The Semicolon Wars

Brian Hayes

If you want to be a thoroughgoing world traveler, you need to learn 6,912 ways to say "Where is the toilet, please?" That's the number of languages known to be spoken by the peoples of planet Earth, according to Ethnologue.com.

If you want to be the complete polyglot programmer, you also have quite a challenge ahead of you, learning all the ways to say:

```
printf("hello, world\n");
```

(This one is in C.) A catalog maintained by Bill Kinnersley of the University of Kansas lists about 2,500 programming languages. Another survey, compiled by Diarmuid Piggott, puts the total even higher, at more than 8,500. And keep in mind that whereas human languages have had millennia to evolve and diversify, all the computer languages have sprung up in just 50 years. Even by the more-conservative standards of the Kinnersley count, that means we've been inventing one language a week, on average, ever since Fortran.

For ethnologists, linguistic diversity is a cultural resource to be nurtured and preserved, much like biodiversity. All human languages are valuable; the more the better. That attitude of detached reverence is harder to sustain when it comes to computer languages, which are products of design or engineering rather than evolution. The creators of a new programming language are not just adding variety for its own sake; they are trying to make something demonstrably better. But the very fact that the proliferation of languages goes on and on argues that we still haven't gotten it right. We still don't know the best notation—or even a good-enough notation—for expressing an algorithm or defining a data structure.

Every programmer knows there is one true programming language. A new one every week

There are programmers of my acquaintance who will dispute that last statement. I expect to hear from them. They will argue—zealously, ardently, vehemently—that we have indeed found the right programming language, and for me to claim otherwise is willful ignorance. The one true language may not yet be perfect, they'll concede, but it's built on a sound foundation and solves the main problems, and now we should all work together to refine and improve it. The catch, of course, is that each of these friends will favor a different language. It's Lisp, says one. No, it's Python. It's Ruby. It's Java, C#, Lua, Haskell, Prolog, Curl.

Sadly, linguistic diversity has a dark side. Communities separated by differences of language don't always get along peaceably; the term "Balkanization" comes to mind. And, like weary, war-torn countries, the computing professions have had their share of sectarian strife and schism. As far as I know, the conflicts have never come to actual bloodshed, but harsh words have been exchanged (in many languages).

The Endian Wars

In 1726 Jonathan Swift told of a dispute between the Little-Endians of Lilliput and the Big-Endians of Blefuscu; 41,000 perished in a war fought to decide which end of a boiled egg to crack. This famous tempest in an egg cup was replayed 250 years later by designers of computer hardware and communications protocols. When a block of data is stored or transmitted, either the least-significant bit or the most-significant bit can go first. Which way is better? It hardly matters, although life would be easier if everyone made the same choice. But that's *not* what has happened, and so quite a lot of hardware and software is needed just to swap ends at boundaries between systems.

This modern echo of Swift's Endian wars was first pointed out by Danny Cohen of the University of Southern California in a brilliant 1980 memo, "On holy wars and a plea for peace." The memo, subsequently published in *Computer*, was widely read and admired; the plea for peace was ignored.

Another feud—largely forgotten, I think, but never settled by truce or treaty—focused on the semicolon. In Algol and Pascal, program statements have to be separated by semicolons. For example, in `x:=0; y:=x+1; z:=2` the semicolons tell the compiler where one statement ends and the next begins. C programs are also peppered with semicolons, but in C they are statement *terminators*, not separators. What's the difference? C needs a semicolon after the last statement, but Pascal doesn't. This discrepancy was one of the gripes cited by Brian W. Kernighan of AT&T Bell Labs in a 1981 diatribe, "Why Pascal is not my favorite programming language." Although Kernighan's paper was never published, it circulated widely in *samizdat*, and in retrospect it can be seen as the beginning of the end of Pascal as a serious programming tool.

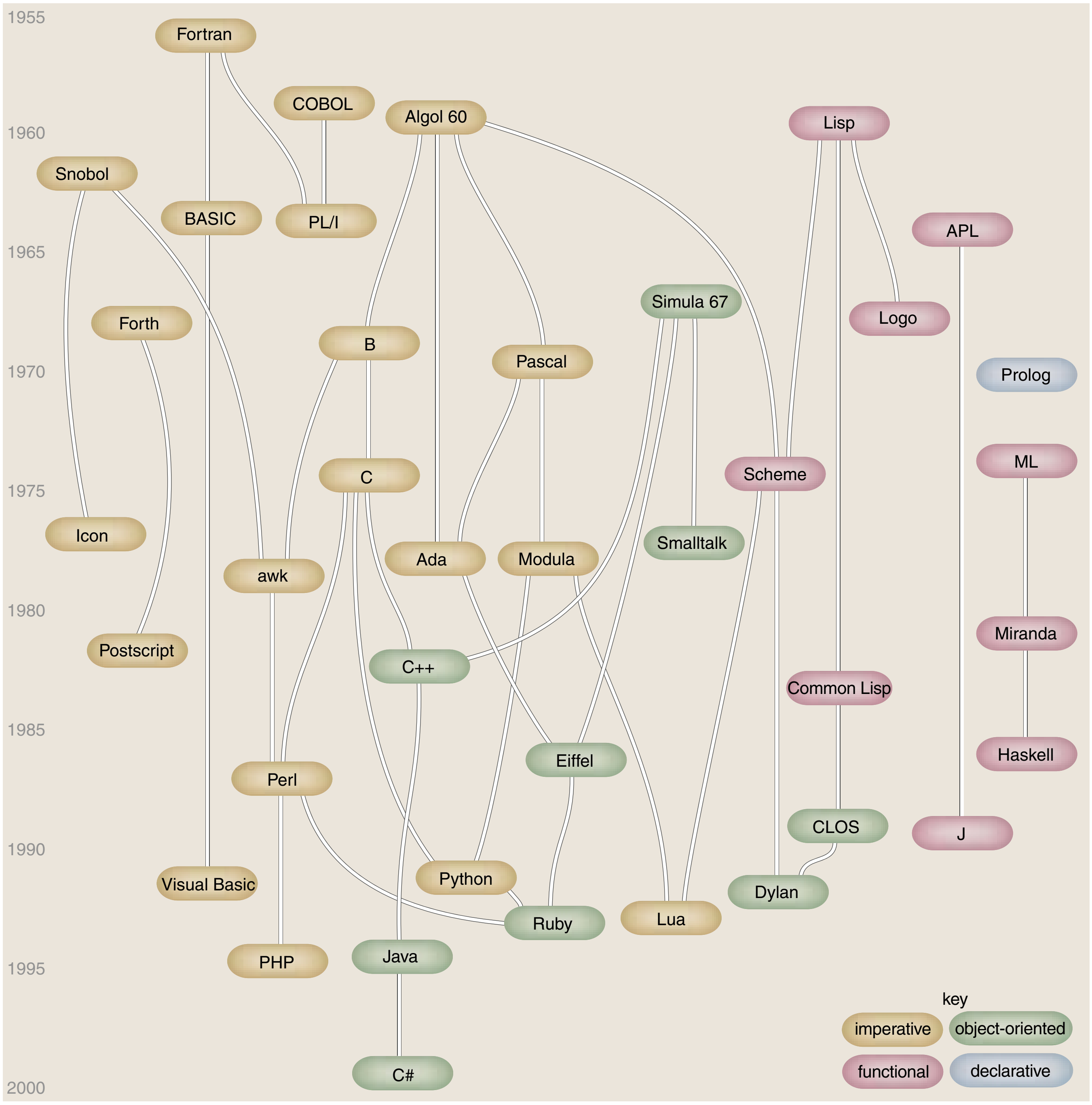
Still another perennially contentious issue is how to count. This one brings out the snarling dogmatism in the meekest programmer. Suppose we have a list of three items. Do we number them 1, 2, 3, or should it be 0, 1, 2?

Brian Hayes is Senior Writer for American Scientist. *Additional material related to the "Computing Science" column appears in Hayes's weblog at http://bit-player.org. Address: 211 Dacian Avenue, Durham, NC 27701. Internet: bhayes@amsci.org*

Everyone in computerdom knows the answer to that question, and knows it as an eternal truth held with the deepest, visceral conviction. Only one of the alternatives is logically tenable. But which is it? Consider the Java expression `Date(2006,1,1);` what calendar date do you suppose that specifies? The answer is February 1, 3906. In Java we count months starting with 0, days starting with 1, and years starting with 1,900.

Even the parts of a program that aren't really part of the program can provoke discord. "Comments" are meant for the human reader and have to be marked in some way so that the computer will ignore them. You might think it would be easy to choose some marker that could be reserved for this purpose in all languages. But a compendium of programming-language syntax compiled by Pascal Rigaux—a marvelous resource, by the way—lists some 39 incompatible ways to designate comments: `#` in awk, `\` in Forth, `(*...*)` in Pascal, `/*...*/` in C, and so on. There's also a running debate over whether comments should be "nestable"—whether it's permissible to have comments inside comments.

Then there's the CamelCase controversy. Most programming languages insist that names of things—variables, procedures, etc.—be single words, without spaces inside them;



A chronology of selected programming languages shows a few of the links between them. The diagram is not a genealogy but merely indicates major patterns of influence. The classification of languages as imperative, functional, object-oriented or declarative is also approximate; only a few "pure" languages belong exclusively to one of these categories. The chronology is based in part on time lines constructed by Éric Lévénez and by Pascal Rigaux and on information from the Association for Computing Machinery History of Programming Languages conferences.

but runningthewordstogether makes them unreadable. Hence CamelCase, with humps in the middle (also known as BumpyCaps and NerdCaps; but sTuDLy CaPs are something else). To tell the truth, I don't think there's much actual controversy about the use of CamelCase, but the name has occasioned lively and erudite discussions, revisiting old questions about *Camelus dromedarius* and *C. bactrius*, and offering glimpses of such further refinements as sulkingCamelCase (with a droopy head).

Organizing Babel

I mock the pettiness of these squabbles—and I believe some of them deserve mocking—and yet I don't want to give the impression that only cosmetic issues are in dispute, or that programming languages are really all alike under the skin. On the contrary, what's most fascinating about programming languages is how dramatically they differ. I would argue that the distance between C and Lisp, for example, is greater than that between any pair of human languages.

Noam Chomsky asserts that all human languages have the same "deep structure," which may even be hardwired into the brain. In computer languages, too, certain features seem to be universal. Almost all programming languages are built on the same kind of grammatical scaffold, called a context-free grammar. At the semantic level, almost all programming languages have the same computational power: If you can compute something in one language, you can get the same answer in any other, given enough effort. But this formal equivalence is misleading. Raw computational power is not what people care about in a programming language; the real criterion is how readily you can express your ideas.

In the 1930s the linguists Edward Sapir and Benjamin Lee Whorf argued that what you can think is conditioned by what language you think in. For natural languages, the Sapir-Whorf hypothesis has met with much skepticism, but for computer languages the idea seems more plausible. Different categories of programming languages elicit quite different modes of thinking and problem solving.

Programming languages are usually classified in four families. Imperative languages are built on commands: *do this, do that, do the next thing*. The commands act on stored data, modifying the overall state of the system. The imperative approach was the default in most early programming languages, including Fortran, COBOL and Algol.

A functional language is modeled on the idea of a mathematical function, such as $f(x) = x^2$. The function is a black box that accepts arguments as input and returns values as output. A key point is that the calculation depends *only* on the arguments and affects *only* the value; there are no extraneous side effects. This property makes it easier to reason about functional programs, since there's no need to keep track of the state of the entire machine. Functional programming began with Lisp, although most versions of Lisp allow other styles of programming as well. John Backus, the lead developer of Fortran and a contributor to Algol, later became an advocate of functional languages. Several "pure" functional languages have emerged since then, including ML, Miranda and Haskell.

In object-oriented programming languages the root idea is to bind together imperative commands and the data they act on, forming encapsulated objects. Instead of defining a procedure to manipulate a data structure, one "teaches" the data structure how to carry out operations on itself. Most object-oriented languages also have some notion of inheritance, whereby an object is born already knowing default behaviors. The object-oriented languages trace their heritage back to SIMULA 67, but they began to attract attention only in the 1980s with Smalltalk. In a curious turn of events, object-oriented principles became wildly popular, but the result was not the widespread adoption of Smalltalk; instead, object-oriented features were bolted onto other languages. From C, for example, came C++ and Objective C and eventually C#; Java is also in this family. Object-oriented notions are now so deeply ingrained that they influence almost every new language.

The languages of the fourth category are variously known as logic, relational or declarative languages. What they have in common is the idea of programming not by spelling out step-by-step algorithms but by stating facts or relations. The best-known exemplar of this technique is Prolog, which relies on an method called unification to make deductions from stated facts. Related concepts also turn up in less-exotic areas such as database-query languages and spreadsheets.

These four categories suggest the breadth of the programming-language spectrum, but there are further variations across many other dimensions. At the most superficial level, the various languages simply *look* different. C is terse, COBOL quite verbose. Lisp is full of parentheses. Perl, said some wag, looks like Snoopy swearing: `@&$^^#@!`.

Languages can also be distinguished as "low-level" or "high-level." The low-level ones allow more-direct access to aspects of the underlying hardware, such as addresses in memory or input and output devices. High-level languages provide an insulating layer of abstraction.

A generation of languages created in the 1970s emphasized "structured programming"—otherwise known as bondage and discipline. Pascal is in this group: It enforces strict rules about types of data and the flow of control through a program. The reaction against such constraints produced "hacker-friendly" languages, including C.

Languages also differ in their intended audience or area of application. Fortran began as a language for scientific computing, COBOL for business. Quite a few interesting languages were designed for teaching or for children. BASIC, Pascal and Smalltalk are all in this class, and so is Logo. (All of them have had to struggle to be taken seriously as languages for grownups.)

Zealotry

The remarkably wide range of programming languages would seem to offer something for everyone. We could celebrate diversity. We could let a thousand flowers bloom. What actually happens, more often, is that we launch a crusade to convert the infidels—or else exterminate them.

In 1975 Edsger W. Dijkstra, a major figure in the structured-programming movement, wrote a memo titled "How Do We Tell Truths that Might Hurt?" The "truths" were mostly Dijkstra's opinions of programming languages; how he told them was very bluntly. Fortran is "an infantile disorder," PL/I "a fatal disease," APL "a mistake, carried through to perfection." Students exposed to BASIC "are mentally mutilated beyond hope of regeneration," he said. "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense." When the memo was published a few years later, defenders of COBOL and BASIC replied in kind, although none of them were quite able to match Dijkstra's acid rhetoric.

In fairness, I should note that most disputes over programming languages are neither as vicious nor as humorless as the affair of Dijkstra's "truths." Today's missionaries take an upbeat approach, spending more time in promoting their own religion and less in dissing the other person's beliefs. The message is no longer "You'll burn in hell if you write C." It's "Look what a paradise Python offers you!" (I think maybe I liked the old sermons better.)

Much of this proselytizing is done with the best of intentions. When you have found a tool that seems artful and elegant, you want to spread the good news. This is a generous impulse. But there is also self-interest at work. For programming language *P* to prosper, it must have a community of users—people who write *P* programs and buy books about *P*, who teach *P* to students, who agitate to get *P* supported on new platforms, who hire *P* programmers. Every convert to *P* improves *P*'s chance of survival; if the convert comes from the rival language *Q*, so much the better.

Quarrels over notation are hardly unique to the world of computing. In mathematics there was the famous impasse between the Leibnizian *dx/dt* and the Newtonian $\dot{x}$ (known as the war between deity and dotage). Chemists wrangle about how to name molecules. Even chess players have fought over how to record moves. But the situation in computer science is of a different order. Calculus never had 2,500 ways to write a derivative.

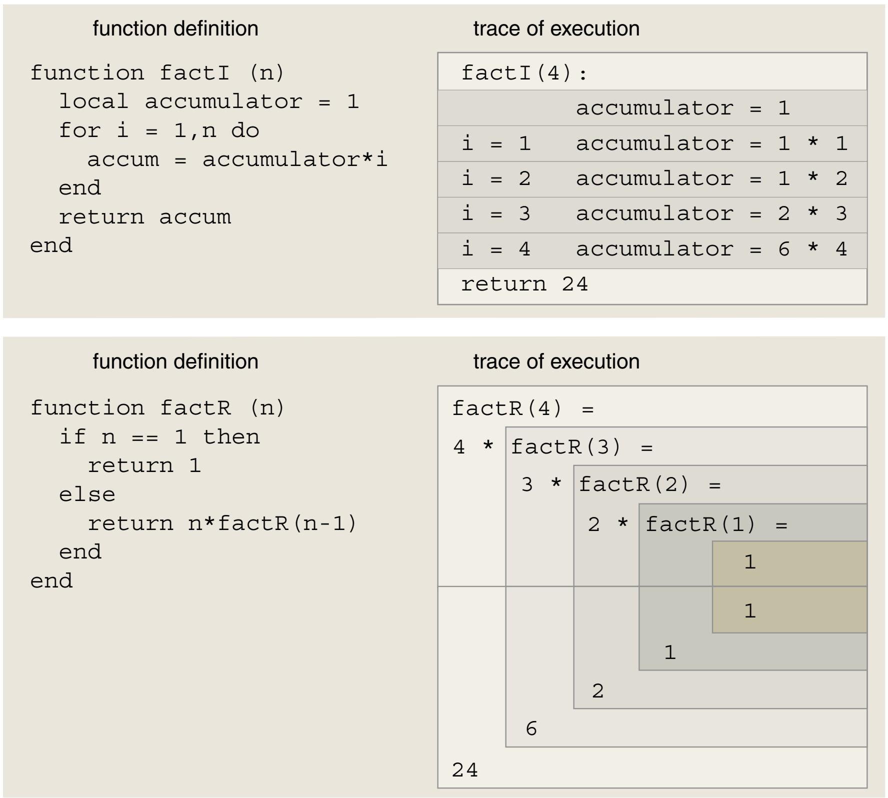
Over the years, the cacophony of programming languages has repeatedly been cited as a threat to further progress in computing. The usual response has been—what else?—to propose yet another programming language. "If we could all just get together and agree on one last, greatest language...." In the 1960s this was the ambition of PL/I, the language that Dijkstra called a fatal disease. Later, Ada was to reunify all of computing—by mandate of the U.S. Department of Defense. A decade ago Java was the shining hope, promoted with the slogan "Write once, run anywhere."

A few programming languages—most notably Fortran and Lisp—seem to be all but immortal; the rest are like waves washing ashore and then draining into the sand. Riding the crest of the latest wave are the scripting languages, especially Python and Ruby. Their origins are humble. The idea of scripting began with batch-command languages, used as "glue" to bind together other programs, and with extension languages, meant to be embedded inside programs. But scripting languages have grown up into general-purpose programming languages. They are popular now for writing Internet applications. Python also has a following in scientific computing.

The Internet has brought another encouraging development: a new multilingualism. Merely managing a Web site these days requires fluency in half a dozen programming and data-formatting languages. There's HTML (Hypertext Markup Language) for the basic structure of the pages and CSS (Cascading Style Sheets) for details of presentation, as well as JavaScript for annoyances such as pop-up windows. On the server side, content is likely to be encoded in some form of XML (Extensible Markup Language) and accessed through a database query language such SQL. All the pieces are held together by a scripting language, which might be PHP, Perl, Python or Ruby. (Of course this situation cries out for yet another language to unify or replace all the others. At least two languages are already contending for this role—Curl and Links.)

Lisping in Numbers

My plea for peace in programmerhood would carry more weight if I could present myself as an impartial arbiter, with no stake in the outcome of the language race. But it's time to confess. I too have a favorite programming language, which I cling to like a child



```
function factI (n)
  local accumulator = 1
  for i = 1,n do
    accum = accumulator*i
  end
  return accum
end
```

```
factI(4):
        accumulator = 1
i = 1   accumulator = 1 * 1
i = 2   accumulator = 1 * 2
i = 3   accumulator = 2 * 3
i = 4   accumulator = 6 * 4
return 24
```

```
function factR (n)
  if n == 1 then
    return 1
  else
    return n*factR(n-1)
  end
end
```

```
factR(4) =
4 * factR(3) =
     3 * factR(2) =
          2 * factR(1) =
                    1
                    1
              1
         2
     6
24
```

The imperative and the functional styles of programming correspond to different styles of thought, even when applied to the same problem. The two programs shown here both calculate the factorial of a number *n*: the product of all the integers from 1 through *n*. The imperative (or iterative) procedure in the upper panel works by repeatedly overwriting the value of a variable named *accumulator*. A program in the functional style *(lower panel)* relies instead on the mechanism called recursion. The factorial of *n* is defined as *n* multiplied by the factorial of *n*–1. Thus *factR*(4) is equal to 4 × *factR*(3), and *factR*(3) in turn is equal to 3 × *factR*(2). The sequence of nested function calls continues until eventually *factR*(1) returns a simple value of 1, then the product is calculated "from the inside out." Although there are languages specialized for either imperative or functional programming, both of these programs are written in the same language: Lua, developed in the 1990s at the Pontifical Catholic University of Rio de Janeiro.

with a bedraggled teddy bear. Don't you dare try to take away my Lisp!

Without engaging in missionary zealotry of my own, it's hard to explain my fondness for Lisp. I'll just say it's a simple-minded language with one trick that it does very well. Every Lisp expression is a list, evaluated by reading the first element of the list as the name of a function and the remaining elements as the arguments to the function. For example, `(/ (+ 3 5) 2)` is a program for dividing `(+ 3 5)` by `2`, where `(+ 3 5)` is a subprogram for adding `3` and `5`. The value of the entire expression is `4`. The syntax is brutally simple, even primitive, but that's its strength. Lisp evangelists always note that data and programs are represented in the same way, which makes it easy to write programs that manipulate other programs. That's true, but what appeals to *me* is just the uniformity of the notation. Everything is done the same way, and so there's not much to remember. (One thing I *won't* mention is the profusion of parentheses (which annoy some people). (What the world needs (I think) is not (a Lisp (with fewer parentheses)) but (an English (with more.))))

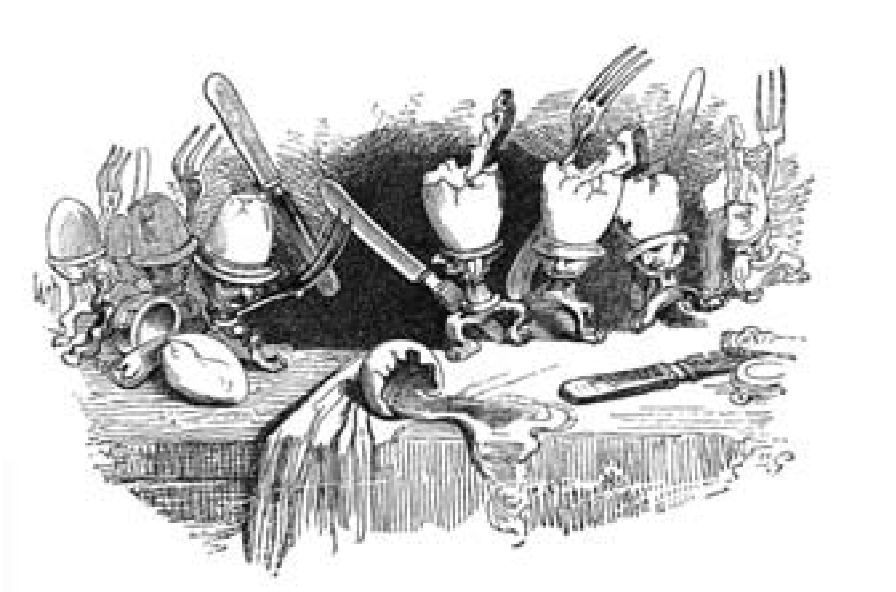
In the chronology of programming languages, Lisp comes from the very dawn of time. It was conceived nearly 50 years ago by John McCarthy, now of Stanford University. My own acquaintance with the language goes back 25 years. To persist in using such an antique idiom seems peculiar and affected, like speaking in Miltonic verses. There's something stubborn and curmudgeonly about it. It looks like a rebuke to all the effort expended on programming-language design since the 1950s. Do I really mean to suggest that not one of the 2,500 newer languages has been able to improve on Lisp?

No, I don't. And of course the Lisp I speak today is not the language McCarthy introduced 50 years ago. It has been augmented, overhauled, updated, split into multiple dialects, then reassembled in a standard called Common Lisp. (Still, the parts of the language I like best were there at the beginning and have changed little.)

An International Lisp Conference was held at Stanford a year ago. This was a gathering of the faithful, and naturally there was talk about how to bring enlightenment to the rest of the world. It was also an occasion showing that even advocates of the *same* language are quite capable of arguing among themselves deep into the night.

At the end of the final session, John McCarthy rose to speak. He looked around at his audience and remarked, "If someone set off a bomb in this room, it would wipe out half of the worldwide Lisp community. That might not be a bad thing for Lisp, because it would have to be reinvented." His meaning, as I understood it, was partly that the Common Lisp standard had stifled innovation. But he went on to say that if he could go all the way back to the beginning, there were things he would do differently. Even the maker of the language did not see it as beyond improvement. I found McCarthy's candor refreshing, but I also had the thought: No, no, don't tamper with it. I like it just the way it is.

I do believe there are real differences among programming languages—better ones and worse ones—and I rank Lisp among the better. When you get to the bottom of it, however, I write programs in Lisp for the same reason I write prose in English—not because it's the best language, but because it's the language I know best.


Carnage at the breakfast table is the outcome of the war between the Little-Endians and the Big-Endians. (The engraving, from the 1838 edition of *Gulliver's Travels*, is by J. J. Grandville.)

Bibliography

Association of Lisp Users. International Lisp Conference, Stanford University, June 19–22, 2005. http://international-lisp-conference.org/2005/index.html

Bergin, Thomas J., and Richard G. Gibson, Jr. (editors). 1996. *History of Programming Languages II*. New York: ACM Press.

Cohen, Danny. 1980. On holy wars and a plea for peace. Internet Experiment Note IEN-137. http://www.ietf.org/rfc/ien/ien137.txt. Reprinted in *Computer* 14(10):48–54.

Cooper, Ezra, Sam Lindley, Philip Wadler and Jeremy Yallop. 2006. Links: Web programming without tiers. http://groups.inf.ed.ac.uk/links/papers/links-icfp06/links-icfp06.pdf

Dijkstra, Edsger W. 1975. How do we tell truths that might hurt? Memo EWD498. http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD498.html Reprinted in *ACM SIGPLAN Notices* 17(5):13–15; also reprinted in Dijkstra, Edsger W. 1982. *Selected Writings on Computing: A Personal Perspective*, pp. 129–131. New York: Springer-Verlag.

Gordon, Raymond G., Jr. (editor). 2005. *Ethnologue: Languages of the World*. Fifteenth edition. Dallas, Texas: SIL International. Online version: http://www.ethnologue.com/

Ierusalimschy, Roberto, Luiz Henrique de Figueiredo and Waldemar Celes. 2006. The Evolution of Lua. http://www.tecgraf.puc-rio.br/~lhf/ftp/doc/hopl.pdf

Kernighan, Brian W. 1981 (unpublished). Why Pascal is not my favorite programming language. Computer Science Technical Report 100, AT&T Bell Labs. http://www.lysator.liu.se/c/bwk-on-pascal.html

Kinnersley, Bill. The language list. http://people.ku.edu/~nkinners/LangList/Extras/langlist.htm

Knuth, Donald E. 2003. *Selected Papers on Computer Languages*. Stanford, Calif.: CSLI Publications.

Lévénez, Éric. 2006. Computer languages history. http://www.levenez.com/lang/

Patterson, Meredith L.. Weblog. Programming languages and their relationship styles. http://maradydd.livejournal.com/293666.html

Piggott, Diarmuid. 1995–2006. HOPL: An interactive roster of programming languages. http://hopl.murdoch.edu.au/

Rigaux, Pascal. Undated. Syntax across languages. http://merd.sourceforge.net/pixel/language-study/syntax-across-languages/

Wexelblat, Richard L. (editor). 1978. *History of Programming Languages*. New York: Academic Press.

Wiki. Camel Case. http://c2.com/cgi/wiki?CamelCase

Wikipedia. CamelCase. http://en.wikipedia.org/wiki/CamelCase