Old Machines Running Old Languages

by Gabriel Egan - 03 Aug 2015

The Minimal Computing Lab at the Centre for Textual Studies (CTS) at De Montfort University (Leicester, England) was created to help students of arts and humanities subjects to understand how computers work, with a special focus on how they can store and process writing. Almost all young people quite literally use computers all day long, and yet almost all have virtually no idea how these machines work at the most fundamental level and would never consider programming one for themselves. To introduce students to how computers work and how to program them, modern computers–even stripped-down ones such as the Raspberry Pi–are much too complicated. Before one can use a Windows or Macintosh computer or a Pi to teach a modern, simplified programming language such as Scratch, one has to 'boot' the computer to load an operating system of bewildering complexity. If a student asks "What is happening now?" as the machine boots, we have to answer "It is complicated, so let us not discuss it now: wait until we get to the simple Scratch environment". This, I submit, is a highly unsatisfactory answer. We have to just pretend not to notice all the complexity that is needed as a prior step to reaching the simple programming environment, the simplicity of which is entirely false since it depends on an underlying complexity that it exists to conceal.

Why Old Computers?

We need a genuinely simple approach to programming, and for that we need genuinely simple computers. The Minimal Computer Lab uses two kinds: the Altair 8800 and the Amstrad PCW. These machines have no hard drives, no built-in operating systems, and no Read-Only-Memory (ROM). When such a machine is switched on it has empty memory and a processor doing nothing. The first task is to get something into the machine's memory (and thence the processor) to start a 'boot' procedure. With the Altair 8800 that requires toggling switches on the front panel to force into memory a dozen or two bytes of code that enable the machine to 'boot' from its mass-storage medium, in our case a paper-tape punch/reader. With the Amstrad PCW it means inserting a floppy-disk: the bare machine has just enough built-in code–a tiny bit of pseudo-ROM in the printer controller–to read and execute the contents of the first sector on the first floppy disk.

In the Lab we make much use of paper tape because it renders readily apparent the fact that digital text *is* (despite all preconceptions) stored on a physical medium: there is nothing virtual about it. In other modern media the zeroes and ones are very small (say microscopic pits on a metal surface) and/or stored in a form such as magnetism that our eyes cannot detect. With paper-tape, the zeroes and ones can be read directly off the tape by the naked eye, and one of the first tasks that students are given is to manually decode the binary ASCII of a strip of paper-tape containing a famous quotation. Once the students understand the essential architecture of the machine, they begin programming in the language BASIC. This choice of language has been controversial in some quarters, and the Lab director is often asked why we do not teach a 'real' programming language that the students can use elsewhere, such as C or Python. Or the director is told "Teach

them something they can use to program their smartphones" because that will catch their interest.

Why BASIC?

Let us compare how the ubiquitous "Hello, World!" program looks in C, Python, and BASIC. In C (depending on the installation), it might be:

```
#include <stdio.h>
main()
{
printf("Hello, World!");
```

If you already know quite a bit about computers, this makes perfect sense. Of course you need to include a least one standard library when your program is compiled. ("Compiled?", asks the student; "Yes, hold on, we will come to that later".) And of course there has to be a 'main' section to every program. And the curly and round braces? Well, you just need those, they are required by the rules. At this point, knowledgeable readers may be thinking "but the braces too can be explained quite logically". Indeed they can. But can they be explained in a way that makes sense to someone who is at the level of learning about the "Hello, World!" program? I have not found a way to do this.

What about "Hello, World!" in Python (version 3, the latest)? This is much better:

```
print("Hello, World!")
```

But we still have those braces. Why? (And, if you have been following the development of Python, how come we did not need them in Python version 2?) The answer is that in Python (version 3), `print` is not a statement–in common parlance, an imperative to 'do this'–but is instead a function. Being a function, Python's `print` is an intellectual construct that takes an argument (the string "Hello, World!") as a specific instance of a generic object that it has been programmed to work upon in a certain way. This presents a significant pedagogical problem, since in order to explain how the simplest program ever devised works, we must (if we use Python) first explain the tricky concept of a programmable function.

In BASIC, of course, our "Hello, World!" program is just:

```
10 PRINT "Hello, World!"
```

The only bit of computer code is a simple imperative ("do this!") and, as in English, the words that are the object of a verb about a piece of language are in quotation marks. If you already understand how to make sense of (to parse) this line from a fictional narrative work

> She said "Go! Leave me!"

then you already know enough to make sense of BASIC's "Hello, World!" program.

But what about that number "10"? Surely that is an unfortunate part of BASIC's underlying design philosophy that we do not want students to have to understand, since as the referents of branch instructions a program's line numbers tell the reader nothing about the instructions to which control is

being passed. Thus, instead of "GOSUB 1000" " we would prefer students to learn to label parts of their programs by what they do, as in "GOSUB Get_Age".

In fact, as the referents of jumps, numbers are at least as familiar to novices as alphabetic labels are. "Open your books at page 32 and begin reading", "Let us start again from the top of the second movement", "Act Three, Scene Two, everybody!". By contrast, there are few precedents by which a novice might understand that an alphabetic label can work as precisely as a numeric one. Indeed, the implied codes in the following printed page are entirely familiar to students:

> 2.2 *The Tragedy of Hamlet*
>
> controversy. There was, for a while, no money bid for argument, unless the poet and the player went to cuffs in the question. 350
>
> HAMLET Is't possible?
>
> GUILDENSTERN O, there has been much throwing about of brains.

Every line of a play that is being studied seriously has a line number–although publishers usually print the number only for every 5th or 10th line–and it is part of a generalized numbering system incorporating the larger units of acts and scenes. Even quite weak students are comfortable with the requirement to direct their attention to line 350 and to reference a quotation of it using its wider context as "2.2.350". We are already not a million miles away from the use of dotted quads as Internet Protocol addresses.

If we want to build on what students already understand, line numbering in programs is fine. And if they go on to advanced programming, line numbers will return in many contexts, since serious text editors for programming and XML coding provide them and computer error messages often quote them. Moreover, a line number is a convenient analogy for the notion of 'addresses' as labels by which we refer to computer memory locations. Just as we say in respect of our BASIC program that "line 10 contains the instruction to . . .", we may say when introducing machine code programming that "memory location 0000 contains the instruction to . . .".

Beyond BASIC, and Still Minimal

The Altair 8800 and Amstrad PCW computers of the CTS's Minimal Computing Lab enable students with no computational experience whatsoever–especially arts and humanities students who may think of themselves as utterly non-technical–to learn about and try their hand at algorithmic thinking. As a number of commentators have observed, those with any practical experience of algorithmic thinking–those who have tried breaking a process into its fundamental, often repetitive, operations–are by this experience (and especially by their instructive failures) set apart from those with no experience of this approach to problem solving. In the CTS we focus our teaching on language problems. We are a Centre for *Textual* Studies and our students are already highly skilled in expressing ideas in words. But they are, on the whole, uncomfortable with, if not wholly allergic to, expressing ideas using numbers. We use BASIC to show how a machine can perform such useful tasks as alphabetically sorting a list of words or counting the frequencies of various features of a piece of writing such as the rates of function word usage or punctuation, or the commonest lengths of words,

sentences, or paragraphs. When used to quantify features of language, rather than say sine functions or the Fibonacci series (examples common in many programming primers), numbers are least terrifying to our students.

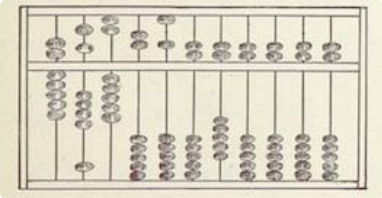
At some point, BASIC and the Altair 8800 or the Amstrad PCW are inevitably not powerful enough for the students' ambitions. With even a relatively short piece of text such as Chapter One of Jane Austen's novel *Persuasion*, a word-frequency analysis running in BASIC might take days to complete. (We can mitigate this by using our Teletype machine as the console and leaving it on all week; the results will be captured on paper even if they appear at 3am on a Tuesday, and cannot 'scroll' off the screen as with a VDU.) Moreover, although we might just be able to squeeze the whole of *Persuasion* into the computer's memory at one time, we are not going to be able to also squeeze in a few of Charles Dickens's novels as well for comparison. The limitations of these machines soon become apparent to students in their project-work.

For faster-running programs, the Altair 8800 and Amstrad PCW may be programmed by more advanced students using machine code instructions or assembly language. This dispels the last of the mysteries about how the machine operates by revealing the fundamental level of logic gate operation, busses, and clocks. Students who crave still more power may use the university's many labs filled with modern computers, which no longer seem to them to be operating by magic but are revealed to be simply faster, more capacious versions of the machines whose fundamental operations the students now understand.

For the problem of comparing multiple literary texts, the limitations of the Altair 8800 and Amstrad PCW are blessings in disguise. When one teaches computational stylistics in a lab of modern computers, students can go a very long way before they reach the limitations of the machines' capacity to store and process texts. Having counted the word frequencies across all of *Persuasion* it is trivial to rewrite the program to do the same for all of Austen's novels, and then all Dickens's too. In the students' minds, the problem of scale is, at least for a while, deferred by today's machines' prodigious memory and mass storage capacities. But the problem of scale will nonetheless at some point return. If one tries to initialize a memory array to hold all the words of all the eighteenth- and nineteenth-century novels currently available in digital form–totalling several billion words–even the latest computers run out of memory space. An algorithm that worked for a few novels will not work for hundreds of novels considered at once.

With the Altair 8800 and the Amstrad PCW this limitation of scale is reached much earlier in a project and the students have to learn the methods for overcoming it. If they cannot fit even the whole of *Persuasion* into their data structure at one time, they have to learn how we analyse texts in smaller chunks, compiling the results for Chapter One before clearing the data structure to work on Chapter Two, and so on, and collating all the results once the final chapter has been processed, That is, the use of Minimal Computing resources forces attention onto the inevitable problems of scale right from the start, and teaches students not to assume that everything can be solved by just adding more memory and mass storage. Instead, they learn to rethink their algorithms so that when scaled up to work on more texts they take longer to execute but do not employ substantially greater resources of memory or mass storage. In other words, the principles of Minimal Computing are an excellent introduction, for those who wish to pursue the topic, to the principles of Supercomputing.

Minimal Computing



a working group of GO::DH

About
News & Announcements
Thought Pieces
Links & Resources
Mailing List
People

Contribute

« No Connect