An Interview with Gilad Bracha

Laurence Tratt, King's College London

Adam Welc, Oracle Labs

GILAD BRACHA'S PROGRAMMING language CV speaks for itself: he's the creator of the Newspeak programming language, a software engineer at Google (where he works on Dart), and a coauthor of the Java Language Specification. Previously, he worked on Strongtalk. Put simply, there are few greater authorities on the realities of modern programming languages. As part of this special issue, we talked to Bracha about his background, thoughts on the current state of programming languages, and where they might go in the future.

Q: What have been the major steps in your evolution as a language designer?

The first step was learning to program. My first programming language was BASIC, in high school. To quote Dijkstra, "it is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration." Draw your own conclusions.

After that, it was a very gradual evolution. I encountered Pascal and many others at university—APL was a high point, COBOL a low. I think that being exposed to a diverse set of languages at an early age was important. You might say that was a second step, but it wasn't a very advanced one. I remember reading about Ada and being so impressed with all the features it had compared to Pascal—I'm ashamed to say that at the time I actually thought that was a good thing.

For my undergraduate thesis, I worked on an implementation of Knuth's Metafont. It had declarative aspects to it, which was an eye opener. After I graduated, I encountered Smalltalk and Prolog. Both left a huge impression. I had begun to appreciate simplicity. Maybe that's the third step.

In grad school, I started working on programming languages professionally, doing research, and inventing new things, if you will. That would be step four—the point of no return.

Step five was when I started working on Java. Caring for such a beast was a whole new experience in terms of complexity, constraints, resources, and so on.

Finally, there came a time where I felt I could take all I had learned and build a complete language, Newspeak.

With Newspeak, were you trying to solve challenges that you thought other languages missed (or did poorly)?

Newspeak was intended to address four points: modularity, security, reflectivity, and interoperability. I don't think modularity has ever been properly addressed, and I think we succeeded on that front. Newspeak modules are first-class values and can be mutually recursive. You can subclass and mix-in modules, so you can extend entire class libraries as a unit. All modules are parametric, so all external dependencies are provided from the outside rather than by imports. This means modules are inherently sandboxed and always configurable. You might say dependency injection is built in to the language, but not in the obvious way.

Security was addressed well in Mark Miller's E language, but, as a rule, other languages haven't dealt with security properly. We're set up to make Newspeak secure, because the language supports the object capability model very naturally. In practice, the implementation is still lacking, but we're making progress on that.

Some languages/systems do well on reflectivity—which is closely related to what's now being called liveness—but most don't. We used

 0740-7459/14/$31.00 © 2014 IEEE

Smalltalk as our point of departure. We could have used Self or certain Lisps, but Smalltalk was the most realistic basis for me. There's actually a real tension between security and reflectivity. Happily, we've managed to preserve that wonderful property of Smalltalk in Newspeak, and in fact we're trying to improve upon it in some ways.

Finally, with interoperability, we wanted Newspeak to play well with others. Smalltalk has been notoriously bad at this, so we made sure we had a principled approach to foreign function calls. We use aliens, special objects that represent the world outside Newspeak. As a result, the same Newspeak GUI can bind to a native UI on Windows and to HTML on the browser, for example.

While Smalltalk's support for reflectivity is superb, it did less well in modularity, security, and interoperability. So the overall challenge for Newspeak was to address all four properties in one system.

You've been responsible for documenting and extending existing languages, as well as creating new languages. What are the challenges of a blank sheet of paper versus stepping into the design of a language with real users?

There's never a blank sheet of paper. There's always what came before, for better or worse. The question is how much freedom you have. In the mainstream, there's very little freedom. You try to make things a little bit better. You can only move a tiny bit forward, but each move can have a huge impact.

With a new language, you have a lot more freedom, but it's never absolute. You can push the state of the art, but it might be decades before the impact is felt, if ever.

The flip side of these challenges is the satisfaction you get from your work. In the mainstream, satisfaction comes from influencing the industry, affecting how many people work and see the world of programming. The rewards are external. With a new, pure design, the satisfaction comes from within—knowing that you've done something really elegant, even if the world at large pays little attention.

Some language communities have sharp distinctions between language designers and implementers. Is there an optimum balance?

I think that you need to be aware of the implementation, but you should be careful not to let those considerations dominate. If you insist on doing the principled thing, clever people will figure out how to implement it.

You've had experience in designing languages individually and as part of a committee. Is one approach better than the other, and will one win out?

Yes. Design by committee is very much inferior. You tend to get the intersection of the abilities of the committee members rather than the union. The best role for such committees is to review a stable design for minor inconsistencies, not to come up with new designs or major revisions.

I don't think one approach will ever win out. We're seeing a long tail of languages built by individuals while the major languages tend to be designed by a group in industry (not as bad as a real committee, but not as good as a single designer) and then taken over by standards bodies.

You've been working on the specification of Dart, which got its start by being translated into JavaScript. Is translating a new language into an existing one something you expect to see more of, or will people keep writing traditional compilers and VMs?

I expect we'll see a lot of translation into existing languages, especially JavaScript, but possibly Objective-C and Java. The root issue is the same in all cases. The major client platforms today don't always provide traditional compilers and VMs with the facilities they need. Hopefully, platforms will provide better primitives and become more open, but it's not clear we'll see the flexibility the traditional computer provided. Nevertheless, we'll still use traditional compilers and VMs, especially on servers.

Does it influence language design at all?

With a new language, you have a lot more freedom, but it's never absolute.

Sadly, it does. It tends to impact decisions around any feature that's very hard or costly to implement. For example, Dart doesn't support nonlocal returns because they're expensive to support in JavaScript (you compile them using exceptions, and browsers often provide poor performance for exception-handling code),

which in turn means that you can't have user-defined control constructs.

What examples have you seen of language features that seemed great on paper but fared badly in practice?
Checked exception handling. It seemed such a clear win. You could statically check something that hadn't been checkable before. The problem is that once you declare one function with the exception types it throws, everyone who calls it typically has to repeat the annotations. The contagion spreads like a plague.

Also, generics—especially variance schemes. I've done generics several times and have never been fully satisfied with the results. Nor am I pleased with other efforts in this area. One of these days, I'll write a screed entitled, "Generics considered harmful." The only reason I don't is because I don't have a great alternative (other than just don't do it, which often isn't acceptable).

> One of these days, I'll write a screed entitled, "Generics considered harmful."

Do the current batch of "mainstream(ish)" languages look like you might have envisioned when you started out?
What flows in the mainstream isn't water. The mainstream finds the path of least resistance, like a real stream, and it picks up a considerable amount of impurities along the way.

It's amazing and disappointing that given all the brilliant work that has been done in programming languages, what is widely used is so conservative. This is largely because programming languages are a cultural artifact. Progress is limited by people's ability to deal with change, and so things move very slowly.

Do debates like static versus dynamic typing have deep technological roots, or are they mostly cultural issues?
Both. The difference between the typed and untyped world views goes all the way to the basic foundations of logic, as Phil Wadler will tell you. It's as deep as you can go. On the other hand, most of the discussion is based on preconceived beliefs, very much like religion.

I believe one should try and utilize types where it makes sense, without being bound to one world view. Happily, that idea, which I and others in the research community have been promoting for decades, is finally making headway in the mainstream programming culture, as evidenced by Dart, Typescript, and Hack. I hope we'll see purer realizations of this notion as well.

What aspects of current mainstream languages do you think will look quaint in 50 years time?
Quaint is such a kind word. I think you mean obsolete or antiquated. I'd like to think that almost everything we program in today will fall into that category by then. I assume AI will get to the point where most application programming will be done by instructing a machine in a natural language and getting immediate feedback. When some detail needs to be disambiguated, one would use mathematical logic—and only a few people would ever have to do that.

Of course, I don't expect to be around in 50 years, so you can't hold me accountable if I'm wrong. But the two themes I mentioned—more logic and more interactive feedback—should certainly become much more the norm.

Will more programming be done in syntactically distinct domain-specific languages, or will the tower-of-Babel problem always drive us back to general-purpose languages?
Prophecy is given to fools and children. DSLs are hugely valuable, but designing languages is hard. And as you say, general-purpose languages are such a great tool. So I'd argue that internal DSLs, and general-purpose languages and tools that support making them, are a more promising direction.

Why do you think that, despite all the effort put into parallel programming languages, we still don't seem to have good solutions?
It's the damn humans. The critters simply can't think in parallel. Any number of people who have dedicated their lives to devising schemes for parallel programming are eager to sell you their solution. They tend to say something like, "Now that we can't get sequential speed-ups, we'll have to program in parallel so you should learn my obscure approach to doing it." The problem is that the solutions aren't fit for human consumption, no matter how great the need. This is also aggravated by the hardware designers, who are incentivized to create machines with more

cores with no regard or concept of how to program them.

So you're pessimistic about our ability to think in parallel, but optimistic that we'll design AI that can create software for us?
It's more likely that we'll build electronic brains that program in parallel than it is that we'll genetically reengineer our own brains to do this, so, yes. How might that happen? I don't know for sure, but I would speculate that at some point there will be a huge revolution and programming as we know it will become irrelevant. Until then, we'll see increments that will help but likely won't really resolve the issue.

Are there any emerging languages that you think might point the way to the future?
I think that we may see functional languages finally focus on useful things. Two examples: Clojure is already a bit of a hit, and Lisp is eternal. Elm is a functional language for UIs, which incorporates liveness.

Logic programming is ripe for a renaissance. A good example is Bloom, which is a Datalog like language. I expect to see more of this sort of work over time.

What advice would you give to the next generation of language designers?
Keep it simple. Dead simple. Much simpler than you think is necessary. And don't re-invent the wheel. Yours is likely to be square. Study the great languages of the past—not just the popular ones, but the really good ones.

ABOUT THE AUTHORS

GILAD BRACHA is a software engineer at Google, where he works on Dart. He's also the creator of the Newspeak programming language and previously worked on Strongtalk, as well as coauthored the Java Language Specification. Contact him at gilad@bracha.org.

LAURENCE TRATT is a Reader at King's College London, where he leads the software development team in the Department of Informatics. Contact him via http://tratt.net/laurie.

ADAM WELC is a Principal Member of Technical Staff at Oracle Labs, where he works in the Virtual Machine Research group. Contact him via http://adamwelc.org.