

How to pick a computing language

Eric Blair

18 October 2006

I can not stand how much debate there is about computing languages. I hate the fact that the Web is filled with it, I hate the fact that so many postings on Usenet in the way of 'I need a Matlab routine to...' get replies like 'Why aren't you using Algol!?', and I hate that my own work is so often evaluated based on choice of computing platform rather than actual output.

So, in my little effort for world peace, here are my notes on picking a language. I'll generalize this a bit next time, but the basic theme is that there is no One True Way. The process of picking a language is picking which is the least annoying trade-off, over the course of a series of many trade-offs.

The moral here is that, even though you will no doubt have a preference on one side or the other with all of the debates below, there is indeed a sensible other side, that other people prefer. That is, there are languages on both sides of any debate because there are valid reasons, both in terms of aesthetic preference and in terms of practical issues, for picking both options. Anybody who tells you otherwise, for example insisting that we must all use dynamically-typed languages from now on, is just being an ass. Pick the least annoying paradigm for yourself, and let your neighbors pick the least annoying paradigm for themselves. We'll all get our work done in the end.

Having ranted, here's a list of a dozen primary axes along which general-purpose computing languages differ. Work out which side you prefer, find a language that is on the same side as you, and go code.

1. Are the libraries what you need? The primary joy in using an existing computing language is the strong hope that somebody has already written the code you need. However, I have never seen a language that really has a good library for everything I've wanted. I see the big schism as languages with lots of libraries for numerical and otherwise number-crunching routines versus languages with lots of support for Web handling, though you will no doubt find your own divisions among what type of code base supports what languages.

2. Does it assign types dynamically or statically? The evangelists here all seem to be on the dynamic-typing side, but the issue is more muddled than a one-side-or-the-other split, and neither extreme is great.

If a system is enthusiastic about dynamically typing, then odd problems crop up. What if the first piece of text input happens to be "14"—will your system

then assume that the input is all integers, and then crash when the next input is "fifteen"? Say you are trying to build a list of lists. Start with an empty list, then add the first list, then add the second list, and so on. The interpreter may cast an empty list to NULL, and may flatten a list of a single list to just a list—and if the first list happens to have no elements or one element, there may be still more casting until you're left with no list at all. With experience, you'll figure out what tricks you need to fix the problem, and will see it as no big deal.

With static typing, the annoyance is that you'll need to explicitly re-cast variables from time to time. For many but not all static languages, you will need to declare the type at first use, which is also potentially annoying—just as playing guess-the-type can be with a language that doesn't allow you to just come out and declare a variable's type.

Anyway, the question is not dynamic versus static, but how dynamic the language is. What is its list of auto-casts and what is its list of casts that you have to make yourself, and how well does that list fit your expectations?

3. What are the scoping rules? Every language has different rules for when a variable is in or out of scope. One paradigm is the location of a variable in a file or a directory path. E.g.: in C, a file-global variable can be read by every function beneath it in the file; or in Matlab/Octave, a function is accessible if its `.do` file is in a specified path. The other paradigm is to define objects, and say that most variables have scope only within the object or its friends. In the first paradigm, files or portions of files can be read in (included) in other files so that scope is somehow shared; in the second, there is a syntax to say that one object inherits another.

The intent in both cases is to allow modularity and encapsulation, wherein one unit represents some self-contained concept and doesn't interfere with other self-contained concepts. Every useful language in existence has this concept, and every one of 'em does it in a slightly different manner. I.e., the choice is the epitome of personal taste.

Evangelists here tend to be on the object-oriented side. But the scope-by-file method manages the same encapsulation and flexibility.

4. Is it too verbose or terse? There is general agreement that too much verbosity in code is a bad thing.

But too little verbosity can also be bad. Coding textbooks often show off one-line programs that calculate the GDP per capita of Bangladesh based on the price of toothbrushes, with fourteen intermediate steps neatly cascading on a single line. This looks very impressive, and looks much more pleasing than the half-page version with all the intermediate mess on fourteen separate lines. It's certainly fun to write one-liners, but guess which routine is easier to debug. Hint: most of your time debugging is spent tracing a process through its intermediate steps.

Some languages are very good at hiding intermediate cruft—so good that

you have no idea of what's going on. Some languages are terrible at hiding anything, and require that you always go through every step yourself. There's a practical and æsthetic balance to be drawn here; let's not pretend that terse code is always superior to more verbose code.

5. How does it handle aliases? Let us say that you want to put a box at the front door so anybody can leave things or take things as necessary. For this to work, you have to tell everybody where to find the box. In computing land, you are not assigning to `var1` and `var2` the *contents* of the box, but its *location*.

So there needs to be two types of assignment: one that goes to the box and copies its contents into `var1`, and one that indicates that `var1` is hereby an alias for the box itself. This is an inherently confusing concept, because the two types of assignment have so much in common. But gosh golly, you've gotta have it.

C and friends use variables and pointers to variables. Some languages use immediate and lazy evaluation for this. Some languages always assume that you mean aliasing unless forced otherwise. And some languages, which I consider to be too limited for anything more than teaching, eliminate confusion by disallowing aliasing entirely.

6. Call-by-what? How are arguments passed in to subfunctions? Typically, languages do this by passing the value itself or by passing an alias for the value. Again, I'd consider a system to be braindead if it doesn't allow both, but, due to speed issues, the ability to pass aliases is the one that really shouldn't be passed up.

7. How does the language pass functions? It would be nice to write a function that takes a function as an argument, such as `apply(fn, list)`, that replaces every element of the list with `fn(element)`. More generally, there are many reasons for passing functions as arguments to other functions. Some languages encourage this by treating functions like any other data, Lisp being the paradigmatic example. Others make it not-so-easy, like C, which lets you pass function pointers, but has no syntax for writing in-line minifunctions. There exist systems where functions can't be passed at all, which I again recommend throwing in the waste bin. But among the mainstream options, it's a question of how convenient you want function-passing to be—is this something you expect to use every other line, or something that is good to have handy when necessary?

8. Does it let you hang yourself? This one is self-descriptive. The best example would be in the dynamic/static type issue. If you have a function that acts on text strings, and you give it the number 14, should it automatically cast it to "14" or give you a gentle 'I'm sorry Dave, I can't do that'? C is famous for not stopping you when you access element fifteen of a fourteen element list; some hate it for that and some rely on it heavily.

Every system glosses over errors in some directions and refuses to act in other directions, so the question here should really be: in what ways does the system let you hang yourself and when does it stop you, and are those the constraints that will help or frustrate you?

9. Is it fast or is it easy? You can't have both, though every proselytizer advertises that their fave has achieved such a miracle. If you want something that is superpaternalistic and takes care of everything without your thinking about it, then you're asking the processor to do a lot of work. For example, if a system is enthusiastically dynamically typed, then the system will check the type of the variable at every single use; if you have a vector of a million data points, that's a lot of overhead. If your system thinks you're too dumb to understand aliases or call-by-reference, then it will copy the contents of the box where other systems just point to the box, again creating overhead. As I've noted before, all that ease and convenience can mean not just a ten or twenty percent speed drag, but a slowdown of about fifty times¹.

Joel the Guru has complained² about people who just assume away all computational issues, and recently berated an evangelist³ who insisted on not caring. You're allowed to pick user friendliness over speed, but you have to acknowledge that you're making that decision.

10. Does it lean toward ease of use or ease of initial use? Ease of initial use means that a new user can intuitively guess at what needs to be done with few errors. To achieve this, there are usually restrictions in place that prevent the user from doing the wrong thing, details are elided, and lots of in-place documentation produced. Ease of long-term use means that the user has few restrictions, can tweak as many detailed as desired, and is not frequently interrupted. Most systems focus on either making the details easily accessible or on hiding them; few if any do a truly good job of guiding early users and at the same time getting out of the advanced user's way.

Evangelists miss this trade-off all the time, and pretend that because it is so easy to type `print 'Hello, world'` it must be that the system will always be easy to use. This is a condescending belief that users are unable to learn and adapt. Lisp coders have reason to brag about their great effectiveness even though their code is just a pile of parens to outsiders, and C coders who really understand pointers do things that are impossible in other languages, and all those guys have valid reasons for why Visual Basic isn't working for them. Let's not pretend that everybody's idea of ease of use is identical to that of a dilettante who will never go much further than printing 'Hello, world'.

Anyway, in terms of picking the language, you can ask yourself if you will be using this darn system for the rest of your life, or are hoping to just do a one-off project that will not grow into a big mess, and pick a language accordingly.

¹<http://fluff.info/blog/arch/00000172.htm>

²<http://www.joelonsoftware.com/articles/fog0000000319.html>

³<http://joelonsoftware.com/items/2006/09/12.html>

11. Does its fluff seem useful to me? Perl handles regular expressions as part of the language, which provides some nifty text-handling that is much more ornery in other languages. Some languages have built-in databases, or hashes, or lists. This may seem more fun and useful to you than calling these things in from a library. Every language has a library to handle regular expressions, interface with databases, and use hashes and lists; the question is only whether they are immediately on hand or up on the shelf, and what you want to have on hand. But if *everything* is immediately on hand in the form of a quirk in the grammar then you get, well, Perl, which has a very complex grammar and can often be hard to read.

12. Does it use too many parentheses? Finally, there is the actual visual appeal: is the code filled with parens, tabs, stars? This is the last question on the list because, frankly, you'll get used to it.

So, there you have it. A dozen ways by which different languages distinguish themselves, all of which require balance rather than a direct proclamation of The Right Answer.