We Need More Than One

Why students need a sophisticated understanding of programming languages

Kathleen Fisher
AT&T Labs Research
kfisher@research.att.com

> Some believed we lacked the programming language to describe your perfect world.
>
> — Agent Smith, *The Matrix*

In the fall of 1995, when I was almost finished with my Ph.D., another graduate student asked what I was doing for my thesis. When I explained I was working in programming languages, he was taken aback, asking why anyone needed any language besides C++. His reaction was not unusual. At the time, there was a wide-spread sense that C++ was the last language, and students need only master it and be done with studying programming languages for the rest of their careers. The following year, JAVA burst on the scene, and shortly thereafter, new voices claimed that JAVA was the only language that a student really needed. This pattern echoed earlier claims made about FORTRAN, and COBOL, and C.

Why the desire, reflected in those voices, to have only one language? Perhaps because, like natural language, it is hard to learn a new programming language, and people would prefer to have everyone use the language they themselves are already comfortable with.

But we will never have just one programming language.

Like natural languages, programming languages affect how we think. In particular, how we think when we communicate with computers. Each language comes with a natural domain of discourse. Ideas within or close to that domain are easy to express, while ideas further afield are harder. For example, the machine model that C provides matches the underlying hardware of today's machines very closely, making it a good language in which to write low-level code such as device drivers. In contrast, the pattern-matching, parametric polymorphism, and higher-order functions of ML make it a good language for writing compilers. The abstractions provided by object-oriented languages are well suited to programming user interfaces. Apparently the machines in *The Matrix* felt they could not build a world without suffering because no language existed that suited such a task.

The domain of a language is determined by the model of computation that it provides its programmers. It builds this model through a combination of the constructs it provides and the runtime services it makes available to programmers. In C, the model closely reflects the underlying hardware, and the language provides few runtime services. In SCHEME, the model is the lambda calculus, and the language provides runtime services to convert the underlying hardware into a system that behaves, to the programmer, like the lambda calculus.

A language is not just defined by what it includes, but by what it leaves out as well. The underlying model associated with SQL is that of relational algebra. The language leaves out transitive closure to permit more efficient implementations. The CRYPTOL language is designed to express cryptographic algorithms clearly and concisely [1]. Its designers left out general recursion so that they could always compile CRYPTOL programs directly to hardware to produce highly efficient encryption, sacrificing the ability to write certain kinds of programs. C provides direct control over pointers and memory allocation, which gives the programmer enormous freedom. But enormous responsibility as well! The epidemic of security vulnerabilities caused by buffer overflows in C code suggest that many programmers are not up to the task of managing that responsibility. Languages like JAVA, ML, and HASKELL don't let programmers do low-level pointer manipulation and insist upon garbage collection, reducing the freedom allowed to programmers. But programs written in such languages are guaranteed to be memory safe, avoiding a huge class of safety-critical bugs. Such programs hide the underlying machine though, and introduce run-time overhead, which is unacceptable for certain kinds of applications.

As these examples illustrate, we will never have just one language because *a single language cannot be well-suited to all programming tasks*. It may be possible to solve a problem in an ill-suited language, but it is harder. Trying to do so is akin to hammering with a wrench: it may get the nail in, but it would be better to use a hammer! Programmers who

choose a well-suited language will accomplish their tasks faster and more cheaply and will produce more maintainable code. This reality has led to the proliferation of languages that we see today. C++ and JAVA, yes. But also PERL, PHP, PYTHON, RUBY, VISUAL BASIC, XQUERY, XSLT, *etc.* Just within my center at AT&T, which includes roughly a hundred people, my colleagues program in C, PERL, C++, JAVA, AWK, LATEX, CYCLONE, PHP, XSLT, BASH, R, MAKE, OCAML, and SML. And those are just the languages I know they are using off the top of my head.

Not only is the set of languages in use today large, but it is not fixed. The collection of languages evolves over time in response to changing circumstances. For example, garbage collection was too expensive for the benefits it provided when it was first invented because CPUs were slow, memory was expensive, and programs were relatively simple. But as the field evolved, CPUs sped up, memory became much cheaper, and programs became extraordinarily complex. And so garbage collection became not just viable, but necessary for a large collection of tasks. Other changes come about because programming language researchers figure out better ways to do things, and those ideas eventually make it into mainstream languages. Parametric polymorphism is an example of a concept that has recently made such a jump. Multi-core machines are likely to stir things up still further, as finding easier ways to leverage the power of parallelism will become increasingly important.

Consequently, it is unreasonable to expect that teaching students to program well in a single language is sufficient to prepare them for all the programming tasks they are likely to face during their careers. One language can embody only one domain, and such limited exposure will not prepare students for the many other domains they will encounter. And encounter them they will, because the languages will change, the problems will change, and the underlying assumptions will change.

So, we need to give students to ability to learn new languages, which means we need to teach them more than one language. Just as learning a foreign language is difficult, so is learning a programming language. But the second foreign language is easier than the first, and the third is easier still. So too with programming languages. As a computer scientist becomes more comfortable with a range of languages, learning the next becomes easier. When students begin to see the range of possible domains and the variety of programming language features, they develop the ability to learn new languages independently and much more quickly. Of course, giving students this ability is not a simple matter. Like learning a foreign language, simply reading a text book is not sufficient. Students need to actually use a language to be able to really understand it, and such use takes time.

As part of learning about programming languages, it is important that students learn to identify the domain supported by each language, so that when faced with a programming task, they know which languages are well-suited to solving the problem. Considerations include intrinsic features of the language such as the abstractions it provides and the resource assumptions it makes as well as extrinsic features such as the availability of useful tools and libraries, the availability of knowledgeable programmers, and the need to interoperate with existing systems and code bases.

The languages I have mentioned so far are mostly examples of general purpose languages, which strive to make it possible to express many ideas reasonably well. There are also domain-specific languages, which attempt to make it very easy to solve a narrower range of problems. Such languages are somewhat akin to the specialized vocabularies that exist in natural language, such as legal or medical jargon. Domain-specific languages include widely-used examples, such as SQL for querying relational databases, MAKE for expressing how to build programs from sources, YACC for describing grammars, POSTSCRIPT for describing how to print documents, LATEX for typesetting, XSLT for transforming XML and XQUERY for querying it, formulae in Excel spreadsheets, *etc.* But there are many more domain-specific languages for even more specialized purposes.

Such languages can be powerful tools for solving problems for a number of reasons. They can dramatically increase productivity while expanding the range of people who can program by providing tailored abstractions that facilitate thinking about programming tasks in the domain of the language. When one of my colleagues saw for the first time a signature program for tracking usage profiles written in the domain-specific language HANCOCK [5], he was amazed, commenting "You can talk about what is really going on!" — in contrast to the earlier versions of the program, written in C, where the meaning of the program was obscured by all the code necessary to implement it. Programs written in domain-specific languages tend to be significantly shorter than corresponding code in general-purpose languages because the domain-specific language already knows about the kind of problem being solved and so can generate or provide in the runtime system a lot more of the code. Also, by leaving unnecessary things out of the language, the compiler can do a better job of optimization or code generation, as we saw earlier with SQL and CRYPTOL. Finally, if the domain-specific language is declarative, meaning programs in the language describe the problem rather than telling the computer how to solve it, the system can generate multiple artifacts from a single description. In PADS [7], a language for describing ad hoc data formats, the system generates not just a parser for the data, but also a printer. And a statistical analyzer. And a format translator, *etc.* As another example, the ESP language [8] for coding device drivers generates not just code to manage the device, but also input to the SPIN model checker to verify that the implementation is correct.

Because of these advantages, sometimes the best way to solve a programming task is to first invent a new lan-

guage in which to solve the problem. It is this reality that leads to the many domain-specific languages described in van Deursen, Klint, and Visser's annotated bibliography of domain-specific languages [9] and the USENIX conferences on Domain-Specific Languages. Within my center at AT&T, people have designed languages to describe graphs for visual layout [2], process massive transaction streams to build time-varying profiles [5], describe ad hoc data formats [7], query high-volume data streams [6], describe how to load data into a relational database, construct software for monitoring web hosting infrastructure [3], and build user interfaces [4].

The ubiquity of domain-specific languages means that students are even more likely to need to know more than a single language or two to have successful careers as computer scientists. The utility of domain-specific languages as a technique for solving particular problems means that many more computer scientists actually build languages than might be obvious from listing high-visibility general-purpose languages.

In sum, it is important for computer scientists to know how to wield programming languages effectively, as programming languages are the most powerful and flexible mechanism by which people communicate with computers. To acquire this skill, it is important that students study and use a range of programming languages, acquiring in the process the ability to assess the strengths and weaknesses of those languages. With this knowledge, students can assess programming tasks and choose the existing languages that are best suited to the tasks facing them. Or they can conclude that the best approach is to design a new, domain-specific language.

As computer science evolves, new languages will be invented and released into the wild regularly. Historically, this pattern has repeated for as long as computers have existed, and there is no reason to think it will change anytime soon. In fact, the expanding scope of computer science, the introduction of multi-cores, the ubiquity of the Internet, and the proliferation of diverse hardware platforms all suggest that new languages will appear and thrive with increasing frequency. Consequently, the challenges of programming computers will not be solved by one language, but rather by many, and so it behooves us to train our students to thrive with the many rather than pining for the one.

Acknowledgments

John Launchbury provided inspiration for this essay and useful comments during its drafting.

References

[1] Cryptol. `http://www.cryptol.net`.

[2] Dot graph description language. `http://graphviz.org`.

[3] Vizgems. `http://www.research.att.com/areas/visualization/projects_software/visualdiscovery.php`.

[4] Yoix scripting language. `http://www.yoix.org`.

[5] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for analyzing transactional data streams. *TOPLAS*, 26(2):301–338, 2004.

[6] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*, 2002.

[7] K. Fisher and R. Gruber. PADS: A domain specific language for processing ad hoc data. In *PLDI*, pages 295–304, June 2005.

[8] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A language for programmable devices. In *PLDI*, pages 309–320, New York, NY, USA, 2001. ACM.

[9] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. `http://homepages.cwi.nl/~arie/papers/dslbib`.