Understanding the new programmes of study for computing

Simon Peyton Jones

http://www.computingatschool.org.uk

Version 2.2

December 2014

Contents

1. Introduction 3

1.1 Audience and scope 4
1.2 A simple framework 4
1.3 Acknowledgements 5
1.4 License 5

2. Algorithms 6

2.1 Algorithms versus programs 7
2.2 Algorithmic thinking 7
2.3 Searching and sorting 8

3. Data and data representation 10

3.1 Bits 10
3.2 Bits for everything 11
3.3 Counting 11
3.4 Clever representation 12
3.5 Just bits 12

4. Programs and programming 13

4.1 What is a program? 13
4.2 Why learn to program? 14
4.3 Programming languages 15
4.4 What does it mean to “learn to program”? 15
4.5 Natural languages and programming languages 17

5. Ideas, not technology 17

5.1 Abstraction 17
5.2 Computational thinking 20

6. What is computer science? 20

6.1 Computer Science is a discipline 21
6.2 Computer Science and IT are complementary 21
6.3 Computing is a STEM subject 22

7. Onwards and upward 23

7.1 The CAS community 23
7.2 Other organisations 23
7.3 Have a go 24
7.4 Further reading 24

8. Appendix: “programming” versus “coding”? 25

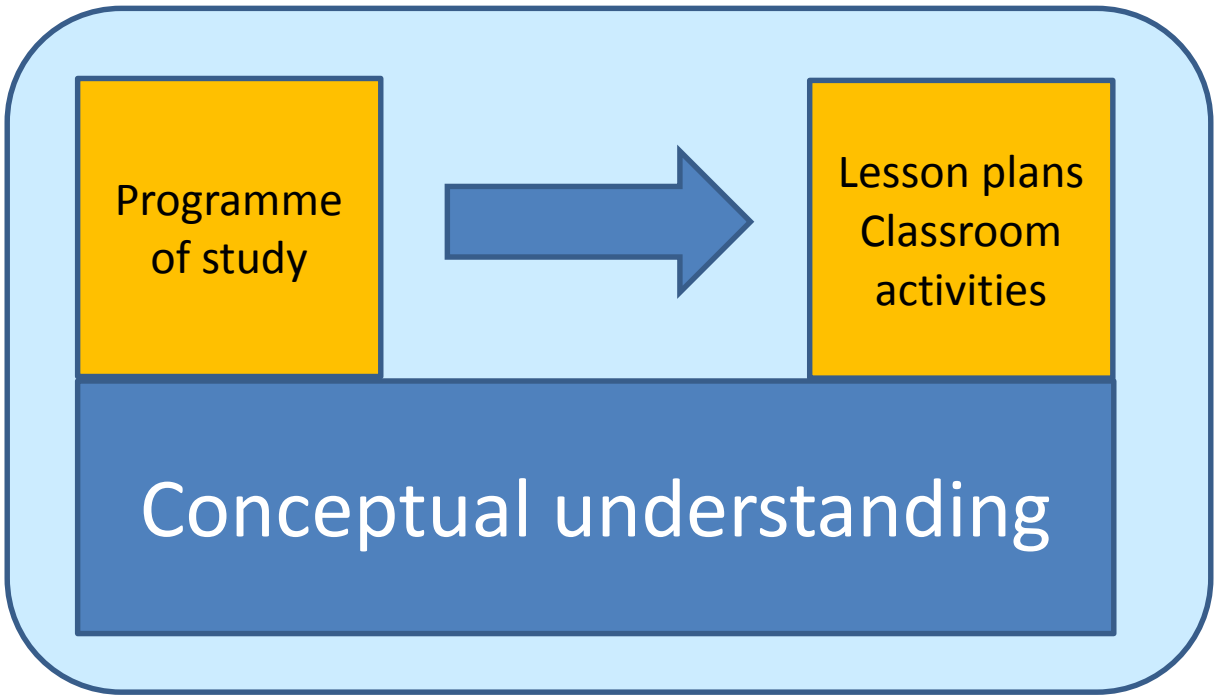
8.1 Formal languages 25
8.2 Universal languages 25
8.3 Coding and programming 26
8.4 Is HTML “programming”? 26

1. Introduction

The new Programmes of Study (POS) for Computing embodies some substantial changes compared to its predecessor, the POS for ICT. It is very brief, with only three sides of A4 covering eleven years of education. This leaves welcome opportunity for schools to adopt different approaches, but it means that the POS themselves are spare, dry, and use technical language.

You may be considering a published scheme of work, or your school may have already purchased one. Some of these schemes were written to provide a single solution for schools new to computing. Some of them fail to explain adequately the conceptual understanding that is needed by teachers and children to make sense of the classroom activities being taught through these schemes of work.

Whatever you may be doing, the 'glue' that joins all the elements of any scheme together will be *the development of children's understanding of some key concepts that underpin the curriculum*. These are the 'dry' terms we mentioned above, which I hope to make clear in this document. I picture it like this[1]:



A solid conceptual understanding makes sense of the programmes of study, and provides context and meaning for the various learning activities that you may undertake with your pupils. For every classroom activity, however engaging, we must ask why we are teaching it, and what the pupils are learning from it. We must not move from "death by PowerPoint" into "death by Scratch".

To take an analogy, you could teach numeracy as a mechanical activity, with lots of rules that children should follow by rote. But that would miss a huge opportunity! Good teachers are careful to convey the conceptual idea of a number, and although they do teach rules for

[1] This model has parallels with Shulman's pedagogic content knowledge in which the importance of subject knowledge and the importance of 'how to teach' that subject knowledge are emphasised.

performing sums, they want their children to understand *why* the rules work, to suggest their own approaches, and to be able to make mental checks that the answer is plausible. All of this is anchored in our own conceptual understanding of arithmetic; and we see arithmetic itself as one of the first steps in the great adventure of learning mathematics.

It is the same with computer science. We need to understand *why* as well as *what*.

1.1 Audience and scope

The intended audience for this document is teachers, both primary and secondary, ***specifically including those who have no previous experience of computer science or programming*.**

If you are a teacher, motivated to understand the Computing curriculum, and you don't understand something in here, that is my fault not yours. Please tell me.

I have a very specific purpose in mind: *to explain some of the key concepts behind the new Computing curriculum*. There are other important matters that I do not attempt to cover, including these:

- This document is about the computer science component of the Computing curriculum, and says little about IT or digital literacy. This is not because they are less *important*, but because computer science is less *familiar*. There is plenty of other excellent material that supports teaching and learning in IT and digital literacy.
- It says nothing about using technology to support and inspire teaching and learning across the entire curriculum. I call this Technology Enhanced Learning (TEL). It is hugely important, but it is a completely separate topic.
- It is not a scheme of work. I make no attempt to say "teach this in Year 3 and that in Year 5". You are the professionals who will address that, with the help of communities of practices such as CAS, as well as commercially-produced resources. Rather, I want to convey a visceral feel for the key concepts that will link together and make sense of your scheme of work.

I use the first person, because the document expresses my own views. I do not speak for CAS, still less for the Department for Education. Nor am I a school teacher, so I may sometimes not "connect" with you. But since I chaired the group that drafted the programmes of study, I hope that some insight into the thinking behind it may be useful to you.

1.2 A simple framework

Here are the stated aims of the Computing POS:

The national curriculum for computing aims to ensure that all pupils:

1. *can understand and apply the fundamental principles and concepts of computer science, including abstraction, logic, algorithms and data representation*
2. *can analyse problems in computational terms, and have repeated practical experience of writing computer programs in order to solve such problems*
3. *can evaluate and apply information technology, including new or unfamiliar technologies, analytically to solve problems*
4. *are responsible, competent, confident and creative users of information and communication technology*

Of these, aims 3 and 4 are relatively familiar, but aims 1 and 2 are quite new. If asked, most people would probably characterise computer science as a highly specialised university-level subject that you might study if you want to get a job in the software industry. The major new feature of the new POS is that **it introduces computer science as a foundational subject discipline that, like maths, music, or natural science, every child should have the opportunity to learn, from primary school onwards, alongside IT and digital literacy.**

This is a big change. The new Programmes of Study does not *replace* ICT with computer science. Rather the newly titled subject of Computing is a balanced combination of ICT and computer science:

Computer Science + IT + digital literacy = Computing

Our focus in this document is exclusively on the computer science part of this equation, because that is the part that is unfamiliar to many teachers. Computer science is a big subject, but broadly it is the study of **computation** and of **information**:

Computation + Information = Computer Science

More concretely we might say (using the title of a famous book by Niklas Wirth):

Algorithms + Data = Programs

In this guide I want to be as concrete as possible, so I will take this latter formulation as a guide, by studying **algorithms** (Section 2), **data** (Section 3), and **programs** (Section 4). After that I will talk about the thinking skills that Computing teaches, including abstraction and computational thinking (Section 5). Finally, in the light of all this, I'll stand back and ask again what Computer Science is (Section 6).

Please don't take these "equations" too seriously! They are broad approximations (like "Inorganic + Organic = Chemistry") that give you a way to start thinking about the topic, but they will mislead you if you take them as the whole truth. For example, one could certainly add "communication" to "computation" and "information"; just think of the internet, or how much of the emergent behaviour of an ant colony is bound up with the communication between ants.

1.3 Acknowledgements

I warmly thank Miles Berry, Ray Chambers, Nicki Cooper, Quintin Cutts, Graham Hastings, Sue Sentance, Zoe Ross, Yvonne Walker, Peter Warwick, and John Woollard for their help and guidance in writing this document.

1.4 License

2. Algorithms

The POS for KS1 says *"understand what algorithms are, how they are implemented as programs on digital devices, and that programs execute by following precise and unambiguous instructions"*[2]. So what exactly is an "algorithm"?

An algorithm is a precise method for solving a given problem. For example

- **Problem:** repair a puncture on your bike
  - **Algorithm:** take off the wheel, remove the tyre, remove the inner tube, find the hole, patch it, replace the inner tube, replace the tyre, put the wheel back on.
- **Problem:** find your way to the exit of a maze
  - **Algorithm 1**: walk around at random until you find the exit. (This *is* an algorithm!)
  - **Algorithm 2:** walk forward, keeping your right hand touching the wall at all time.
  - **Algorithm 3:** Walk forward until you reach an intersection. Turn right unless there is a doughnut in the way. If you can turn right, do so, and leave a doughnut on the ground to make sure you don't do the same thing again.
- **Problem**: find which of your classmates picked up your calculator by mistake.
  - **Algorithm 1:** whenever you bump into a classmate, ask them, until you find the right person
  - **Algorithm 2:** like Algorithm 1, except never ask the same classmate twice
  - **Algorithm 3:** find your classmates in alphabetical order, and ask each in turn. Stop when you find the calculator, or when you have asked the last pupil.
  - **Algorithm 4:** whenever you meet a classmate, ask them. If they don't have the calculator, get them to join you in the search by running Algorithm 4 themselves. When someone finds the calculator, they should find you.

So an algorithm simply says how to do something, or accomplish some task. Notice that:

- Before we can speak of an algorithm we must be clear about the problem it is trying to solve.
- The audience for an algorithm is a human being, not a computer. The goal is to convey to the mind of your audience the essential insight of *how* the problem can be solved.
- Because the goal is to convey the "essential insight", the description of an algorithm will usually suppress lots of incidental detail. For example, Algorithm 1 above did not specify the words to use to your classmates, still less which muscles to move when speaking those words.
- Nevertheless, an algorithm should be precise, in the sense that the listener can say "oh yes, I see; I could do that" (or perhaps "I could build a machine to do that").
- There may be lots of different algorithms that solve the same problem; I gave several algorithms for the final two example above.

[2] I will use red italics when I quote directly from the POS

- Some algorithms are simpler than others. For example, walking randomly through a maze is simpler than leaving doughnuts on the ground.
- Some algorithms are faster than others. For example, walking randomly around a maze might take a very long time indeed.
- You sometimes have to think very carefully

2.1 Algorithms versus programs

So what are the differences between an algorithm and a program?

	Algorithm	**Program**
Audience	A person	A computer
Language	Usually expressed in informal language	Expressed in a programming language
Level of detail	Incidental detail suppressed	Every detail is specified
Level of precision	Precise enough for a human, with a reasonable level of common sense and background knowledge, to say "I can see how to do that".	Precise enough that a mindless machine can execute the program without human intervention

Looking at this table, you can see that there isn't a hard-and-fast distinction between algorithms and programs; a program is simply an algorithm whose description is so precise that it can be executed even by as stupid a device as a computer.

2.2 Algorithmic thinking

Algorithms often embody great insight and ingenuity; reading an algorithm can give you a real "aha!" moment. Better still, good algorithms are useful as well as beautiful.

For example, suppose you are given a pile of coins, all of equal weight except one that is heavier than the others, and a weighing balance. The task is to find the heavy coin. Here are two algorithms for doing so

- **Algorithm 1** (simple): pick two coins and weigh them against each other. If one is heavier, you have found the heavy coin. If not, discard one of the two, and repeat until you find the heavy coin.
- **Algorithm 2** (clever): divide the pile into two equal halves, and weigh the two halves against each other. The heavier pile contains the heavy coin, so divide it into two and weigh them against each other. And so on.

  What if the pile has an odd number of coins, so it can't be divided in two? Easy! Divide into two equal piles with one left over; if the two equal piles have the same weight, the leftover one is the heavy one.

Algorithm 2 is a bit more complicated; it took six lines to explain instead of three. Complications are bad: more code to write, more chances of getting it wrong. But Algorithm

2 has a very compelling advantage: it takes fewer weighings than Algorithm 1, perhaps *many* fewer. Let's try it:

Number of coins	Number of weighings needed (in the worst case)	
	Algorithm 1	Algorithm 2
2	1	1
4	3	2
8	7	3
16	15	4
1,000	999	10
1,000,000	999,999	20

You can easily work out the numbers for small piles. Can you see the pattern? Since computers often work with millions or thousands of millions of data items, the difference between Algorithm 1 and 2 can become very important indeed.

The exact details aren't important. The big points I want to bring out are:

- Algorithm 2 embodies a clever insight, an "aha moment", namely that we can halve the size of the problem in one weighing. And that insight leads to an algorithm that is much, much faster than the blindingly obvious one. Computer science is chock-full of these amazing insights, and communicating their joy and beauty is the mission of a computing teacher.
- This particular algorithm, and many like it, are well within the reach of a primary school child. With a couple of simple balances and some coins you can race two teams against each other, each running a different algorithm. Here's a CS Unplugged video of a similar activity.
- To compare Algorithms 1 and 2 we didn't need to program them, run them on a computer, and time them with a stopwatch. Algorithm 2 is so much faster than Algorithm 1 that it will beat the latter on *any* computer, if the number of coins is big enough. This is what the KS3 POS means when it says *"use logical reasoning to compare the utility of alternative algorithms for the same problem"*

There is plenty of room for extension work for bright pupils. For example, you can make Algorithm 1 run faster by discarding *both* coins if they have equal weight; and you can make Algorithm 2 run even faster if you divide into *three* piles, two exactly equal and one either equal or off by one.

2.3 Searching and sorting

The POS for KS3 says that pupils should be taught to *"understand several key algorithms that reflect computational thinking [for example, ones for sorting and searching]".* Although sorting and searching are not mandatory, they are called out explicitly here. Why? There are several reasons why this class of algorithms are particularly useful in a teaching context:

- It is particularly easy to explain what sorting and searching algorithms do, and to explain why they are useful.
- They are ubiquitous inside computer systems, so knowledge of sorting and searching algorithms is particularly useful.

- There are an astonishingly large number of algorithms known for sorting and searching, each useful in different circumstances. People write entire books about them!

Let's start with searching. Suppose I want to find Jane Smith's name in a telephone book (a searching problem). Here are two possible algorithms:

- **Algorithm 1 (linear search).** Start at page 1 and read page by page until you find Jane Smith's name.
- **Algorithm 2 (binary search).** Open the directory in the middle and see what name is written there. Suppose it is Bill Manlove. That is alphabetically before Jane Smith, so she must be in the second half. So, split the second half in half, and look at the middle page. Keep doing this until you are down to one page. Now split the page in half, and so on until you get down to one name.

Algorithm 2 is a bit more complicated than Algorithm 1. If the telephone book only had ten names in it, Algorithm 1 would be fine. Moreover, Algorithm 2 requires the telephone book to be listed in alphabetical order, whereas Algorithm 1 will work just fine whatever order the names are in.

But if it had a two million names in it, and it took one second to read each name, it would take you, on average, a million seconds to find the name you were looking for, or over ten days. But Algorithm 2 would take only about 20 "probes" (counting each halving as a probe) to find the right name. Each probe might take longer; let's be pessimistic and say a minute. So Algorithm 2 takes 20 minutes worst-case. That's a **lot** better than ten days! (Incidentally do you notice the same pattern as the in the coin-weighing problem?)

Now suppose that you sometimes wanted to look a name up, and sometimes wanted to add a name to the directory. Adding a name to an unsorted directory (which works fine for Algorithm 1) is easy: just add it to the end. But to add a name to a sorted directory we first need to find the right place (easy) and then make space for it. That might involve shuffling all the other entries along in memory. So

- an unsorted directory is good for insertion, but bad for lookup;
- a sorted directory is good for lookup but bad for insertion.

Is any structure good for both? Yes – but that would take us into a discussion of balanced trees, which is too much for this document.

The really important points are these:

- There is a huge variety of algorithms for sorting and searching
- They vary in
    - how complicated they are to write;
    - how much time they take to run
    - how well they support operations like insertion, deletion, and lookup

CS Unplugged has a rich page on sorting algorithms, aimed squarely at teachers, with lots of useful links.

3. Data and data representation

Much of the power of computers comes from their ability to store and manipulate very large quantities of **data** very quickly. The way in which this data is stored and manipulated can make enormous differences to the speed, robustness, and security of a computer system.

Not much of this section is *directly* relevant to the KS1/2 POS, but I hope that you may nevertheless find it useful.

3.1 Bits

Data in a computer is (almost) invariably represented as bits (short for "binary digit"). A bit is either a 1 or a 0.

It's easy to see how bits can represent a black and white picture. For example, here is a crude picture of a table, represented as a 5x4 grid of "pixels". If we display 1's as black and 0's as white, we get a picture.

0	0	0	0	0
1	1	1	1	1
0	1	0	1	0
0	1	0	1	0

If we devoted two bits to each pixel we could represent shades of grey, like this:

The two bits	Shade
0 0	
0 1	
1 0	
1 1	

Now we can draw a table with light grey legs and dark grey feet:

00	00	00	00	00
11	11	11	11	11
00	01	00	01	00
00	10	00	10	00

If I want more shades of grey, I can use more bits for each pixel. Two bits can represent four shades, three bits can represent eight:

The three bits	Shade
0 0 0	
0 0 1	
0 1 0	

0 1 1	
1 00	
1 0 1	
1 1 0	
1 1 1	

Four bits can represent 16, and so on. The number of shades we can represent doubles each time we add one more bit.

We can represent colour pictures too, of course, using three groups of bits for each pixel, one for the degree of redness, one for blueness, and one for greenness.

3.2 Bits for everything

Bits can represent things other than pictures.

- Bits can represent **characters**, like the ones on this page. Again we need a table to tell us which bit-pattern stands for which character. Here is part of the ASCII table, just one such convention:

Bit pattern	Character that it stands for
0100 0001	A
0100 0010	B
...	

- Bits can represent **formatting information** in a document. For example, this document has characters, but also some indication of which words are in bold, where the bullet points are, and so on.
- Bits can represent **sounds**. Just imagine measuring the height of an audio waveform 10,000 times a second. How might we measure each height? Just like the shades of grey, we can do so with a binary number. That sequence of 10,000 binary numbers each second describes the waveform quite well, and it is just a bunch of numbers; more bits.
- Bits can represent **videos**: a video is just a sequence of pictures.
- Bits can represent **numbers** (see the next section).

In short, bits can stand for, or represent, absolutely anything. When you stop to think about it, that's quite remarkable. Bits are a kind of universal medium for representing information.

3.3 Counting

In the table of characters I gave above, the bit patterns are not entirely arbitrary

Bit pattern	Character that it stands for
0100 0001	A
0100 0010	B
0100 0011	C
…	…

If you regard the bit pattern as a binary (base-2) number, then the code for each letter is one greater than the code for the previous one. For example, if you add one to the code for B you get the code for C: that is, 01000010 + 1 = 01000011.

Similarly, in the tables of bit-patterns for the shades of grey, if you regard the bits as a 3-bit binary number, I am counting in binary: the darker the shade, the bigger the number. Using numbers sequentially to represent shades, rather than randomly assigning a bit-pattern to each shade, makes it easier to figure out what to display, and easier to transform the picture (e.g. to make it darker add one to each pixel).

Finally, of course, many programs manipulate numbers, and bits can certainly represent numbers just by regarding the bits as a binary number.

I am not going to explain about binary numbers and arithmetic, because you can find many excellent tutorials, and classroom exercises, on the CAS community site, or CS Unplugged, or elsewhere.

3.4 Clever representation

Lots of computer science concerns clever ways to use bits to represent data, in a way that makes it more amenable to processing, or take less space (compression), or be less easy for a bad person to steal (encryption).

For example, consider a video. We could store a complete picture for each frame of the video, but successive frames are nearly the same, so that would mean lots of duplication. It might take a lot less space to store just the *differences* from one frame to the next.

Similarly, even within a single picture there might be quite big blocks where the colour is more or less uniform, so again recoding the differences from a baseline could reduce the number of bits required. In the extreme an all-black picture can be represented very compactly!

3.5 Just bits

You can explore much of this with primary-aged children (e.g. look at the Data section of CS Unplugged), although this is not something the POS requires at KS1/2 level. There are some deep lessons here:

- **It's all just bits**. We can store those bits on a USB stick, or transmit the bits over the internet. The USB stick or the internet neither know nor care whether you are storing or transmitting pictures, sounds or characters. To the USB stick or the internet, it's just bits.
- **It's all conventions.** In our first example we used 20 bits to describe a 5x4 array of black-and-white pixels. But how did we *know* it was 5x4 and not 10x2 or 20x1? The bits don't tell us that; we need some extra information often called meta-data, to tell us how to interpret the bits.

  The suffix of a file name is a simple kind of meta-data. A file `song.mp3` is probably a sound file; `cv.txt` contains textual characters. If you change the filename suffix, which you can readily do, your data will be interpreted with a different set of conventions.

4. Programs and programming

"Coding is the new Latin" has become a bit of a catch-phrase. There is much talk of coding (i.e. programming[3]), to the extent that you might think that programming pretty much *was* the new part of the Computing POS. So perhaps

Computer Science = Programming?

Absolutely not! Programming plays the same role in computer science that investigations do in maths or science. Programming animates the subject and brings computer science to life; it is creative, and engaging. It illustrates otherwise-abstract concepts in completely concrete terms. It is also an incredibly useful skill. Indeed programming is writ large in Aim 2 of the POS: *"can analyse problems in computational terms,* ***and have repeated practical experience of writing computer programs in order to solve such problems***. "

Nevertheless computer science is more than programming, just as chemistry is more than Bunsen burners and test tubes. Articulating this "more than" is a big purpose of this document.

Nevertheless, programming makes a good starting point, just because it is so concrete.

4.1 What is a program?

Here is an excerpt from a knitting pattern for a jersey:

> Cast off 2 sts beg next 2 rows. 67 (**73**-79-**83**-97-**107**-117-**123**) sts.
> **1st row:** (K1. P1) twice. Sl1. K1. psso. Knit to last 6 sts. K2tog. (P1. K1) twice.
> **2nd row:** (P1. K1) twice. P2tog. Purl to last 6 sts. P2togtbl. (K1. P1) twice.
> **3rd row:** As 1st row.
> **4th row:** (P1. K1) twice. Purl to last 4 sts. (K1. P1) twice.
> Rep last 4 rows 0 (**0**-1-**1**-0-**3**-6-**6**) time(s) more. 61 (**67**-67-**71**-91-**83**-75-**81**) sts.

A knitting pattern is *exactly* like a program:

- All the creativity is in the pattern, none (or very little) with the person doing the knitting. If you simply follow the instructions, you will end up with a jersey.
- It is fully precise. There is nothing like "knit a nice wavy pattern here".
- It looks like gibberish to anyone who does not know how to knit.
- Sophisticated patterns are made by combining together very simple pieces a variety of ways.
- The "simple pieces" are things like "K1" meaning "knit one stitch", and "P1"meaning "purl one stitch".
- The "combining together" are things like
    - Put one thing after another (e.g. "K1. P1" means "knit one then purl one".
    - Repeat things a fixed number of times (e.g. (K1. P2) twice means "knit one then purl one, twice".
    - Repeat things until something happens (e.g. "Purl to last 6 sts" means "keep doing purl one until the last six stitches")

[3] Are "coding" and "programming" the same thing? At this level of detail, yes; but see the Appendix for a fuller discussion.

- To be of use, a knitting pattern must be knitted, or *executed*, by a person or a machine.

Here are some other familiar examples of program

- A cooking recipe is a program for constructing a dish. It is executed by a human following the instructions.
- A musical score is a program that says how to create a sound. It can be executed (played) by a computer (think of a Midi player) or a human being.
- Driving instructions from your house to the Eiffel Tower is a program for getting to the Eiffel Tower. It is executed by getting in your car and following the instructions.
- The settings of an oven timer constitute a program for controlling the oven. Typically it is limited to rather simple programs like "wait until 10.30, switch on, wait an hour, switch off".
- The instructions for a floor turtle, such as "Pen down; forward 4; turn right; forward 2" are a program. It is executed by the floor turtle, which draws a line on the floor to show where it has been.
- The sorting network drawn on the floor in this lovely CS Unplugged video is a program for sorting numbers. It is executed by a group of children, walking along the lines and swapping over in the boxes.
- The formulae of a spreadsheet constitute a program that says how to compute the values of some cells from those of others. For example, if the cell C3 contains =A1+B1, the value of C1 is computed by adding the value of A1 and B1, having first computed those if necessary.

Notice that, although programs can, and often are, executed by machines, that is not an essential property. The essential thing is that the program is *precise*. The execution engine, whether it is a person or a machine, simply follows instructions blindly.

Notice, too, that in each of these examples there is a different *programming language*. For the knitting pattern it was the "K1. P1" stuff. Recipe books have their own language. Driving instructions use stylised language. And so on. The language is sometimes extremely simple and limited (such as the oven timer); indeed you might baulk at calling it a "language" at all. Or it might be very rich and complicated. But it has fixed form and vocabulary.

So here's an attempt at a definition:

> **A program is a fully-precise description of how to achieve a goal. It is expressed in a specific programming language, and can be executed blindly by a person or machine.**

4.2 Why learn to program?

Why is it useful to learn to program?

- **It is hugely creative**. When you write a program you are making a computer do something it has never done before. The only limits are the limits of your imagination and ability.
- **It can be extremely engaging and enjoyable;** it encourages playful experimentation, and perseverance in the face of repeated failure**.**
- **It rewards precision of thought**. If the program is wrong, it won't work.

- **It encourages the ability to reason**. If you see "(Forward 3; Turn right) four times" you may imagine the turtle drawing a square. You are reasoning in your head about the behaviour of a program when it is executed. This is a pretty abstract thing to do, but you have a very concrete reason to want to do it!
- **It is an extremely marketable skill**. Nowadays it is not just professional software developers who write programs. Scientists, engineers, data analysts, and many other professions in the knowledge economy, all increasingly involve some level of programming.

Children are naturally creative. They want to make things. They have fertile imaginations which they are bursting to express. They also love to create things that look cool and impress their peers. Programming allows children to do all of these things at once. The sheer joy that a child displays as she watches her friends play the latest computer game she has written is a wonder to behold. The problem experienced by many teachers is not how to get children started, but how to break it to them that the lesson is over and it is time to stop.

4.3 Programming languages

During the last ten years there has been huge progress in the variety and richness of programming environments[4] intended specifically for teaching. Below is a list of some examples, with particular emphasis on ones that might be useful at primary school. It is emphatically not an exhaustive list; my motivation for giving it is to be concrete about my claims about variety and richness, and to give you a starting point for finding out more.

- Scratch http://scratch.mit.edu
- Kodu http://www.kodugamelab.com
- TouchDevelop https://www.touchdevelop.com
- Logo e.g. http://www.calormen.com/jslogo/ or http://education.mit.edu/starlogo
- Greenfoot http://www.greenfoot.org (secondary)

Computer science is about **ideas**, not about **technology**. A scheme of work that says *"we start with HTML and Logo, move on to Scratch, and use TouchDevelop for ones who need to be stretched"* is focusing too much on technology (in this case, the programming language).

Ultimately, the language you use is not that important. What *is* important is that your pupils become confident in programming; understand that there are many languages, each with different strengths; and are intensely relaxed about learning a new language. This is one reason that (albeit not until KS3) the POS specifies "use two or more programming languages, at least one of which is textual".

4.4 What does it mean to "learn to program"?

"Cooking" can mean a lot of different things, ranging from simply experimenting with different food tastes, following recipes, exploring variations of recipes, up to creating new recipes. Similarly, "music" can mean anything from trying out the sounds that different instruments can make, right up to composing a symphony.

[4] I say "programming environments" rather than simply "programming languages" because each tends to come with its own way of editing, running, sharing, debugging, and testing programs. The term "environment" covers all of these things whereas "language" describes only the notation in which you write the program itself.

In the same way, "programming" can mean a lot of different things. For example:

- **Simply experimenting with the medium**. Programming environments like Scratch and Kodu make it easy to try things out in a playful, exploratory way: "I wonder what happens if I press that button/drag that shape?". At this stage the goal is to experiment, gain confidence that nothing bad will happen, and to gain intuition about what happens. It's rather like a toddler playing with building bricks.
- **Simply copy an existing program, run it, and then start *making small changes* to it**. The program solves the "blank sheet of paper" problem. Some changes are limited but fun (e.g. change the colour of the monster). As confidence builds, pupils will become more ambitious (e.g. can we have more than one monster?).
- **Start to *predict* what a change will do**. One important aspect of computational thinking is to be able to predict what a program will do, or what effect a change to the program will have. For simple, straight-line programs (i.e a simple sequence of instructions) this is pretty easy; the more complicated the program, the harder it gets. But at every level the ability to reason logically about the program is key. (The phrase "reason logically" about programs is used in KS1, 2, and 3 of the POS.)
- ***Debug* a program that is not working properly.** For example, if you want to draw a square with a floor turtle, you might forget to put the pen down, so the turtle crawls around but doesn't draw anything. Debugging always involves coming up with a guess (or hypothesis) about what is going wrong, performing experiments to confirm the guess, and making a change that you predict will fix it[5].
- ***Explain to someone else* how/why your program works**. The simple act of explaining often reveals latent bugs in your program, or potential simplifications to your code.
- ***Read* a program and figure out its purpose**. For example

  ```
  T := 0
  for I = 1..N { T := T+I }
  ```

  You could talk about loops and variables, but an experienced programmer would say "oh, that just adds up the numbers between 1 and N, and puts the total in T". That is, she has worked out the *purpose* of the code, rather than just following the individual steps it takes.
- **Starting from an idea of what you want your program to do, *write a program from scratch* to do it**.

These aspects of programming are arranged in roughly increasing order of sophistication, but even a KS1 child should be able to do most of them *for simple programs* (turtle graphics being the archetypal example). The important thing is to focus on the *thinking* aspects (prediction, explanation, debugging, testing, design) rather than the *technological* aspects (where do the semicolons go).

These computational thinking skills are articulated in the POS from KS1 onwards; for example

- *"create and debug simple programs"* (KS1)
- *"use logical reasoning to predict the behaviour of simple programs"* (KS1)

[5] Pupils often try to debug programs by making random changes, unsupported by any reasoning, and running the changed program to see if it behaves better. This isn't computational thinking; it's simply guesswork.

- *"use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs"* (KS2).
- "design, write and debug programs that accomplish specific goals" (KS3)

The big message here is: *even the programming part of the POS is not just about writing programs*; it's about reading, explaining, debugging, predicting, and reasoning about programs. Encourage playfulness, creativity, and exploration.

4.5 Natural languages and programming languages

Natural languages (like English or French) and programming languages (like Java or Python) are both called "languages", but they are very different:

- **Natural language leaves stuff out.** A listener can only understand a sentence by using his or her general knowledge, and knowing the context of the sentence. For example "Take that bishop" could be an instruction in a chess game, or the beginning of a story about a clergyman. Even when the general sense is understood, which particular bishop is involved might only be clear from the previous conversation, or from a gesture. In contrast, programs are precise. A program has one, and only one meaning.
- **Natural language has lots of extra stuff.** You could delete at least half the words in this document and still leave something that was roughly comprehensible. In contrast, virtually any insertion or deletion n a program will turn it into a program that does something else (usually not what you want) or that is simply rejected as unintelligible.

Pupils often find it frustrating that the computer doesn't understand what they intend from a "nearly correct" program. But computer programs are executed mindlessly by a machine that has no innate understanding of what they intended.

5. Ideas, not technology

The new Computing curriculum is primarily about **ideas** rather than about **technology**.

It is easy to get sucked into thinking about technology, whether it be how to make Word lay out a paragraph right, or how to write a loop in Java, or how to configure a network driver (to take three very different examples). It is even easier for our children to come to believe that Computing is primarily about technology, because technology is so immediate, seductive, and often rewarding.

So as teachers you must constantly strive to articulate the principles and ideas of the subject, even though those principles and ideas are less immediate, and hence perhaps seemingly less relevant, than the sleek devices in our pocket. This section collects a few reflections on this theme.

5.1 Abstraction

One of the Aims of the POS is that every child can "can understand and apply the fundamental principles and concepts of computer science, including abstraction, …". The first bullet of the KS3 POS is this highly-compressed sentence: *"Design, use, and evaluate computational abstractions that model the state and behaviour of real-world problems and physical systems".* What is a "computational abstraction" and what does it mean to it to "model" a real-world problem?

The best way to get hold of the idea of abstraction is by looking at examples:

- The London Underground map is a simple model of a complex reality. It contains precisely the information necessary to plan a route from one station to another. But it omits masses of other information: the physical location of each station, and the physical distance between them; the size of the station; the number of staff that work there; the frequency of trains; and so on. All of the detail is stripped away, leaving only the information about which station is connected to which other station.
- When you apply for a job, your CV summarises something hugely rich and complicated: your whole life. You choose the information in your CV to be useful to a potential employer, leaving out masses of information that won't interest them. Your CV is an abstraction of your life. The longer it is the more accurate it may be, but abstractions must fit their purpose: if it is too long, the recruiter won't read it.
- In our knitting pattern above, the instruction "knit 1 stitch" was taken as primitive, not needing any further explanation. But in reality, knitting one stitch involves hundreds of small movements of muscles, some simultaneous, some one after another. Knitting would be far too complicated if we thought of all of these individual movements.

  At the next level up, an experienced knitter might say "knit a stretchy 16-inch neck". Again, much detail is suppressed (how many stitches? what kind of stitches?), but in exchange the expert can think about garments at a higher level. ("How could I make it a bit more fitted around the wrists?")
- If we throw a ball and want to predict where it will go, we might proceed like this. First, the important thing about the ball is its *position* and its *speed*. Then, in each time interval (say 1/10 second), using its speed we can figure out how far it will go in that time interval; and using the force of gravity on the ball, we can work out how its speed will change. So now we know its position and speed after 1/10 second. If we just repeat this process we can work out its position and speed after 2/10, 3/10 seconds, etc.

  The details are not important here. But notice that we have suppressed many details: the colour of the ball, the day of the week, whether the sun is shining, and so on. All that remains is the position and speed of the ball (this is its **state**), and the way the position and speed change over time (this is its **behaviour**). We are "modelling" an enormously complex process (air molecules bumping into the ball, etc.) with two numbers.
- A turtle program to draw a rectangle hides its implementation (forward 2, turn right, forward 1, turn right, forward 2, turn right, forward 1) behind a simple call "DrawRectangle". You can combine multiple calls to DrawRectangle to make a procedure that draw a house, DrawHouse, say. Then you can combine multiple calls to DrawHouse and DrawRectangle to draw a village of houses. And so on.

  When you were drawing the village, if you had to think of all the individual "forward" and "turn right" that were involved, your head would explode. Success depends on calling procedures knowing **what they do,** without thinking of **how they do it**.

All of these examples demonstrate abstraction, probably the most powerful, pervasive, and simple Big Idea in computer science. Specifically:

- Every abstraction has a *client*, or *purpose*. For example, the client of the underground map is a person wanting to make a journey, and her purpose is to plan her journey.

- *Different abstractions of the same thing may serve different purposes*. The underground map abstraction would be no good for an engineer wanting to dig a new tunnel, because she needs to know exactly where the existing tunnels are, and the underground map doesn't tell you that. A different abstraction, showing geography and soil types, but perhaps not the names of the stations, would be needed for that.
- *Every abstraction suppresses unnecessary detail*, to allow the client to focus on the important aspects. The choice of what to suppress and what to expose is the central challenge.
- *Designing good abstractions (ones that fit their purpose) is highly creative*. For example, in the case of the thrown ball, isn't the mass of the ball important? Actually, it isn't: the path of the ball does not depend on its mass. But that's not obvious!

  The fact that one abstraction might be better than another (for a given purpose) is what the "Design, use, **and evaluate** computational abstractions" phrase in the KS3 POS.
- *Abstractions almost always embody approximations*; for some purposes they hide too much detail. For example, in our abstraction of the ball, the day of the week really isn't relevant. But we also stripped away information about the air pressure, and that does matter if you cared about a highly accurate predication of the ball's flight. By making the abstraction simple, we necessarily made it approximate.

  Similarly in our turtle graphics abstraction, we did not provide any way to control the thickness or colour of the lines. If we wanted more control, we would have to elaborate the abstraction by adding SetLineWidth(n), and SetLineColour(c).
- Computer programs are among the most complex artefacts that human beings have ever created. *Abstraction is the key to managing that complexity*. We can program a complex task by breaking it down into a sequence of simpler tasks, and considering them one by one. Each of those simpler tasks we can in turn break down into sub-tasks, and so on. Each task is an abstraction of its perhaps-complicated implementation.

What is a "computational" abstraction? Simply, an abstraction that is amenable to treatment by a computational process, or (in plain language) a computer. In the case of the thrown ball, the two measurement of position and speed are simply numbers, and we can calculate mechanically (i.e. with a program) what the position and speed will be over time. The map of the Underground looks less "computational" but we could represent it as a table with two columns: each station and the stations it is directly connected to. Using that table we could write a program to get from A to B with the smallest number of stops.

In short, abstraction is a fancy word for something we do instinctively every day. The reason that it is highlighted particularly in computer science is because abstraction takes very concrete form: every program uses abstraction again and again.

In computer science, abstraction is repeatedly applied, layer upon layer upon layer. Once we have DrawRectangle and DrawTriangle, we can write DrawHouse by calling DrawRectangle repeatedly. Once we have DrawHouse we can write DrawVillage by calling DrawHouse repeatedly. And so on. Real programs are built using tens or hundreds of layers of this kind. As Bertrand Russell is supposed to have said "It's turtles all the way down".

5.2 Computational thinking

Much writing about the new Computing curriculum mentions **computational thinking**. But what exactly is "computational thinking"? It is a term that has broad resonance, but which is surprisingly hard to pin down. Here is an attempt:

> Computational thinking is the process of *recognising* aspects of computation in the world that surrounds us, and *applying* tools and techniques from computing to understand and reason about both natural and artificial systems and processes.

Rather than getting hung up on a precise definition, it is more helpful to recognise *characteristics* of computational thinking:

- Computational thinking is something that *people* do (not something that computers do), and includes the ability to think logically, algorithmically and (at higher levels) recursively and abstractly.
- You can't write a program without engaging in computational thinking, but computational thinking is a broader skill than programming. Thinking about how to get all the children out of school fastest when there is a fire alarm is a computational problem, even though it may never be expressed in a computer program.
- Computational thinking gives an understanding of the artificial world, but it also gives a new way of understanding the *natural world.* For example, cells are driven by DNA, which works very much like a program, written on a "tape" of nucleotides; termites build mounds of astonishing complexity, not driven by a central brain, but by the interaction of lots of tiny termite brains running tiny programs; the global ecosystem is the result of multiple sub-ecosystems all interacting with each other; and so on. Insights from computer science are becoming directly useful in understanding the natural order, often thorough simulations that abstract the key pieces of a complex biological system, and implement their behaviour in an algorithmic way.

There is a collection of pointers to resources supporting computational thinking on this CAS Community resource.

Computational thinking is not everything! A well-rounded student of computing will also be proficient in other generic skills and processes, including: thinking critically, reflecting on one's work and that of others, communicating effectively both orally and in writing, being a responsible user of computers, and contributing actively to society.

6. What is computer science?

Computer Science is the study of principles and practices that underpin an understanding and modelling of computation, and of their application in the development of computer systems. At its heart lies the notion of computational thinking: a mode of thought that goes well beyond software and hardware, and that provides a framework within which to reason about systems and problems. This mode of thinking is supported and complemented by a substantial body of theoretical and practical knowledge, and by a set of powerful techniques for analysing, modelling and solving problems.

Computer Science is deeply concerned with how computers and computer systems work, and how they are designed and programmed. But pupils studying computing also gain insight into computational systems of all kinds, whether or not they include computers. Computational thinking influences fields such as biology, chemistry, linguistics, psychology, economics and

statistics. It allows us to solve problems, design systems and understand the power and limits of human and machine intelligence. It empowers us, and for that reason all pupils should be aware of and have some competence in it. Furthermore, pupils who can think computationally are better able to conceptualise and understand computer-based technology, and so are better equipped to function in modern society.

Computer Science is a practical subject, where invention and resourcefulness are encouraged. Pupils are expected to apply the understanding they have developed to real-world systems, and to the purposeful creation of artefacts. This combination of principles, practice, and invention makes it an extraordinarily useful and an intensely creative subject, suffused with excitement, both visceral ("it works!") and intellectual ("that is so beautiful").

6.1 Computer Science is a discipline

Education enhances pupils' lives as well as their life skills. It prepares young people for a world that doesn't yet exist, involving technologies that have not yet been invented, and that present technical and ethical challenges of which we are not yet aware.

To do this, education aspires primarily to teach disciplines with long-term value, rather than skills with short-term usefulness, although the latter are certainly useful. A "discipline" is characterised by:

- **A body of knowledge**, including widely-applicable ideas and concepts, and a theoretical framework into which these ideas and concepts fit.
- **A set of techniques and methods** that may be applied in the solution of problems, and in the advancement of knowledge.
- **A way of thinking and working** that provides a perspective on the world that is distinct from other disciplines.
- **Longevity**: a discipline does not "date" quickly, although the subject advances.
- **Independence from specific technologies**, especially those that have a short shelf-life.

Computer Science is a discipline with all of these characteristics. It encompasses foundational principles (such as the theory of computation) and widely applicable ideas and concepts (such as the use of relational models to capture structure in data). It incorporates techniques and methods for solving problems and advancing knowledge, and a distinct way of thinking and working that sets it apart from other disciplines (computational thinking). It has longevity (most of the ideas and concepts that were current 20 or more years ago are still applicable today), and every core principle can be taught or illustrated without relying on the use of a specific technology – or indeed *any* technology.

6.2 Computer Science and IT are complementary

Computer Science and IT are complementary subjects. Computer Science teaches a pupil how to be an effective *author* of computational tools (i.e. software), while IT and digital literacy teaches how to be a thoughtful *user* of those tools. This neat juxtaposition is only part of the truth, because it focuses too narrowly on computers as a technology, and computing is much

broader than that. As Mike Fellows[6] remarked "Computer Science is no more about computers than astronomy is about telescopes". More specifically:

- **Computer Science** is a discipline that seeks to understand and explore the world around us, both natural and artificial, in computational terms. Computer Science is particularly, but by no means exclusively, concerned with the study, design, and implementation of computer systems, and understanding the principles underlying these designs.
- **Information Technology** deals with the purposeful application of computer systems to solve real-world problems, including issues such as the identification of business needs, the specification and installation of hardware and software, and the evaluation of usability. It is the productive, creative and explorative use of technology.

We want our children to understand and play an active role in the digital world that surrounds them, not to be passive consumers of an opaque and mysterious technology. A sound understanding of computing concepts will help them see how to get the best from the systems they use, and how to solve problems when things go wrong. Moreover, citizens able to think in computational terms would be able to understand and rationally argue about issues involving computation, such as software patents, "net neutrality", identity theft, genetic engineering, electronic voting systems for elections, and so on. In a world suffused by computation, every school-leaver should have an understanding of computing.

6.3 Computing is a STEM subject

Computing (embracing computer science, IT, and digital literacy) is a quintessential STEM subject, sharing attributes with Engineering, Mathematics, Science, and Technology:

- Like **mathematics**, it has its own theoretical foundations and mathematical underpinnings, and involves the application of logic and reasoning.
- Like **science**, it embraces measurement and experiment.
- Like **engineering**, it involves the design, construction, and testing of purposeful artefacts.
- It requires understanding, appreciation, and application of a wide range of **technologies**.

Moreover, Computer Science provides pupils with insights into other STEM disciplines, and with skills and knowledge that can be applied to the solution of problems in those disciplines.

Although they are invisible and intangible, software systems are among the largest and most complex artefacts ever created by human beings. The marriage between software and hardware that is necessary to realize computer-based systems increases the level of complexity, and the complex web of inter-relationships between different systems increases it yet further. Understanding this complexity and bringing it under control is the central challenge of our discipline. In a world where computer-based systems have become all pervasive, those individuals and societies that are best equipped to meet this challenge will have a competitive edge.

The combination of computational thinking, a set of computing principles, and a computational approach to problem solving is uniquely empowering. The ability to bring this

[6] This famous phrase is often erroneously attributed to Dijkstra.

combination to bear on practical problems is central to the success of science, engineering, business and commerce in the 21st century.

7. Onwards and upward

This document can make no more than a small contribution to your journey as a computing teacher. What else can you do?

7.1 The CAS community

First and foremost, you should join the Computing at School community. It is a vibrant, fizzing community of practice, in which thousands of teachers and software professionals are working out together how to make the new POS into a live reality in every classroom.

At the time of writing CAS has over 16,000 members, and is growing at over 600 each month. Most are teachers, and most come from the UK, but (crucially) CAS is open to any adult, anywhere in the world, that cares about our children's education in computing.

CAS has more than 130 "hubs" across the country, in-person meetings of teachers where you can share best practice, offer ideas, learn, give back, and encourage each other.

CAS is like the open source movement. You do not pay a subscription in order to receive a service; rather, you join a community of practice to share what you know, and to learn from what others can offer. Membership is free, and you get no email spam. The CAS community site has over 2,500 resources, each created by a CAS member, almost all under a Creative Commons license so that you can download, adapt, and use it in your classroom.

Here is the Getting started in CAS page.

CAS runs a number of projects specifically in support of teachers grappling with the new POS

- CAS Barefoot project: resources and training for primary
- CAS Quickstart project: a CPD course for primary, and another for secondary.
- The CAS Network of Teaching Excellence in Computer Science is an umbrella for the CAS network of hubs, master teachers, lead schools, and universities.

7.2 Other organisations

CAS is just one of a number of a number of non-profit organisations that are each focused on supporting computing for school-age children, including

- Code Club: after-school programming clubs at primary level
- Code.org, although based in the USA, has tons of extremely useful material for teachers, from primary onwards
- Raspberry Pi Foundation created the now-legendary Raspberry Pi computer, and have a very broad range of activities.
- Young Rewired State and CoderDojo run programming workshops for children around the country.

7.3 Have a go

Don't be satisfied with *reading* about computing. Do some yourself! Go to code.org and dive into some of the Hour of Code examples. You'll be writing code in less than five minutes. Then move on to Scratch, Kodu, or TouchDevelop.

Computing is more than programming, but there is no faster way to get a visceral sense of the subject than to start writing some simple code. And it's such fun too.

7.4 Further reading

Here are some places you might look for inspirational material about the concepts and principles of computer science:

- Inspirational books, and articles/lectures/blog posts about computing. These are mostly of a popular-science nature, or aimed directly at teachers. They are not highly technical. For example "Nine algorithms that changed the world", "Computational fairy tale", "Algorithms unplugged", and so on.
- CAS has produced guides to the Programmes of Study, one for primary, and one for secondary.
- **Unplugged activities** are classroom activities that teach computer science *without using computers*. For example, pupils hold balloons and sort themselves into order, or one pupil instructs a blindfolded colleague how to walk a maze, or simulate a network by passing messages written on bits of paper.

  These activities are extremely successful in practice, and have the huge merit of *visibly separating the ideas from the technology* --- because there is no technology. Moreover, it is simple, cheap, fun, and works equally well at primary and secondary level. Using unplugged activities is probably the single most powerful pedagogical strategy for teaching computer science that has emerged in the last decade.

  Here are some unplugged resources, but this list is far from exhaustive:

  - The famous Computer Science Unplugged website, book, and videos.
  - A new set of unplugged activities developed by Teaching London Computing. Follow the "inspiring classroom activities" link.
  - Search for "unplugged" on the CAS Community site.

8. Appendix: "programming" versus "coding"?

You may have seen a lot in the press about "Why we must teach our kids to code". But what exactly is "coding", and is it the same as "programming".

8.1 Formal languages

Computers are programmed using **formal languages**. By "formal" I just meant that they are highly prescribed. They have a fixed grammar, and a computer can execute them blindly just by following the instructions. Knitting patterns are also written in a formal language, for example.

Formal languages vary a lot, and there are many, many such languages designed for highly specific purposes.

- If you add a rule for Outlook to tell it how to process an email message, you are using a formal language.
- Even if you set your out-of-office message to say "start on 3 October, finish on 8 October, send this reply" you are using an (extremely limited) formal language.
- When the phone says "press 1 for sales and 2 for letting", and you press 2, you are using a (very limited) formal language.
- Excel's formula language is a formal language (much more expressive than the previous three)
- HTML (HyperText Markup Language) is a formal language used to describe web pages. You might write

  ```
  <h1>Cloud types</h1>
  <p>One kind of cloud is <em>cumulus</em>.</p>
  ```

  The tag "<h1>" says "A level-1 heading starts here", while "</h1>" says where it ends. Then the "<p>" says a "a paragraph starts here", and the "," pair encloses a word to be emphasised. The web page might look like this:

  Cloud types
  One kind of cloud is *cumulus*.

  The tags constitute a highly-prescribed formal language that describes how to render the text on the web page
- SQL is a formal language, designed for writing database queries.
- Traditional programming languages, like Basic, C, Python, Java, and so on, are all formal languages.
- Conway's Game of Life is a formal language. It may not look like it, because it's non-textual, but it is. The "program" is the initial layout of blobs, and the program "executes" by following the rules of the game.

8.2 Universal languages

Many of the formal languages mentioned above are limited, special-purpose languages. But some formal languages, usually called *programming languages*, are sufficiently expressive to

be **universal**. That is, they can compute *anything*. C, C++, Java, Pascal, FORTRAN, Cobol, etc are all universal languages.

One of the most remarkable results in computer science is that all of these languages are, in a precise sense, equally expressive. That is, anything that can be computed by any of the above languages can be computed by any of the others.

The British computer scientist Alan Turing worked this out, and he designed a super-simple computer called a Turing Machine to make the ideas concrete. A modern microprocessor is mind-bogglingly more complicated than a Turing Machine, and yet anything the microprocessor can compute, a Turing Machine can compute too. It's totally amazing.

We usually reserve the term "programming" and "programming language" for universal languages. To be universal, they usually offer loops or recursion. (But it is sometimes not obvious when a language is universal. For example, totally surprisingly, Conway's Game of Life is in fact universal. If you can write a Java program to compute something you can compute that same thing by setting up the initial blobs of a Conway game, and pressing "start". Then you might have to wait a long time.)

8.3 Coding and programming

As we have seen, there is a spectrum,

- from a huge range of not-very-expressive formal languages, each of which do different things
- to a huge range of universal (programming) languages, which (in quite a precise sense) all do the same thing, though they vary hugely in how convenient they are to use

I think it's quite appropriate to use

- "Coding" for any formal language, no matter how limited
- "Programming" for any universal formal language. So all programming is coding.

Please do not worry about the coding/programming distinction. I have taken time on it here because people often ask, and because the idea of a "universal" language is rather remarkable. But if you ask someone else "What is the difference between coding and programming" you might well get a different answer, and these definitions don't really matter anyway. We should not waste time asking whether a creature is a "frog" or a "toad". It's more interesting to count how many legs it has, or to see how far it can jump.

8.4 Is HTML "programming"?

In this sense, using HTML is coding, but not programming. That is not to downgrade HTML/CSS. It's a very accessible and creative way into a whole collection of useful ideas:

- That you tell a computer what to do, in a very precise way.
- That you can code (HTML web pages) that no one has ever written before: coding is creative.
- That you may write HTML code that doesn't work (i.e. does not display the web page you wanted), and must then reason about what is going wrong and how to fix it.
- That it is possible to write HTML in a more clear or less clear way, which affect how easy it is for others to understand and modify.

- That there is a strong element of *abstraction* going on. When you say `<h1>` in HTML, you mean "render a section heading". You aren't saying *how* do to that. When we add CSS we add the ability to control the "how to do that" part.
- There is nested structure (e.g. you can set the font for a whole chunk of a document)

In fact HTML has one interesting feature that most programming languages don't: it is **declarative**. Most programming languages go "do this, and then do that"; they are **imperative**, and have the notion of flow of control. HTML doesn't. There is no program counter, and no sense of one instruction following another. You just make declarative statements like "this is a heading". There are universal languages like this, called **functional languages**. (I warmly recommend Haskell.)

So far as the POS is concerned, in KS3 it speaks of "at least two programming languages", and I don't think that includes HTML. But HTML/CSS can (and for many teachers will) have an honourable place in a teaching sequence that leads children into the rich and wonderful world of coding and programming.