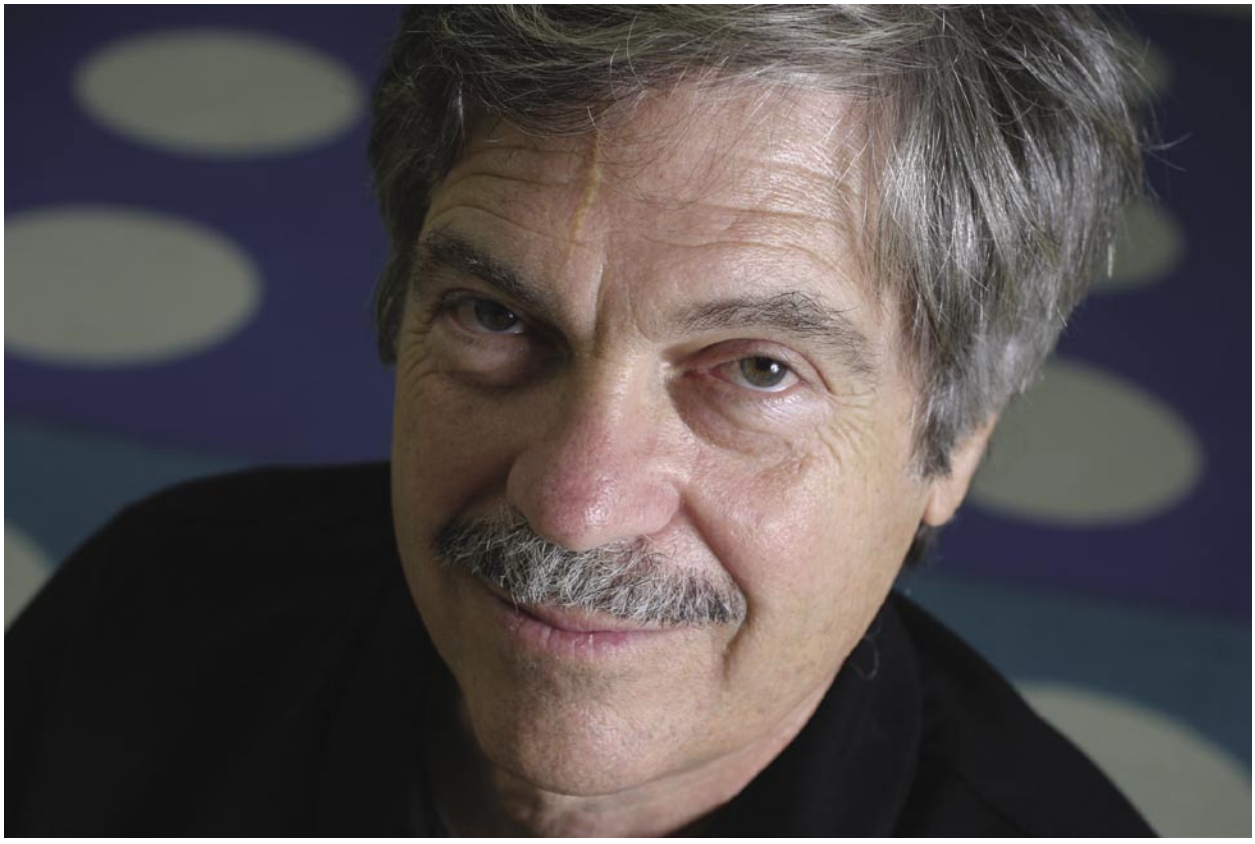
Big talk with THE CREATOR OF SMALLTALK—AND MUCH MORE.

When you want to gain a historical perspective on personal computing and programming languages, why not turn to one of the industry's preeminent pioneers? That would be Alan Kay, winner of last year's Turing Award for leading the team that invented Smalltalk, as well as for his fundamental contributions to personal computing.

Kay was one of the founders of the Xerox Palo Alto Research Center (PARC), where he led one of several groups that together developed modern workstations (and the forerunners of the Macintosh), Smalltalk, the overlapping window interface, desktop publishing, the Ethernet, laser printing, and network client-servers.

Prior to his work at PARC, Kay earned a Ph.D. in 1969 from the University of Utah, where he designed a graphical object-oriented personal computer and was a member of the research team that developed pioneering 3-D graphics work for the Advanced Research Projects Agency (ARPA). Kay was also a "slight participant" in the original design of the ARPANet, which later became the Internet. He holds undergraduate degrees in mathematics and molecular biology from the University of Colorado. After leaving Xerox PARC, Kay went on to become chief scientist of Atari, a Fellow of Apple Computer, and vice president of research and development at The Walt Disney Company.

Today he is Senior Fellow at Hewlett-Packard Labs and president of Viewpoints Research Institute, a nonprofit organization whose goal is to change how children are educated by creating a sample curriculum with support-



PHOTOGRAPHY BY GILLES MINGASSON

ing media for teaching math and science. This curriculum will use Squeak as its media, and will be highly interactive and constructive. Kay's deep interests in children and education have been the catalysts for many of his ideas over the years.

In addition to winning the Turing Award, Kay recently received the Draper Prize from the National Academy of Engineering and the Kyoto Prize in Advanced Technology, awarded every four years by the Inamori Foundation.

Guiding our tour through personal computing history with Kay is Stuart Feldman of IBM Research, where he is vice president and on-demand business transformation area strategist. Since joining IBM in 1995, Feldman has also served as vice president for Internet technology and was head of computer science in the research division.

Feldman also spent 11 years at Bellcore, where he held several research management positions in software engineering and computing systems, and 10 years at Bell Labs, where he was a computer science researcher. Feldman was a member of the original Unix team and is best known as the creator of the Make configuration management system and as the author of the first Fortran-77 compiler. He has a Ph.D. in applied mathematics from the Massachusetts Institute of Technology. He is a member of the *Queue* Advisory Board.

STUART FELDMAN One of the topics that some of the younger people on our *Queue* editorial board keep asking about is the history of programming languages. The *Queue* board has a bimodal generation distribution, and those members who are in their 20s or 30s seem genuinely confused about where programming languages might actually come from. It's my observation that we have one big language and one smaller language every decade—that appears to be all the field can afford. Smalltalk is one of those five- or 10-year events.

ALAN KAY In the late 1960s, Jean Sammet was able to track down and chronicle about 3,000 programming languages that were extant then. When things were simpler in a sense—but theoretically harder because the machines were slower, smaller, didn't have hard drives most of the time, and had bad tools—people nonetheless rolled their own operating systems and programming languages whenever they felt like it. So there are zillions of them around.

For a *Scientific American* article 20 years ago, I came up with a facetious sunspot theory, just noting that there's a major language or two every 10½ years, and in between those periods are what you might call hybrid languages. These could be looked at as either an improvement on the old thing or almost a new thing. I chronicled Fortran as an improvement on an old thing or almost a new thing, and Algol and Lisp were the new thing.

Then there was Simula, which the designers thought of as an extension of Algol. It was basically a preprocessor to Algol the way C++ was a preprocessor for C. It was a great concept and I was lucky enough to see it as almost a new thing. Smalltalk and Prolog happened in the early 1970s. The predecessor of Prolog was a wonderful thing that Carl Hewitt did in the late 1960s called Planner.

Perhaps it was commercialization in the 1980s that killed off the next expected new thing. Our plan and our hope was that the next generation of kids would come along and do something better than Smalltalk around 1984 or so. We all thought that the next level of programming language would be much more strategic and even policy-oriented and would have much more knowledge about what it was trying to do. But a variety of different things conspired together, and that next generation actually didn't show up. One could actually argue—as I sometimes do—that the success of commercial personal computing and operating systems has actually led to a considerable retrogression in many, many respects.

You could think of it as putting a low-pass filter on some of the good ideas from the '60s and '70s, as computing spread out much, much faster than educating unsophisticated people can happen. In the last 25 years or so, we actually got something like a pop culture, similar to what happened when television came on the scene and some of its inventors thought it would be a way of getting Shakespeare to the masses. But they forgot that you have to be more sophisticated and have more perspective to understand Shakespeare. What television was able to do was to capture people as they were.

So I think the lack of a real computer science today, and the lack of real software engineering today, is partly due to this pop culture.

SF So Smalltalk is to Shakespeare as Excel is to car crashes in the TV culture?

AK No, if you look at it really historically, Smalltalk counts as a minor Greek play that was miles ahead of what most other cultures were doing, but nowhere near what Shakespeare was able to do.

If you look at software today, through the lens of the history of engineering, it's certainly engineering of a sort—but it's the kind of engineering that people without the concept of the arch did. Most software today is very much like an Egyptian pyramid with millions of bricks piled on top of each other, with no structural integrity, but just done by brute force and thousands of slaves.

SF The analogy is even better because there are the hidden chambers that nobody can understand.

AK I would compare the Smalltalk stuff that we did in the '70s with something like a Gothic cathedral. We had two ideas, really. One of them we got from Lisp: late binding. The other one was the idea of objects. Those gave us something a little bit like the arch, so we were able to make complex, seemingly large structures out of very little material, but I wouldn't put us much past the engineering of 1,000 years ago.

If you look at [Doug] Engelbart's demo [a live online hypermedia demonstration of the pioneering work that Engelbart's group had been doing at Stanford Research Institute, presented at the 1968 Fall Joint Computer Conference], then you see many more ideas about how to boost the collective IQ of groups and help them to work together than you see in the commercial systems today. I think there's this very long lag between what you might call the best practice in computing research over the years and what is able to leak out and be adapted in the much more expedient and deadline-conscious outside world.

It's not that people are completely stupid, but if there's a big idea and you have deadlines and you have expedience and you have competitors, very likely what you'll do is take a low-pass filter on that idea and implement one part of it and miss what has to be done next. This happens over and over again. If you're using early-binding languages as most people do, rather than late-binding languages, then you really start getting locked in to stuff that you've already done. You can't reformulate things that easily.

Let's say the adoption of programming languages has very often been somewhat accidental, and the emphasis has very often been on how easy it is to implement the programming language rather than on its actual merits and features. For instance, Basic would never have surfaced because there was always a language better than Basic for that purpose. That language was Joss, which predated Basic and was beautiful. But Basic happened to be on a GE timesharing system that was done by Dartmouth, and when GE decided to franchise that, it started spreading Basic around just because it was there, not because it had any intrinsic merits whatsoever.

This happens over and over again. The languages of Niklaus Wirth have spread wildly and widely because he has been one of the most conscientious documenters of languages and one of the earlier ones to do algorithmic languages using p-codes (pseudocodes)—the same kinds of things that we use. The idea of using those things has a common origin in the hardware of a machine called the Burroughs B5000 from the early 1960s, which the establishment hated.

SF Partly because there wasn't any public information on most of it.

AK Let me beg to differ. I was there, and Burroughs actually hired college graduates to explain that machine to data-processing managers. There was an immense amount of information available. The problem was that the DP managers didn't want to learn new ways of computing, or even how to compute. IBM realized that and Burroughs didn't.

SF If memory serves, I was fascinated by that machine at the time, but I was unable to get the detail that made me understand it.

AK In fact, the original machine had two CPUs, and it was described quite adequately in a 1961 paper by Bob Barton, who was the main designer. One of the great documents was called "The Descriptor" and laid it out in detail. The problem was that almost everything in this machine was quite different and what it was trying to achieve was quite different.

The reason that line lived on—even though the establishment didn't like it—was precisely because it was almost impossible to crash it, and so the banking industry kept on buying this line of machines, starting with the B5000. Barton was one of my professors in college, and I had adapted some of the ideas on the first desktop machine that I did. Then we did a much better job of adapting the ideas at Xerox PARC (Palo Alto Research Center).

Neither Intel nor Motorola nor any other chip company understands the first thing about why that architecture was a good idea.

Just as an aside, to give you an interesting benchmark—on roughly the same system, roughly optimized the same way, a benchmark from 1979 at Xerox PARC runs only 50 times faster today. Moore's law has given us somewhere between 40,000 and 60,000 times improvement in that time. So there's approximately a factor of 1,000 in efficiency that has been lost by bad CPU architectures.

The myth that it doesn't matter what your processor architecture is—that Moore's law will take care of you—is totally false.

SF It also has something to do with why some languages succeed at certain times.

AK Yes, actually both Lisp and Smalltalk were done in by the eight-bit microprocessor—it's not because they're eight-bit micros, it's because the processor architectures were bad, and they just killed the dynamic languages.

Today these languages run reasonably because even though the architectures are still bad, the level 2 caches are so large that some fraction of the things that need to work, work reasonably well inside the caches; so both Lisp and Smalltalk can do their things and are viable today. But both of them are quite obsolete, of course.

The stuff that is in vogue today is only about "one-half" of those languages. Sun Microsystems had the right people to make Java into a first-class language, and I believe it was the Sun marketing people who rushed the thing out before it should have gotten out. They made it impossible for the Sun software people to do what needed to be done.

SF What should Java have had in it to be a first-quality language, not just a commercial success?

AK Like I said, it's a pop culture. A commercial hit record for teenagers doesn't have to have any particular musical merits. I think a lot of the success of various programming languages is expeditious gap-filling. Perl is another example of filling a tiny, short-term need, and then being a real problem in the longer term. Basically, a lot of the problems that computing has had in the last 25 years comes from systems where the designers were trying to fix some short-term thing and didn't think about whether the idea would scale if it were adopted. There should be a half-life on software so old software just melts away over 10 or 15 years.

It was a different culture in the '60s and '70s; the ARPA (Advanced Research Projects Agency) and PARC culture was basically a mathematical/scientific kind of culture and was interested in scaling, and of course, the Internet was an exercise in scaling. There are just two different worlds, and I don't think it's even that helpful for people from one world to complain about the other world—like

people from a literary culture complaining about the majority of the world that doesn't read for ideas. It's futile.

I don't spend time complaining about this stuff, because what happened in the last 20 years is quite normal, even though it was unfortunate. Once you have something that grows faster than education grows, you're always going to get a pop culture. It's well known that I tried to kill Smalltalk in the later '70s. There were a few years when it was the most wonderful thing in the world. It answered needs in a more compact and beautiful way than anything that had been done before. But time moves on. As we learned more and got more ambitious about what we wanted to do, we realized that there are all kinds of things in Smalltalk that don't scale the way they should—for instance, the reflection stuff that we had in there. It was one of the first languages to really be able to see itself, but now it is known how to do all levels of reflection much better—so we should implement that.

We saw after a couple of years that this could be done much better. The object model we saw after a couple of years could be done much better, etc. So the problem is—I've said this about both Smalltalk and Lisp—they tend to eat their young. What I mean is that both Lisp and Smalltalk are really fabulous vehicles, because they have a meta-system. They have so many ways of dealing with problems that the early-binding languages don't have, that it's very, very difficult for people who like Lisp or Smalltalk to imagine anything else.

Now just to mention a couple of things about Java: it really doesn't have a full meta-system. It has always had the problem—for a variety of reasons—of having two regimes, not one regime. It has things that aren't objects, and it has things that it calls objects. It has real difficulty in being dynamic. It has a garbage collector. So what? Those have been around for a long time. But it's not that great at adding to itself.

For many years, the development kits for Java were done in C++. That is a telling thing.

We looked at Java very closely in 1995 when we were starting on a major set of implementations, just because it's a lot of work to do a viable language kernel. The thing we liked least about Java was the way it was implemented. It had this old idea, which has never worked, of having a set of paper specs, having to implement the VM (virtual machine) to the paper specs, and then having benchmarks that try to validate what you've just implemented—and that has never resulted in a completely compatible system.

The technique that we had for Smalltalk was to write the VM in itself, so there's a Smalltalk simulator of the VM that was essentially the only specification of the VM. You could debug and you could answer any question about what the VM would do by submitting stuff to it, and you made every change that you were going to make to the VM by changing the simulator. After you had gotten everything debugged the way you wanted, you pushed the button and it would generate, without human hands touching it, a mathematically correct version of C that would go on whatever platform you were trying to get onto.

The result is that this system today, called Squeak, runs identically on more than two dozen platforms. Java does not do that. If you think about what the Internet means, it means you have to run identically on everything that is hooked to the Internet. So Java, to me, has always violated one of the prime things about software engineering in the world of the Internet.

Once we realized that Java was likely not to be compatible from platform to platform, we basically said we'll generate our own system that is absolutely compatible from platform to platform, and that's what we did.

Anybody can do that. If the pros at Sun had had a chance to fix Java, the world would be a much more pleasant place. This is not secret knowledge. It's just secret to this pop culture.

SF If nothing else, Lisp was carefully defined in terms of Lisp.

AK Yes, that was the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwell's Equations of Software!" This is the whole world of programming in a few lines that I can put my hand over.

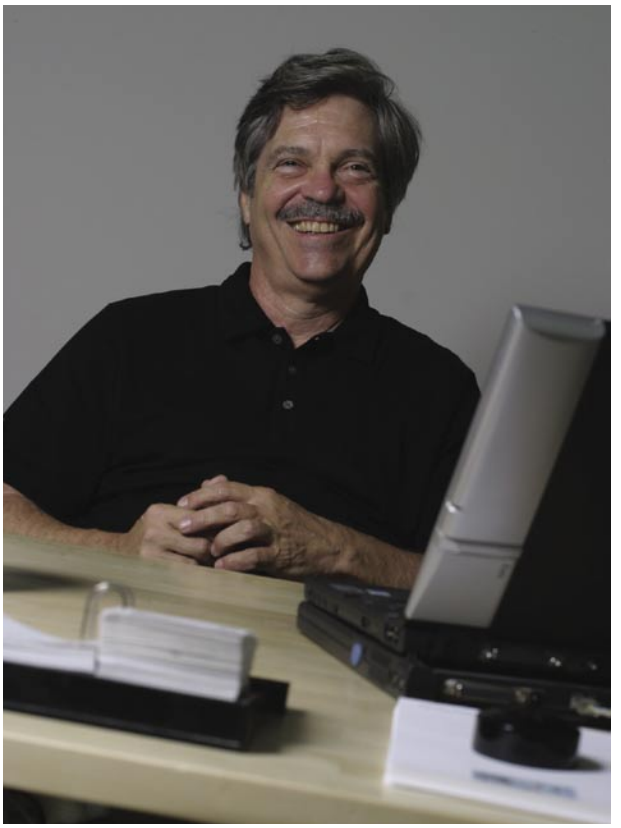
I realized that anytime I want to know what I'm doing, I can just write down the kernel of this thing in a half page and it's not going to lose any power. In fact, it's going to gain power by being able to reenter itself much more readily than most systems done the other way can possibly do.

All of these ideas could be part of both software engineering and computer science, but I fear—as far as I can tell—that most undergraduate degrees in computer science these days are basically Java vocational training. I've heard complaints from even mighty Stanford University with its illustrious faculty that basically the undergraduate computer science program is little more than Java certification.

SF Well, I must admit I was surprised recently when I discovered in a group of very good developers I managed, almost none of them knew C well enough to write expert low-level stuff. All of them were really good Java jocks.

AK In the 1960s Ted Steele spent several years promoting an idea called UNCOL (universal computer-oriented language), and, to me, by a weird and interesting process—mainly because it's easy to implement—C turned out to be UNCOL. I don't think any human being should write in it, but it's a great target for anybody who wants to do multiplatform things—especially if you pick the right subset.

The problem with the Cs, as you probably know if you've fooled around in detail with them, is that they're not quite kosher as far as their arithmetic is concerned. They are supposed to be, but they're not quite up to the IEEE standards. You have to pick a subset of C and you have to have some side information to get to a mathematically perfect transform of your VM.



SF To what do you attribute the long-term love of Smalltalk? There is a certain set of languages that I would assert people seem to love, not just use. I know many people who love C. I know very few who love C++, even though they may make their living on it.

AK You have to be a different kind of person to love C++. It is a really interesting example of how a well-meant idea went wrong, because [C++ creator] Bjarne Stroustrup was not trying to do what he has been criticized for. His idea was that first, it might be useful if you did to C what Simula did to Algol, which is basically act as a preprocessor for a different kind of architectural template for programming. It was basically for super-good programmers who are supposed to subclass everything, including the storage allocator, before they did anything serious. The result, of course, was that most programmers did not subclass much. So the people I know who like C++ and have done good things in C++ have been serious ironmen who have basically taken it for what it is, which is a kind of macroprocessor. I grew up with macro systems in the early '60s, and you have to do a lot of work to make them work for you—otherwise, they kill you.

SF Well, C++, after all, was programmed as a macro processor, in essence.

AK Yes, exactly. But so was Simula.

SF I put Smalltalk in this category of languages that have true devotees—people who genuinely like it or love it, not simply appreciate and use it.

AK In a history of Smalltalk I wrote for ACM, I characterized one way of looking at languages in this way: a lot of them are either the agglutination of features or they're a crystallization of style. Languages such as APL, Lisp, and Smalltalk are what you might call style languages, where there's a real center and imputed style to how you're supposed to do everything. Other languages such as PL/I and, indeed, languages that try to be additive without consolidation have often been more successful. I think the style languages appeal to people who have a certain mathematical laziness to them. Laziness actually pays off later on, because if you wind up spending a little extra time seeing that "oh, yes, this language is going to allow me to do this really, really nicely, and in a more general way than I could do it over here," usually that comes back to help you when you've had a new idea a year down the road. The agglutinative languages, on the other hand, tend to produce agglutinations and they are very, very difficult to untangle when you've had that new idea.

Also, I think the style languages tend to be late-binding languages. The agglutinative languages are usually early-binding. That makes a huge difference in the whole approach. The kinds of bugs you have to deal with, and

when you have to deal with them, is completely different.

Some people are completely religious about type systems and as a mathematician I love the idea of type systems, but nobody has ever come up with one that has enough scope. If you combine Simula and Lisp—Lisp didn't have data structures, it had instances of objects—you would have a dynamic type system that would give you the range of expression you need.

It would allow you to think the kinds of thoughts you need to think without worrying about what type something is, because you have a much, much wider range of things. What you're paying for is some of the checks that can be done at runtime, and, especially in the old days, you paid for it in some efficiencies. Now we get around the efficiency stuff the same way Barton did on the B5000: by just saying, "Screw it, we're going to execute this important stuff as directly as we possibly can." We're not going to worry about whether we can compile it into a von Neumann computer or not, and we will make the microcode do whatever we need to get around these inefficiencies because a lot of the inefficiencies are just putting stuff on obsolete hardware architectures.

I just think that's a two-culture divide. I've seen many meetings where people are unable to communicate just because of the stylistic differences in approaches.

SF I would characterize style languages as those with a very rigorous kernel that describes them intellectually. As Smalltalk went through a number of revolutions, to what extent did those change the core kernel, as opposed to improving the range of usefulness?

AK We'll never know the exact answer to your question because during the development of the system, from when Xerox put it out to this day, all the changes happened in a single thread of development at Xerox PARC. To the outside world, Smalltalk has changed almost not at all. Basically, it's just built on bigger and bigger libraries of different kinds.

But the good thing about the changes in Smalltalk was that it never got diluted, and the scope of the practical things you could think about doing in Smalltalk expanded dramatically during the period at Xerox PARC.

Basically what happened is this vehicle became more and more a programmer's vehicle and less and less a children's vehicle—the version that got put out, Smalltalk '80, I don't think it was ever programmed by a child. I don't think it could have been programmed by a child because it had lost some of its amenities, even as it gained pragmatic power.

So the death of Smalltalk in a way came as soon as it got recognized by real programmers as being something useful; they made it into more of their own image, and it started losing its nice end-user features.

But that's OK. This project that we started in 1995 was to make Squeak as an implementation vehicle for another end-user system for children. That was done quite well and is being used by many, many thousands of children around the world. The other way of looking at this is to realize that computers are made to be programmed by human beings. Let's just roll our own. Let's not complain about Java, or even about Smalltalk.

In fact, let's not even worry about Java. Let's not complain about Microsoft. Let's not worry about them because we know how to program computers, too, and in fact we know how to do it in a meta-way. We can set up an alternative point of view, and we're not the only ones who do this, as you're well aware.

There are numerous examples on the Internet of people who have gone to one level or another by making their own point of view. Squeak is the most comprehensive because it spans the whole field. It doesn't require any particular operating system to run because it's self-sufficient and has a full set of tools and applications and so forth, but there are many interesting functional languages, particularly in Europe, that are of interest.

One of my favorite old languages is one called Lucid by Ed Ashcroft. It was a beautiful idea. He said, "Hey, look, we can regard a variable as a stream, as some sort of ordered thing of its values and time, and use Christopher Strachey's idea that everything is wonderful about tail recursion and Lisp, except what it looks like." When he looked at Lisp, he had a great insight: which was that tail-recursive loops and Lisp are so clean because you're generating the right-hand side of all the assignment statements before you do any rebinding. So you're automatically forced to use only old values. You cannot rebind, so there are no race conditions on anything.

You just write down all of those things, and then when you do the tail recursion, you rebind all of those variables with these new values. Strachey said, "I can write that down like a sequential program, as a bunch of simultaneous assignment statements, and a loop that makes it easier to think of." That's basically what Lucid did—there is no reason that you have to think recursively for things that are basically iteration, and you can make these iterations as clean as a functional language if you have a better theory about what values are.

This idea, by the way, was used in [Squeak contributor] Dave Reed's fantastic thesis for coordinating object siblings where you have one logical object but many physical manifestations of the same object on different machines, and you have to make them track each other by transactions.

The way to get rid of these things (like Smalltalk) is to make something that is much, much more powerful as a computation model and much more expressive for the core programmer who is trying to write programs. In these late programming languages, you can disappear the old guy and just leave the new guy behind. So we are doing that at this moment.

SF What do you think a programming language should achieve and for whom, and then what is the model that goes with that idea?

AK Even if you're designing for professional programmers, in the end your programming language is basically a user-interface design. You will get much better results regardless of what you're trying to do if you think of it as a user-interface design. PARC is incorrectly credited with having invented the GUI. Of course, there were GUIs in the '60s. But I think we did do one good thing that hadn't been done before, and that was to realize the idea of change being eternal.

SF You never walk in the same river, otherwise known as Strachey streams.

AK The user interface, which is still the predominant approach today, is a user interface as the access to function. If the area is interesting, you eventually wind up with something that looks like the control panel of a nuclear reactor. So this is the agglutination of features.

SF Yes, a button on every pixel.

AK Corporate buyers often buy in terms of feature sets. But at PARC our idea was, since you never step in the same river twice, the number-one thing you want to make the user interface be is a learning environment—something that's explorable in various ways, something that is going to change over the lifetime of the user using this environment. New things are going to come on, and what does it mean for those new things to happen?

This means improvements not only in the applications but also in the user interface itself. Some of those ideas were quite manifest in the original Macintosh, but are much less manifest in the Macs of today—and of course never really made it to Microsoft. That just wasn't their way of thinking about things, and I think a programming language is the same way. Even if the user is an absolute expert, able to remember almost everything, I'm always interested in the difference between what you might call stark meaning and adjustable meaning.

I did quite a bit of study on that over the years to understand the influence of having something that you can read. It's known that our basic language mechanism for both reading and hearing has a fast and a slow process. The fast process has basically a surface phrasal-size nature, and then there's a slower one. This is why jokes require pauses; the joke is actually a jump from one context to another, and the slower guy, who is dealing with the real meanings, has to catch up to it.

There have been many, many studies of this. This argues that the surface form of a language, whatever it is, has to be adjustable in some form.

SF As you probably know, recent research has looked at how different parts of the brain recognize and react to jokes. Physically, they are quite distinct.

AK Yes. All creativity is an extended form of a joke. Most creativity is a transition from one context into another where things are more surprising. There's an element of surprise, and especially in science, there is often laughter that goes along with the "Aha." Art also has this element. Our job is to remind us that there are more contexts than the one that we're in—the one that we think is reality.

In the '60s, one of the primary goals of the computer science community was to arrive at an extensible language. As far as I know, only three ever actually worked, and the first Smalltalk was one of those three. Another very interesting one was done by Ned Irons, who invented the term *syntax-directed compiler* and did one of the first ones in the '60s. He did a wonderful extensible language called Imp.

One of the things that people realized from these extensible languages is that there is the unfortunate difficulty of making the meta-system easy to use. Smalltalk-72 was actually used by children. You're always extending the language without realizing it when you are making

ordinary classes. The result of this was that you didn't have to go into a more esoteric place like a compiler compiler—Yacc or something like that—to add some extension to the language.

But the flip side of the coin was that even good programmers and language designers tended to do terrible extensions when they were in the heat of programming, because design is something that is best done slowly and carefully.

SF And late-night extensible programming is unsupportable.

AK Exactly. So Smalltalk actually went from something that was completely extensible to one where we picked a syntax that allowed for a variety of forms of what was fixed, and concentrated on the extensibility of meaning in it.

This is not completely satisfactory. One of the things that I think should be done today is to have a fence that you have to hop to forcibly remind you that you're now in a meta-area—that you are now tinkering with the currency system itself, you are not just speculating. But it should allow you to do it without any other overhead once you've crossed this fence, because when you want to do it, you want to do it.

I could go on and on. I feel like my answers are quite trivial since nobody really knows how to design a good language, including me.

SF What do you wish you had done differently in the Smalltalk era?

AK I had the world's greatest group, and I should have made the world's two greatest groups. I didn't realize there are benefits to having real implementers and real users, and there are benefits to starting from scratch every few months. I hired finishers because I'm a good starter and a poor finisher, but it took me a long time to realize that I was interfering with them by trying to improve things.

I believe that the only kind of science computing can be is like the science of bridge building. Somebody has to build the bridges and other people have to tear them down and make better theories, and you have to keep on building bridges.

SF And every so often, you have to watch one fall into the drink. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums