

Natural Language Principles in Perl

Learn it once, use it many times

You learn a natural language once and use it many times. The lesson for a language designer is that a language should be optimized for expressive power rather than for ease of learning. It's easy to learn to drive a golf cart, but it's hard to express yourself in one.

Learn as you go

You don't learn a natural language even once, in the sense that you never stop learning it. Nobody has ever learned any natural language completely. Unfortunately, in the interests of orthogonality, many computer languages are designed so that every degree of freedom (dimension) is available everywhere. This has its good points if you understand the whole language, but can lead to confusion if you don't. You'd like to ignore some of the dimensions to begin with. You'd like to be able to talk baby talk and be understood. It's okay if a language is difficult to learn, as long as you don't have to learn it all at once.

Many acceptable levels of competence

This is more of a sociological feature, compared to "learn as you go", which is a psychological feature. People don't mind if you speak a subset of a natural language, especially if you are a child or a foreigner. (Except in Paris, of course.) If a language is designed so that you can "learn as you go", then the expectation is that everyone is learning, and that's okay.

Multiple ways to say the same thing

This one is more of an anthropological feature. People not only learn as they go, but come from different backgrounds, and will learn a different subset of the language first. It's Officially Okay in the Perl realm to program in the subset of Perl corresponding to sed, or awk, or C, or shell, or BASIC, or Lisp, or Python. Or FORTRAN, even. Just because Perl is the melting pot of computer languages doesn't mean you have to stir.

No shame in borrowing

In English (and other languages not suffering an identity crisis), people don't mind swiping ideas from other languages and making them part of the language. Efforts to maintain the "purity" of a language (whether natural or artificial) only succeed in establishing an elite class of people who know the shibboleths. Ordinary folks know better, even if they don't know what "shibboleth" means.

Indeterminate dimensionality

Scientists like to be able to locate things by giving a "vector", that is, a list of coordinates in a space of known dimensionality. This is one of the reasons they like orthogonality--it means the various components of the vector are independent of each other. Unfortunately, the real world is not usually set up to work that way. Most problems, including linguistics problems, are a matter of "getting from here to there", and the geography in-between has a heavy influence on which solutions are practical. Problems tend to be solved at several levels. A typical journey might involve your legs, your car, an escalator, a moving sidewalk, a jet, maybe some more moving sidewalks or a tram, another jet, a taxi, and an elevator. At each of these levels, there aren't many "right angles", and the whole thing is a bit fractal in nature. In terms of language, you say something that gets close to what you want to say, and then you start refining it around the edges, just as you would first plan your itinerary between major airports, and only later worry about how to get to and from the airport.

Local ambiguity is okay

People thrive on ambiguity, as long as it is quickly resolved. Generally, within a natural language, ambiguity is resolved rapidly using recently spoken words and topics. Pronouns like "it" refer to things that are close by, syntactically speaking. Perl is full of little ambiguities that people never even notice because they're resolved so rapidly. For instance, many terms and operators in Perl begin with identical characters. Perl resolves them based

on whether it's expecting to see a term or an operator, just as a person would. If you say `1 & 2`, it knows that the `&` is a bitwise AND, but if you say `&foo`, it knows that you're calling subroutine `foo`.

In contrast, many strongly typed languages have "distant" ambiguity. C++ is one of the worst in this respect, because you can look at `a + b` and have no idea at all what the `+` is doing, let alone where it's defined. We send people to graduate school to learn to resolve distant ambiguities.

Punctuation by prosody and inflection

Natural language is naturally punctuated by the pitches, stresses and pauses we use to indicate how words are related. So-called "body language" also comes into play here. *Some* of this punctuation is written in English, but much of it is not—or is only approximated. The trend in recent electronic communications has been to invent various forms of punctuation. :-)

Some computer language designers seem to think that punctuation is evil; I doubt their English teachers would agree.

Disambiguation by number, case and word order

Part of the reason a language can get away with certain local ambiguities is that other ambiguities are suppressed by various mechanisms. English uses number and word order, with vestiges of a case system in the pronouns: "The man looked at the men, and they looked back at him." It's perfectly clear in that sentence who is doing what to whom. Similarly, Perl has number markers on its nouns; that is, `$dog` is one pooch, and `@dog` is (potentially) many. So `$` and `@` are a little like "this" and "these" in English. Perl also uses word order: "sub use" means something quite different from "use sub". Perl doesn't do much with case distinctions, unlike the shells, which make use-vs-mention distinctions using a `$` prefix. Though I guess if you allow that, you could count Perl quotes as a form of case marker. On a slightly more abstruse level, Perl's `\` operator is a sort of case marker or preposition indicating mention rather than use. But as with most computer languages, prepositional notions are usually expressed by position within an argument list. (Though it's certainly possible to write calls using named parameters in Perl, and keys of hashes sometimes function as prepositions.)

```
move $rook from => $qr_pos, to => "kb3";
```

Topicalization

With regard to topicalization, I should point out that this sentence starts with one. A topicalizer simply introduces the subject you're intending to talk about. There are several syntactic forms in English, the simplest one of which is simply a noun: "Carrots, I hate 'em." Pascal has a "with" clause that functions as a topicalizer. Topicalizers can sometimes give a list of topics, at which point you see words like "for BLAH and BLAH, do BLAH". In Perl, there are various things that work as topicalizers. You can say

```
foreach (@dog) { print $_ }
```

This can even be used singularly:

```
for ($some_long_name) { s/foo/bar/g; tr/a-z/A-Z/; print; }
```

Pattern matches (and indeed any conditionals) tend to function as topicalizers in Perl:

```
/^Subject: (.*)/ and print $1;
```

Discourse structure

Discourse structure is how an utterance longer than a sentence is put together. Different languages and cultures have different rules for how to tell a joke or a story, for instance, or how to write a book about Perl. Some computer languages have rather fixed rules for larger structures. COBOL and Pascal come to mind. Perl tends to be pretty free about what order you put your statements, except that it's rather Aristotelian in requiring you to provide an explicit beginning and end for larger structures, using curlies. But you could almost claim that `#!/usr/bin/perl` corresponds to "Once upon a time", while `__END__` means "And they lived happily ever after."

Pronominalization

We all know about pronouns and their uses. There are a number of pronouns in Perl: `$_` means "it", and `@_` tends to mean "them". (But `$1`, `$2` etc. are also pronominal references back to antecedent substrings in the last pattern match, which we mentioned can function as topicalizers.) Within a `foreach` loop or a `grep`, `$_` is not just a copy of the item in question, but an alias for it. Similarly, `@_` is a list of references to the function's arguments, and the arguments can be modified by changing elements of `@_`.

No theoretical axes to grind

Natural languages are used by people who for the most part don't give a rip how elegant the design of their language is. Except for a few writers striving to make a point in the most efficient way possible, ordinary folks scatter all sorts of redundancy throughout their communication to make sure of being understood. They use whatever words come to hand to get their point across, and work at it till they beat the thing to death. Normally this ain't a problem. They're quite willing to learn a new word occasionally if they see that it will be useful, but unlike lawyers or computer scientists, they feel little need to define lots of new words before they say what they want to say. In terms of computer languages, this argues for predefining the commonly used concepts so that people don't feel the need to make so many definitions. Quite a few Perl scripts contain no definitions at all. I dare you to find a C++ program without a definition.

Style not enforced except by peer pressure

We do not all have to write like Faulkner, or program like Dijkstra. I will gladly tell people what my programming style is, and I will even tell them where I think their own style is unclear or makes me jump through mental hoops. But I do this as a fellow programmer, not as the Perl god. Some language designers hope to enforce style through various typographical means such as forcing (more or less) one statement per line. This is all very well for poetry, but I don't think I want to force everyone to write poetry in Perl. Such stylistic limits should be self-imposed, or at most policed by consensus among your buddies.

Cooperative design

Nobody designs a natural language by themselves, unless their name happens to be Tolkien. We all contribute to the design of our language by our borrowing and our coinages, by copying what we think is cool and eschewing what we think is obfuscatinal. The best artificial languages are collaborations--even with a language like Perl where one person seems to be in charge of it. Most of Perl's good ideas were not original with me. Some of them came from other languages, and some of them were suggestions made by various folks as we went along. If you consider the language to include the various cultural trappings (libraries, bin directories) that go along with the language, then even languages like C, or Ada, or C++, or even the Unix shells are collaborations by many, many people. Perl is no exception to this.

"Inevitable" Divergence

Because a language is designed by many people, any language inevitably diverges into dialects. It may be possible to delay this, but for any living language the forces of divergence are nearly always stronger than the forces of convergence. POSIX tried to unify System V and BSD, and as soon as they squeezed things together in that dimension, the number of Unix variants exploded in several other dimensions. The lesson for a language designer is to build in explicit mechanisms so that it's easy to identify which variant of the language is being dealt with. Perl 5 has an explicit extension mechanism for which you specify, using "use" clauses, which kinds of special semantics or "dialects" you're going to be relying on. Perl 4 didn't have this, and there was considerably more pressure to put various things into the language that didn't belong in the core language. Hopefully now we can stabilize "basic" Perl so that there is less need to invent `oraperl`, `sybperl`, `isqlperl`, etc.