acmqueue

Sir, Please Step Away from the ASR-33!

To move forward with programming languages we need to break free from the tyranny of ASCII.

Poul-Henning Kamp

One of the naughty details of my Varnish software is that the configuration is written in a domain-specific language that is converted into C source code, compiled into a shared library, and executed at hardware speed. That obviously makes me a programming language syntax designer, and just as obviously I have started to think more about how we express ourselves in these syntaxes.

Rob Pike recently said some very pointed words about the Java programming language, which if you think about it, sounded a lot like the pointed words James Gosling had for C++, and remarkably similar to what Bjarne Stroustrup said about good ol' C.

I have always admired Pike. He was already a giant in the field when I started, and his ability to foretell the future has been remarkably consistent.[1] In front of me I have a tough row to hoe, but I will attempt to argue that this time Pike is merely rearranging the deckchairs of the *Titanic* and that he missed the next big thing by a wide margin.

Pike got fed up with C++ and Java and did what any self-respecting hacker would do: he created his own language—better than Java, better than C++, better than C—and he called it Go.

But did he *go* far enough?

```
package main

import "fmt"

func main() {

        fmt.Printf("Hello, World\n")

}
```

This does not in any way look substantially different from any of the other programming languages. Fiddle a couple of glyphs here and there and you have C, C++, Java, Python, Tcl, or whatever.

Programmers are a picky bunch when it comes to syntax, and it is a sobering thought that one of the most rapidly adopted programming languages of all time, Perl, barely had one for the longest time. The funny thing is, what syntax designers are really fighting about is not so much the proper and best syntax for the expression of ideas in a machine-understandable programming language as it is the proper and most efficient use of the ASCII table real estate.

IT'S ALL ASCII TO ME...

There used to be a programming language called ALGOL, the lingua franca of computer science back in its heyday. ALGOL was standardized around 1960 and dictated about a dozen mathematical glyphs such as ×, ÷, ¬, and the very readable subscripted 10 symbol, for use in what today we call scientific notation. Back then computers were built by hand and had one-digit serial numbers. Having a teletypewriter customized for your programming language was the least of your worries.

A couple of years later came the APL programming language, which included an extended character set containing a lot of math symbols. I am told that APL still survives in certain obscure corners of insurance and economics modeling.

Then ASCII happened around 1963, and ever since, programming languages have been trying to fit into it. (Wikipedia claims that ASCII grew the backslash [\] specifically to support ALGOL's /\ and \/ Boolean operators. No source is provided for the claim.)

The trouble probably started for real with the C programming language's need for two kinds of and and or operators. It could have used just or and bitor, but | and || saved one and three characters, which on an ASR-33 teletype amounts to 1/10 and 3/10 second, respectively.

It was certainly a fair tradeoff—just think about how fast you type yourself—but the price for this temporal frugality was a whole new class of hard-to-spot bugs in C code.

Niklaus Wirth tried to undo some of the damage in Pascal, and the bickering over begin and end would no } take.

C++ is probably the language that milks the ASCII table most by allowing templates and operator overloading. Until you have inspected your data types, you have absolutely no idea what + might do to them (which is probably why there never was enough interest to stage an International Obfuscated C++ Code Contest, parallel to the IOCCC for the C language).

C++ stops short of allowing the programmer to create new operators. You cannot define :-: as an operator; you have to stick to the predefined set. If Bjarne Stroustrup had been more ambitious on this aspect, C++ could have beaten Perl by 10 years to become the world's second write-only programming language, after APL.

How desperate the hunt for glyphs is in syntax design is exemplified by how Guido van Rossum did away with the canonical scope delimiters in Python, relying instead on indentation for this purpose. What could possibly be of such high value that a syntax designer would brave the controversy this caused? A high-value pair of matching glyphs, { and }, for other use in his syntax could. (This decision also made it impossible to write Fortran programs in Python, a laudable achievement in its own right.)

The best example of what happens if you do the opposite is John Ousterhout's Tcl programming language. Despite all its desirable properties—such as being created as a language to be embedded in tools—it has been widely spurned, often with arguments about excessive use of, or difficult-to-figure-out placement of, {} and [].

My disappointment with Rob Pike's Go language is that the rest of the world has moved on from ASCII, but he did not. Why keep trying to cram an expressive syntax into the straitjacket of the 95 glyphs of ASCII when Unicode has been the new black for most of the past decade?

Unicode has the entire gamut of Greek letters, mathematical and technical symbols, brackets, brockets, sprockets, and weird and wonderful glyphs such as "Dentistry symbol light down and

horizontal with wave" (0x23c7). Why do we still have to name variables OmegaZero when our computers now know how to render 0x03a9+0x2080 properly?

The most recent programming language syntax development that had anything to do with character sets apart from ASCII was when the ISO-C standard committee adopted trigraphs to make it possible to enter C source code on computers that do not even have ASCII's 95 characters available—a bold and decisive step in the wrong direction.

While we are at it, have you noticed that screens are getting wider and wider these days, and that today's text processing programs have absolutely no problem with multiple columns, insert displays, and hanging enclosures being placed in that space?

But programs are still decisively vertical, to the point of being horizontally challenged. Why can't we pull minor scopes and subroutines out in that right-hand space and thus make them supportive to the understanding of the main body of code?

And need I remind anybody that you cannot buy a monochrome screen anymore? Syntax-coloring editors are the default. Why not make color part of the syntax? Why not tell the compiler about protected code regions by putting them on a framed light gray background? Or provide hints about likely and unlikely code paths with a green or red background tint?

For some reason computer people are so conservative that we still find it more uncompromisingly important for our source code to be compatible with a Teletype ASR-33 terminal and its 1963-vintage ASCII table than it is for us to be able to express our intentions clearly.

And, yes, me too: I wrote this in vi(1), which is why the article does not have all the fancy Unicode glyphs in the first place. Q

REFERENCE

1. Pike, R. 2000. Systems software research is irrelevant; http://herpolhode.com/rob/utah2000.pdf.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

POUL-HENNING KAMP (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikeshed.org. His software has been widely adopted as "under the hood" building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large Web sites such as Facebook.