Programming is Hard, Let's Go Scripting...

By Larry Wall on December 6, 2007 12:00 AM

I think, to most people, scripting is a lot like obscenity. I can't define it, but I'll know it when I see it. Here are some common memes floating around:

```
Simple language
"Everything is a string"
Rapid prototyping
Glue language
Process control
Compact/concise
Worse-is-better
Domain specific
"Batteries included"
```

...I don't see any real center here, at least in terms of technology. If I had to pick one metaphor, it'd be easy onramps. And a slow lane. Maybe even with some optional fast lanes.

Easy Onramps

But basically, scripting is not a technical term. When we call something a scripting language, we're primarily making a linguistic and cultural judgment, not a technical judgment.

I see scripting as one of the humanities. It's our linguistic roots showing through. So speaking of roots...

The Past

Suppose you went back to Ada Lovelace and asked her the difference between a script and a program. She'd probably look at you funny, then say something like: Well, a script is what you give the actors, but a program is what you give the audience. That Ada was one sharp lady...

Since her time, we seem to have gotten a bit more confused about what we mean when we say scripting. It confuses even me, and I'm supposed to be one of the experts.

So I'm afraid all I can do is give you my own worm's eye view of the past, the present, and the future. Let me warn you that I am not without a few prejudices here and there.

BASIC

Now, however it was initially intended, I think BASIC turned out to be one of the first major scripting languages, especially the extended version that DEC put onto its minicomputers called BASIC/PLUS, which happily included recursive functions with arguments. I started out as a BASIC programmer. Some people would say that I'm permanently damaged. Some people are undoubtedly right.

But I'm not going to apologize for that. All language designers have their occasional idiosyncracies. I'm just better at it than most. `:-)`

RSTS BASIC/PLUS

Anyway, when I was a RSTS programmer on a PDP-11, I certainly treated BASIC as a scripting language, at least in terms of rapid prototyping and process control. I'm sure it warped my brain forever. Perl's statement modifiers are straight out of BASIC/PLUS. It even had some cute sigils on the ends of its variables to distinguish string and integer from floating point.

But you could do extreme programming. In fact, I had a college buddy I did pair programming with. We took a compiler writing class together and studied all that fancy stuff from the dragon book. Then of course the professor announced we would be implementing our own language, called PL/0. After thinking about it a while, we announced that we were going to do our project in BASIC. The professor looked at us like were insane. Nobody else in the class was using BASIC. And you know what? Nobody else in the class finished their compiler either. We not only finished but added I/O extensions, and called it PL 0.5. That's rapid prototyping.

Unix?

I remember one day our computer center got a letter from Bell Labs telling us that we could get a tape of Unix V6 for cheap, only $100 because they were coming out shortly with V7. We all looked at each other and said, Why would we ever want to use this thing called Unix? We have RSTS.

JAM (no not that one)

My first scripting language was written in BASIC. For my job in the computer center I wrote a language that I called JAM, short for Jury-rigged All-purpose Meta-language. Story of my life...

JAM was an inside-out text-processing language much like PHP, except that HTML hadn't been invented yet. We mostly used it as a fancy macro processor for BASIC. Unlike PHP, it did not have 3,000 functions in one namespace. We wouldn't have had the memory, for one thing.

LISP

For good or ill, when I went off to grad school, I studied linguistics, so the only computer language I used there was LISP. It was my own personal McCarthy era.

Is LISP a candidate for a scripting language? While you can certainly write things rapidly in it, I cannot in good conscience call LISP a scripting language. By policy, LISP has never really catered to mere mortals.

And, of course, mere mortals have never really forgiven LISP for not catering to them.

Pascal, Ada

Visit the home of the Perl programming language: Perl.org

Download
Documentation
Perl Bloggers
Foundation News

Sponsored by

Subscribe to this website's feed

Monthly Archives

February 2014 (1)
January 2014 (1)
October 2013 (1)
January 2013 (1)
December 2012 (1)
November 2012 (1)
October 2012 (2)
August 2012 (2)
June 2012 (11)
May 2012 (18)
April 2012 (17)
February 2012 (1)
December 2011 (1)
September 2011 (1)
August 2011 (2)
June 2011 (1)
May 2011 (3)
April 2011 (1)
March 2011 (1)
February 2011 (1)
January 2011 (1)
November 2010 (1)
October 2010 (2)
September 2010 (1)
August 2010 (3)
July 2010 (2)
April 2010 (2)
March 2010 (4)
May 2008 (1)
April 2008 (2)
March 2008 (1)
February 2008 (1)
January 2008 (1)
December 2007 (2)
September 2007 (1)
August 2007 (1)
July 2007 (1)
June 2007 (1)
May 2007 (1)
April 2007 (1)
March 2007 (1)
February 2007 (1)
January 2007 (1)
December 2006 (1)
November 2006 (2)
October 2006 (1)
September 2006 (1)
August 2006 (1)
July 2006 (1)
June 2006 (1)
May 2006 (1)
April 2006 (1)
March 2006 (1)

Once I got into industry, I wrote a compiler in Pascal for a discrete event simulator, and slavered over the forthcoming Ada specs. As a linguist, I don't think of Ada as a big language. Now, English and Japanese, those are big languages. Ada is just a medium-sized language.

Unix, shell

After several years I finally became acquainted with Unix and its various scripting languages. OK, to be more precise, BSD, and csh.

BSD, csh

Yeah, yeah, I know. More brain damage...

I also learned a little C.

C

That's because a little C is all there is. I'm still learning those libraries though.

shell + awk + sed + find + expr...

But the frustrations of Unix shell programming led directly to the creation of Perl, which I don't really have time to tell. But essentially, I found that shell scripting was intrinsically limited by the fact that most of its verbs are not under its control and hence largely inconsistent with each other. And the nouns are impoverished, restricted to strings and files, with who-knows-what typology.

C xor shell

More destructive was the mindset that it was a one-dimensional universe: you either programmed in C or you programmed in shell, because they're obviously at opposite ends of the One True Continuum. Perl came about when I realized that scripting did not always have to viewed as the opposite of programming, but that a single language could be pretty good for both. That opened up a huge ecological niche. Many of you have seen my old clamshell diagram, with the two dimensions of manipulexity and whipuptitude.

Tcl

After Perl came Tcl, which in a sense is a purer scripting language than Perl. Perl just pretends that everything is a string when it's convenient, but Tcl really believes that as a controlling metaphor. The string metaphor tends to have bad performance ramifications, but that's not why Tcl languished, I think. There were two reasons for that.

First, Tcl stayed in the Unix mindset that controlling tools was the opposite of creating tools, so they didn't optimize much. The fast parts can always be written in C, after all.

The second reason was the lack of a decent extension mechanism, so you ended up with separate executables for expect, incr-tcl, etc.

I must say, though, that I've always admired Tcl's delegational model of semantics. But it fell into the same trap as LISP by expecting everyone to use the One True Syntax. Speaking of the One True Syntax:

Python

After Tcl came Python, which in Guido's mind was inspired positively by ABC, but in the Python community's mind was inspired negatively by Perl. I'm not terribly qualified to talk about Python however. I don't really know much about Python. I only stole its object system for Perl 5. I have since repented.

Ruby

I'm much more qualified to talk about Ruby--that's because a great deal of Ruby's syntax is borrowed from Perl, layered over Smalltalk semantics. I've always viewed Ruby as a much closer competitor for Perls ecological niche, not just because of the borrowed ideas, but because both Perl and Ruby take their functional programming support rather more seriously that Python does. On the other hand, I think Ruby kind of screwed up on its declaration syntax, among other things.

***sh**

Meanwhile, the Bourne shell was extended into the Korn shell and bash. I didn't have much to do with those either. Thankfully. I will say that the continued evolution of the shell shows just how crufty a language can get when you just keep adding on ad hoc syntactic features.

PHP

We've also seen the rise of PHP, which takes the worse-is-better approach to dazzling new depths, as it were. By and large PHP seems to be making the same progression of mistakes as early Perl did, only slower. The one thing it does better is packaging. And when I say packaging, I don't mean namespaces.

JavaScript

Then there's JavaScript, a nice clean design. It has some issues, but in the long run JavaScript might actually turn out to be a decent platform for running Perl 6 on. Pugs already has part of a backend for JavaScript, though sadly that has suffered some bitrot in the last year. I think when the new JavaScript engines come out we'll probably see renewed interest in a JavaScript backend.

Monad/PowerShell

I've looked a bit at Microsoft's Monad, and I'm pleased to note that it has object pipes like Perl 6. I just hope they don't patent it.

Lua, AppleScript

There are other scripting languages in wide use. Sadly, I must confess I never looked closely at Lua or AppleScript, probably because I'm not a game designer with a Mac.

Actually, I suspect it runs deeper than that, which brings us up to the present time.

The Present

When I look at the present situation, what I see is the various scripting communities behaving a lot like neighboring tribes in the jungle, sometimes trading, sometimes warring, but by and large just keeping out of each other's way in complacent isolation.

I tend to take an anthropological view of these things. Many of you here are Perl programmers, but some of you come from other programming tribes. And depending on your tribal history, you might think of a string as a pointer to a byte array if you're a C programmer, or as a list if you're a functional programmer, or as an object if you're a Java programmer. I view a string as a Text, with a capital T.

Text

I read that word from a postmodern perspective. Of course, the term Postmodern is itself context-sensitive. Some folks think Postmodernism means little more than the Empowerment of the Vulgar. Some folks think the same about Perl.

But I take Postmodernism to mean that a Text, whether spoken or written, is an act of communication requiring intelligence on both ends, and sometimes in the middle too. I don't want to talk to a stupid computer language. I want my computer language to understand the strings I type.

Perl is a postmodern language, and a lot of conservative folks feel like Postmodernism is a rather liberal notion. So it's rather ironic that my views on Postmodernism were primarily informed by studying linguistics and translation as taught by missionaries, specifically, the Wycliffe Bible Translators. One of the things they hammered home is that there's really no such thing as a primitive human language. By which they mean essentially that all human languages are Turing complete.

When you go out to so-called primitive tribes and analyze their languages, you find that structurally they're just about as complex as any other human language. Basically, you can say pretty much anything in any human language, if you work at it long enough. Human languages are Turing complete, as it were.

Human languages therefore differ not so much in what you *can* say but in what you *must* say. In English, you are forced to differentiate singular from plural. In Japanese, you don't have to distinguish singular from plural, but you do have to pick a specific level of politeness, taking into account not only your degree of respect for the person you're talking to, but also your degree of respect for the person or thing you're talking about.

So languages differ in what you're forced to say. Obviously, if your language forces you to say something, you can't be concise in that particular dimension using your language. Which brings us back to scripting.

How many ways are there for different scripting languages to be concise?

How many recipes for borscht are there in Russia?

Language designers have many degrees of freedom. I'd like to point out just a few of them.

early binding / late binding

Binding in this context is about exactly when you decide which routine you're going to call for a given routine name. In the early days of computing, most binding was done fairly early for efficiency reasons, either at compile time, or at the latest, at link time. You still tend to see this approach in statically typed languages. With languages like Smalltalk, however, we began to see a different trend, and these days most scripting languages are trending towards later binding. That's because scripting languages are trying to be dwimmy (Do What I Mean), and the dwimmiest decision is usually a late decision because you then have more available semantic and even pragmatic context to work with. Otherwise you have to predict the future, which is hard.

So scripting languages naturally tend to move toward an object-oriented point of view, where the binding doesn't happen 'til method dispatch time. You can still see the scars of conflict in languages like C++ and Java though. C++ makes the default method type non-virtual, so you have to say virtual explicitly to get late binding. Java has the notion of final classes, which force calls to the class to be bound at compile time, essentially. I think both of those approaches are big mistakes. Perl 6 will make different mistakes. In Perl 6 all methods are virtual by default, and only the application as a whole can tell the optimizer to finalize classes, presumably only after you know how all the classes are going to be used by all the other modules in the program.

single dispatch / multiple dispatch

In a sense, multiple dispatch is a way to delay binding even longer. You not only have to delay binding 'til you know the type of the object, but you also have to know the types of all rest of the arguments before you can pick a routine to call. Python and Ruby always do single dispatch, while Dylan does multiple dispatch. Here is one dimension in which Perl 6 *forces* the caller to be explicit for clarity. I think it's an important distinction for the programmer to bear in mind, because single dispatch and multiple dispatch are philosophically very different ideas, based on different metaphors.

With single-dispatch languages, you are basically sending a message to an object, and the object decides what to do with that message. With multiple dispatch languages, however, there is no privileged object. All the objects involved in the call have equal weight. So one way to look at multiple dispatch is that the objects are completely passive. But if the objects aren't deciding how to bind, who is?

Well, it's sort of a democratic thing. All the routines of a given name get together and hold a political conference. (Well, not really, but this is how the metaphor works.) Each of the routines is a delegate to the convention. All the potential candidates put their names in the hat. Then all the routines vote on who the best candidate is, and the next best, and the next best after that. And eventually the routines themselves decide what the best routine to call is.

So basically, multiple dispatch is like democracy. It's the worst way to do late binding, except for all the others.

But I really do think that's true, and likely to become truer as time goes on. I'm spending a lot of time on this multiple dispatch issue because I think programming in the large is mutating away from the command-and-control model implicit in single dispatch. I think the field of computation as a whole is moving more toward the kinds of decisions that are better made by swarms of insects or schools of fish, where no single individual is in control, but the swarm as a whole has emergent behaviors that are somehow much smarter than any of the individual components.

eager evaluation / lazy evaluation

Most languages evaluate eagerly, including Perl 5. Some languages evaluate all expressions as lazily as possible. Haskell is a good example of that. It doesn't compute anything until it is forced to. This has the advantage that you can do lots of cool things with infinite lists without running out of memory. Well, at least until someone asks the program to calculate the whole list. Then you're pretty much hosed in any language, unless you have a real Turing machine.

So anyway, in Perl 6 we're experimenting with a mixture of eager and lazy. Interestingly, the distinction maps very nicely onto Perl 5's concept of scalar context vs. list context. So in Perl 6, scalar context is eager and list context is lazy. By default, of course. You can always force a scalar to be lazy or a list to be eager if you like. But you can say things like `for 1..Inf` as long as your loop exits some other way a little bit before you run into infinity.

eager typology / lazy typology

Usually known as static vs. dynamic, but again there are various positions for the adjustment knob. I rather like the gradual typing approach for a number of reasons. Efficiency is one reason. People usually think of strong typing as a reason, but the main reason to put types into Perl 6 turns out not to be strong typing, but rather multiple dispatch. Remember our political convention metaphor? When the various candidates put their names in the hat, what distinguishes them? Well, each candidate has a political platform. The planks in those political platforms are the types of arguments they want to respond to. We all know politicians are only good at responding to the types of arguments they want to have...

There's another way in which Perl 6 is slightly more lazy than Perl 5. We still have the notion of contexts, but exactly when the contexts are decided has changed. In Perl 5, the compiler usually knows at compile time which arguments will be in scalar context, and which arguments will be in list context. But Perl 6 delays that decision until method binding time, which is conceptually at run time, not at compile time. This might seem like an odd thing to you, but it actually fixes a great number of things that are suboptimal in the design of Perl 5. Prototypes, for instance. And the need for explicit references. And other annoying little things like that, many of which end up as frequently asked questions.

limited structures / rich structures

December 1998 (1)
November 1998 (1)
July 1998 (2)
June 1998 (1)
March 1998 (1)

POWERED BY MT 5 MOVABLE TYPE®

Awk, Lua, and PHP all limit their composite structures to associative arrays. That has both pluses and minuses, but the fact that awk did it that way is one of the reasons that Perl does it differently, and differentiates ordered arrays from unordered hashes. I just think about them differently, and I think a lot of other people do too.

symbolic / wordy

Arguably APL is also a kind of scripting language, largely symbolic. At the other extreme we have languages that eschew punctuation in favor of words, such as AppleScript and COBOL, and to a lesser extent all the Algolish languages that use words to indicate blocks where the C-derived languages use curlies. I prefer a balanced approach here, where symbols and identifiers are each doing what theyre best at. I like it when most of the actual words are those chosen by the programmer to represent the problem at hand. I don't like to see words used for mere syntax. Such syntactic functors merely obscure the real words. That's one thing I learned when I switched from Pascal to C. Braces for blocks. It's just right visually.

Actually, there are languages that do it even worse than COBOL. I remember one Pascal variant that required your keywords to be capitalized so that they would stand out. No, no, no, no, no! You don't want your functors to stand out. It's shouting the wrong words: IF! foo THEN! bar ELSE! baz END! END! END! END!

Anyway, in Perl 6 we're raising the standard for where we use punctuation, and where we don't. We're getting rid of some of our punctuation that isn't really pulling its weight, such as parentheses around conditional expressions, and most of the punctuational variables. And we're making all the remaining punctuation work harder. Each symbol has to justify its existence according to Huffman coding.

Oddly, there's one spot where we're introducing new punctuation. After your sigil you can add a twigil, or secondary sigil. Just as a sigil tells you the basic structure of an object, a twigil tells you that a particular variable has a weird scope. This is basically an idea stolen from Ruby, which uses sigils to indicate weird scoping. But by hiding our twigils after our sigils, we get the best of both worlds, plus an extensible twigil system for weird scopes we haven't thought of yet.

We think about extensibility a lot. We think about languages we don't know how to think about yet. But leaving spaces in the grammar for new languages is kind of like reserving some of our land for national parks and national forests. Or like an archaeologist not digging up half the archaeological site because we know our descendants will have even better analytical tools than we have.

Really designing a language for the future involves a great deal of humility. As with science, you have to assume that, over the long term, a great deal of what you think is true will turn out not to be quite the case. On the other hand, if you don't make your best guess now, you're not really doing science either. In retrospect, we know APL had too many strange symbols. But we wouldn't be as sure about that if APL hadn't tried it first.

compile time / run time

Many dynamic languages can eval code at run time. Perl also takes it the other direction and runs a lot of code at compile time. This can get messy with operational definitions. You don't want to be doing much file I/O in your `BEGIN` blocks, for instance. But that leads us to another distinction:

declarational / operational

Most scripting languages are way over there on the operational side. I thought Perl 5 had an oversimplified object system till I saw Lua. In Lua, an object is just a hash, and there's a bit of syntactic sugar to call a hash element if it happens to contain code. Thats all there is. They don't even have classes. Anything resembling inheritance has to be handled by explicit delegation. That's a choice the designers of Lua made to keep the language very small and embeddable. For them, maybe it's the right choice.

Perl 5 has always been a bit more declarational than either Python or Ruby. I've always felt strongly that implicit scoping was just asking for trouble, and that scoped variable declarations should be very easy to recognize visually. Thats why we have `my`. It's short because I knew we'd use it frequently. Huffman coding. Keep common things short, but not too short. In this case, 0 is too short.

Perl 6 has more different kinds of scopes, so we'll have more declarators like `my` and `our`. But appearances can be deceiving. While the language looks more declarative on the surface, we make most of the declarations operationally hookable underneath to retain flexibility. When you declare the type of a variable, for instance, you're really just doing a kind of tie, in Perl 5 terms. The main difference is that you're tying the implementation to the variable at compile time rather than run time, which makes things more efficient, or at least potentially optimizable.

immutable classes / mutable classes

Classes in Java are closed, which is one of the reasons Java can run pretty fast. In contrast, Ruby's classes are open, which means you can add new things to them at any time. Keeping that option open is perhaps one of the reasons Ruby runs so slow. But that flexibility is also why Ruby has Rails.

Perl 6 will have an interesting mix of immutable generics and mutable classes here, and interesting policies on who is allowed to close classes when. Classes are never allowed to close or finalize themselves, for instance. Sorry, for some reason I keep talking about Perl 6. It could have something to do with the fact that we've had to think about all of these dimensions in designing Perl 6.

class-based / prototype-based

Here's another dimension that can open up to allow both approaches. Some of you may be familiar with classless languages like Self or JavaScript. Instead of classes, objects just clone from their ancestors or delegate to other objects. For many kinds of modeling, it's actually closer to the way the real world works. Real organisms just copy their DNA when they reproduce. They don't have some DNA of their own, and an `@ISA` array telling you which parent objects contain the rest of their DNA.

The meta-object protocol for Perl 6 defaults to class-based, but is flexible enough to set up prototype-based objects as well. Some of you have played around with Moose in Perl 5. Moose is essentially a prototype of Perl 6's object model. On a semantic level, anyway. The syntax is a little different. Hopefully a little more natural in Perl 6.

passive data, global consistency / active data, local consistency

Your view of data and control will vary with how functional or object-oriented your brain is. People just think differently. Some people think mathematically, in terms of provable universal truths. Functional programmers don't much care if they strew implicit computation state throughout the stack and heap, as long as everything *looks* pure and free from side-effects.

Other people think socially, in terms of cooperating entities that each have their own free will. And it's pretty important to them that the state of the computation be stored with each individual object, not off in some heap of continuations somewhere.

Of course, some of us can't make up our minds whether we'd rather emulate the logical Sherlock Holmes or sociable Dr. Watson. Fortunately, scripting is not incompatible with either of these approaches, because both approaches can be made more approachable to normal folk.

info hiding / scoping / attachment

And finally, if you're designing a computer language, there are a couple bazillion ways to encapsulate data. You have to decide which ones are important. What's the best way to let the programmer achieve separation of concerns?

object / class / aspect / closure / module / template / trait

You can use any of these various traditional encapsulation mechanisms.

transaction / reaction / dynamic scope

Or you can isolate information to various time-based domains.

process / thread / device / environment

You can attach info to various OS concepts.

screen / window / panel / menu / icon

You can hide info various places in your GUI. Yeah, yeah, I know, everything is an object. But some objects are more equal than others.

syntactic scope / semantic scope / pragmatic scope

Information can attach to various abstractions of your program, including, bizarrely, lexical scopes. Though if you think about it hard enough, you realize lexical scopes are also a funny kind of dynamic scope, or recursion wouldn't work right. A `state` variable is actually more purely lexical than a `my` variable, because it's shared by all calls to that lexical scope. But even state variables get cloned with closures. Only global variables can be truly lexical, as long as you refer to them only in a given lexical scope. Go figure.

So really, most of our scopes are semantic scopes that happen to be attached to a particular syntactic scope.

You may be wondering what I mean by a *pragmatic* scope. That's the scope of what the user of the program is storing in their brain, or in some surrogate for their brain, such as a game cartridge. In a sense, most of the web pages out there on the Internet are part of the pragmatic scope. As is most of the data in databases. The hallmark of the pragmatic scope is that you really don't know the lifetime of the container. It's just out there somewhere, and will eventually be collected by that Great Garbage Collector that collects all information that anyone forgets to remember. The Google cache can only last so long. Eventually we will forget the meaning of every URL. But we must not forget the *principle* of the URL. That leads us to our next degree of freedom.

use Lingua::Perligata;

If you allow a language to mutate its own grammar within a lexical scope, how do you keep track of that cleanly? Perl 5 discovered one really bad way to do it, namely source filters, but even so we ended up with Perl dialects such as Perligata and Klingon. What would it be like if we actually did it right?

Doing it right involves treating the evolution of the language as a pragmatic scope, or as a set of pragmatic scopes. You have to be able to name your dialect, kind of like a URL, so there needs to be a universal root language, and ways of warping that universal root language into whatever dialect you like. This is actually near the heart of the vision for Perl 6. We don't see Perl 6 as a single language, but as the root for a family of related languages. As a family, there are shared cultural values that can be passed back and forth among sibling languages as well as to the descendants.

I hope you're all scared stiff by all these degrees of freedom. I'm sure there are other dimensions that are even scarier.

But... I think its a manageable problem. I think its possible to still think of Perl 6 as a scripting language, with easy onramps.

And the reason I think its manageable is because, for each of these dimensions, it's not just a binary decision, but a knob that can be positioned at design time, compile time, or even run time. For a given dimension X, different scripting languages make different choices, set the knob at different locations.

You can't even think about X!
There's only one way to do X!
There's more than one way to do X!
There are too many ways to do X!

You may recognize some slogans in here.

Curling Up

So I'm not suggesting that all scripting languages have to take all these dimensions into account, even if Perl 6 tries to. The scripting paradigm is not any one of these dimensions. According to various theories the universe may be laid out in ten or twenty dimensions, but generally we get by with only about three and a half of those dimensions. The rest are said to be curled up. Maybe we live in a scripting universe.

Most of the scripting languages we call Perl 6 will have most of these dimensions curled up most of the time. But unlike the real universe, where it takes huge machines to uncurl these dimensions, we'll make the dimensions uncurl just by keeping our declarations straight. Well, we'll try. And where that fails, we'll rely on the culture to keep things straight.

For example, that's exactly what happened already with Perl 5. We have the declarations, `use strict; use warnings;`. But it's the culture that decided to enforce the use of them. So much so that we've decided that they should be the default for most of Perl 6. It was one of those decisions by the hive. In this case the swarm turned out to be smarter than the language designer. And that's as it should be.

The Future

Well, so what's the future of scripting?

In my completely unbiased opinion, that would be Perl 6. `:-)`

Seriously though, it's always safe to predict that the ecological landscape will end up with many small languages and a few dominant ones. Some languages like AppleScript have particular ecological niches and are unlikely to grow out of them. Other languages get used outside their original niche. There will always be the generalists, like crows and mockingbirds, and the specialists, like penguins and dodos. (Well, maybe not always the dodos...)

Among the generalists, the conventional wisdom is that the worse-is-better approach is more adaptive. Personally, I get a little tired of the argument: My worse-is-better is better than your worse-is-better because I'm better at being worser! Is it really true that the worse-is-better approach always wins? With Perl 6 we're trying to sneak one better-is-better cycle in there and hope to come out ahead before reverting to the tried and true worse-is-better approach. Whether that works, only time will tell.

Tags: language design, Larry Wall, Perl 5, Perl 6, programming languages, scripting, scripting languages, state of the onion, worse is better

This blog is licensed under a Creative Commons License.