Programming Paradigms and Computational Thinking

Greg Michaelson

School of Mathematical and Computer Sciences,
Heriot-Watt University, Edinburgh, Scotland, EH14 4AS.
Email: G.Michaelson@hw.ac.uk. Tel: +44-131-451-3422.

Abstract. Given the extraordinarily wide range of both programming languages and methodologies, it is worthwhile seeking some organising principles, both to enable understanding of how to choose amongst them for particular problems, and to teach others how to do so.
The notion of programming paradigms, with associated programming languages and methodologies, is a a well established tenet of Computing education, enshrined in international curricula. However, this notion sits ill with the classic conceptualisation of a scientific paradigm as a dominant world view, which supersedes its predecessors through superior explanatory power.
Furthermore, even granted that there are programming paradigms, it is not at all clear how they are to be characterised and differentiated. Indeed, on closer inspection, apparently disparate programming paradigms are very strongly connected. Rather, we should view them as complementary approaches within a unitary paradigm of computational thinking.

1 Kuhn's paradigms

The idea of a scientific paradigm derives from the work of the Americal philosopher Thomas Kuhn and his highly influential 1962 book "The Structure of Scientific Revolutions"[17]. For Kuhn, a paradigm is a way of conceptualising the world with a unitary body of theory and practice. Kuhn characterises *normal science* as the practice of the prevailing paradigm, and explores the processes underlying *paradigm shifts*, from an old to a new paradigm, in the face of explanatory failure. In turn, these new paradigms become the normal science. Kuhn explores a number of fundamental paradigm shifts, noting that they invariably take place against the intense resistance of practitioners of the prevailing normal science, who have personal and social interests in maintaining the status quo.

For example, in the 16th and 17th centuries, the Ptolemaic geocentric cosmology was displaced by the Copernican heliocentric cosmology. This was driven by new observations, enabled by new optical instruments, which could not be accounted for by the mathematics underpinning the geocentric cosmology. There was intense resistance to the new cosmology, in particular from the Catholic Church which saw heliocentrism as directly contradicting scripture. Indeed, the

Church did not drop its proscription of Copernicus's and Galileo's books until 1835, despite the heliocentric model's universal adoption for astronomy and hence navigation.

It is important to emphasise that, in Kuhn's conception, paradigms do not coexist within normal science. One is always supreme until displaced by another.

Nonetheless, Kuhn notes that the dominant paradigm is not uniform for all practitioners. For example, in the context of different people using quantum mechanics in different contexts, he says:

> What quantum mechanics means to each of them depends upon what courses he has had, what texts he has read, and which journals he studies. It follows that, though a change in quantum-mechanical law will be revolutionary for all of these groups, a change that reflects only on one or another of the paradigm applications of quantum mechanics need be revolutionary only for the members of a particular professional subspeciality. For the rest of the profession and for those who practice other physical sciences, that change need not be revolutionary at all. In short, though quantum mechanics (or Newtonian dynamics, or electromagnetic theory) is a paradigm for many scientific groups, it is not the same paradigm for them all. Therefore, it can simultaneously determine several traditions of normal science that overlap without being coextensive. A revolution produced within one of these traditions will not necessarily extend to the others as well. (p52)

That is, different traditions that develop differentially from a common basis may coexist within normal science. We shall argue that this is a better characterisation of computing than the prevailing notion of co-existing paradigms.

2 Before programming paradigms

When I studied Computer Science in the early 1970s, the world seemed much simpler than today. I was taught of a progression from machine code, via assembly language and CPU specific autocodes, to the three great languages: FORTRAN, for scientific and engineering problems; COBOL for data processing; and Algol60 for algorithm formulation. Here, there is an implicit taxonomy by distance from hardware, and by function.

In first year I learnt Algol60 and ICL 1900 assembly language, and in second year COBOL [1] and PDP 9/15 assembly language. The sole programming methodology was flowcharts: structured and modular programming were very new, and mentioned in passing.

In the second and third year Theory of Computing modules, I learnt about languages motivated by recursive function theory and λ calculus: LISP, PAL and, latterly, ML. These were presented within a canonical mathematical logic framework, of systematised formalisms for proving properties about mathematics

[1] I was very poor at COBOL.

itself. Thus, formal languages and automata, the arithmetic hierarchy and Gödel encodings were central in terms of programming practice.

In the third year Programming Languages module, I learnt about BCPL, SNOBOL, LISP again, Algol68, and the new educational language Pascal. This was a far more eclectic course, focusing on language properties, case by case. We wrote small programs in each.

Shortly after I started teaching Computing, in the late 1970s, there was a popular taxonomy of *generations* of languages. The first three corresponded to the machine code to assembly language to high level language progression presented above: all thoroughly imperative.

The *fourth generation languages* (4GLs), popularised by the spread of mini- and then micro-computers into smaller commercial enterprises, were aimed at non-specialists for data processing, especially for database manipulation and report generation. These were essentially syntactically sugared scripting languages for libraries of general purpose sub-programs, removing the need to write new code for each new problem case. 4GLs have almost entirely given way to spreadsheets and SQL.

The fifth generation *declarative* languages (5GLs) were heralded by the Japanese 5th Generation Program as the basis of new ways of conceptualising hardware and software. These were derived from the mathematical logic roots discussed above. The fundamental distinction was between "logic" languages like Prolog, based on Horn clause logic, and "functional" languages like ML, based on λ calculus and recursive function theory. Perhaps the key contribution of both language styles was the introduction of case based rules or functions structured by pattern matching, and, from functional languages, polymorphic typing. Curiously, these astonishingly useful forms have yet to permeate into mainstream imperative languages.

Despite world wide 5th Generation research throughout the 1980s, in particular in the UK Alvey programme, 5GLs have largely been superseded by object oriented languages, principally Java, C++ and Python, running on the same old Von Neumann processors. Of the declarative languages, the principal survivors are Haskell, and ML in the guise of OCaml.

3 A brief history of programming paradigms

3.1 From Floyd to the ACM Curriculum

The contemporary idea of a programming paradigm may well have originated with Floyd's 1978 Turing Award Lecture[6] "The Paradigms of Programming".

Floyd's notion draws explicitly on Kuhn's. However, his focus is on programming as a problem solving methodology supported by, rather than characterised by, a language. Thus, he contrasts structured programming, based on stepwise refinement followed by information hiding, with branch-and-bound, divide-and-conquer and state-machine paradigms.

Floyd also emphasises that support environments are as important as languages in supporting paradigms.

Finally, Floyd is insistent that language design should be driven by programming practice:

> To the designer of programming languages, I say: unless you can support the paradigms I use when I program, or at least support my extending your language into one that does support my programming methods, I don't need your shiny new languages; like an old car or house, the old language has limitations I have learned to live with. To persuade me of the merit of your language, you must show me how to construct programs in it. I don't want to discourage the design of new languages; I want to encourage the language designer to become a serious student of the details of the design process. (p459-50)

In 1986, Gibbs and Tucker[10], in their proposal for a new Computer Science curriculum, explored the notion of *programming styles* as part of the "Core Principles of Programming" topic:

> This course emphasizes the principles and programming styles that govern the design and implementation of contemporary programming languages. (p208)

Here, they identify as language styles:

> procedural, functional programming, object programming, logic programming, modular programming, data flow, and their uses. (p208)

These are our familiar programming paradigms, reflecting the rapid changes in languages since Floyd, in particular the growth of OO and declarative programming. While these styles differ markedly from Floyd's problem analysis methodologies, as Floyd urges, the style drives the language.

The term and current usage have certainly chrystalised in a contemporaneous special issue of IEEE Software on Multiparadigm Languages and Environments. In his Chief Editor introduction, Shriver[24], bemoans the absence of a common definition of software paradigm. For Shriver, in general:

> a paradigm is a model or approach employed in solving a problem. (p2)

He then characterises a software paradigm as being "induced by or inducing" a confluence of language class, programming environment and software engineering discipline. His examples of language classes include:

> procedural, nonprocedural, functional, visual, logic programnming, and object-oriented languages. (p2)

intersecting with, but not corresponding to, Gibbs' and Tucker's styles. Finally, Shriver says that:

> Software engineering refers to the set of rules and structured thinking that one applies within the environment to produce software systems or components. (p2)

In his Guest Editor introduction to the same edition, Hailpern[12] expands on Shriver's distinction between language-induced paradigms and paradigm-induced languages. For language-induced paradigms, which historically came first, a programming paradigm is:

> An abstract view of a class of programming that describes one means of solving programming problems. (p8)

For the more recent paradigm-induced languages, a programming paradigm is:

> A way of approaching a programming problem. A way of restricting the solution set...A paradigm allows the programmer to use only restricted set of concepts (sic). (p8)

Thus, for Hailpern:

> Now that we can see the abstract paradigms behind the concrete languages, we can begin to design new languages based on these fundamental concepts. (p8)

Hailpern's paradigms include: access oriented, dataflow, data-structure oriented, functionl, imperative, object oriented, parallel, real-time and rules-oriented.

Three years later, Wegner[23], in his Guest Editor introduction to a special issue of ACM Computing Surveys on programming paradigms, also identifies them with classes of language, citing those covered in subsequent papers: distributed, parallel, functional and log programming paradigms. Here, the paradigms are based on commonalities in language constructs, that is, in Shriver's terminology, they are language-induced.

This notion of paradigm encompassing a strong link between programming language and programming style still has considerable currency. It is this notion that I question here. Still, Wegner's comment remains pertinent:

> Computer science has grown so fast, and fashions within computer science change so rapidly, that it is more difficult to identify its dominant paradigms or characterise its essence than in established scientific disciplines like physics or mathematics. (p257)

3.2 Paradigms in the ACM/IEEE Curricula

Since 1991, the internationally influential ACM/IEEE-CS Computing Curricula have deployed various notions of programming and language paradigm.

1991 The 1991 Curricula[9] does not explicitly define paradigm. The curricula is structured as eleven Knowledge Units, with aspects of the above conceptions of paradigm distributed amongst them.

First of all, Floyd's programming approaches appear in "AL: Algorithms, AL8: Problem-Solving Strategies":

1. Greedy algorithms...
2. Divide-and-conquer algorithms...
3. Backtracking algorithms... (p43)

In "PL: Programming Languages, PL1: History and Overview of Programming Languages", we meet:

1. Early Languages; Algol, Fortran, Cobol
2. The evolution of procedural languages [from the Alogols to Ada]
3. Non procedural paradigms and languages; functional (e.g., Lisp), logic (e.g., Prolog), object-oriented (e.g. Smalltalk), and parallel (e.g. Occam) (p59)

Here we have the paradigm supported by the language, from Gibbs and Tucker, Shriver and Hailpern. "PL11: Programming Paradigms" develops this further:

> Introduction to alternative programming paradigms (e.g. functional, logic and object-oriented) and languages (e.g. Lisp, Prolog and Smalltalk, respectively). Program construction using at least two of these paradigms. Advantages and disadvantages vs. the procedural programming paradigm (p63)

However, there is no indication of the link between paradigm and program design, and the suggested laboratories are based on programming in languages corresponding to paradigms.

Finally, in "SE: Software Methodology and Engineering, SE1: Fundamental Problem-Solving Concepts", we return to Floyd with:

> Introduction to the basic ideas of algorithmic problem solving and programming, using principles of top-down design, stepwise refinement and procedural abstraction. basic control structures, data types and input/output conventions
> ...
> 1. Procedural abstraction; parameters
> 2. Control structures; selection, iteration, recursion
> 3. Data types...and their use in problem solving
> 4. The software design process; from specification to implementation; stepwise refinement; graphical representation (p65)

This is the procedural paradigm, presented quite independently of any programming language.

2001 The 2001 revision to the 1991 curriculum[21] again does not define programming or language paradigm but refers to them throughout the document. A central concern is the status of the programming-first model of teaching computing, and alternatives to it. This is posed in terms of alternative paradigms:

> ...we offer three implementations of a programming-first model and three that adopt an alternative paradigm. The programming-first implementations are an imperative-first approach that uses the traditional imperative paradigm, an objects-first approach that emphasizes early use of objects and object-oriented design, and a functional-first approach that introduces algorithmic concepts in a language with a simple functional syntax, such as Scheme. (p24)

Note that logic and parallel programming are no longer primary paradigms.

However, "paradigm" now also refers to a model of, or strategy for, teaching. Thus the three alternatives are:

> a breadth-first approach that begins with a general overview of the discipline, an algorithms-first strategy that focuses on algorithms over syntax, and a hardware-first model that begins with circuits and then builds up through increasingly sophisticated layers in the abstract machine hierarchy.

The algorithm-first strategy is reminiscent of Floyd's paradigms.

The "Programming Fundamentals (PF)" Knowledge Area states that:

> This knowledge area consists of those skills and concepts that are essential to programming practice independent of the underlying paradigm. (p89)

However, "PF1. Fundamental programming constructs" lists as topics:

> Basic syntax and semantics of a higher-level language
> Variables, types, expressions, and assignment
> Simple I/O
> Conditional and iterative control structures
> Functions and parameter passing
> Structured decomposition

which is a classic procedural/imperative paradigm syllabus.

We also meet:

- the distributed paradigm (p94)
- Brief survey of programming paradigms: ... Functional languages...Declarative, non-algorithmic languages...Scripting languages (p113)
- event driven paradigm (p120)

Finally, the Advanced Course area of "Programming Languages" includes "CS343 Programming Paradigms" but there is no further guidance.

Interestingly, the "CS11A. Introduction to Algorithms and Applications" course includes "technology as a catalyst of paradigmatic change (p189)". We shall later return to this idea of computing paradigms defined by technology.

2013 The current ACM/IEEE "Computer Science Curricula 2013"[22] makes central use of the term "paradigm", though still without definition and in a number of different contexts. In the discussion of introductory courses, there is a major "Programming paradigm and choice of language" section which starts:

> A defining factor for many introductory courses is the choice of programming paradigm, which then drives the choice of programming language. Indeed, half of the six introductory course models listed in CC2001 were described by programming paradigm (Imperative-first, Objects-first, Functional-first). Such paradigm-based introductory courses still exist and their relative merits continue to be debated. We note that rather than a particular paradigm or language coming to be favored over time, the past decade has only broadened the list of programming languages now successfully used in introductory courses. (p42)

It is interesting that this emphasises the co-existence of programming and language paradigms, undermining the applicability of Kuhn's notion of the emergence of a dominant paradigm. Indeed, the comparative study of paradigms is suggested as an alternative to a single-paradigm approach:

> Some introductory course sequences choose to provide a presentation of alternative programming paradigms, such as scripting vs. procedural programming or functional vs. object-oriented programming, to give students a greater appreciation of the diverse perspectives in programming, to avoid language-feature fixation, and to disabuse them of the notion that there is a single correct or best programming language. (p42)

Here we find the paradigms of Imperative-first, Objects-first/object oriented, Functional-first/functional, scripting and procedural. The "SDF/Fundamental Programming Concepts" unit adds Event-Driven and Reactive Programming paradigms (p169).

In the "Software Engineering SE/Software Design" unit topics we meet the link between paradigm and methodology:

> Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented (p180)

In terms of methodology, procedural programming is not further characterised: functional and object oriented programming are, as are functional and object oriented languages.

The term "paradigm" is used inconsistently in other Knowledge Areas, with references to:

- Parallel programming paradigms including threading, message passing, event driven techniques, parallel software architectures, and MapReduce (p72)
- the primary paradigms used by learning agents (p126)
- the producer-consumer paradigm. (p151)
- hardware as a computational paradigm (p187)

3.3 Paradigms in UK curricula

In the United Kingdom, post-school national benchmarks for curricula are specified by the independent Quality Assurance Agency for Higher Education (QAA). The Computing benchmark[8] explicitly follows the ACM/IEEE curricula(p9). In turn, the British Computer Society, which accredits UK Computing courses, is based on this QAA statement[2]. Such accreditation is accepted for wider UK Engineering Council, and hence European, chartered registration.

In the QAA benchmark, paradigms are mentioned solely as an aside in a statement on software, tools and materials:

> Apart from exposure to a range of languages covering the different programming paradigms in widespread use, institutions might provide access to tools including... (p12)

Although paradigms are not well defined, we once again see the linkage of languages to paradigms.

In English, Welsh and Northern Irish secondary schools, standards are defined by the UK government's Department for Education. The mandated subject content for A Level[2] Computer Science[7] includes:

> the need for and characteristics of a variety of programming paradigms (p2)

again without further qualification.

Examinations compliant with the standards are set by awarding bodies recognised by the government regulator. Schools may then choose amongst these. The Assessments and Qualifications Alliance (AQA), an independent charity, is one of the five recognised bodies. In the AQA syllabuss[1], in the major topic "4.1 Fundamentals of programming", we find:

> 4.1.2.1 Programming paradigms ...
> Understand the characteristics of the procedural and object-oriented programming paradigms, and have experience of programming in each. (p41)

While functional programming is not named as a paradigm, there is strong subsequent coverage.

However, there is also an independent notion of classifying programming languages, in "4.6.2 Classification of programming languages", which distinguishes high level and low level languages, and machine code and assembly language, and introduces a notion of an imperative high-level language (p68). This seems to hark back to how languages were presented in the 1970s, as discussed above.

[2] Year 13 exit qualification enabling University entrance.

3.4 Programming paradigms are not paradigmatic

We have seen a transition in the notion of a programming paradigm, from problem solving methodologies, via their realisation in languages with appropriate constructs, to being characterised by languages. Nonetheless, suppose we take the notion of programming paradigm at face value. From the above potted history, there seems to be agreement over the last 30 years that we can distinguish at least:

- procedural programming, also known as the imperative paradigm, based on modular decomposition and stepwise refinement, realised in 3rd generation languages like Pascal and C;
- object oriented programming, based on class-oriented modelling, realised in OO languages like Smalltalk, C++, C#, Java and Python;
- functional programming, based on functional abstraction, realised in functional languages like LISP, ML and Haskell.

as core paradigms. Alas, on closer inspection, it is not at all clear how distinct these alleged paradigms actually are.

Let's consider the ambiguous case of structured programming, based on algorithm elaboration using sequence, selection and repetition. Structured programming is a core procedural programming approach for expressing algorithms through stepwise refinement within and from modules.

However, perhaps structured programming constitutes a paradigm in its own right. After all, it's about the only applicable methodology for initial programming in contemporary graphical languages, like Alice and Scratch[3].

But structured programming is a key methodology for OO as well as procedural programming, where it's used to elaborate algorithms within and from methods. Indeed, once classes have been identified, OO programming is pretty well indistinguishable from procedural programming. So maybe structured programming is a subset of procedural programming is a subset of OO programming.

Still, functional programming seems markedly different, even though it's hard to characterise a functional programming methodology other than as programming in a functional language. But, once again, structured programming is a key methodology for functional programming once functions have been identified, even though the language constructs seem markedly different. Sequence is function composition, repetition is recursion, all functional languages have explicit selection constructs and modern languages provide case based function definitions. Furthermore, modular decomposition and stepwise refinement are as applicable to functional as to procedural programming. So maybe structured programming is a subset of procedural programming, which is a subset of both OO and functional programming - see Figure 1.

From the perspective of establishing taxonomies of programming languages, both Krishnamurthi[16] and Harper[13] question the efficacy of the notion of distinct programming paradigms tied strongly to distinct languages. Krishnamurthi

[3] Yes, some graphical languages support procedural abstraction.

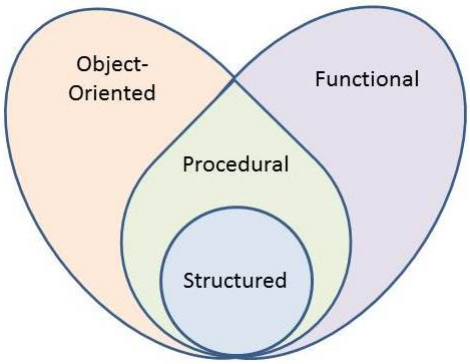

Fig. 1. Programming paradigms are not distinct

focuses on a spectrum of language features through definitional interpreters while Harper promotes understanding of programming languages in terms of types. Contrariwise, van Roy[26] identifies thirty different programming paradigms, using a taxonomy based on both operational properties and types. I have sympathy with these views of programming languages, but argue that they are orthogonal to identifying programming paradigms.

4 Is there a paradigm for Computer Science?

Tedre[25] questions whether Computer Science is mature enough to even speak of paradigms.

We certainly can't identify disjunctions in programming practice comparable to those from, say, Newtonian to relativistic mechanics. Rather, we see an evolution, where older programming practices build on and are absorbed into newer ones. For example, 70 years on from ENIAC, flowcharts are still used, and 1990s UML includes 1960s state machines.

We also can't identify disjunctions in language use; rather, new languages coexist with the old, which slowly fade away or morph to more closely resemble the new. Thus, C and ML now both have OO descendents, in OCaml , and C++ and C#, respectively. Despite the best endeavours of language warriors, programming paradigms are much more akin to Kuhn's co-extant traditions rooted within a common paradigm..

This is hardly surprising. As Turing and Church hypothesised well before digital computers[3], all known models of computability may be shown to be Turing complete, that is equivalent to Turing machines, through sound schema for translating instances of any one model into instances of any other. Similarly, all "full strength" programming languages may be shown to be equivalent, through schema for translating a program in any one language into any other. Indeed, it would be most disconcerting if programs that supposedly solved the

same problem in different languages gave different outputs from the same inputs. Ultimately, just as all flesh is grass, all executable code is machine code.

Thus, I strongly contest Tedre. As we saw, for Kuhn, one paradigm is dominant in any given epoch, until it is displaced by another paradigm with greater explanatory power. Thus, in Kuhnian terms, Turing complete finite computation, now over 80 years old, is patently the dominant paradigm for Computer Science, providing a substantial body of mathematical theory underpinning the world changing practice of constructing and programming computers.

Arguably, Turing complete computation is the first as well as the only Computier Science paradigm. It was thought that analogue computing, based on differential equations reified as differential analysers and then amplifiers, constituted a prior paradigm, but analogue computing has also been shown to be Turing complete[11]. Similarly, quantum computing is widely heralded as disruptive but Deutch[5] long established that it is Turing complete.

Trans-finite hypercomputation would constitute a new paradigm by definition, as hypercomputers can allegedly solve decidability problems that Turing machines can't, as, for example, Hogarth argues[14]. However, the possibility of hypercomputation is hotly contested[4].

5 Computational thinking is the programming paradigm

If Turing complete computation is the paradigm for Computer Science, then I argue that Computational Thinking (CT) is the complementary paradigm for programming. Following Wing's highly influential intervention[27] there has been sustained interest in CT as a basis for teaching programming. Michaelson[20] characterises a spectrum, from a weak CT, which sees pretty well all intellectual endeavour as CT, to a strong CT of [4], systematic problem solving. Strong CT is based on Kao's four elements[15] of:

- Decomposition: the ability to break down a problem into subproblems;
- Pattern recognition: the ability to notice similarities, differences, properties, or trends in data;
- Pattern generalization: the ability to extract unnecessary details and generalize those that are necessary in order to define a concept or idea in general terms.
- Algorithm design: the ability to build a repeatable, stepbystep process to solve a particular problem. (p6)

Furthermore, pretty well all historical and contemporary approaches to programming can be encompassed in CT[18]. These may be summarised as:

- programming language oriented: full strength language, pedagogic or full strength subset, functional, logic, object oriented, language independent/pseudo code;

[4] a practice to which I subscribe.

– simple to complex: stepwise refinement, structured programming, iterative prototyping;
– components: modular programming, algorithms, data structures, types, classes, libraries;
– design: flowcharts, data flow, entity relationship, UML.

These are all are variants of decomposition and algorithm design, with functional and logic programming, and type/data structure/class based approaches, also encompassing pattern recognition and generalization.

More important, all of these approaches are in principle programming language independent. For example, it is unremarkable to implement a design derived in what might be characterised as a procedural approach in an OO language: indeed, much initial teaching is based on just this. Similarly, it is straightforward to implement an OO design in a procedural language, with disciplined use of global sub-programs and data structures.

Implementing strongly imperative designs in functional languages is more complicated, with copying substituting for in-place update. In contrast, implementing functional designs that don't take strong advantage of functional abstraction is straightforward in procedural or OO languages which now invariably support recursion. And functional abstraction may be realised through cunning use of jump tables or classes.

Of course, some programming approaches are more easily realised in languages that directly support corresponding constructs. And, in practice, a prespecified language will strongly influence the programming approach. Nonetheless, I argue that different programming approaches and languages represent complementary traditions within a unitary Computational Thinking paradigm, rather than distinct paradigms.

Acknowledgements

This paper grew from an invited seminar on Programming Paradigms, for Computing school teachers in the North West of England in December 2017. Subsequently, I summarised the seminar in a brief account for the magazine *(Hello World)*[19].

I'd like to thank:

– Roger Davies, Paul Revell and participants at the CAS NW England HTP Day;
– Allen Tucker for stimulating discussion about the origins of programming paradigms;
– Andrew McGettrick and Mark Hogarth for help in locating references.

References

1. Assessments and Qualifications Alliance. *AS AND A-LEVEL COMPUTER SCIENCE: AS (7516), A-level (7517).* Version 1.4. AQA, December 2016. `http://filestore.aqa.org.uk/resources/computing/specifications/AQA-7516-7517-SP-2015.PDF`.

2. British Computer Society. *Guidelines on course accreditation Information for universities and colleges*. BCS, January 2018. http://www.bcs.org/upload/pdf/2018-guidelines.pdf.
3. P. Cockshott, L. M. Mackenzie, and G. Michaelson. *Computation and its Limits*. OUP, 2012.
4. P. Cockshott and G. Michaelson. Are there new models of computation: a reply to Wegner and Eberbach. *Computer Journal*, 50(2):232–247, 2007.
5. D. Deutch. Quantum theory, the ChurchTuring principle and the universal quantum computer. *Proceedings of the Royal Society A*, 400(1818), July 1985.
6. R.W. Floyd. The Paradigms of Programming. *Computer Journal*, 22(8):455–460, 1979.
7. Department for Education. *GCE AS and A level subject content for computer science* . DFE-00359-2014. DFE, April 2014. https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/302105/A_level_computer_science_subject_content.pdf.
8. Quality Assurance Agency for Higher Education. *Subject Benchmark Statement: Computing*. QAA, February 2016. http://www.qaa.ac.uk/en/Publications/Documents/SBS-Computing-16.pdf.
9. ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. ACM/IEEE-CS, 1991.
10. N. E. Gibbs and A. B. Tucker. A model curriculum for a liberal arts degree in Computer Science. *CACM*, 29(3):202–210, 1986.
11. D. S. Graca and J. S. Costa. Analog computers and recursive functions over the reals. *Journal of Complexity*, 19(5):644–664, October 2003.
12. B. Hailpern. Multiparadigm Languages. *IEEE Software*, pages 6–9, January 1986.
13. R. Harper. What, if anything, is a programming paradigm? *fifteeneightfour blog, CUP*, May 2017. http://www.cambridgeblog.org/2017/05/what-if-anything-is-a-programming-paradigm/.
14. M. Hogarth. Non-Turing Computers are the New Non-Euclidean Geometries. *International Journal of Unconventional Computing*, 5:27791, 2009.
15. E. Kao. Exploring Computational Thinking at Google. *CSTA Voice*, 7(2):6, May 2011.
16. S. Krishnamurthi. Teaching Programming Languages in a Post-Linnaean Age. In *2008 SIGPLAN Workshop on Programming Language Curriculum*, May 2008.
17. T. S. Kuhn. *The Structure of Scientific Revolutions*. Chicago University Press, 1962.
18. G. Michaelson. Teaching programming with computational and informational thinking. *Journal of Pedagogic Development*, 5(1), March 2015.
19. G. Michaelson. Are there programming paradigms? *(Hello World)*, 4:32–33, January 2018.
20. G. Michaelson. *Microworlds, Objects First, Computational Thinking and Programming*. 2018. (in press).
21. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computing Curricula 2001 Computer Science Final Report*. ACM/IEEE-CS, December 2001.
22. ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM/IEEE-CS, 2013. https://www.acm.org/binaries/content/assets/education/cs2013_web_final.pdf.
23. P.Wegner. Programming Language Paradigms. *ACM Computing Surveys*, 21(3):253–258, September 1989.

24. B. D. Shriver. Software paradigms. *IEEE Software*, page 2, January 1986.
25. M. Tedre. *The Development of Computer SCience: A Sociocultural perspective*. PhD thesis, University of Joensuu, 2006.
26. P. van Roy. *Programming Paradigms for Dummies: What Every Programmer Should Know*. IRCAM/Delatour, France, 2009.
27. J. M. Wing. Computational Thinking. *CACM Viewpoint*, pages 33–35, March 2006.