17 Computational Thinking

Paul Curzon, Tim Bell, Jane Waite, and Mark Dorling

17.1 Motivational Context

The term "computational thinking" was popularized by Wing (2006) as the form of thinking computer scientists practice. Computational thinking has since been widely accepted and promoted both as the skill set that programmers develop and as the general thinking skills that should be developed by computer scientists as they learn the discipline. Wing also advocated it as a generally useful problem-solving skill set that all should learn. Computational thinking also arguably offers a powerful way of both thinking and doing across a wide range of subject disciplines, transforming the way that they are carried out, such as through the use of computational modeling.

17.1.1 Computation

Computational thinking is not primarily about the development of electronic computer systems. It is about computation and the development of systems based on computation. Computation dates back millennia. The first algorithms were developed thousands of years before digital computers. One of the earliest, and most famous, is Euclid's algorithm (c. 300 BCE; cited in Euclid, 1997) for computing the greatest common divisor of two numbers. The word "algorithm" derives from the name of the Muslim scholar Muḥammad ibn Mūsā al-Khwārizmī and is most closely associated with his work *On the Calculation with Hindu Numerals* (al-Khwārizmī, c. 825). It concerns the algorithms for doing arithmetic with decimal positional numbers. Computation is not just about numeric calculation, however. It concerns symbol processing more generally. Early algorithms, predating electronic computers, include encryption-related algorithms that concern the manipulation of letters and other symbols. Computation does not need to be done by machines, of course. Humans can follow algorithms, and al-Khwārizmī's book was about algorithms for people to follow. Indeed, the first actual "computers" were people, not machines. The term was originally used to describe the people tasked with doing the astronomical calculations needed to develop maritime tables for navigation at sea (OED, 1993). Indeed, Charles Babbage did this job, and it was a motivation for him to develop machines that could do the calculations automatically. The developers of these pre-computer age algorithms were certainly engaged in a form of computational thinking, in the sense of solving computational problems through precise algorithmic solutions.

Turing (1936) famously articulated a formal idea of computation in the thought experiment of a Turing machine, and a variety of other models of computation have been devised that have been proved equivalent. These models define the limits of what computation, and so algorithms, can do. Since computational thinking concerns the design of computational systems, these theories give limits on the possible.

Computation is not restricted to the manipulation of abstract symbols. It can and does happen to physical things in the world that embody information, and not just inside computer chips (which are embodiments of computation in the physical world too). Computation in such a computational system involves information processing through, for example, the movement and transformation of information between different physical objects. This is a core idea behind distributed cognition (Hutchins, 1995), where the brain is seen as an information processing agent and cognition is seen as extending to incorporate such computational systems in the world. Hutchins' core example is analysis of the computational properties of ship navigation, exploring how information is transformed as it passes between different forms, physical and mental. This richer view of computation is actually vital in the development of the modern computer systems that play an increasingly physical role in the world, augmenting human processes in complex ways.

This view of computation as including movement and transformation of physical objects means that "unplugged computing," where physical objects and role play are used to illustrate computing concepts (Bell, Alexander, Freeman, & Grimley, 2009; Bell, Rosamond, & Casey 2012), is not just the use of analogy, but is actually about computation itself. Computational thinking is being done in devising unplugged computational systems, whether inventing a self-working magic trick (an algorithm for a magical effect) as illustrated by Curzon and McOwan (2017) or devising an activity of searching for numbered balls under cups using a binary search algorithm. This mirrors real-world, everyday uses of computational thinking too, such as when a teacher, presented with a pile of 400 paper exam scripts that must be put in sorted order by ten-digit student number, devises a form of radix sorting as an efficient way to do so in preference to using some variation of bubble sorting. A more forward-thinking computational thinker might later redesign the system as a whole, allocating desks to students in the required order, allowing the students to physically sort themselves and so their scripts. Even without turning to digital solutions, algorithmic thinking is useful.

17.1.2 What Is Computational Thinking?

Wing (2006) is clear that computational thinking is about thinking like a computer scientist. It is, however, also a fundamental analytical skill for everyone, not just for computer scientists. She is also clear that the concept she is defining is about computing processes, whether they are executed by a human or by a machine. It is specifically not just the skill of computer programming, but

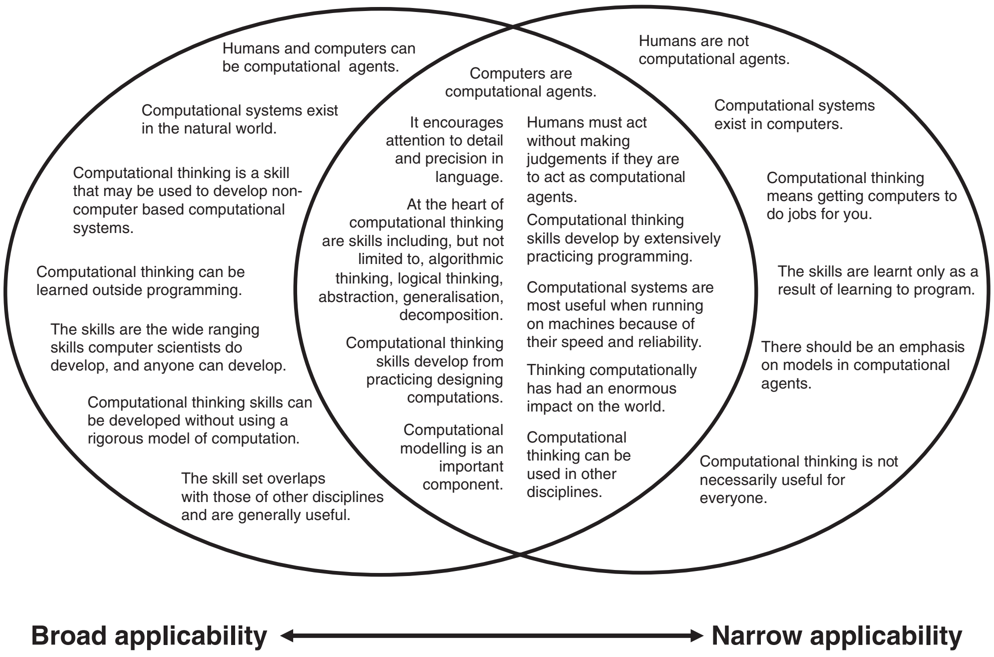
Humans and computers can be computational agents.
Computational systems exist in the natural world.
Computational thinking is a skill that may be used to develop non-computer based computational systems.
Computational thinking can be learned outside programming.
The skills are the wide ranging skills computer scientists do develop, and anyone can develop.
Computational thinking skills can be developed without using a rigorous model of computation.
The skill set overlaps with those of other disciplines and are generally useful.
Computers are computational agents.
It encourages attention to detail and precision in language.
At the heart of computational thinking are skills including, but not limited to, algorithmic thinking, logical thinking, abstraction, generalisation, decomposition.
Computational thinking skills develop from practicing designing computations.
Computational modelling is an important component.
Humans must act without making judgements if they are to act as computational agents.
Computational thinking skills develop by extensively practicing programming.
Computational systems are most useful when running on machines because of their speed and reliability.
Thinking computationally has had an enormous impact on the world.
Computational thinking can be used in other disciplines.
Humans are not computational agents.
Computational systems exist in computers.
Computational thinking means getting computers to do jobs for you.
The skills are learnt only as a result of learning to program.
There should be an emphasis on models in computational agents.
Computational thinking is not necessarily useful for everyone.
Broad applicability
Narrow applicability

Figure 17.1 *Agreement and disagreement around two views of what computational thinking should be.*

the much wider way of thinking that computer scientists (not specifically programmers) develop.

There are, unfortunately, now a wide variety of sometimes polarized views over what computational thinking should be (Denning, 2017; Tedre & Denning, 2016; Denning & Tedre, 2019). This has led to problems, not least that research studies use different definitions, often without being clear what they mean by the term. This diversity of views is largely a result of how successful the original definition was, resonating around the world. This success has led to it being incorporated into education systems globally, and this has made its meaning an issue of politics, with different groups using it with their own definition to fit their own priorities and agendas. Views mainly differ on the breadth of applicability and the nature of computational agents (Figure 17.1). Most literature is closer to the middle of this diagram, but the authors have regularly encountered professionals who argue strongly for one of the extreme views.

Despite the different views, it is ultimately more useful as an educator to focus on the agreement, which as Figure 17.1 shows is large, and not worry which end of the spectrum resonates personally. There is general agreement around a large central core (see Section 17.3 for a deeper summary) that computational thinking is the way of thinking used to develop solutions in a form that ultimately allows "information processing" or "computational" agents to execute those solutions. The computational agent should be guaranteed to achieve some specified result without further thought or problem-solving involved, just by blindly and precisely following the solution. Ultimately, solutions are not one-off answers like "the cheapest route is via Hong Kong," but rather are *algorithms* that solve a general case (e.g., "find the cheapest route"). Computational thinking is thus concerned with the development of systems involving information processing, and it is the focus on algorithmic solutions that differentiates it from other problem-solving approaches. There are different views, however, on what can be a computational agent. It could be a machine or human (or possibly even an animal or other biological system if it can follow those instructions precisely and blindly). It could also be a combination of both.

Programming relies directly on this skill set, but computational system design and development is about far more than just coding itself. The development of higher levels of computational systems relies on these skills, as does innovation in computing more generally. As Wing (2006) notes, "Thinking like a computer scientist means more than being able to program a computer. It requires thinking at multiple levels of abstraction."

Although the idea of "computational thinking" has been around for centuries, the term was first used by Papert (1980) as part of his call for a new approach to teaching mathematics based on computational methods (see also Chapters 1, 19, 20, and 22). This original definition is about the idea that computational thinking is a way of doing other subjects differently. He suggested it as part of a teaching methodology for computational environments through the Logo programming language. In this context, it can be seen more as a novel way of gaining understanding rather than narrowly about solving problems. It is this idea that is transforming science and leading to innovation more generally (see Section 17.1.9). However, it was Wing's use of the term, not Papert's, which led to the concept being widely adopted.

17.1.3 A "Traditional" View or Not?

Denning (2017) has brought differing views to a head. He identifies what he calls a "traditional view". Essentially, this boils down to the idea that computational thinking should be based on computational models and algorithms that have definite computational steps. Part of this view is that computational agents must act like machines (or, at least, well-defined models of machines) and therefore are most likely to be encountered by beginners when developing software (i.e., developing instructions in formally defined languages for electronic computers of the kind that currently exist). Denning claims that there is no evidence that developing programming skills alone extends to more general problem-solving, so this justification of computational thinking for all should be dropped, at least until evidence is produced that it has broader benefits. He argues that the ultimate goal of computational thinking is computational design. More widely, its goal is computational *systems* design.

As many of the concepts that have been espoused as fundamental to the term "computational thinking" are well known in other disciplines, Denning (2017) argues that computational thinking should not be given the special status it has as a general problem-solving approach.

Wing (2006), on the other hand, argues that computational thinking is much more than this. It is both a skill that leads to programming ability and a generally useful skill. A consequence of this view is that it can be learned separately from programming. Even if programming does not lead to general problem-solving skill, this wider definition of computational thinking that intersects with other subject views of problem-solving may lead to more general problem-solving skills.

Denning outlines a series of precursors to Wing in discussing the general skill set developed by programmers and advises we stick to Aho's more recent though "historically well-grounded definition":

> Mathematical abstractions called models are at the heart of computation and computational thinking. Computation is a process that is defined in terms of an underlying model of computation and computational thinking is the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms.
> (Aho, 2012, pp. 834–835)

Denning grounds the skill set of computer scientists firmly in working throughout with defined models of computation, and rules out calling anything computation that is not based on such a model. It appears to rule out both informal demonstrations (such as the classic sandwich-making exercise where instructions are followed literally) and working at higher levels of abstraction without a specific model targeted. However, a major point of thinking at higher levels of abstraction for a problem means the details (and model) at lower levels are explicitly ignored, and this skill of working at all levels needs to be developed. That is part of the power of computational thinking and is certainly important in creating programs.

17.1.4 What Is a Computational Agent?

A key question is whether only machines should be classed as computational agents. If so, this leads to the position that computational thinking is only concerned with the creation of programs. If so, then arguably there is no need for a new term of "computational thinking" at all, as programming itself is the skill set.

Wing and many others since have argued for a wide definition. At the outset, Wing (2006), for example, stated that computational agents can be humans, not just machines. This wider view puts the emphasis not on machines or programming, but on information processing and the design and understanding of systems that do such information processing. Humans can and do perform such information processing, though they are clearly less capable of following instructions precisely. This is embodied in computing curricula in England and other countries, and it is because of this wide definition that the idea of computational thinking has become so widespread. It is also the foundation of the arguments for computing for all and the basis of the resultant push around the world that it, and not just programming, should be taught in school, not just in higher education. A significant reason for this push is because it makes clear that computational thinking is a useful tool for all to learn, not just programmers. If it is just useful for programmers as a skill, then there is far less justification for teaching it to all from primary upwards.

Denning (2017) argues for a narrower definition: that a computational agent should not involve human judgment. This frames it as a skill for students who are learning to program, since this is the environment in which a beginner might encounter such computational agents. He argues that there is no evidence that this narrow version has any transferable benefits beyond computing itself and therefore such claims should be dropped. Lee (2016) takes a related view, arguing that computational thinking is definitely not about creating algorithms for humans to follow, but that it is more than just programming. She puts the emphasis on it being about taking *real-world problems* and creating abstractions of them and algorithms that solve them, which are then implemented on computers.

17.1.5 An Evolving Definition

Denning, in part, is reacting to the way a range of authors have adapted the meaning of the term. According to Dagienė et al. (2017), authors that have further developed the meaning include Grover and Pea (2013), Kalelioglu et al. (2016), Lu and Fletcher (2009), Selby and Woollard (2013), and Wolz et al. (2011). These authors have argued that there is a place in the progression of learning to think computationally for activities that do not necessarily result in implementing a programmed solution.

For example, Lu and Fletcher (2009), though taking computational thinking to be about solving problems with computers, explicitly argued that it should be split from programming in the early years. They argued that the focus should be on

> establishing vocabularies and symbols that can be used to annotate and describe computation and abstraction, suggest information and execution, and provide notation around which mental models of processes can be built. (Lu & Fletcher, 2009, p. 260)

They posited that doing so would lead to students being in a better position, with such a foundation, to learn both programming and more advanced computing. Their argument is that a computational thinking language (CTL) must permeate the pedagogy. They give wide-ranging examples of how such language and unplugged computational thinking might be developed in an interdisciplinary way from US curricula, all concerning the students doing computation, not writing programs. The focus is on computation and information processing tasks generally, not on how they are specifically implemented in computers. Curzon's practical approach to teaching programming itself, as well as data structure and algorithms concepts (Curzon, 2002), takes a similar explanatory approach, teaching programming ideas divorced from writing programs, using wide-ranging analogies with real-world processes to explain general computing concepts, but getting completely away from the syntax and detailed semantics of a specific language. He advocates the same approach for introducing computational thinking as embodied by the Teaching London Computing website (http://teachinglondoncomputing.org).

Selby and Woollard (2013), in searching for an appropriate definition to use in school education, looked for consensus. They surveyed the literature concerned with computational thinking and came up with a definition based on the most commonly agreed-upon components of computational thinking. Computing at School (CAS) adopted this basic definition to promote computational thinking in England (Csizmadia et al., 2015). This is based on the five top-level categories that were most commonly encountered and so showed the most consensus from the research community: algorithmic thinking, abstraction, decomposition, generalization, and evaluation.

Further evolution of the term may also be needed if "computational thinking" is taken to be the skill set needed to develop computational systems in the future. For example, Chapter 20 argues for the need to include more explicitly the underlying skills needed for new areas such as machine learning, distributed computing, and quantum computing paradigms. This implies that not just logical thinking, but also statistical and probabilistic thinking skills will be needed for the development of computational systems in the future.

17.1.6 Problem-Solving, Expression, Creativity, and Communication

Computational thinking is currently mainly associated with problem-solving, but this may limit the opportunities for getting the most from it, as well as limiting those who might be attracted to using it. Bers (2017) links back to Papert's call for technological fluency as well as computational thinking (Papert, 1980), describing technological fluency as when one can express oneself "creatively, in a fluent way, effortlessly and smoothly as one does with language" (Bers, 2008). She sees computational thinking as too associated with problems and draws out the potential for using it for communication, creativity, and expression beyond science, technology, engineering, and mathematics (STEM). This resonates with classroom practice, where students probably do not see their programs as algorithms that solve a problem, but as instructions that make something happen. Brennan and Resnick's (2012) computational thinking concepts, practices, and perspectives meet Denning's point about the link between computational thinking and programming, but they highlight the importance of expression, connecting, and questioning as a means to "code to learn" rather than just learning to code (Resnick, 2013). Kafai (2016) advocates for "computational participation" rather than just computational thinking. Calling it "participation" allows for more emphasis on community, where code is written to be shared rather than a disposable exercise, where developing software occurs in the context of a community, and where others' work can be remixed.

17.1.7 Pedagogy and Pragmatism

There are two distinct issues in this debate that need to be separated: (1) What is the skill set involved in computational thinking? (2) How does one take children on the journey to gain those skills? Teaching is a pragmatic endeavor. Many subjects are taught using a spiral learning model where "lies to children" (misconceptions) are used to simplify concepts in order to make them accessible at a younger age. Denning and Tedre (2019) highlight that what computational thinking really is varies by necessity as students progress from beginner to professional, and that failing to make this distinction leads to conflict. Whether or not one wishes to take a narrow or wide definition of computational thinking, there is still a great deal of benefit to be gained by looking for a simplified progression and taking a constructivist approach, building on everyday ideas that are already understood. There are good pedagogic reasons for doing this. It involves presenting more accessible versions of computational thinking to younger age groups – ones that fudge the details. A perfectly sensible first step is to work on writing clear instructions with young primary school students (such as telling a friend how to walk the outline of a square, writing recipes, or writing out a dance routine). Later, the limitations of human instructions such as recipes can be explored, and this can motivate the need for the more rigorous, formal treatment found in programs. Foundational concepts can be introduced in this way, then built on and refined as one progresses through the education system.

Even if analogical approaches are rejected, the issue is not whether there is a formal model per se, but whether a computational agent, in principle, could blindly follow the resulting algorithm, and if it did so accurately, whether it would guarantee the same outcome over and over again. Attempts at creating algorithms by younger learners are likely to be incomplete, inaccurate, and clumsy, but as they make progress, precision, completeness, cohesion, and elegance will improve, thereby showing such progression. This will apply whether one is starting to write programs, recipes, or other instructions for humans. Just because the young, novice developer has not provided or worked with a precise model of computation, a mathematical semantic model, or implemented a fully working, complete computational stack all the way down doesn't mean they are not thinking computationally or about computation. Human-centered software development also does not start with algorithms and models, but rather with understanding human and socio-technical needs. It is only later in good development processes that underlying models, precision, and completeness enter the scene.

Despite issues with definitions, pragmatically, the idea of computational thinking has proved immensely useful, especially in putting a focus on the importance of the general skill set of computer scientists, as well as promoting computing for all in education systems worldwide. It has helped improve the standing of computing in schools as a rigorous subject that is more than office IT skills in a variety of countries.

Whichever definition we adopt, it is clear that programming is a key step in teaching computational thinking, but also that learners build upon existing knowledge and skills starting with relevant and familiar contexts. Similarly, we also need to be aware that becoming a programmer is not the goal for all students, in the same way that becoming a professional novelist is not the goal for all those learning to write.

17.1.8 Changes to School Curricula

A consequence of the push arising from Wing's seminal article (Wing, 2006) has been that computational thinking has become a foundation stone of new syllabuses of computing. It has also provided a useful term to articulate changes that were already planned for curricula and enabled curriculum designers to articulate the broader principles intended in revised curricula. This has especially avoided the perception that the changes were only about esoteric topics of interest to programmers or just about developing skills to groom students for work purely in the software development industry. For example, the purpose of study of the English National Curriculum for Computing (Department for Education, 2013), which applies from primary school upwards, starts: "A high-quality computing education equips pupils to use computational thinking and creativity to understand and change the world." A key aim, in line with the Royal Society report (Royal Society, 2012), was that computing should be more than just programming. Computational thinking was placed at its heart in part to emphasize this, following Wing's definition. Explicit aims include the following:

- "can understand and apply the fundamental principles and concepts of computer science, including abstraction, logic, algorithms and data representation"
- "can analyse problems in computational terms"

An outline of skill and knowledge progression is set out. For example, at ages 5–7, pupils should be able to "understand what algorithms are; how they are implemented as programs on digital devices; and that programs execute by following precise and unambiguous instructions." At ages 7–11, pupils should be able to (among other things): "solve problems by decomposing them into smaller parts," and also "use logical reasoning to explain how some simple algorithms work and to detect and correct errors in algorithms and programs." At ages 11–14, pupils should be able to "design, use and evaluate computational abstractions," and "use logical reasoning to compare the utility of alternative algorithms for the same problem." For more on this, see also reference to the US curriculum in Chapter 20 and in other countries in Chapter 18.

Computational thinking is increasingly being made a central framework as countries update their school curricula. A review of computing education in K–12 schools across 12 countries (Hubwieser et al., 2015) revealed that computational thinking or algorithmic concepts were now addressed by curricula in Germany/Bavaria, France, New Zealand, Finland, the USA, Israel, Russia, the UK, Korea, Sweden, and India. New Zealand has even called the core computing part of their proposed curriculum “computational thinking” (NZ Ministry of Education, 2017).

As these ideas are embedded in national curricula from primary school upwards, making sure such interventions now deliver practical benefits to the students involved is a vital and pressing issue.

17.1.9 Practical Benefits

One purported benefit of computational thinking skills is that they are the basis of being able to program. With such thought processes in place, programming becomes easier and better programs are written. However, it is more than just about low-level programming. Many of the same skills apply in designing hardware systems too, and in developing systems at higher levels. Developing maintainable, usable, and used software systems needs more than just coding skills. Modern computer systems are socio-technical systems, and the design of real-world systems requires an understanding of the wider systems, so development of the skills to design such complex computational systems through experience is more than just programming.

The explosion of interest in computational thinking has come about not because of its basis in programming per se, but because of the argument that it is a general problem-solving skill set and mode of thought that is desirable for more than just programmers or even computer scientists to possess. The world is now digital as well as physical, and it is argued that everyone can benefit from being able to think algorithmically and understand deeply how the digital world works, particularly how it is driven by algorithms. This is important as if you understand how something is constructed, whether physical or digital, then you have a stronger basis for understanding its potential uses and its effects on society (Royal Society, 2012, 2017a, 2017b). For example, a policy-maker who understood how GPS algorithms work would be in a better position to see how it would transform the way we do so many things and to see new possibilities for it, including new cyber-threats, such as it potentially being spoofed. A hiker would also better understand the risk of losing that signal when entering a deep, narrow canyon. To take a different real case, the lives of several innocent nurses were blighted due to such a lack of understanding (Thimbleby, 2018). In 2018, a UK court case was brought against these nurses, accusing them of negligence and, in particular, of fabricating paper patient records that differed from automated computer logs of tests administered. At the last minute, the prosecution offered no evidence when expert witnesses showed that hospital administrators, police, and prosecutors had not understood enough about the way the system worked or was used to realize that those computer logs could be inaccurate. Had the hospital administrators understood the algorithms more deeply, they might also have procured a more reliable system. It also matters in the sense of being able to contribute to the development of appropriate algorithmic socio-technical solutions to problems. Participatory design is a powerful way to develop systems that truly work for the people involved.

Thinking about socio-technical systems as computational systems is a new way of thinking about them that is important to anyone who is already operating in, who could be operating in, or who is interacting with the digital world. Note that this is not a point just about using technology or the introduction of the term "computational thinking" per se (the introduction of the term didn't represent the point in time when that way of thinking started to exist), but about the way of thinking it embodies, which long predates the term, as discussed in Section 17.1.4. It matters not just in terms of designing interactive systems or in understanding how digital devices work, but also in making informed decisions as citizens about ethical issues where we may choose (or not) to place limits on how we use computation (such as artificial intelligence in decision-making or self-driving vehicles).

A more general argument still is that even aside from understanding and making the best use of digital technology, thinking of systems explicitly as computational systems and of algorithmic ways of doing things can make us all more effective in everyday life, whether working in a coffee shop, running a factory, or sorting exam scripts. If we think computationally about the things we do, then we can develop more effective ways of working or achieving tasks more generally. Whether we call this "computational thinking" or not is merely a matter of definition of terms.

Thinking computationally is about more than just problem-solving: it provides a whole new way of thinking. Millican and Clark (1996) and Millican (n.d.), for example, argue that new modes of explanation based on the ideas stemming from Turing have had a revolutionary impact on philosophy and the intellectual world more generally, providing a new algorithmic mode of explanation. This is one basis for the idea that computational thinking is of use for all. There is "clear potential for algorithmic explanation in such fields as psychology, politics, sociology, and economics" (Millican, n.d.) as well as the traditional sciences. This is as big a revolution as those of Newton or Einstein on our modes of thought.

These reasons apply whether for scientists, lawyers, artists, or politicians. It provides a new lens through which to look at (and understand) the world and so craft new ways of doing things, new ways of working. Taking law as an example, Susskind (2017) makes the case that lawyers must "start to innovate, to practice law in ways that we could not have done in the past," in part due to computing innovation. Lawyers who can drive this themselves and even directly contribute rather than relying on computer scientists will have a big advantage. "We require a new cadre of self-sufficient legal technologists whose impact on modern society will be profound." He outlines a range of future careers for lawyers with computer science skills, including systems engineering and programming, and suggests a computer science degree will be one future route to becoming a lawyer. Such lawyers of the future will need computational thinking skills as well as legal skills in order to both see and grab opportunities. Similar arguments to Susskind's apply across the spectrum of professions.

The way science is conducted has already changed profoundly. In the past, science was moved forward by theory – rigorous thinking about the possibilities – and empirical experiment in the real world. There is now a third way. Phenomena of interest can be modeled algorithmically: theory is encoded with algorithmic rules. The phenomena can then be explored through simulation or proof. Virtual experiments can be performed on the models created, exploring the consequences of the rules, including emergent properties. This can be compared with the results of experiments. If the results differ, then it suggests the rules, and therefore the underlying understanding, need further refinement. It also leads to prediction for real experiments. It can thus drive both theory and empirical research, and it applies even to massively complex systems such as the climate.

Computational modeling dates back to some of the earliest uses of computers, where complex calculations were needed to understand phenomena. The early EDSAC (Electronic Delay Storage Automatic Calculator) family of computers contributed in this way to the work of three Nobel Prize winners, in Chemistry, Medicine, and Physics. John Kendrew and Max Perutz credited it for the discovery of the structure of myoglobin, Andrew Huxley for work on understanding the way nerves function, and Martin Ryle for work in radio astronomy. All acknowledged EDSAC in their Nobel Prize speeches. The astronomer, Joyce Wheeler, also used EDSAC to investigate the nuclear reactions that keep stars burning. Computational modeling of the weather by Edward Lorenz led to the observations that small changes to inputs led to widely differing results. This ultimately led to the development of chaos theory, showing how computational modeling can contribute to whole new theory. Now, computational modeling is a standard approach across science.

This idea is closely tied to that of Papert (1980) that computational methods could be used as the basis for learning mathematics and other subjects (see Chapter 19). Simulation can be used to explore and understand subjects that are new to the learner. Application of computational thinking by learners in non-computing subjects, such as in mathematics, economics, or physics, to find mathematical solutions to problems at different levels of abstraction (i.e., algorithms and then computer code) is a demonstration of how computational thinking skills can lead to deeper, more meaningful learning. A similar approach can be used in biology (e.g., by coding the behavior of ants laying and following trails, which can lead to a deeper understanding of that behavior as a learner).

17.2 The Elements of Computational Thinking

While there are differing views as to the details, there is a lot of agreement at least as to the core elements that make up computational thinking: algorithmic thinking, logical thinking, abstraction, generalization, and decomposition, for example, are generally agreed to play a part (Selby & Woollard, 2013). A range of other aspects have also been suggested, including recursive thinking, pattern matching, representation, heuristic thinking, scientific thinking, probabilistic and statistical reasoning, understanding people, concurrency and parallelism, and attention to detail. We discuss the core, uncontroversial elements in depth here. Many other aspects are arguably sub-skills of, or closely linked to, these core elements. For example, recursive thinking can be thought of as an advanced form of decomposition. In the examples given below, the skills described draw upon a combination of the core elements – particularly generalization, decomposition, and abstraction. Other definitions (e.g., Google, n.d.; ISTE/CSTA, 2014) are in part different because they group the separate aspects differently, such as pulling out pattern matching as a separate skill from generalization or linking abstraction and decomposition together as a single core element.

17.2.1 Algorithmic Thinking

Algorithmic thinking is the idea that solutions to problems are not limited to one-off answers like, "The vending machine will give a $5 and $10 note as the change," but rather are algorithms that can give answers whenever needed for general cases: instructions that, if followed blindly and precisely, are guaranteed to lead to an answer, such as, "Here's how to work out the notes and coins to give if you buy an item worth x dollars and give the vending machine y dollars." If a person can express the solution to a problem as a general algorithm that will solve it for all cases, then they have shown a deeper understanding of the problem than otherwise. It is possible to be able to do related tasks without that deeper understanding of the algorithm. For example, most people can give correct change, but articulating the process exactly (see if the largest note is too much, if not, give one out, then …) is quite difficult to do. This is akin to the fact that people can catch a ball without being able to explain the laws of gravity. If a student writes an algorithm that another person can follow or implements an algorithm as a program that works correctly for any input (such as giving change for any amount of money), then they have demonstrated that they do deeply understand the process.

An important issue in creating algorithms, and therefore algorithmic thinking, is in trying to get the most efficient algorithm for the job, where this could, for example, mean the fastest or alternatively the least memory-hungry algorithm. Often the best answer involves trade-offs in choosing between algorithms, rather than there being a single right answer.

A key part of algorithmic thinking, given Turing's result on the essence of computation and Turing completeness (Böhm & Jacopini, 1966; Aho, 2012), is that a computational thinker can give instructions making use of all three of sequence, selection, and iteration. Without this basic minimum, one cannot claim to have a true grasp of computation. Because these "big three" define everything that computational devices can do in terms of flow of control, having an understanding of them opens the full power of computation. It also defines the limits of computation and underpins our understanding of what computers *can't* do (Harel, 2003).

Algorithmic thinking is the core part of the computational thinking skill set that makes it different from the thinking skills of other disciplines such as scientific thinking, mathematical thinking, design thinking, and so on where the other building blocks of computational thinking arise. Computing overlaps and draws on many other subjects. The core of computing is centered around algorithms, however. Similarly, computational thinking draws on and overlaps with other problem-solving approaches from those disciplines. In this sense, algorithmic thinking is the defining part that makes it different. However, on its own, algorithmic thinking is not enough to be generally useful as a way of problem-solving (unless one subsumes all of the other aspects into the term "algorithmic thinking").

17.2.2 Logical Thinking

Being able to think logically is a core skill that underpins all versions of computational thinking. Logic underpins the semantics of programming languages, and thinking in a logical way is needed to develop algorithms, to implement these as programs, and to verify whether or not they work correctly, either informally or formally. Computer scientists developing algorithms need to be able to think through a problem, being sure that their solutions cover all possibilities that might arise and that they are guaranteed to always give the correct solution. Of course, with the advent of machine learning approaches, this becomes a question of probabilistic reasoning, rather than pure logical reasoning (see Chapter 20). At one end of the logical thinking spectrum lies simply thinking clearly and precisely, including avoiding errors and with attention to detail. At the other end lies an ability to reason about algorithmic solutions using formal logic. In between lies being able to put together rigorous arguments based on deductive or inductive reasoning. While formal reasoning in logic is a core aspect of computing, few computer scientists learn to do it well, so what usually seems to be meant by commentators in the context of computational thinking is the less rigorous versions. Formal logical reasoning is, however, a sophisticated aspect of computational thinking that is certainly desirable for programmers and can be considered a part of the highest levels of progression in computational thinking skill.

17.2.3 Abstraction

Abstraction is the process of simplifying and hiding detail to get at the essence of something of interest. As part of computational thinking, it provides a way to manage complexity in order to make problem-solving easier and allow truly massive computational systems to be designed. By building in levels of abstraction, the fine details of lower levels can be ignored when working on higher levels. Once you have logic gates, you can ignore the details of transistors. Once you have a computer architecture, you can ignore the details of logic gates, and so on. Once you have a programming language, you can ignore assembly language, which itself allowed you to ignore machine code. This is a core skill underpinning the way that the subject of computing has developed. It is also a core skill of computer scientists because without it, building the immensely large and complex systems that we now rely on is intractable. It is only by building in layers with clean interfaces between them that complex systems can be built, so that the complexity of each new layer is simple once the complexity of the lower layers has been hidden by the interface. A course has even been given where students build all of the layers of abstraction one at a time, starting with logic gates, and ending up with a working program running on an operating system (Schocken & Nisan, 2004). Such an endeavor is made possible by breaking it into 12 levels of abstraction.

Computer scientists make use of a wide variety of forms of abstractions both in programming and in system design more generally. These include control abstraction, which is the core of developing programs based on procedures and functions, and data abstraction, which is the core idea behind building complex data types from simpler ones.

Being able to think at multiple levels of abstraction and move between levels is a key ability. This is needed as one develops solutions, moving back and forth, for example, between the level of the problem, design levels, and programming levels. Linked to this is being able to view systems through different abstractions: the bus map intended for passengers may not, for example, include locations and times where drivers swap as their shifts start and end, whereas the abstract version for drivers would have very different information.

Abstraction is not just important for building systems, but also for the development of theory. For example, O-notation focuses on critical operations rather than all operations or processor cycles. Hiding that detail allows us to reason effectively about the efficiency of algorithms. Abstract models of computation (such as Turing machines, finite-state machines, and random access models) allow us to understand computation itself, including its limits.

17.2.4 Generalization

Generalization involves taking the solution to a problem and creating a more general version that is applicable to a wider set of problems. In a computational thinking context, this is first and foremost applied to algorithms. Having come up with a way to solve a specific problem, can the details be abstracted away to give a more general algorithm that is not just specific to that problem?

At a simple level, a sequence of instructions that are applied repetitively can be generalized to a loop. A more sophisticated generalization is to develop general-purpose functions. For example, if there are several situations where the user enters a date, a function could be developed once and for all that allows the user to do this, verifying that it is valid. A more general version of the function might have parameters that restrict the range of dates (e.g., a booking website would not allow a user to enter a date in the past). Generalization can be applied to both problems and solutions. For example, the problem of listing the top ten scores in a game could be generalized to creating a list of any number of scores. A possible solution – sorting the scores into order – could be generalized to a procedure that will sort any list of values into ascending or descending order.

Generalization is closely related to pattern finding (another idea that is often given as an element of computational thinking). When a pattern is noticed either in a program or in data, there is an opportunity to express it more generally by capturing the pattern rather than the specific case. For example, students might use a programming language to draw a square by giving the sequence of instructions "turn right, forward ten steps, turn right, forward ten steps, turn right, forward ten steps, turn right, forward ten steps." This could be generalized by a loop that repeats the two-instruction pattern four times; and that in turn could be generalized to draw polygons by changing the number of repetitions and the angle of the turn.

Generalization skills do not just apply to programming, but also to problem-solving more generally. Whether or not the ultimate intention is a program, generalizing a problem or situation in the same way can, for example, lead to a deeper understanding of that problem or situation, which may be important in its own right.

17.2.5 Decomposition

Decomposition is the idea that to solve a complex problem, including writing a complex program, it can often be broken into smaller parts that can each be solved separately and much more easily. This is closely connected to control abstraction. The simplest forms are task-based, or procedural, decomposition. For example, if one is creating a robot face that shows "emotions" through expressions, one could break the problem into that of solving each whole task (i.e., how to present each emotion). Program a happy face first, then separately program a sad face, and so on.

A different take on decomposition is to focus on the real-world problem or context, focusing on the real-world objects making up the system to be modeled and splitting the problem into one of modeling each of those aspects separately. This leads to an object-based decomposition. Taking the same example of programming a robot face, one could instead decompose the problem into that of programming a mouth for all emotions, then separately programming an eye, and so on. Object-based decomposition potentially provides a higher level of structuring than a procedural decomposition.

Another focus of decomposition can be on the processing of data structures. Rather than process the whole of a data structure, it can be split into parts and either the same or different algorithms can then be developed for processing those parts.

Decomposition links to generalization in that if we can decompose a problem into subproblems that generalize to ones that we have solved before, then we can just take those sub-solutions and reuse them. Abstraction means we do not have to worry about how those sub-solutions work, just that they solve the given subproblem. This leads to more sophisticated forms of decomposition and, in particular, recursive and divide-and-conquer problem-solving.

17.2.6 Evaluation

Evaluation is a potential element of computational thinking. However, there is no complete consensus on its inclusion. Practically, it is clearly an important part of any problem-solving approach. It is also very clearly a vital part of the skill of programming, where more time is typically spent testing solutions than writing code itself. The contentious issue is simply whether it should be considered as a part of "computational thinking." For example, Berry (2014), who takes computational thinking to be "looking at problems or systems in a way that considers how computers could be used to help solve or model these," omits evaluation from his description of computational thinking for primary schools. Selby and Woollard (2013), however, identified it as one of the more widely claimed terms that are used in relation to computational thinking in a survey of educators and other experts. That in itself does not mean it should be part of a definition, just that it is widely accepted as being so. It was consequently included in the UK Computing at School definition (Csizmadia et al., 2015). This is just one example of the different ways of defining computational thinking.

One argument for its inclusion is that unlike in school mathematics problem-solving, where an answer is right or wrong, in computing there are lots of ways to achieve the same result, some of which are better than others. Importantly, coming up with a solution requires trade-offs to be made (e.g., with respect to speed and memory usage). Evaluation of whether requirements are met (beyond just producing correct answers) matters. Programmed solutions may technically meet a functional specification, but be unfit for purpose because they are too slow or don't scale. This might also be due to usability or user experience issues. Evaluation of whether solutions are fit for purpose therefore has to be a core part of any successful computing-related problem-solving approach. If computational thinking does not include elements of evaluation, it would be only a partial approach for computer scientists.

17.2.7 Computational Modeling

Some of the apparent differences between authors over the definition of computational thinking are really just to do with what one considers to be the top-level skills. For example, we have argued that computational modeling is a key computing approach that has changed research and development in other subjects. As such, modeling is an important aspect of computational thinking. Denning (2017) proposes it is an absolutely central component. Computational modeling can be thought of as a separate topic or as one aspect of algorithmic thinking where it is applied to problems that can be simulated.

On the other hand, computational modeling applies not just to simulation approaches (so programming solutions). At higher levels of abstraction, it can be used with models that are not executable. One can do proof and model checking of appropriately designed computational models too. This leads to similar ends as with programmed simulation models, but allows exhaustive experiments to be conducted. This is an example where computational thinking is potentially about much more than the skill of coding. It leads to the application of verification tools and techniques rather than programming ones, and applying them to the understanding of the world too. These tools require models to be written in formal logical languages with an axiomatic basis (e.g., Peano's axioms) rather than programming languages with a model of computation as their basis, but otherwise the issues are similar – we are just working at a higher level of abstraction. The more sophisticated logical thinking skills mentioned above – working with formal logic – are needed here. The use of such tools and thinking is a part of formal program development for safety-critical systems, so any definition of computational thinking as being about the wider development of programs rather than narrowly as coding should include it.

17.2.8 Expressing Algorithms in Formal Languages

None of the computational thinking concepts above explicitly addresses the step of writing actual code (i.e., expressing the algorithm in a precise syntax that has a detailed, formally defined semantics). A student may have developed a design including algorithms in a loose pseudocode or in a flow chart language, but it is another step – and another skill – to be able to implement this as code in a specific language correctly. This often seems to be implicitly assumed by commentators, rather than explicitly stated. If computational thinking is the skill of developing programs, then expressing an algorithm as code must be part of the computational thinking skill set. This transition from a logical design to a physical implementation is part of the ability to work at multiple levels of abstraction. This is more than just about programming, though – expressing an algorithm precisely needs a level of rigor, and more formal pseudocode- or logic-based specification languages require similar skills in using formal language precisely. Guzdial (2008) explicitly discusses issues around this skill. For example, there are higher-level issues to be explored, such as the way people naturally omit certain steps, like else cases, from formal descriptions. A separate issue again, beyond having mastery of the language constructs, is having mastery over their pragmatic use to best develop readable and maintainable code. Machines must be able to follow code, but humans must be able to understand it.

17.2.9 A Holistic View

In practice, computer scientists use mixtures of these separate skills at different times, and when combined they are much more powerful than alone. For example, thinking in terms of layers of abstraction to decompose a problem and drawing on previous generalized solutions makes it much easier to create algorithmic solutions to problems. Understanding the separate elements is important. However, it is also important that this holistic aspect is understood too, not just the individual skills. Educators need to consider how best to help students develop both of these elements and how to develop the skill of combining the separate parts into computational thinking as a whole. In practice, most lessons and activities will use a range of the skills at the same time.

17.2.10 Links to the Skill Sets of Other Disciplines

Is computational thinking something new and totally different from thinking and problem-solving in other subjects? Computing itself emerged from a range of subjects, including mathematics, engineering, design, and the social sciences. Likewise, computational thinking builds on problem-solving and modes of thinking from other subjects. Generalization, decomposition, abstraction, logical thinking, and other components all play important parts in other disciplines. It draws on design (“the ultimate goal is computational design” [Denning, 2017], and interaction design is also a key part of making usable systems), mathematics, scientific methods (e.g., in evaluation A/B testing, virtual experiments, etc.), engineering methods, and general problem-solving methods.

There is no reason why computational thinking has to be totally unique in order to be an important concept and skill. If it were to be no different from other problem-solving skill sets, then that is an argument for the importance of teaching it to all. However, as argued, from a philosophical point of view, it *has* led to a seismic change in modes of thought. The difference is ultimately in the importance placed on algorithms in the skill set and how the separate skills used in other disciplines apply to algorithmic thinking, not the elements themselves. Algorithmic solutions in turn lead to the possibility of programmed solutions.

17.3 Research: What Is Known

There has been increasing research in how to teach and assess computational thinking explicitly, especially since the 2006 revolution. This has built on earlier work on teaching computer science and programming, given that even without the name, the component skills were still being taught, if implicitly. There has been an explosion of interest on the back of Wing’s work, culminating in 2017 with a new international conference series being launched with a sole focus on computational thinking education (CSE, 2017). This bodes well for the future as researchers investigate the best ways to teach different aspects, their actual effects on student learning and skills, and whether there is general benefit to be had or not, divorced from programming skills. However, care has to be taken in that different authors often use their own interpretations of what computational thinking is across the full range of possibilities discussed and more, so results are not necessarily about the same thing. We overview some major themes in the existing research below.

17.3.1 Unplugged Computational Thinking

Unplugged activities (Bell, Rosamond, & Casey, 2012) have long been successfully used at all levels, from primary to master's levels, as well as when teaching adult teachers, as a way to teach programming and computing concepts more generally in constructivist ways. This covers a variety of techniques, including role-playing, puzzles, games, and magic, to illustrate concepts (Curzon & McOwan, 2017). These activities often help develop computational thinking in its widest sense too. Activities can also be used to explicitly illustrate the high-level elements of computational thinking, like decomposition, generalization, and abstraction. Teaching London Computing (http://teachinglondoncomputing.org), the Digital Schoolhouse (www.digitalschoolhouse.org.uk), and the lesson plans on csunplugged.org, for example, make these opportunities explicit across a wide range of cross-curricula activities.

Curzon (2014) argues that the core ideas of computational thinking can be explained in powerfully memorable ways using a combination of contextually rich stories and unplugged activities. He gives example activities embedded in such stories that have successfully been used. Examples given include stories concerned with helping people with locked-in syndrome, using games and role-play, and the design of medical devices using magic trick-based activities. These ideas are expanded upon in Curzon and McOwan (2017). This approach has successfully been used as part of continuous professional development for teachers. A series of workshops given following this approach had shown strongly positive evaluation results (Curzon, 2014; Meagher, 2017).

However, much of the work on unplugged approaches is anecdotal and much more research is needed, including on the important issue of how such approaches are linked to programming itself.

17.3.2 Computational System Design and Programming

Computational system design involves a wider set of activities than programming, and those activities include computational thinking elements. When analyzing the requirements and designing the solution, the task is broken into manageable parts to attend to, and so decomposition is used; at each stage of increasing exploration of the task, abstraction is needed to work at an appropriate level of detail, and so on. Research on effective teaching of overall design processes is therefore relevant. Here, we focus on research where there are links between computational thinking ideas and programming.

McCracken et al. (2001) describe a five-step process for problem-solving that learners should use to aid computational design. These fives step map to the computational thinking core concepts of Selby and Woollard (2013):

1. Abstract the problem from its description (abstraction)
2. Generate subproblems (decomposition)
3. Transform subproblems into subsolutions (generalization and algorithmic thinking)

4. Recompose (algorithmic thinking)
5. Evaluate and iterate (evaluation)

However, McCracken et al. (2001) highlighted that few students were able to use this process and noted that students appeared "clueless." Lister et al. (2004, 2011) and Lopez et al. (2008) highlight the importance of being able to read and trace code as precursors to the problem-solving skills needed to write code, so these may be precursors to any form of programming-based computational thinking. For example, this suggests that before one can do functional abstraction, one needs to be able to read and trace existing code that uses such abstraction. A precursor to that is understanding the basic concepts and their semantics.

Fuller et al. (2007) identified 11 programming skills needed by students, and mapped them to Bloom's taxonomy, presented in a matrix format. For example, the ability to debug requires "application" and "analysis" thinking skills. Each of the 11 skills is underpinned by the computational thinking core elements discussed above. For example, tracing and adapting code develop evaluation and generalization skills. Designing and modeling solutions (as algorithms and/or programs) develop algorithmic thinking, abstraction, and evaluation skills (Sentance & Csizmadia, 2017). Developing the computational thinking skill appears to provide a foundation for the corresponding programming skill.

Computational thinking skill is not the only thing needed by programmers, of course. They also need to understand the syntax of the language they are using as well as the programming constructs available to them in order to implement the design. This leads back to some of the earlier extensions discussed, as it implies that issues to do with language concepts and terminology are precursors to computational thinking.

17.3.3 Abstraction

Abstraction is a particularly important pillar of computational thinking and has attracted specific attention. An important question now that computational thinking is part of school syllabuses is: How young can you start to learn about abstraction? It is sometimes suggested that Piaget's work implies that children cannot learn about abstraction until they reach a particular age and stage of development – that of formal operational at around the age of 12. The National Research Council report on computational thinking (National Research Council, 2011) asked for a review of this. However, Piaget himself suggested that children use abstraction from before the age of two and that abstraction is used continuously when learning "without end and especially without an absolute beginning" (Piaget, 2001, p. 136).

Armoni (2013) pointed out that abstraction, as part of the process of developing programs from solutions, is hard to teach. She suggested a "level of abstraction" framework and gave guidelines for teaching abstraction. Several authors have argued that programming ability can be developed by explicitly focusing students on abstraction, particularly different levels of abstraction, as part of the process of writing programs. This has been considered both with respect to tertiary institution students (Aharoni, 2000; Cutts et al., 2012; Hazzan, 2003) and school students (Armoni, 2013; Statter & Armoni, 2016; Waite et al., 2016, 2017). Cutts et al. (2012), for example, argued that focusing on a model of three levels of abstraction helps students develop their programming ability. Their levels were: English descriptions, computer science speak (i.e., a halfway house such as pseudocode, where some of the terminology of code, like variable and procedure names, is embedded in English phrasing), and code. Statter and Armoni's (2016) model is similar, but with four levels: the statement of the problem, its description as an algorithm or design level, the program itself, and finally the concrete execution of that program. Grade 7 students who were explicitly taught these different levels did focus more on the algorithm level in their descriptions. Thus, explicitly teaching about abstraction, even with a simple set of levels, can help the development of programming skill.

17.3.4 Assessment

A critical research area is how to assess computational thinking (see also Chapters 10 and 14). Denning (2017) suggests that it should be assessed as a skill, with others focusing on knowledge frameworks such as those of Computing at School (Csizmadia et al., 2015) and K12CS (https://k12cs.org). However, skills and knowledge coexist and, in particular, conceptual computer science knowledge, computational thinking skills, and programming skills can and should coexist. As with programming itself, assessing it as a skill does not preclude the pedagogical importance of assessing understanding of knowledge too. Having a strong conceptual knowledge of a discipline can also support the development of related skills – if you understand how a gearbox works, learning the skill of changing gears in a car can be easier. Similarly, if you have a deep understanding of the concept of abstract data types, then using that form of abstraction in programs is easier. Knowledge helps refine skills as you know more of what you are trying to do and why.

A variety of researchers have explored ways to assess computational thinking as a skill. This could be done by assessing the individual component skills or by assessing computational thinking as a holistic single skill. One approach is to directly assess the skills based on evidence in programs. If programming is seen as the whole point, then the idea is that computational thinking can be assessed by the quality of the programs that a student produces. Another approach is to assess the skills using more general problems at a higher level of abstraction than that of writing programs. We provide an overview of some of the research on these topics below.

17.3.4.1 Assessing Computational Thinking through Programming

Several automated approaches have been suggested to assess computational thinking based on evaluating programs. For example, both Dr. Scratch

(Moreno-León, Robles, & Román-González, 2015) and Seiter and Foreman's (2013) "Progression of Early Computational Thinking" (PECT) model are applied to Scratch programs to assess primary-aged students' development of computational thinking skills. These approaches are based on the idea that computational thinking skills should ultimately be evident in programs written. As such, they may therefore not assess more general application of the skills and are dependent on sub-skills concerned with actually embodying an algorithm in a formal language.

PECT (Seiter & Foreman, 2013) aims to combine direct measures of programs with broad design patterns that are linked to computational thinking concepts. Seiter and Foreman applied PECT to 150 Scratch projects of primary students of differing ages, concluding that it showed that progression in students' skills improved as they got older.

Seiter (2015) has also used the Structure of Observed Learning Outcomes (SOLO) taxonomy (Biggs & Collis, 1982) to give insight into computational thinking ability as embodied in Scratch programming. This was based on how well students could understand the structure of the problem. It focused on such things as their ability to synchronize the costumes and motions of single and multiple sprites. However, the low numeracy and literacy of some students meant that those students could not understand the tasks at all. Seiter concluded that students above this level can understand multiple concerns and incorporate them into a single script. They can also synchronize a single concern between more than one script. However, synchronizing many concerns across many scripts was a challenge.

17.3.4.2 Assessing Computational Thinking through Problem-Solving

An alternative to basing assessment of computational thinking skill on programs is to assess proficiency at more general problem-solving tasks. Several authors have aimed to do this based on Bebras (Dagiene & Futschek, 2008). Bebras is an international competition with questions on both computing concepts and computational thinking skill. Hubwieser and Mühling (2014) suggested that Bebras tasks were suitable as an international benchmark test for computing ability in the style of the Programme for International Student Assessment (PISA) tests. They give a methodology for finding and validating groups of questions that measure specific competencies. Such an approach could be used to identify problems that test specific computational thinking competencies. Dagienė and Sentance (2016) give recommendations on how Bebras tasks can be used to develop and assess children's computational thinking skills specifically. They created an explicit two-dimensional classification scheme for questions (Dagienė et al., 2017). Computational thinking aspects act as one dimension and content knowledge as the other.

Project Quantum (Oates et al., 2016) is a crowdsourced multiple-choice computer science question bank being pioneered in the UK to provide formative assessment. Quality assurance is integral via a feedback loop based on big data that will be generated from its expected widespread use. Questions are machine-markable and algorithms will generate data about the quality of questions. Bebras questions are one of the sources used, and so this could be a way of determining and/or improving the quality of the computational thinking questions, ultimately generating a large, quality-assured set of questions.

Several other specific "computational thinking" tests have been developed. The "Computational Thinking Test" (CTt) (Román-Gonzáles, 2015) is a multiple-choice questionnaire involving 28 questions, such as whether a particular program will lead a character along a given maze path. It tests understanding of programming concepts such as loops and conditionals. Korkmaz, Çakirb, and Özdenc (2017) similarly developed a set of 29 five-point Likert scale questions to assess computational thinking. Tested on over 1,000 students, the authors concluded that it is a valid and reliable tool for measuring computational thinking skills. Brennan and Resnick (2012), however, suggest that assessment requires a combination of approaches. They used an analysis of projects, artifact-based interviews, and pupils completing design scenario challenges. They concluded that this triangulation leads to an understanding of computational thinking concepts and practices, but that these approaches do not effectively reveal changes in expressing, connecting, and questioning perspectives.

17.3.4.3 Determining Progression and Age-Appropriate Curricula

Designing assessment requires both an understanding of what is to be assessed and a methodology for capturing the specific knowledge, skills, and understanding at a point in time. In school, teachers develop lesson activities to teach objectives for learners that help them make progress. What those objectives are and how one might move from one objective to another to provide progression matter as computational thinking is brought into the school curriculum.

Dorling and Walker (2014) interpreted the English curriculum in the form of an easily digestible table. This table presented the learning statements by either topic area taken from the Computing at School Curriculum for Schools document (Computing at School, 2012) or by subject strands: Computer Science, Information Technology, and Digital Literacy. Dorling, Selby, and Woollard (2015) suggested that this interpretation of the curriculum had aligned computational thinking core elements to all of the statements in the table. A later version of the grid was cross-referenced to the computational thinking concepts outlined in Csizmadia et al. (2015). Rich et al. (2017) also consider progression of conceptual ideas and links to computational thinking based on a detailed review of over 100 computing education research articles. They use concept maps to show progress and also propose an alternative model to the spiral curriculum.

Barefoot (2014a, 2014b), which provides material for primary school teaching of computing, suggests ideas for progression in computational thinking for children aged 3–11. However, this is not a complete progression, as it only provides suggestions for a limited set of lessons, some set in programming contexts, others in a cross-curricula scenarios.

Bebras (Dagiene & Futschek, 2008) is also structured by age, using six groupings of questions from ages 5 to 19 with the complexity of the problems increasing. This is a loose organization, and individual countries can choose the questions that they think are appropriate. However, the groups of questions are linked to age, and therefore a progression is implied.

Denning (2017) suggests existing progression frameworks are focused on progression of knowledge and that this is misguided. This is not entirely true – as noted above, several groups have considered skills-based progression. However, he makes the important point that there are two separate issues: knowledge-based progression and skills-based progression. Both should be addressed. Frameworks for progression of both skills and conceptual knowledge within subjects are needed, and arguably integrated versions are needed too. Research about the appropriateness and effectiveness of progression frameworks is needed.

17.3.4.4 Validity

In all approaches to assessment, validation of the underlying models and/or the tools and techniques based on them is needed. Methods might concern computational thinking as a whole, some specific subset of it, or individual foundational skills. Much research is needed in this area. For example, according to Armoni (2013), in 2013, there were no validated methods to assess abstraction ability.

An important issue is whether different approaches produce the same answers – their convergent validity. Román-Gonzáles et al. (2017) explore this for three approaches: Dr. Scratch (Moreno-León & Robles, 2015), Bebras (Dagiene & Futschek, 2008) and CTt (Román-Gonzáles, 2015). Their results suggest that CTt partially converges with the other two. They suggest that the three approaches are complementary, and they use a revised version of Bloom's taxonomy (Krathwohl, 2002) as a way to classify this. They conclude that Dr. Scratch assesses the very top "create" and "evaluate" levels of Bloom's taxonomy, Bebras assesses the "analyze" and "apply" levels, and CTt assesses the "understand" and "remember" levels, as it focuses on the concepts related to computational thinking rather than the practice of it. In essence, this is saying that Dr. Scratch assesses the programming part of computational thinking, Bebras targets more general thinking skills, and CTt targets conceptual knowledge of computational thinking.

17.4 Implications for Practice

17.4.1 Implications Depending on the View Taken

If one takes the view that computational thinking is primarily associated with learning to program, and it doesn't directly support learning in other subjects, then the important focus becomes how to teach programming itself well. Studying the component skills can still enhance insight into how to teach programming. One such insight is that making students explicitly focus on levels of abstraction when programming helps develop programming ability (Cutts et al., 2012; Statter & Armoni, 2016).

If one takes the intermediate view expounded by Lee (2016) that computational thinking is about more than coding itself, but the person concerned must have an aim that the resulting algorithm be carried out on a computer, not by a human, then the focus is different. The key practical point Lee recommends is that repeated practice is needed of working with real-world problems and how they are solved using information processing devices. Practice is needed in taking real-world problems, developing from them statements of the problems in a form that can be solved by computers, designing algorithms that solve those problems, and implementing those algorithms as programs, specialized hardware, or combinations of the two. This involves both understanding and taking into account, from the outset, the actual goals and needs of stakeholders and verifying and validating those solutions in the real world.

If one takes the wide view that computational thinking is a transferable skill for all and that algorithms go beyond computers and may be usefully followed by humans too (as in the original definition of the words "algorithm" and "computer"), the implications are different again. This implies that computational thinking can and should be developed both through programming and through other means. The focus then turns to how to develop the individual skills across a wide range of information processing situations, physical and otherwise, in computing and other subject contexts. Exploring how best to use them together is also critical, so developing the holistic skill matters. Any skill is developed with practice: the more, the better. Therefore, students need to be encouraged to practice as much as possible, in as many contexts as possible, not just programming contexts. In this view, starting to develop general computational thinking skills, not just programming skills, should start early in primary school, as some countries are now doing. Making links from activities such as writing clear instructions to early programming tasks is also important.

Whichever view is taken, intrinsic motivation to practice the skills needs to be developed (see also Chapter 11). Ensuring such practice is fun and engaging is one important element, as is providing realistic context. The educational community also needs to develop appropriate progression pathways for their pupils from primary school upwards that develop and refine the skills over time.

Developing the component skills separately provides a foundation for learning computational thinking as a whole. Knowledge supports the development of skill, and teachers need to understand the barrier concepts and points so that they can help students overcome them.

As with any skill, having an understanding of the underlying concepts and having the vocabulary to express them by themselves can help develop the skills in a reflective way. Therefore, the skills and concepts need to be developed in parallel. Unplugged methods provide a powerful, constructivist way to do this at all levels if used well. The theory of semantic waves (Macnaught et al., 2013; Maton, 2013) provides guidance on how to do this – as suggested by Curzon et al. (2018), one should travel up and down the semantic wave from abstract concepts to concrete examples of them (whether unplugged, real world, or programming) and back to the abstract ones, making clear the links between the levels.

Whichever view one takes of the definition of computational thinking, it is important to be pragmatic regarding developing the best ways to teach it. Whether one considers computational thinking as something that can be developed separately from programming or not, and whether or not it includes physical computation in the world, analogies with real-world ideas *are* powerful ways of teaching concepts and of developing skills. According to the theory of semantic waves (Macnaught et al., 2013; Maton, 2013), good explanation involves moving from technical, abstract concepts to concrete illustration, and then back to technical concepts. This is what good use of analogy and unplugged teaching does. Analogy and simplified explanations are used widely across other subjects as effective ways to teach. This should not be lost to computing because of ideology. Ideas such as unplugged teaching should not be dropped just because one thinks of them as only analogy. Instead, the fact that they are analogy should be made clear. For example, whether or not one believes writing a recipe involves any aspect of computational thinking, a recipe book is still a useful initial way to help students understand concepts including breaking a problem down into parts (procedural abstraction) and the ordering of the parts (how the flow of control involved in procedure call works). Having such understanding about concepts is a critical foundation for learning to program.

It certainly does help to keep the focus on the general value of skills and conceptual understanding, even if very specific examples are being taught; for example, students might be learning the syntax of a Python "for" loop, but the point is to understand iteration in programs; they might be learning a version of binary search, but the wider picture is that it is an example of the power of using divide and conquer to decompose a problem.

Also, whatever view is taken, to develop computational thinking skills fully does, of course, ultimately involve programming too. This is another kind of example that can be used to travel a semantic wave of good explanation (Macnaught et al., 2013; Maton, 2013). Ideally, programming skills should be developed in conjunction with more general computational thinking skills and understanding. For example, the Computing at School Working Group suggests:

> Computer Science is more than programming, but programming is an absolutely central process for Computer Science. In an educational context, programming encourages creativity, logical thought, precision and problem-solving, and helps foster the personal, learning and thinking skills required in the modern school curriculum. Programming gives concrete, tangible form to the idea of "abstraction," and repeatedly shows how useful it is.
> (Computing at School, 2012)

17.4.2 Practical Resources for Teaching

A wide variety of practical resources and tools exist to support the teaching of computational thinking. These include:

- CS Unplugged (http://csunplugged.org)
- Teaching London Computing (http://teachinglondoncomputing.org)
- Barefoot (http://barefootcas.org.uk)
- The International Society for Technology in Education's Computational Thinking Toolkit (Sykora, 2014)
- Google's Exploring Computational Thinking (Google, n.d.)
- Bebras (Bebras, n.d.)
- Dr. Scratch (www.drscratch.org)
- Digital Schoolhouse (www.digitalschoolhouse.org.uk)
- Computational thinking rubric (Dorling & Stephens, 2016)

There are many more such resources and resource collections, with more being developed all the time.

17.5 Open Questions

Computational thinking is still a relatively new idea, and designing curricula that use it is even newer. There are many open questions, making it a very fertile area for future research. The most fundamental open question is just what definition of computational thinking should be adopted and how wide it should stretch. In the absence of agreement about definitions of the term, those doing such research need to be precise about the definition that they are working with.

What definition is appropriate depends to a large extent on the answers to more specific open questions. For example, we need to determine the true extent of the transferability of the skills (see also Chapter 9, which explores transfer of learning). How useful are or can be the skills in practice to learning in other areas if either a narrow or a wide view is taken? Is there a difference in general usefulness if you learn them only as programming versus taking a wider approach to teaching them? Are they useful at all? Is knowledge of computational thinking useful in understanding the digital world and how? Can computational thinking skills be developed effectively outside of programming? How effective are the various unplugged methods for teaching computational thinking? For example, does early practice using logic puzzles to refine logical thinking skills actually lead to better computational thinking skills and so make programming easier to learn? Similar issues apply to the other components of computational thinking. What makes an effective unplugged computational thinking activity in general and what makes them ineffective? How does one best link unplugged and programming techniques? Rigorous evidence is needed of what actually does work and why.

If computational thinking is primarily useful for programmers and can only usefully be taught through programming, then the question becomes how to enhance those skills more effectively through programming. Even if they can be developed in other ways, this is still an important question. Either way, we need to better understand the importance of the conceptual knowledge, programming skills, and computational thinking skills for developing independence and resilience in learners. In particular, we need further consideration of the relationships between programming skills and the core computational thinking concepts. Does a better grasp of computational thinking concepts and subskills lead to better holistic computational thinking and programming skills, and if so, how best do knowledge and skills combine? It is often suggested that math is an important precursor to being able to cope on a tertiary institution computing course. However, it is also often suggested that it is not the math content that matters. What exactly are those mathematical precursor skills? Perhaps it is because math does develop some of the relevant precursor skills, such as attention to detail, logical thinking, or abstraction skills (e.g., in algebra). This might suggest that teaching the subskills of computational thinking in other contexts does help.

Validated progression frameworks are needed for both skills and knowledge. What do you teach at different levels from primary upwards to achieve the best learning? And how best do you then teach at each level of progression and each topic? The questions are not just about how to teach. We need to know what the effective means of both formative and summative assessment are too. There are very big unanswered questions as to how to assess both programming and computational thinking skills. How are each of the progression levels best assessed both formatively and summatively? This applies both to computational thinking overall and to the separate subskills, such as abstraction and generalization.

At the moment, arguments are being made and policy implemented based on opinion and early results, as there is a lack of evidence. Experiments need to be founded in rigorous theories of the mechanisms involved. For many of the research areas outlined, some work has been done, though often on a small scale and in uncontrolled ways. What is needed is really rigorous evidence around all of these issues that is more than just action research suggesting an intervention was a positive experience in a single context. Research needs to be replicated, including situating the studies in real classrooms, with real teachers, over longer periods of time, and on larger scales. We need large-scale, longitudinal comparison of teaching, learning, and assessment of computational thinking across schools, cultures, and age groups. We then need continuous professional development for teachers and resources to be developed based on the research. This material needs to be organized in a validated progression, affording educators the means to plan lessons and evaluate students' progress, allowing students to show what they know and can do.

References

Aharoni, D. (2000). Cogito, ergo sum! Cognitive processes of students dealing with data structures. *ACM SIGCSE Bulletin*, 32(1), 26–30.

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, 55(7), 832–835.

al-Khwārizmī, M. (c. 825). *On the Calculation with Hindu Numerals.*

Armoni, M. (2013). On teaching abstraction in computer science to novices. *Journal of Computers in Mathematics and Science Teaching*, 32(3) 265–284.

Barefoot (2014a). Barefoot Computing. Retrieved from http://barefootcas.org.uk/

Barefoot (2014b). Computational thinking: What does computational thinking look like in the primary curriculum? Retrieved from https://barefootcas.org.uk/barefoot-primary-computing-resources/concepts/computational-thinking/

Bebras (n.d.). Bebras International Challenge on Informatics and Computational Thinking. Retrieved from www.bebras.org

Bell, T., Alexander, J., Freeman, I., & Grimley, M. (2009). Computer science unplugged: School students doing real computing without computers. *New Zealand Journal of Applied Computing and Information Technology*, 13(1), 20–29.

Bell, T., Rosamond, F., & Casey, N. (2012). Computer Science Unplugged and related projects in math and computer science popularization. In H. L. Bodlaender, R. Downey, F. V Fomin, & D. Marx (Eds.), *The Multivariate Algorithmic Revolution and Beyond: Essays Dedicated to Michael R. Fellows on the Occasion of His 60th Birthday, Lecture Notes in Computer Science* (pp. 398–456). Berlin, Germany: Springer.

Berry, M. (2014). Computational Thinking in Primary Schools. Retrieved from http://milesberry.net/2014/03/computational-thinking-in-primary-schools/

Bers, M. U. (2017). *Coding as a Playground: Programming and Computational Thinking in the Early Childhood Classroom*. New York: Routledge.

Bers, M. U. (2008). *Blocks to Robots: Learning with Technology in the Early Childhood Classroom*. New York: Teachers College Press.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. New York: Academic Press.

Böhm, C., & Jacopini, G. (1966). Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5), 366–371.

Brennan, K., & Resnick, M. (2012). New frameworks for studying and assessing the development of computational thinking. Vancouver, Canada: Educational Research Association. Retrieved from https://scholar.harvard.edu/kbrennan/publications/new-frameworks-studying-and-assessing-development-computational-thinking

Computing at School (2012). Computer science: A curriculum for schools. Computing at School Working Group. Retrieved from www.computingatschool.org.uk/data/uploads/ComputingCurric.pdf

CSE (2017). Proceedings of the 1st International Conference on Computational Thinking Education, July, Hong Kong. Retrieved from www.eduhk.hk/cte2017/

Csizmadia, A., Curzon, P., Dorling, M., Humphreys, S., Ng, T., Selby, C., & Woollard, J. (2015). Computational thinking: A guide for teachers. Retrieved from http://computingatschool.org.uk/computationalthinking

Curzon, P. (2002). Computing without Computers: A Gentle Introduction to Computer Programming, Data Structures and Algorithms. Retrieved from https://teachinglondoncomputing.org/resources/inspiring-computing-stories/computingwithoutcomputers/

Curzon, P. (2014). Unplugged computational thinking for fun. In T. Brinda, N. Reynolds, & R. Romeike (Eds.), *KEYCIT – Key Competencies in Informatics and ICT,* *Commentarii Informaticae Didacticae* (pp. 15–28). Potsdam, Germany: Universitätsverlag Potsdam.

Curzon, P., & McOwan, P. W. (2017). *The Power of Computational Thinking: Games, Magic and Puzzles to Help You Become a Computational Thinker*. Hackensack, NJ: World Scientific.

Curzon, P., McOwan, P. W., Donohue, J., Wright, S., & Marsh, D. W. R. (2018). Teaching of concepts. In S. Sentance, E. Barendsen, & C. Schulte (Eds.), *Computer Science Education: Perspectives on Learning and Teaching in School* (pp. 91–108). London, UK: Bloomsbury.

Cutts, Q., Esper, S., Fecho, M., Foster, S., & Simon, B. (2012). The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research* (pp. 63–70). New York: ACM.

Dagienė, V., & Sentance, S. (2016). It's computational thinking! Bebras tasks in the curriculum. In A. Brodnik & F. Tort (Eds.), *Informatics in Schools: Improvement of Informatics Knowledge and Perception (ISSEP 2016). Lecture Notes in Computer Science* (pp 28–39). Berlin, Germany: Springer.

Dagienė, V., Sentance, S., & Stupienė, G. (2017). Developing a two-dimensional categorization system for educational tasks in informatics. *Informatica* 28(1), 23–44.

Dagiene, V., & Futschek, G. (2008). Bebras international contest on informatics and computer literacy: Criteria for good tasks. In R. T. Mittermeir & M. M. Sysło (Eds.), *Informatics Education – Supporting Computational Thinking. ISSEP 2008. Lecture Notes in Computer Science* (pp. 19–30). Berlin, Germany: Springer.

Denning, P. (2017). Remaining trouble spots with computational thinking. *Communications of the ACM*, 60(6), 33–39.

Denning, P., & Tedre, M. (2019). *Computational Thinking*. Cambridge, MA: MIT Press.

Department for Education (2013). National Curriculum in England: Computing programmes of study. Retrieved from www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study

Dorling, M., Selby, C., & Woollard, J. (2015). Evidence of assessing computational thinking. In A. Brodnik & C. Lewin (Eds.), *IFIP 2015: A New Culture of Learning: Computing and Next Generations* (pp. 1–11). Laxenburg, Austria: IFIP.

Dorling, M., & Walker, M. (2014). Computing Progression Pathways. Retrieved from http://community.computingatschool.org.uk/files/5098/original.xlsx

Dorling, M., & Stephens, T. (2016). Computational Thinking Rubric: Dispositions, Attitudes and Perspectives, Retrieved from https://community.computingatschool.org.uk/resources/4793/

Euclid (1997). *Elements* [c. 300 BCE]. D. E. Joyce (Ed.). Retrieved from http://aleph0.clarku.edu/~djoyce/java/elements/toc.html

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., & Thompson, E. (2007). Developing a computer science-specific learning taxonomy. In *Proceedings of the ITiCSE-WGR '07 Working Group Reports on Innovation and Technology in Computer Science Education* (pp. 152–170). New York: ACM.

Google (n.d.). Exploring Computational Thinking, Google for Education. Retrieved from https://edu.google.com/resources/programs/exploring-computational-thinking/

Grover, S., & Pea, R. (2013). Using a discourse-intensive pedagogy and Android's App inventor for introducing computational concepts to middle school students. In *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education* (pp. 723–728). New York: ACM.

Guzdial, M. (2008). Education: Paving the way for computational thinking. *Communications of the ACM*, 51(8), 25–27.

Harel, D. (2003). *Computers Ltd: What They REALLY Can't Do*. Oxford, UK: Oxford Paperbacks.

Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education*, 13(2), 95–122.

Hubwieser, P., & Mühling, A. (2014). Playing PISA with Bebras. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (pp. 128–129). New York: ACM.

Hubwieser, P., Giannakos, M. N., Berges, M., Brinda, T., Diethelm, I., Magenheim, J., Pal, J., Jackova, J., & Jasute, E.(2015) A global snapshot of computer science education in K–12 schools. In *Proceedings of the 2015 ITiCSE on Working Group Reports* (pp. 65–83). New York: ACM.

Hutchins, E. (1995). *Cognition in the Wild*. Cambridge, MA: MIT Press.

ISTE/CSTA (2014). Operational Definition of Computational Thinking for K–12 Education. Retrieved from www.iste.org/docs/ct-documents/computational-thinking-operational-definition-flyer.pdf

Kafai, Y. B. (2016). From computational thinking to computational participation in K–12 education, *Communications of the ACM*, 59(8), 26–27.

Kalelioglu, K., Gülbahar, Y., & Kukul, V. (2016). A framework for computational thinking based on a systematic research review. *Baltic Journal of Modern Computing*, 4(3), 583–596.

Korkmaz, Ö., Çakir, R., & Özden, M. Y. (2017). A validity and reliability study of the Computational Thinking Scales (CTS). *Computers in Human Behavior*, 72, 558–569.

Krathwohl, D. R. (2002). A revision of Bloom's taxonomy: An overview. *Theory into Practice*, 41(4), 212–218.

Lee, I. (2016). Reclaiming the roots of CT. *CSTA Voice: The Voice of K–12 Computer Science Education and Its Educators,* 12(1), 3–4.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., & Simon, B. (2004). A multinational study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin,* 36(4), 119–150.

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In *Proceedings of the Thirteenth Australasian Computing Education Conference* (pp. 9–18). Darlinghurst, Australia: Australian Computer Society, Inc.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (pp. 101–112). New York: ACM.

Lu, J. J., & Fletcher, G. H. (2009). Thinking about computational thinking. *ACM SIGCSE Bulletin*, 41(1), 260–264.

Maton, K. (2013). Making semantic waves: A key to cumulative knowledge-building. *Linguistics and Education*, 24(1), 8–22.

Macnaught, L., Maton, K., Martin, J. R., & Matruglio, E. (2013). Jointly constructing semantic waves: implications for teacher training. *Linguistics and Education*, 24, 50–63.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-year CS Students. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, Working Group Reports (ITiCSE-WGR '01)* (pp. 125–180). New York: ACM.

Meagher, L. (2017). Teaching London Computing Follow-up Evaluation through Interviews with Teachers, Technology Development Group, Summer. Retrieved from https://teachinglondoncomputing.org/evaluation/

Millican, P., & Clark, A. (Eds.) (1996). *The Legacy of Alan Turing, Volume 1: Machines and Thought*. Oxford, UK: Oxford University Press.

Millican, P. (n.d.). A New Paradigm of Explanation? Retrieved from www.philocomp.net/home/paradigm.htm

Moreno-León, J., & Robles, G. (2015). Dr. Scratch: A web tool to automatically evaluate Scratch projects. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 132–133). New York: ACM.

Moreno-León, J., Robles, G., & Román-González, M. (2015). Dr. Scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, 46(10), 1–23.

National Research Council (2011). *Committee for the Workshops on Computational Thinking: Report of a Workshop of Pedagogical Aspects of Computational Thinking*, Washington, DC: The National Academies Press.

NZ Ministry of Education (2017). The New Zealand Curriculum Online: Technology: Learning area structure. Retrieved from http://nzcurriculum.tki.org.nz/The-New-Zealand-Curriculum/Technology/Learning-area-structure

Oates,T., Coe, R., Peyton-Jones, S., Scratcherd, T., & Woodhead S. (2016). Quantum: Tests worth teaching. White Paper, March, Computing at School. Retrieved from http://community.computingatschool.org.uk/files/7256/original.pdf

OED (1993). *The New Shorter Oxford English Dictionary*. Oxford, UK: Oxford University Press.

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books.

Piaget, J. (2001). *Studies in Reflecting Abstraction*. Edited and translated by R. L. Campbell. Hove, UK: Psychology Press.

Resnick, M. (2013). Learn to Code, Code to Learn. Edsurge, May 8. Retrieved from www.edsurge.com/news/2013-05-08-learn-to-code-code-to-learn

Rich, K. M., Strickland, C., Binkowski, T. A, Moran C., & Franklin, D. (2017). K–8 learning trajectories derived from research literature: Sequence, repetition, conditionals. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER'17)* (pp. 182–190). New York: ACM.

Román-González, M. (2015). Computational thinking test: Design guidelines and content validation. In *Proceedings of the 7th Annual International Conference on Education and New Learning Technologies (EDULEARN 2015)* (pp. 2436–2444). Valencia, Spain: IATED Academy.

Román-Gonzáles, M., Moreno-León, J., & Robles, G. (2017). Complementary tools for computational thinking assessment. In *Proceedings of the International*

Conference on Computational Thinking Education (CTE2017) (pp. 154–159). Ting Kok, Hong Kong: The Education University of Hong Kong.

Royal Society (2012). *Shut Down or Restart? The Way Forward for Computing in UK Schools*. London, UK: The Royal Society.

Royal Society (2017a). *After the Reboot: Computing Education in UK Schools*. London, UK: The Royal Society.

Royal Society (2017b). *Machine Learning: The Power and Promise of Computers That Learn by Example*. London, UK: The Royal Society.

Schocken, S., & Nisan, N. (2004). From NAND to Tetris in 12 easy steps. In *Proceedings of the 34th Annual Conference on Frontiers in Education* (p. 1461). New York: IEEE.

Seiter, L., & Foreman, B. (2013). Modeling the learning progressions of computational thinking of primary grade students. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research (ICER'13)* (pp. 59–66). New York: ACM.

Seiter, L. (2015). Using SOLO to classify the programming responses of primary grade students. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 540–545). New York: ACM.

Selby, C., & Woollard, J. (2013). Computational thinking: The developing definition. Retrieved from http://eprints.soton.ac.uk/356481

Sentance, S., & Csizmadia, A. (2017). Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies*, 22(2), 469–495.

Statter, D., & Armoni, M. (2016). Teaching abstract thinking in introduction to computer science for 7th graders. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 80–83). New York: ACM.

Susskind, R. (2017). *Tomorrow's Lawyers: An Introduction to Your Future*, 2nd edn. Oxford, UK: Oxford University Press.

Sykora, C. (2014). Computational thinking for all. Arlington: ISTE. Retrieved from www.iste.org/explore/articleDetail?articleid=152&category=Solutions&article=Computational-thinking-for-all

Tedre, M., & Denning, P. J. (2016). The long quest for computational thinking. In *Proceedings of the 16th Koli Calling Conference on Computing Education Research* (pp. 120–129). New York, NY: ACM.

Thimbleby, H. (2018). Misunderstanding IT: Hospital cybersecurity and IT problems reach the courts. *Digital Evidence and Electronic Signature Law Review*, 15, 11–32.

Turing, A. M. (1936) (published 1937). On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society*, 2(42), 230–265.

Waite, J., Curzon, P., Marsh, D. W., & Sentance, S. (2016). Abstraction and common classroom activities. In *Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (pp. 112–113). New York: ACM.

Waite, J., Curzon, P., Marsh, W., & Sentance, S. (2017). Teachers' uses of levels of abstraction focusing on design. In *Proceedings of the 12th Workshop in Primary and Secondary Computing Education* (pp. 115–116). New York: ACM.

Wing, J. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.

Wolz, U., Stone, M., Pearson, K., Pulimood, S. M., & Switzer, M. (2011). Computational thinking and expository writing in the middle school. *ACM Transactions on Computing Education (TOCE)*, 11(2), 9.