Home
Essays
H&P
Books
YC
Arc
Bel
Lisp
Spam
Responses
FAQs
RAQs
Quotes
RSS
Bio
Twitter

Succinctness is Power

May 2002

> "The quantity of meaning compressed into a small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid."
>
> - Charles Babbage, quoted in Iverson's Turing Award Lecture

In the discussion about issues raised by Revenge of the Nerds on the LL1 mailing list, Paul Prescod wrote something that stuck in my mind.

> Python's goal is regularity and readability, not succinctness.

On the face of it, this seems a rather damning thing to claim about a programming language. As far as I can tell, succinctness = power. If so, then substituting, we get

> Python's goal is regularity and readability, not power.

and this doesn't seem a tradeoff (if it *is* a tradeoff) that you'd want to make. It's not far from saying that Python's goal is not to be effective as a programming language.

Does succinctness = power? This seems to me an important question, maybe the most important question for anyone interested in language design, and one that it would be useful to confront directly. I don't feel sure yet that the answer is a simple yes, but it seems a good hypothesis to begin with.

Hypothesis

My hypothesis is that succinctness is power, or is close enough that except in pathological examples you can treat them as identical.

It seems to me that succinctness is what programming languages are *for.* Computers would be just as happy to be told what to do directly in machine language. I think that the main reason we take the trouble to develop high-level languages is to get leverage, so that we can say (and more importantly, think) in 10 lines of a high-level language what would require 1000 lines of machine language. In other words, the main point of high-level languages is to make source code smaller.

If smaller source code is the purpose of high-level languages, and the power of something is how well it achieves its purpose, then

the measure of the power of a programming language is how small it makes your programs.

Conversely, a language that doesn't make your programs small is doing a bad job of what programming languages are supposed to do, like a knife that doesn't cut well, or printing that's illegible.

Metrics

Small in what sense though? The most common measure of code size is lines of code. But I think that this metric is the most common because it is the easiest to measure. I don't think anyone really believes it is the true test of the length of a program. Different languages have different conventions for how much you should put on a line; in C a lot of lines have nothing on them but a delimiter or two.

Another easy test is the number of characters in a program, but this is not very good either; some languages (Perl, for example) just use shorter identifiers than others.

I think a better measure of the size of a program would be the number of elements, where an element is anything that would be a distinct node if you drew a tree representing the source code. The name of a variable or function is an element; an integer or a floating-point number is an element; a segment of literal text is an element; an element of a pattern, or a format directive, is an element; a new block is an element. There are borderline cases (is -5 two elements or one?) but I think most of them are the same for every language, so they don't affect comparisons much.

This metric needs fleshing out, and it could require interpretation in the case of specific languages, but I think it tries to measure the right thing, which is the number of parts a program has. I think the tree you'd draw in this exercise is what you have to make in your head in order to conceive of the program, and so its size is proportionate to the amount of work you have to do to write or read it.

Design

This kind of metric would allow us to compare different languages, but that is not, at least for me, its main value. The main value of the succinctness test is as a guide in *designing* languages. The most useful comparison between languages is between two potential variants of the same language. What can I do in the language to make programs shorter?

If the conceptual load of a program is proportionate to its complexity, and a given programmer can tolerate a fixed conceptual load, then this is the same as asking, what can I do to enable programmers to get the most done? And that seems to me identical to asking, how can I design a good language?

(Incidentally, nothing makes it more patently obvious that the old chestnut "all languages are equivalent" is false than designing languages. When you are designing a new language, you're

constantly comparing two languages-- the language if I did x, and if I didn't-- to decide which is better. If this were really a meaningless question, you might as well flip a coin.)

Aiming for succinctness seems a good way to find new ideas. If you can do something that makes many different programs shorter, it is probably not a coincidence: you have probably discovered a useful new abstraction. You might even be able to write a program to help by searching source code for repeated patterns. Among other languages, those with a reputation for succinctness would be the ones to look to for new ideas: Forth, Joy, Icon.

Comparison

The first person to write about these issues, as far as I know, was Fred Brooks in the *Mythical Man Month*. He wrote that programmers seemed to generate about the same amount of code per day regardless of the language. When I first read this in my early twenties, it was a big surprise to me and seemed to have huge implications. It meant that (a) the only way to get software written faster was to use a more succinct language, and (b) someone who took the trouble to do this could leave competitors who didn't in the dust.

Brooks' hypothesis, if it's true, seems to be at the very heart of hacking. In the years since, I've paid close attention to any evidence I could get on the question, from formal studies to anecdotes about individual projects. I have seen nothing to contradict him.

I have not yet seen evidence that seemed to me conclusive, and I don't expect to. Studies like Lutz Prechelt's comparison of programming languages, while generating the kind of results I expected, tend to use problems that are too short to be meaningful tests. A better test of a language is what happens in programs that take a month to write. And the only real test, if you believe as I do that the main purpose of a language is to be good to think in (rather than just to tell a computer what to do once you've thought of it) is what new things you can write in it. So any language comparison where you have to meet a predefined spec is testing slightly the wrong thing.

The true test of a language is how well you can discover and solve new problems, not how well you can use it to solve a problem someone else has already formulated. These two are quite different criteria. In art, mediums like embroidery and mosaic work well if you know beforehand what you want to make, but are absolutely lousy if you don't. When you want to discover the image as you make it-- as you have to do with anything as complex as an image of a person, for example-- you need to use a more fluid medium like pencil or ink wash or oil paint. And indeed, the way tapestries and mosaics are made in practice is to make a painting first, then copy it. (The word "cartoon" was originally used to describe a painting intended for this purpose).

What this means is that we are never likely to have accurate

comparisons of the relative power of programming languages. We'll have precise comparisons, but not accurate ones. In particular, explicit studies for the purpose of comparing languages, because they will probably use small problems, and will necessarily use predefined problems, will tend to underestimate the power of the more powerful languages.

Reports from the field, though they will necessarily be less precise than "scientific" studies, are likely to be more meaningful. For example, Ulf Wiger of Ericsson did a study that concluded that Erlang was 4-10x more succinct than C++, and proportionately faster to develop software in:

> Comparisons between Ericsson-internal development projects indicate similar line/hour productivity, including all phases of software development, rather independently of which language (Erlang, PLEX, C, C++, or Java) was used. What differentiates the different languages then becomes source code volume.

The study also deals explictly with a point that was only implicit in Brooks' book (since he measured lines of debugged code): programs written in more powerful languages tend to have fewer bugs. That becomes an end in itself, possibly more important than programmer productivity, in applications like network switches.

The Taste Test

Ultimately, I think you have to go with your gut. What does it feel like to program in the language? I think the way to find (or design) the best language is to become hypersensitive to how well a language lets you think, then choose/design the language that feels best. If some language feature is awkward or restricting, don't worry, you'll know about it.

Such hypersensitivity will come at a cost. You'll find that you can't *stand* programming in clumsy languages. I find it unbearably restrictive to program in languages without macros, just as someone used to dynamic typing finds it unbearably restrictive to have to go back to programming in a language where you have to declare the type of every variable, and can't make a list of objects of different types.

I'm not the only one. I know many Lisp hackers that this has happened to. In fact, the most accurate measure of the relative power of programming languages might be the percentage of people who know the language who will take any job where they get to use that language, regardless of the application domain.

Restrictiveness

I think most hackers know what it means for a language to feel restrictive. What's happening when you feel that? I think it's the same feeling you get when the street you want to take is blocked off, and you have to take a long detour to get where you wanted

to go. There is something you want to say, and the language won't let you.

What's really going on here, I think, is that a restrictive language is one that isn't succinct enough. The problem is not simply that you can't say what you planned to. It's that the detour the language makes you take is *longer.* Try this thought experiment. Suppose there were some program you wanted to write, and the language wouldn't let you express it the way you planned to, but instead forced you to write the program in some other way that was *shorter.* For me at least, that wouldn't feel very restrictive. It would be like the street you wanted to take being blocked off, and the policeman at the intersection directing you to a shortcut instead of a detour. Great!

I think most (ninety percent?) of the feeling of restrictiveness comes from being forced to make the program you write in the language longer than one you have in your head. Restrictiveness is mostly lack of succinctness. So when a language feels restrictive, what that (mostly) means is that it isn't succinct enough, and when a language isn't succinct, it will feel restrictive.

Readability

The quote I began with mentions two other qualities, regularity and readability. I'm not sure what regularity is, or what advantage, if any, code that is regular and readable has over code that is merely readable. But I think I know what is meant by readability, and I think it is also related to succinctness.

We have to be careful here to distinguish between the readability of an individual line of code and the readability of the whole program. It's the second that matters. I agree that a line of Basic is likely to be more readable than a line of Lisp. But a program written in Basic is is going to have more lines than the same program written in Lisp (especially once you cross over into Greenspunland). The total effort of reading the Basic program will surely be greater.

total effort = effort per line x number of lines

I'm not as sure that readability is directly proportionate to succinctness as I am that power is, but certainly succinctness is a factor (in the mathematical sense; see equation above) in readability. So it may not even be meaningful to say that the goal of a language is readability, not succinctness; it could be like saying the goal was readability, not readability.

What readability-per-line does mean, to the user encountering the language for the first time, is that source code will *look unthreatening*. So readability-per-line could be a good marketing decision, even if it is a bad design decision. It's isomorphic to the very successful technique of letting people pay in installments: instead of frightening them with a high upfront price, you tell them the low monthly payment. Installment plans are a net lose for the buyer, though, as mere readability-per-line probably is for the programmer. The buyer is going to make a *lot* of those low,

low payments; and the programmer is going to read a *lot* of those individually readable lines.

This tradeoff predates programming languages. If you're used to reading novels and newspaper articles, your first experience of reading a math paper can be dismaying. It could take half an hour to read a single page. And yet, I am pretty sure that the notation is not the problem, even though it may feel like it is. The math paper is hard to read because the ideas are hard. If you expressed the same ideas in prose (as mathematicians had to do before they evolved succinct notations), they wouldn't be any easier to read, because the paper would grow to the size of a book.

To What Extent?

A number of people have rejected the idea that succinctness = power. I think it would be more useful, instead of simply arguing that they are the same or aren't, to ask: to what *extent* does succinctness = power? Because clearly succinctness is a large part of what higher-level languages are for. If it is not all they're for, then what else are they for, and how important, relatively, are these other functions?

I'm not proposing this just to make the debate more civilized. I really want to know the answer. When, if ever, is a language too succinct for its own good?

The hypothesis I began with was that, except in pathological examples, I thought succinctness could be considered identical with power. What I meant was that in any language anyone would design, they would be identical, but that if someone wanted to design a language explicitly to disprove this hyphothesis, they could probably do it. I'm not even sure of that, actually.

Languages, not Programs

We should be clear that we are talking about the succinctness of languages, not of individual programs. It certainly is possible for individual programs to be written too densely.

I wrote about this in On Lisp. A complex macro may have to save many times its own length to be justified. If writing some hairy macro could save you ten lines of code every time you use it, and the macro is itself ten lines of code, then you get a net saving in lines if you use it more than once. But that could still be a bad move, because macro definitions are harder to read than ordinary code. You might have to use the macro ten or twenty times before it yielded a net improvement in readability.

I'm sure every language has such tradeoffs (though I suspect the stakes get higher as the language gets more powerful). Every programmer must have seen code that some clever person has made marginally shorter by using dubious programming tricks.

So there is no argument about that-- at least, not from me. Individual programs can certainly be too succinct for their own

good. The question is, can a language be? Can a language compel programmers to write code that's short (in elements) at the expense of overall readability?

One reason it's hard to imagine a language being too succinct is that if there were some excessively compact way to phrase something, there would probably also be a longer way. For example, if you felt Lisp programs using a lot of macros or higher-order functions were too dense, you could, if you preferred, write code that was isomorphic to Pascal. If you don't want to express factorial in Arc as a call to a higher-order function

```
(rec zero 1 * 1-)
```

you can also write out a recursive definition:

```
(rfn fact (x) (if (zero x) 1 (* x (fact (1- x)))))
```

Though I can't off the top of my head think of any examples, I am interested in the question of whether a language could be too succinct. Are there languages that force you to write code in a way that is crabbed and incomprehensible? If anyone has examples, I would be very interested to see them.

(Reminder: What I'm looking for are programs that are very dense according to the metric of "elements" sketched above, not merely programs that are short because delimiters can be omitted and everything has a one-character name.)

- Japanese Translation
- Russian Translation
- Lutz Prechelt: Comparison of Seven Languages
- Erann Gat: Lisp vs. Java
- Peter Norvig Tries Prechelt's Test
- Matthias Felleisen: Expressive Power of Languages
- Kragen Sitaker: Redundancy and Power
- Forth
- Joy
- Icon
- J
- K