

Alan J. Perlis - Epigrams on Programming

Comments on:

Perlis, Alan J. (1982). Epigrams on Programming. *ACM SIGPLAN Notices* 17(9), September 1982, pp. 7-13.

1. One man's constant is another man's variable.

This to me makes sense because very often in a program, one may declare something as a constant and then may want to change it. One example, being if you fix the size of an array and then may need to expand the array at some point in the program, this needing to vary it and use it as a variable.

I love the "declare something as a constant and then change it". But you are very correct in your analysis.

2. Functions delay binding: data structures induce binding. Moral: structure data late in the programming process.

functions and interfaces provide an abstraction that allows a programmer to write code that isn't dependent on the implementation details of a particular data structure. since interactions between data structures seems to be what generates a lot of code, minimizing the amount of code that has to be changed when a data structure changes seems like a good policy.

One could also argue that ADTs also delay binding, so it is safe to use data structures early, as long as access to them is protected by interfaces.

3. Syntactic sugar causes cancer of the semi-colons.

We read a paper that warned us about syntactic sugar. Hoare also warns against too much syntactic sugar in his hints on programming language design. I forget exactly what he says though and I can't find the paper.

I'm not sure what to make of this comment.

8. A programming language is low level when its programs require attention to the irrelevant.

This again is an interesting comment because in a low level language there is little abstraction between the language and machine language and thus little attention to detail.

I'm not sure why you suggest that there is little attention to detail in a low-level language. Perlis's claim (and most people's experience), is that programming in a low-level language requires significant attention to details. And, unfortunately, most of those details are irrelevant to the overall design of the program.

12. Recursion is the root of computation since it trades description for time.

I'm not sure I understand what he means by description in this context. Recursion does take longer to process through unless you use tail recursion, but what does that have to do with description?

The recursive description of a process is generally more concise (and less prone to error) than the corresponding iterative definition.

15. Everything should be built top-down, except the first time.

I somewhat buy this, but I still think a mixed approach should always be used.

This epigram is a fairly strong attack on top-down programming.

When one usually writes a program it is written top-down, when we initially think about the program, we need not follow the top-down approach, especially when we have loops and cases where we may need to break the top down approach.

I'm not sure how loops and cases break the top-down approach. Perlis is noting that, in practice, it is impossible to think purely top-down. In particular, and as the prior commentor suggested, we sometimes need to think about and even implement some libraries before we implement the clients.

17. If a listener nods his head when you are explaining your program, wake him up.

One of the things I picked up in this class is the need to be patient and disciplined when reading through pieces of code in articles. Going through code is not a passive activity, and as we have seen with code in scheme, a few lines of code may require great deal of understanding.

I think you've caught the key idea - one does not understand code passively.

18. A program without a loop and a structured variable isn't worth writing.

Isn't one of the idea of functional programming to avoid using loops? Is this epigram targeting that idea?

A recursive loop is still a loop.

19. A language that doesn't affect the way you think about programming, is not worth knowing.

I like this one a lot, but it is hard to imagine a language that doesn't affect the way you think about programming. Maybe it should say "...positively affect the way you think about programming..." or something to that extent.

Well, it depends on how close the languages are to others. I'm not sure, for example, that it's worth learning Pascal if you already know C (for example), except to compare the way you do things. On the other hand, learning functional programming for the first time (in Scheme, Lisp, Haskell, whatever) is likely to have a serious effect on how you think about programming.

I disagree with this. Scheme and Script-fu might not be that different but it can still be worth knowing script-fu in addition to scheme. Multiple application specific programming languages are still useful since there is no ideal programming language in the different types.

Good comment.

We saw this in a lot of ways in this class. The languages we discussed were driven by changes in the way people were thinking about programming (from assembly to FORTRAN to LISP to C and the array of OO languages to scripting languages) and the problems they were trying to solve.

This seems to be a misreading of the epigram. Perlis was not suggesting that new ways of thinking lead to new languages; he was suggesting that the best languages to learn are the ones that teach you to think about programming differently. I immediately think of your experience with Ruby. Many of you found the emphasis on easy introspection revealing, and a different way to think about programming.

20. Wherever there is modularity there is the potential for misunderstanding: Hiding information implies a need to check communication.

I'm not sure I buy that. There are languages built around the principle of modularity, of keeping the interface separate from the program so that if you change the program you don't have to re-work everything. That to me seemed like a pretty good reason. So what's the basis of his attack on this?

It seems to me that this goes against what we have learned in class about object oriented programming.

Consider a world in which you don't know how something is implemented, and the underlying documentation is not particularly well written. (Say, think about the standard Sun Java libraries :-). Since you don't know about restrictions on how to call things, you might write incorrect calls. If you knew about the underlying implementation, you'd know what was safe and what was unsafe. (Good documentation can solve this, as can the automatic checking of preconditions and postconditions, but I expect that that's, in part, what he means by "check communication").

30. in programming, everything we do is a special case of something more general - and often we know it too quickly.

I'm not sure that this is necessarily the case. Certainly MOST programs are a specialized case of some other problem. But even this requires us to realize that there is some other problem that we had to first. I think that also largely depends on what you've done already as well. After all if you already knew it was a derivation of something you'd already done that would make most programs almost trivially easy because you could modify pre-existing code rather than write anything from scratch.

Epigrams are pithy and general, but rarely universal.

55. A LISP programmer knows the value of everything, but the cost of nothing.

What does the author mean by this? Is this because LISP is a high-level programming language?

Many of the epigrams seem to be reactions to the LISP community. In this case, I expect that Perlis is noting that many LISP operations are costly, and some LISP programmers don't understand them. In contrast, LISP programmers do understand the value of high-level programming.

Since LISP is used in AI research, does efficiency not matter in such research? Or is the cost of large running time neglected over the aim of getting a certain result?

I don't tend to comment on AI after my initial experience in an AI course that I took simultaneously with an algorithms course. ("Excuse me, but that algorithm you just showed looks like it's exponential." "Yes, that's standard paractice.") However, LISP is certainly intended for use in other domains, so efficiency may matter. (Paul Graham has written a number of essays on why we should not care so much about efficiency, but that's another matter.)

65. Make no mistake about it: computer process numbers -- not symbols. We measure our understanding (and control) by the extent to which we can arithmetize an activity.

at the lowest level everything is just changing one set of bits into another set. changing one base two number into another base two number. floating points numbers, pointers to memory, strings and everything else are built upon that foundation.

This is more a translation of the epigram than a response to it.

71. Documentation is like term insurance: It satisfies because almost no one who subscribes to it depends on its benefits.

I sort of agree with this. Undocumented code is a huge pain to go through. Commenting can take up a lot of time when writing code. However, I think coming back to commented code written even a day or two before is helpful.

To me, this epigram provides a strong contrast to 20. I also think that the use of documentation has changed significantly with the creation of languages with huge libraries and documentation generation system.

83. What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak.

If I had to do everything with a Turing machine instead of a computer I wouldn't be very interested in CS. Way to go Hillary but I'm glad there have been major improvements (although the thought of a Hilton hotel on the peak of Mt. Everest kind of disgusts me).

No comment.

92. The computer is the ultimate polluter: Its feces are indistinguishable from the food it produces.

I am not sure what the author is referring to as feces and food that the computer produces.

Correct output and incorrect output are, in essence, indistinguishable.

93. When someone says “I want a programming language in which i need only say what i wish done,” give him a lollipop.

programming languages are what humans use to communicate to the computer what they want it to do. the options and programming language provides can facilitate some types of commands while making other types of commands harder to do. as with many things, there is almost always a trade off between a programming language offering, or not offering, a feature.

This is more an explanation of the epigram than a reaction, and it lacks the pithy cruelty of the epigram. (Implied: “That is a childish request.”)

103. Purely applicative languages are poorly applicable.

Can you explain this one? What exactly does the author mean by a purely applicative language?

It’s another term for “pure functional language”.

104. The proof of a system’s value is its existence.

Even after the proliferation of programming languages, only a few are still widely used. LISP is a great example because it was one of the first ones but it is still used and it inspired many other languages. Graham argued that most languages are still trying to catch up to LISP.

Actually, compared to all but a few languages, LISP is widely used, particularly if we include its variants.