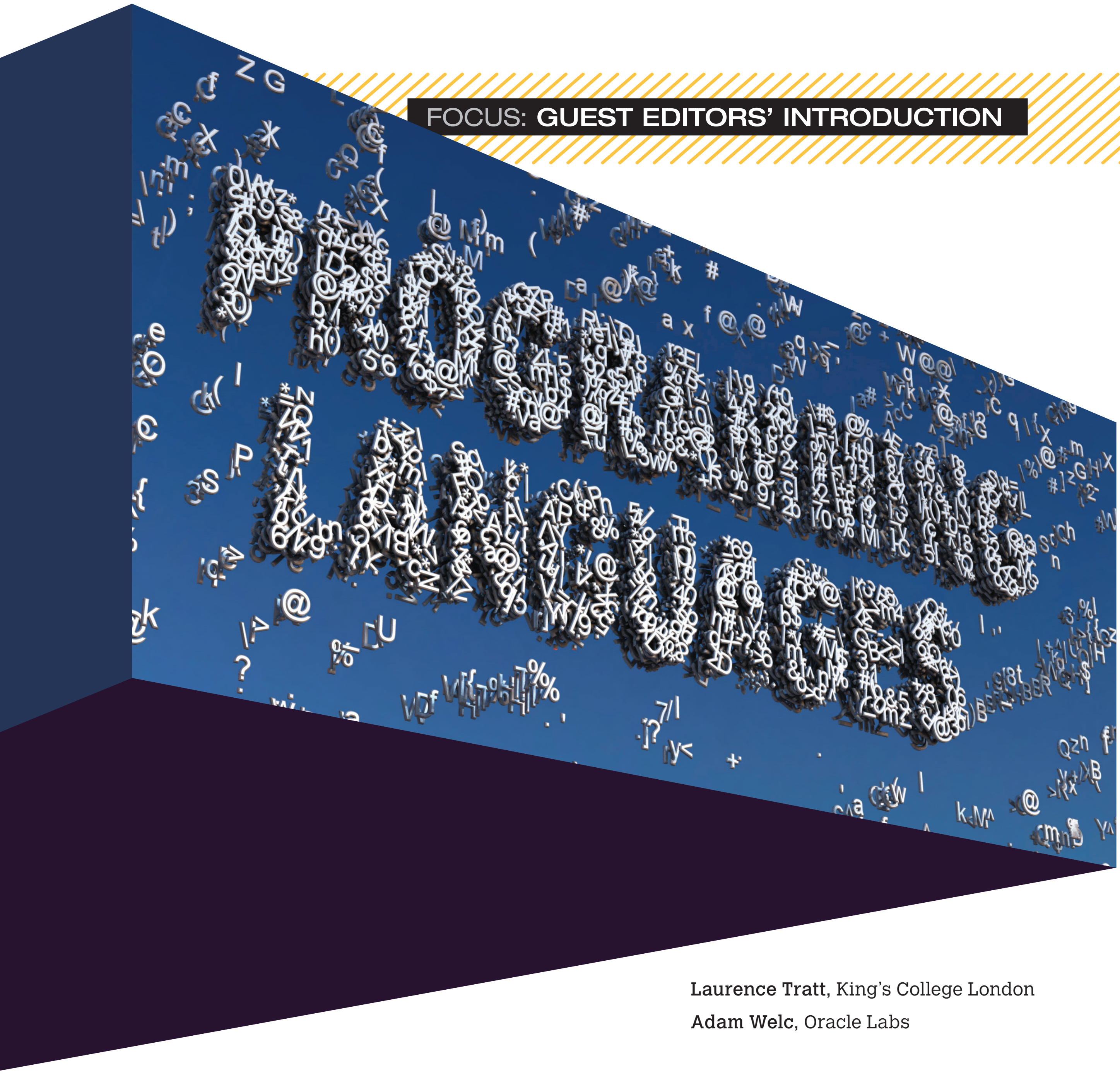

**Laurence Tratt**, King's College London

**Adam Welc**, Oracle Labs

**PROGRAMMING LANGUAGES ARE SO FUNDAMENTAL** to how software is created that we sometimes forget how they came to be. After the first computers were created in the 1940s and '50s, programming languages were somewhat slow to develop. This can be partly attributed to the meager resources of the available computers, which were often too limited for anything other than machine code programming (imagine this: assembly languages were a luxury enjoyed by only a few!).

However, it was far from clear to early computer pioneers what utility programming languages would be, let alone what form they should take. The only languages from the '50s that still have wide(ish) recognition are Fortran (1957) and Lisp (1958). The '60s and '70s were the decades that molded programming languages into the forms we recognize today. ALGOL 60 started the process, spawning children from Pascal (1968) to C (1972); around that same time, Lisp spawned Smalltalk (1972), and innumerable languages since have taken inspiration from this raw DNA.

If we're guilty of forgetting where languages came from, we're also guilty of ignoring where they're going. The notion of "acceptable mainstream programming language" changes slowly. This isn't surprising: changing an organization's programming language can be disruptive. Some staff members can't, or won't, retrain; a new staff is hard to find, and productivity is often low in the changeover period. So, given all these challenges, why would anyone bother changing?

Cast your mind back 20 years. A decent programmer might have been using C for everything. C is still

ABOUT THE AUTHORS

**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.

good for many things—from kernels to resource-efficient utilities—but it's weak in other areas, such as string manipulation in data processing, for example. Trying to operate on C strings is a recipe for buffer overruns and memory leaks. A decent programmer today might write equivalent code in Python in half the time, and the resulting script would be half as short and less prone to long-term bugs. Features that were once unimaginable, or that seemed decadent on slow computers, are now commonplace, from recursive functions to garbage collection to closures. Clever compilation tactics have brought even slow-coach languages up to speed, and programming language paradigms seem ever more fluid as designers look beyond traditional horizons for inspiration.

One of the biggest changes in how we perceive programming languages is that we no longer think of them as just languages. Programmers expect extensive libraries and good performance from a language from its first appearance. We also expect more from the tools that surround a language: good IDEs, debuggers, and profilers are now thought of as essential for a language to be worth trying. It seems likely that expectations such as these will continue to grow. You might think this would increase the barrier to entry for new languages, but there's currently no shortage of new languages.

It often feels that we can never meet user expectations: as fast as we can improve programming languages, we're asked to do more with them. Our success stories are so many that we don't even think of them: today's languages are without a doubt easier to use overall than those of yesteryear for tasks big and small. But sometimes the results are harder to gauge. For example, when the Web took off, users expected programming languages to adapt to the needs of websites; some of the resulting languages made it easy to quickly get something up and running, but made long-term maintenance difficult. Truth be told, we're unrealistic about how easy it is for programming languages to adapt. Most notably, when hardware designers found that they'd run out of easy improvements to sequential program performance, they turned to programming language designers and implementers and said, "You'll have to make your languages and programs concurrent now." We made concurrent languages quickly, but beyond a few examples, we still haven't worked out how to use them very well. Perhaps we one day will—or perhaps we should be realistic that while programming languages have adapted well to many problems, we can't expect them to dig us out of every hole.

One thing we know for certain is that the dominant programming language of today is the legacy language of tomorrow. Sometimes languages are sidelined due to fashion, but changes are generally due to new languages being applicable to a wider or different class of problems than their predecessors. Maybe one day this process will stop, but it seems unlikely that you'd lose money betting on it to continue for a while yet.

We hope this special issue gives a glimpse of what might be coming next in programming languages. We've tried to interpret this liberally because we believe programming languages are as diverse now as at any point in their history, a trend that seems to be increasing. We're fortunate to present you with articles covering a variety of subjects, from different ways of editing programs to different ways of implementing programming languages to implementing different programming language paradigms; from more effective ways of utilizing domain-specific languages to emerging techniques for parallel execution of dynamically typed languages. We're also lucky to have a thought-provoking interview with Gilad Bracha, one of the most interesting programming language designers at work today. This special issue should, at the very least, give you food for thought about what tomorrow might bring.

See www.computer.org/software-multimedia for multimedia content related to this article.