Computing history and its scars

Ben Klemens

19 October 2009

This history of computing is a catalog of scars. Glitches that software ran into when it was around ten years old still leave creaks and aches today.

Punch cards We'll start with punch cards, because that's what they had when FORTRAN was developed in the mid-1950s.

FORTRAN code, going by the 1977 standard, is based on the format of a punch card: the first six columns are reseved for labels, and a mark in the seventh indicates a continuation from last line, which is necessary because anything after the 72nd character in a line is ignored, because that's how wide punch cards were circa the 1950s. FORTRAN later dropped these requirements, but here in 2009, I still deal with active, working code that is bound by the '77 standard for conforming with paper cards first standardized in 1928.

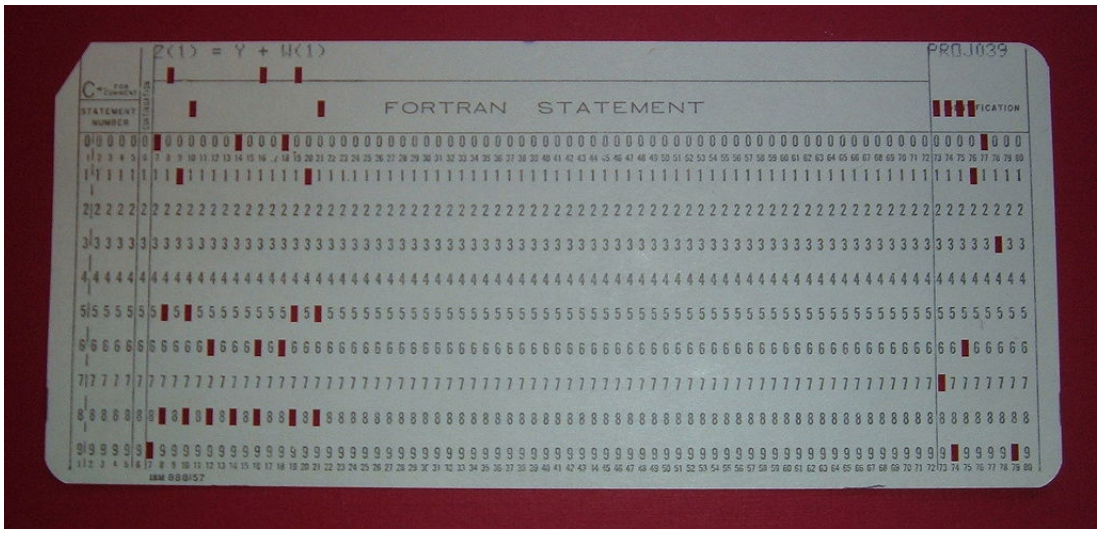

Figure 1: I still need to follow this format when I work on our 64-processor server.

Input via cards is divided into decks. You have a declaration deck, listing variables

and registers which will be used. Another deck initializes constants. Each subroutine will be another deck. Decks should remain as independent as possible.

You can tell a punch card language by this division into decks. COBOL is the big winner on this, with a number of decks that are basically required for every program: identification division, environment division, input-output section, data division, file section, working-storage section, procedure division, and these section titles alone are already longer than a lot of Perl programs.

As noted, modern Fortran really looks nothing like FORTRAN from the mid-century, although Fortran users do still have an annoying custom of writing their code in all caps. The division into decks, however, is still basically there. SAS, for those of you who have the $$$ to use it, hasn't changed at all in its rhythm, requiring that every thought be broken down into a separate deck: CARD ... END CARD; DATA ... END DATA; PROC ... RUN; PROC ... RUN.

Teletype Let us step forward to the mid-60s, and the teletype (aka, the line printer). The teletype's unit of analysis, comparable to the punch card, is the line of text. Bell Labs evidently had a run on these things, because in the early 1970s, they solidified a lot of what we use today, notably C and UNIX, with the teletype interface in mind.

C and UNIX are heavily dependent on plain U.S. English text, divided into reasonably short lines. UNIX has a large number of programs that take in a line of text, find some patterns, make some replacements, delete some lines, and so on; C is built to facilitate this. Here in the modern day, it's still something of a pain to use UNIX utilities to find patterns that span multiple lines.

Programming languages, more than anything, are written so that programmers could make their own jobs easier. Once that part is in place, our programmers may then move on to producing something actually productive.

Much of the paradigm can thus be traced back to the process of writing a compiler for yet another programming language. This involves taking in lines of text using a standard U.S. English alphabet, then recognizing certain patterns, and then outputting some other embodiment of the instructions in that text. For a C compiler, the output would be assembly language for the computer hardware; for any other language here in the modern day, it's probably C code, meaning that you've converted lines of input text in one language into lines of output in another language.

So the tools that grew up fastest in this era are those that filter text to produce other text. They find text patterns and replace them with other patterns, or take an action given some pattern.

Me, I still write in C. It's the first language that is still in very common use today (see, e.g., http://www.langpop.com/[1]). You can see that I find the central importance of newlines to be annoying, but without the structural constraints of punch card decks, and the added ability to define new structures (which Fortran '77 didn't have), you have just about the simplest language in which you can get serious work done. Fundamentally, the process of writing code is still the process of writing plain English text, and the people with teletypes were the first ones who could do that comfortably.

[1] `http://www.langpop.com/`

TV screens But technology marches on, and next up are cathode ray tubes, graphical user interfaces, and window systems. In terms of typing program code into a text editor, not much changed, but outputting those windows is a non-trivial problem, which can be a mess if you're not organized. When a user clicks the button on the mouse, the computer has to be waiting for it, and has to know to send that signal to the program being pointed to, which has to know if it should send the signal to a button or a window border or a text box, and behave accordingly.

Thus, object-oriented programming, which in its modern form is built around accommodating the text-in-box-in-window-on-desktop sort of hierarchy, managing the demands of multiple windows all wanting to do something simultaneously, and letting a click mean different things in different places.

My impression of coding in the 1990s is that it dove in head-first into an object-oriented paradigm. [Not *the* paradigm, because there are other ways to do OO that are very different. For example, your typical C++ OO code [by no means all] has little interest in message-passing between objects.] Nor is OO really an innovation per se, because the ideas existed before. But because of the interest in drawing windows and buttons, objects kinda took over.

Java is the main scar from this period: everything in Java is an object, which still bothers some[2]. After all, the OO setup is perfect for windowing setups, but is not necessarily as applicable for everything else. Is your accounting system or your text editor really best written via exclusive use of a set of object linked via a strict hierarchy? The answer is up for debate, but the influence of windowing technology is such that these object design patterns showed up in accounting programs anyway. Features were taped on to lots of languages, including Fortran and COBOL and Lisp, to include the 1990s idea of how OO should be done.

Fun fact: the Gnome system, which is the desktop on half the world's Linux boxes (that have desktops) is written in plain C, not the object-centric C++. [C was once compilable as C++, but the two are no longer at all mutually intelligible. Using the string *C/C++* is a surefire way to get geeks to taunt you.] I can't guess as to the motivations, but I take this as a pendulum swing back from complex object structures. It's also proof that the whole OO thing didn't really have to happen, and in an alternate universe, windowing systems didn't lead to a trend where every language bolted on an OO syntax.

Network The latest physical technology innovation is the Web. The original language of the Web was HTML, which has done an interesting job of imposing itself on coding in general. The first few drafts of HTML were pretty directly oriented toward putting text on the screen, plus some tags to format the text so it'll blink or turn puce when you click on it.

HTML's structure has since been used for the expression of all sorts of attributes for all sorts of data, via extensible markup language, XML. [HTML is itself a specific case of a broad precursor class of markup, SGML. XML falls somewhere in between.] XML is verbose, and is hard to search, and has lots of little details that make it potentially hard to parse. It requires multiple files to get anything done, because before you get to the part where you describe what's going on, you need another description of the elements in your

[2] `http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html`

data classification system, and of course, the hierarchy of objects that those elements fall into.

But it is Web-friendly. The trouble with writing for the 'Net is that you can quickly get caught up in duct tape. To write something functional, you've got JavaScript (a basically C-like language) sending requests to a server (probably running a scripting language from the 90s like Perl or Python), which might talk to a database sytem running its own ad hoc language, and linking all this so data flows correctly is far from trivial. XML lets you store your data, your form templates, and even the final output, in the same format.

So every language out there has an XML parser, which turns XML into a structure that your program can actually use. Yes, I have an accounting program that stores all its data in XML.

[As a digression, JavaScript and Ruby, two web-friendly languages, do have one other nice feature: they make it very easy to write functions that make use of little functions to do simple tasks. If you want all the elements of an array, the elements being x_i, to each turn into $f(x_i)$, then it's easy to make that happen in one line of code. Like OO, this is nothing new (you can do it in C), but a way of thinking that the language can faciliate. It has an especially strong history in Lisp, which emerged a year or two after FORTRAN. But it still seems to be trendy these days.]

That's all I could think of in terms of what the Web has given to the problem of coding, and it's not the biggest deal. But, like OO, it's now a requisite for any language to have legs: how quickly can I use this to output a XML-compliant web page? In 2078, our systems will still have to be able to read XML, for compatibility with legacy systems written with web standards circa 2006 in mind.