An Introduction to Programs
Professor Richard Harvey FBCS

2 February 2021

There is a video of Steve Balmer, the former CEO of Microsoft leaping about on stage at the 2000 Windows Developers conference shouting the word “developers” fourteen times in a row. The precise psychology of Steve Balmer is not the subject of this lecture but what is interesting is the reverence and enthusiasm for programmers. After all, as we have seen in previous lectures in this series, before one gets to programming one needs to design an algorithm and a data-structure and, frankly once one has chosen those things, the business of “cutting code” is a little more mundane. So, what is this thing called programming?

Computing machines have a history that spans over centuries but if we confine our attention to general purpose programmable computers that actually worked the two most notable early examples were Babbage’s Analytical engine and the Zuse Z4. As soon as one designs a programmable machine, there arises the challenge of constructing a set of instructions that tell the computer what to do. Babbage planned that his computer should receive instructions via punched cards which were also to be found on the Jacquard loom, but the discovery of programming is often assigned to Ada Lovelace who, in the process of translating a French description of lecture by Babbage given in Italy, realised that she needed to fill in the gaps with a series of notes on how programming might work. She did not construct a programming language, however. Konrad Zuse however did formalise a way of writing down formulae in a way that mapped conveniently onto his Z4 computer. I think it’s fair to say neither Lovelace nor Zuse constructed a real programming language and the reason for that is that they had no hardware to program (the analytical engine was never built and the Z4 took a long time to come to fruition and there is no evidence that *Plankalkül* was implemented).

So, programming did not really come to practical reality until the 1950s. The earliest machines were programmed using a set of switches or hard-wired using patch cables. Each low-level command had a unique binary code. The programmer set the switches, or the plugboard, pressed a button to increment the program memory and repeated the process until the program was entered. Given that instructions were in “machine code” they were extremely basic (a typical machine code instruction might be to move the contents of this memory address into a section of memory on the computer), useful programs were lengthy, and the business of programming was troublesome. The programmer might start with a flow-chart of what they hoped to program, then after hours of careful work they would have a list of instructions which that had probably tested by hand (early machines having no debugging capabilities). These would then be entered via switches, or as technology improved on punched cards, paper tape or magnetic tape. The program would be run. The machine would stop, without giving any indication of why it stopped, and the programming would spend hours scratching her head trying to spot the error (the early programming profession was large dominated by women).

This was widely acknowledged[1] to be pretty problematic process but what to do? As usual with technology, several people came to similar conclusions at the same time, but one of the key characters was Grace Hopper. Hopper's idea was to use a "high-level" language. That way the programmer could write instructions in something that looked a bit closer to algebra and the machine would translate it to machine code. This was still quite an intricate process and usually involved a chain of programs. The first would "compile" the high-level code to an intermediate level (object code) and this would then be "linked" to the particular hardware available on that particular computer and then the code could be run. Nowadays we do not usually bother with the linguistic precision of mentioning the linker and assembler stages — we call it compilation and the program that coverts code into zeros and ones is called a compiler.

Writing compilers is one of the standard parts of a computer science course. Compilers usually have two components: there is the part that processes the simple grammar associated with the computer programming language and then there is the part that follows the rules. As you might have guessed, there are compilers for writing compilers - one of the most common is called yacc (which is a little computer science joke - it stands for Yet Another Compiler Compiler to indicate the authors exasperation with computer scientists writing compilers). Compilers for new languages are fun to write but very few problems are directly solved by writing compilers, indeed often a new language makes things whole lot more painful.

From Hopper's early ideas spawned thousands of programming languages. Coming from the rational world of computer science, I find myself somewhat at a loss to explain this irrational plethora of programming languages but there is an entertaining lecture by Simon Peyton-Jones, who was a main mover behind a language called Haskell, in which he points out that most programming languages are designed and used by one person, often a lone graduate student who has designed a language to prove some arcane point. Only a few languages survive in what Peyton-Jones calls a "regrettable absence of death." It is these languages which clutter up the bookshelves of bookshops[2].

As with real languages, measuring the popularity or usage of programming languages is a tricky statistical exercise. Should we measure only people who are using that language on a daily basis? (the equivalent of fluency) or should we measure anyone who has vague knowledge of a language? There are two approaches in common use. One is to measure the amount of interest in a language via the search engines, this is the Tiobe ranking approach [1], the other is to augment this with number of downloads from popular repositories as in the Redmonk rankings [2]. A quick glance at the rankings shows considerable perturbation in or around 2018, which was when GitHub was acquired by Microsoft, but ignoring those blips there is a fair amount of stability in the top programming languages of all time. Top of the pops, probably, is Java but it is falling in popularity. Next comes a batch of languages: C, C++, Java, PHP and Python. Big risers over the last couple of years are Typescript (a special variant of Javascript that uses *static typing* (see later)) and Kotlin. Kotlin is now the Google-recommended language for Android development so its rise is hardly surprising. Just as with human languages there are heated debates about the difference between a language and a dialect[3].

So, what does a program look like? I have a little quiz for you in Figure 1 which shows seven examples of what is known as the "Hello World!" program – the simplest program one could write in a particular language. When learning a new language, one usually starts with getting set-up to write a "Hello World!" program. And the time one spends faffing around installing the compiler, trying to

[1] Well, *narrowly* acknowledged since there were few people who had programmed a computer!
[2] I did a quick search on amazon.co.uk and found around 10,000 books on C++. Given how horrible C++ is, maybe this is not a surprise but even a less rebarbative language such as Swift generates 2000 returns.
[3] The linguist Max Weinreich is well known for the quote "A language is dialect with an army and navy" which expresses the futility of linguistic arguments in the face of such debates.

make the IDE work and wrestling with various security settings on the operating system which are designed to stop naïve users from damaging themselves … that time is the “Time to Hello World!” If you want your language to be adopted, then the time to hello world! should be minutes rather than hours!

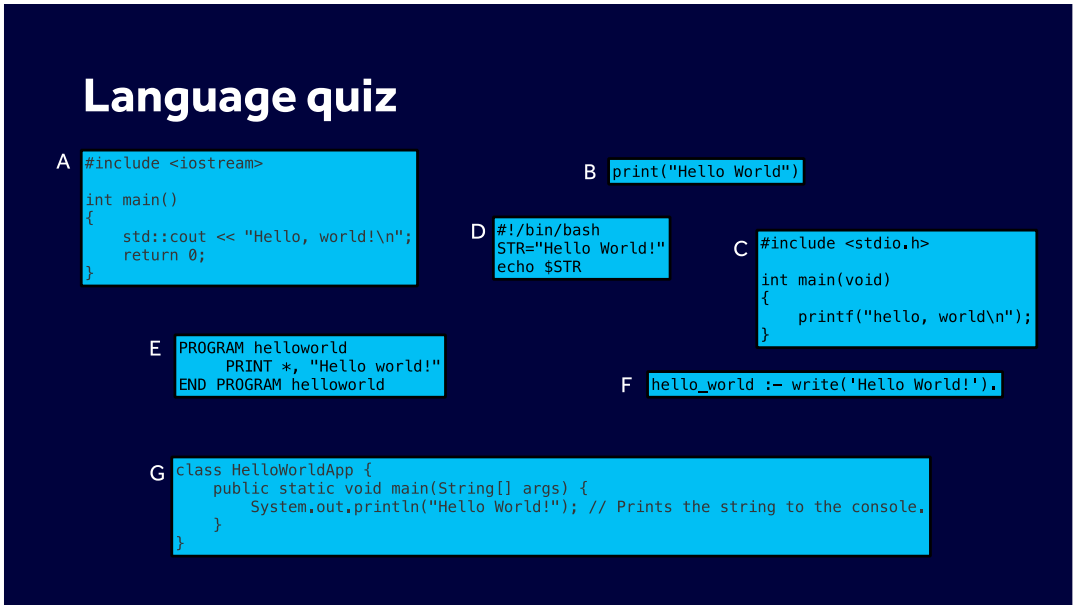

Figure 1: *“Hello World!” programs in a variety of languages. For those of you who enjoy a quiz, turn to the end of the transcript for the answers.*

Notwithstanding the variety of programming languages, there is considerable commonality in a lot of them.

```
//
//  main.swift
//  gresham_example
//
//  Created by Richard Harvey on 21/01/2021.
//

import Foundation

func factorial(_ n: Int) -> Int {
    if n < 1 {
        return 1
    } else {
        return n*factorial(n-1)
    }
}

print("Hello, World!")
print(factorial (13))
```

Figure 2: *Example program in Swift to compute a factorial.*

The start of the program file is usually some commentary or comments. These are meant to be written to be read by the next programmer to encounter my inferior efforts at programming in Swift. This example is particularly egregious since the comments tell one very little. Then comes some preamble which is usually instruction to include some libraries. In this case I am including the

Foundation library which tells Swift how this particular computer represents integers, floats and so on. Then comes definition of functions. As in the list of symbols you often find at the start of textbook or the list of definitions in a contract, these are often at the front, although they need not be. Finally, we reach the program itself which in this case consists of two lines that print things to the console. It is often the case that the main program is rather simple, and the main action are to be found in the functions. Functions have been a recurring topic of some interest to compiler designers. Functions are popular because, as with mathematicians, so long as one agrees definitions, one can share functions between developers, and it keeps the main part of the program free from confusing detail.

In Figure 2, the function is a little unusual is it appears to be defined in terms of itself. Factorial(n) returns 1 if n is less than 1, otherwise it returns n*factorial(n-1) – a recursive definition. When compiling this code, a typical compiler would create some assembly code, the low-level code that the computer understands, that *accepts* one integer argument, n, the algorithm does its stuff and *returns* another integer[4]. At run-time the value assigned to n is stored in an agreed place, called the *stack*. If the function has multiple arguments, then they are stored on the stack in the order they occur. The function then "pops the stack" to retrieve all the values and does the computation. It pushes the returned value on the stack and the main program pops the stack to get the answer. So long as there is common agreement on the location and format of the stack, then this mechanism can allow your function to inter-operate with program fragments written by others, often using different languages. In our example, the function is recursive, so the stack gets the integer 13 pushed onto it, then factorial (12) pops 12 on the stack and so on. Thus, in recursive functions, especially those with many arguments, the stack can grow huge (a stack overflow error). Worse, the capabilities of the stack depend on the particular machine – code that runs perfectly on the developer's machine fails in real-life[5].

So far, I have been discussing a type of function calling called *call-by-value* in which the arguments to the function must be evaluated before being pushed onto the stack. An alternative is *call-by-name* in which the function is passed the thing that has to be computed. The compiler in effect unwraps the function calls and only evaluates them when a value is substituted – this *lazy evaluation* strategy has some advantages not least of which is the security of only evaluating something when it is absolutely needed.

In Figure 2, I declared that the argument to the function, n, was an Int. We say n is of *type* Int. In this language, Swift, we have declared the types, so the compiler can check that the types match at compilation time. This is helpful as it is a common source of error to munge together mixed types. However, it can lead to cluttered code as one is constantly having to change types. Dynamically typed languages, such as Python, change the type of the variable as needed. Thus,

```
x=100
y = 0
y = x/3
```

is legitimate and would give the answer 33.3 but if I now write

[4] There is an obvious bug in this code which is that the factorial function is notorious for creating very large numbers for even small arguments – on my machine this code overflows at n = 21. It is naff to write code that suddenly halts in a machine-dependent way if the arguments to the function are too large. The business of trapping errors caused by you or another developer using the code in the "wrong" way is another discussion point.

[5] In practice most respectable operating systems set an error flag when they run out of stack memory. The programmer is meant to write a special bit of code, called an error handler, which when an error occurs, does something helpful. Even more defensive is to run some tests on the hardware before the progam runs to check the extent of the stack. Of course, such tests take time, and users commonly moan about boot-up time … well there are solutions but, my point is, that there are trade-offs to be made between robust code and usability.

x = “Richard”
y = x/2

then we will have an error, as division is not defined for strings. Depending on your psychological outlook this either makes Python the equivalent of changing out of prison clothes and into your favourite striped loon-pants or it is the equivalent of drinking absinthe – liable to make you go crazy.

You will also refer to the phrases *strongly-* or *weakly-typed* languages. These are not very well-defined terms, but they refer to whether the language insists on all variables having a defined type and there is no implicit conversion between types. Some languages check types at compile time, others at run-time (they are compiled with additional code that does the checking) and others not at all. As with static and dynamic typing it is largely a matter of psychological outlook whether one approves or not.

Here I have only skimmed on the categorisations of various programming languages. The Wikipedia article on programming languages by type [3] identifies 50 different headings including the delightful “Esoteric category”[6]. One interesting group of languages compiles to hardware. The variables are usually connectors and states and the language describes how they are to be connected and the contents of various bits of memory. Compilation involves producing a netlist which is then simulated. Once happy the engineer can either blow a chip on the desk using a technology called Field Programmable Gate Arrays (FPGAs) or via a silicon foundry who, at massively more expense, will build an Application Specific Integrated Circuit or ASIC.

Returning to more conventional languages most programmers settle into a few languages that they use frequently and, just as there is a lot of similarity between English and Romance or Germanic languages, there is enough similarity that one can move fairly swiftly between them. That said, there are various thought-models for programs which, for a new-comer can be daunting. So far, I have been talking about the model known as imperative programming[7]. If one looks at this illustration of FLOW-MATIC, one of the earliest programming languages (Figure 3)

```
(1)   Δ SUMD Δ ← Δ 0 Δ .
(1A)  Δ SUMD2 Δ = Δ 0 Δ .
(1B)  Δ N Δ = Δ 0 Δ .
(2)   Δ READ Δ OD Δ ID Δ WEIGHT Δ THICK Δ IF Δ SENTINEL Δ JUMP Δ TO Δ SENTENCE Δ
      13 Δ .
(2A)  Δ N Δ = Δ N + 1 Δ .
(3)   Δ DEN Δ = Δ WEIGHT / (π/4 ★ (OD² − ID²) ★ THICK) Δ .
(4)   Δ WRITE Δ DEN Δ .
(5)   Δ WRITE Δ EDIT Δ N Δ DEN Δ .
(6)   Δ SUMD Δ = Δ SUMD + DEN Δ .
(7)   Δ SUMD2 Δ = Δ SUMD2 + DEN² Δ .
(7A)  Δ IF Δ N Δ = Δ 19 Δ JUMP Δ TO Δ SENTENCE Δ 9 Δ .
(8)   Δ JUMP Δ TO Δ SENTENCE Δ 2 Δ .
(9)   Δ AVERAGE Δ = Δ SUMD/19 Δ .
(10)  Δ STANDDEV Δ = Δ SQR Δ ((SUMD2-(SUMD)²/19)/18) Δ .
(11)  Δ PRINT-OUT Δ AVERAGE Δ STANDDEV Δ .
(12)  Δ JUMP Δ TO Δ SENTENCE Δ 1 Δ .
(13)  Δ STOP Δ .
```

Figure 3: *Early program written in FLOW-MATIC*

one can see a set of instructions which amount to “do this”, then “do that”. At line 7A there is a “if this then do that, otherwise do that” structure – a conditional branch as we call it. When we look at Figure 3 it is obvious, and precise, what is happening in each step, but the overall picture is

[6] I commend the language SPL (Shakespeare Programming Language) to our readers in which the program is disguised as a Shakespearean play. As with Shakespearean plays, it is verbose.
[7] In the parlance these are often called programming paradigms – the word paradigm being used wrongly as it in [12]. However, to not appear too grouchy about this, I confine my linguistic grumbles to a small footnote.

completely lost. Can you look at Figure 3 and spot that it is computing the density of various types of wheel? If we compare the program to the algorithm, or even the mathematics, then it is an awkward business. It seems that imperative programming has caused us to lose the bigger picture which was clear sight of the problem we are trying to solve. An alternative style of model of programming is known as declarative programming[8].

My explanation of declarative programming involves a meeting between two former Gresham Professors, Sir Christopher Wren and Sir Samuel Pepys. On 21st Feb 1661, Pepys was at Trinity House which in those days was situated on a street called Water Lane near the Thames. Wren was at Gresham College, which was on Broad Street, on the site now occupied by Tower 42. Should Wren wished to instruct Pepys how to arrive at Gresham College then he could have issue instructions which, according to my map of London dating from 1676 would be as follow: Walk up Water Lane, turn left on Tower St, turn right on Idle lane, turn left on Little East Cheap, turn right on Grace Church St, Continuous onto Gratious St, Continue onto Bishops Gate St, turn left into the back entrance of Gresham College (opposite Great St Helens Churchyard). It is not an attractive way to issue directions and Wren is far more likely to have said, come to Gresham College. In my imagination Wren favours the declarative style – he merely declares what he wishes to be true (that Pepys should be at Gresham College) and the system takes care of the rest. To see how this works in practice here is some simple Javascript[9] to square every element of an array:

```
let z = []
let n = [-1, 1676, 42, 5, 10, -3]
let SquareUp = x => x * x
for(let i = 0; i < n.length; i++) {
    z[i] = SquareUp(n[i])
}
```

and here is the same code written declaratively:

```
let n = [-1, 1676, 42, 5, 10, -3]
let z = n.map(v => v * v
```

The map tells the compiler to apply that thing across the array (don't care how you do it – just do it). Declarative programs can be a bit creepy to those of us brought-up on imperative programming (how do I know the machine is doing this efficiently?) but declarative programs are shorter and easier to follow[10].

At this point I think we should take a brief diversion and ponder some of the prominent computer scientists and their views on programming. There is a long and distinguished history of famous computer scientists being rude about other programmers. Here for example, is Edsger Dijsktra letting rip about COBOL which was an early business programming language "Teaching COBOL ought to be regarded as a criminal act[11]." And here is Niklaus Wirth "C++ is an insult to the human brain". While I have some sympathy with Wirth's views on C++ the fact is that this is not science – it is a set of literary opinions masquerading as science. To get to the bottom of the problem I think it is helpful to consider the context in which software development takes place.

[8] Actually, there are programming languages that allow one to mix the procedural style with the declarative of which Javascript is probably the best known.
[9] These examples are modified from https://dev.to/adnanbabakan/declarative-programming-with-javascript-2h97
[10] To be perfectly fair, I should say that the proponents of declarative programs claim that they are easier to follow.
[11] Dijkstra is probably most famous among lay readers for his note "GOTO Consider Harmful". In this note Dijkstra points out, quite correctly, the problem with the GOTO statement is that it can act like an untraceable jump into the middle of the code. So influential was this paper that has spawned a family of "X considered harmful" papers.

The first observation is that programming is only a part of what is now known as development which itself is only a part of software engineering[12] which includes requirements engineering, specification, design, development, testing, maintenance and documentation. Systems and processes which support this activity are nowadays called DevOps. DevOps which is the combination of development and IT Operations is not yet a well-defined term and, as is so often in software development, is regarded by some a slightly faddish methodology of the type that was popular in business in the 1970s (Lean manufacturing and so on) but I am referring to software tools and processes which make development easier. There is an enormous list of these tools and many of them are highly specialised to particular computer architectures or application domains, but one area that deserves wider recognition is source code control. The most widely known system at the moment is called git and was devised by Linus Torvalds the main developer of the Linux operating system[13] but, more generally such systems are called version control systems. When a large number of programmers are working on a project, such as the Linux operating system, keeping track of the changes and accepting working changes into the final source code is a monstrous task. A developer "checks out" a version of the software that they wish to work on, they make their alterations and check it back in again. If their alterations are approved, then their changes are incorporated into a master version. At any point the source code can be snap-shotted to create a release or, if mistakes are made, one can roll-back to a previous version. Two people can be prevented on working on the same part at once by "locking" mechanisms. Some of the ideas can now be found in collaborative word processors such as word, Google docs and Pages but full version control is much more disciplined and Computer Scientists long for wider adoption of version control[14].

Of course, in some ways, software engineering is just like engineering: there are large engineering teams working to an agreed design and there is a customer who is paying the bills. The user, who is often not the customer, has views that may or may not be relevant and each one of these "actors", as they are called in use-case analyses, has a completely different perspective as to success. The customer wants a product that meets all the apexes of the quality triangle: quality; time and cost. The design team know that there are trade-offs to be made: we can have a cheap low-quality system delivered on time, or an expensive high-quality produce that over-runs or sometimes the worst of all worlds: the expensive, low-quality produce that is delivered late. So far, I have said nothing that is not true of all engineering projects. However, software has a great deal of difference from physical engineering projects of which I would adduce:

1. Software can be distributed around the world almost instantaneously and at almost no cost. Thus, however niche the application, it will find users almost instantly.
2. Much software is constructed for free meaning either it is very expensive but given away free (as with your phone's operating system) or it is constructed by charitable efforts or sponsored efforts. This has set the expectation that software development is cheap.
3. Software is virtual – I cannot examine a piece of software and know immediately if it fits.
4. Software has a lot of interdependencies which are hard to manage and tend to cause rapid and fragile failure rather than some progressive decrease in performance.
5. Software projects often contain a lot of bespoke engineering rather than "off-the-shelf" components.

[12] Dijkstra was reputedly rude about software engineering claiming that, as a discipline, it had adopted the charter "How to program if you cannot" but I think he was really talking about a regrettable trend in modern universities which is the failure to realise that programming is a skill that is difficult to master. If the skill of programming cannot be mastered by undergraduates then let us rebadge the course software engineering, take out the programming and all the other hard stuff and fill it with waffly social science. I don't think we should dwell upon this too long but Dijkstra was quite old when he wrote this criticism and old Professors are not immune from becoming grumpy when they see a subject change rapidly in front of them

[13] Torvalds has joked that both Linux and Git are eponymous.

[14] If you have ever struggled to produce an anthology or a newsletter then you need version control – download a free copy of Git and give it a go.

My own view is that if software was designed using the sort of rigorous design processes that are present in large infrastructure projects then there would be less software failure[15] but it's a complex problem and rather outside the scope of this lecture.

However, one welcome development is the rather late recognition that programming is a human activity. For some reason the significance of this observation does not seem to have been fully accounted for in the design of programming languages. Humans make errors, and so one might hope that the topic of programming languages was stuffed full of human observational studies. The more I think about this, the more astonishing it becomes. The Art of Computer Programming (all five volumes and its updates, or fascicles as Donald Knuth calls them) which is one of the bibles of computer science[16] says almost nothing about the subject at all. The volumes are an amazing compendium of algorithms and one could not ask for a more comprehensive, and better written, list of algorithms and their conversion into imperative languages. But, for most programs, even scientific programs the algorithmic part is quite short – the vast majority of the code handles error checking, input/out graphical displays, interfaces and so on. Knuth is also famous for introducing a style of programming, called literate programming [4], in which the code is properly documented, and the program is readable by human beings who may themselves not be skilled programmers. This seems to be excellent practice and indeed there are series of Gresham lectures by Harold Thimbleby where the topic of how to make software better reoccurs in various forms. It is especially acute in healthcare where software errors kill people and I have also touched on this in previous lectures.

The nub of the issue is that programmers and software designers are humans make mistakes. Given that, one would have thought that the whole topic of programming was intimately concerned with how humans operate. The motivation for a yet another programming language would be a detailed and extensive empirical study showing that the new language sped up development or made it higher quality or reduced cost. I have never seen such a paper. There are a few reviews of the errors that undergraduate programmers make but rarely are these comparative between languages. To be controversial for a moment, it seems that "GOTO consider harmful" represents ultimate programming paper which in abbreviated form is … there is something I do not like about this practice … let's do this instead. This is quite an astonishing position to be in – I feel as though I am at the back of creative writing seminar where the Professor spends three hours talking about literary style without mentioning any evidence whatsoever.

When it comes to software engineering, the situation is better. For example, one practice that is commonplace in large software house is "code review" where another experienced programmer reviews your changes and gives you feedback. Code review has been part of what one might regard as good practice for over 40 years. So, a natural question might be…does it work? We don't know. We do know something about code review practices (In [5] and [6] authors make measurements on the rapidity of code review at, among others, Microsoft and Google, and some useful rules have emerged such as the observation that bugs are more likely to be introduced by programmers with a minor role in the project.[17] I should say I picked code review as a practice as I know that there is some quantifiable evidence that measures its effectiveness. The majority of programming practices are supported by hearsay, common-sense and the recommendations of the gurus of programming, some of whom I have mentioned today.

Of course, in the early days of programming such variety of practice hardly mattered and indeed there are many people today who would fiercely defend the rich tapestry of programming practices

[15] Which is not to say software is unique. As I write the London Crossrail project is two years delay and £5Bn over the original budget estimate. A similar project the Amsterdam metro North/South line was delivered seven years late and 40% overbudget.

[16] Possibly *the* bible of computer science. Knuth (pronounced Ka-nooth) refers to it as TAOCP.

[17] Cynics might complain that the observation that people who are the least familiar with the system are the most likely to break it is hardly earth-shattering but quantifying these things is important and deserves kudos.

and languages – many of them built on opposing principles. I do not. I would wish to see a far more quantitative approach to software engineering, and I think recent innovations in DevOps and machine learning will make it feasible to build really reliable code without having the resources of Microsoft, Google, IBM or Apple[18].

Another potential muddle is how the law treats programs. A program is indeed a literary work so is protected by copyright law. The European Patent Office (EPO) specifically excludes programs "as such" but will consider patents involving software if they exhibit something mysterious called "further technical effect". The US patent office is more liberal and software patents are commonplace in the US. All patent offices exclude mathematical functions and algorithms from patenting. Hardware is included, subject to the usual tests. Ladies and Gentlemen, I have just told you that you can compile a program to hardware, and I have told you that programs implement algorithms and programs can be used to describe algorithms. Fortunately, the patent offices of the world do not seem to be bothered about looking ridiculous by not understanding the fundamentals of computer science, so life goes on. One interesting development in the world of intellectual property that did originate form the world of software is the Open Source Movement. Open Source has been an incredibly powerful force not only in software and like all movements as led to new ways of thinking about intellectual property – namely that the money is not in the ideas but by putting those ideas to work.

Answers to the quiz. A is C++; B is Python (a language often praised for its compactness); C is C (a little computer science joke there); D is bash which is one of several interpreted languages available at the unix command line; E is Fortran (old-style Fortran in which the commands are capitalised); F is Prolog and G is Java (much loved by undergraduates but a notoriously verbose language). These examples came mostly from https://excelwithbusiness.com/blog/say-hello-world-in-28-different-programming-languages/.

© Professor Harvey, 2021

Bibliography

[1] TIOBE Company, "TIOBE Programming Community Index Definition," [Online]. Available: https://www.tiobe.com/tiobe-index/programming-languages-definition/. [Accessed 2021].

[2] RedMonk, "The RedMonk Programming Language Rankings: June 2020," 27 July 2020. [Online]. Available: https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/. [Accessed Jan 2021].

[3] Wikipedia, "List of programming languages by type," [Online]. Available: https://en.wikipedia.org/wiki/List_of_programming_languages_by_type#Esoteric_languages. [Accessed Jan 2021].

[4] D. E. Knuth, "Literate Programming," *The Computer Journal,* vol. 27, no. 2, pp. 97--111, 1984.

[5] P. C. Rigby and C. Bird, "Convergent contemporary software review practices," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering,,* pp. 202--212, 2013.

[6] C. Sadowski, E. Söderberg, L. Church, M. Sipko and A. Bacchelli, "Modern Code Review: A Case Study at Google," *2018 ACM/IEEE 40th International Conference on Software Engineering: Software Engineering in Practice,* pp. 181--190, 2018.

[7] T. Kuhn, The Structure of Scientific Revolutions, University of Chicago Press, 1962.

[18] For the avoidance of doubt, these big providers of code go to enormous effort to build what is called solid code. The issue arises for smaller entities particularly those operating in potentially dangerous situations such as navigation, aerospace, transport and medicine. Is it possible for smaller entities to ever match the coding resources of, say, Apple?