Should Your 8-year-old Learn Coding?

Caitlin Duncan
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
caitlin.duncan@pg.canterbury.ac.nz

Tim Bell
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand
tim.bell@canterbury.ac.nz

Steve Tanimoto
Department of Computer Science and Engineering
University of Washington
Seattle, Washington, USA
tanimoto@cs.washington.edu

ABSTRACT

There has been considerable interest in teaching "coding" to primary school aged students, and many creative "Initial Learning Environments" (ILEs) have been released to encourage this. Announcements and commentaries about such developments can polarise opinions, with some calling for widespread teaching of coding, while others see it as too soon to have students learning industry-specific skills. It is not always clear what is meant by teaching coding (which is often used as a synonym for programming), and what the benefits and costs of this are. Here we explore the meaning and potential impact of learning coding/programming for younger students. We collect the arguments for and against learning coding at a young age, and review the initiatives that have been developed to achieve this (including new languages, school curricula, and teaching resources). This leads to a set of criteria around the value of teaching young people to code, to inform curriculum designers, teachers and parents. The age at which coding should be taught can depend on many factors, including the learning tools used, context, teacher training and confidence, culture, specific skills taught, how engaging an ILE is, how much it lets students explore concepts for themselves, and whether opportunities exist to continue learning after an early introduction.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer and Information Science Education — computer science education, curriculum

Keywords

Coding, programming, young students

WiPSCE '14, November 05 - 07 2014, Berlin, Germany

ACM 978-1-4503-3250-7/14/11 ...$15.00
http://dx.doi.org/10.1145/2670757.2670774.

1. INTRODUCTION

As computer science (CS) curricula are appearing in more and more countries, a commonly debated question that arises (particularly in the popular press) is around what age students should have the chance to learn to program. In places where curricula don't exist, clubs, camps and informal programmes are filling the gap, so parents will also be asking this question.

Here we unpack this issue, which involves looking carefully at what we mean by programming, what kind of activities are relevant to various ages, what environment the child is learning in, and what the benefits are. We are not seeking to answer the question, but to work out what the question really is! The goal is to inform curriculum designers, teachers, and parents who must grapple with these issues.

The title of this paper is a response to an article circulated in April 2013, titled "Why your 8-year-old should be coding"[1]. The article is announcing a new creative and engaging learning environment for young children (Tynker), and focuses on job opportunities, helping children to see the options before them ("trying to educate them about those options when they still have years to form opinions and create and live their own dreams"), how "information and computation is coming to every field", and the value of this kind of experience for other STEM subjects. It concludes with the statement: "And that, dear readers, is why your eight-year-old should be coding." Of course, the article is intended to be provocative, but with many such discussions occurring in public and new curricula being deployed, it is important to collate the key arguments around this so that we are able to bring research-based facts to an emotionally charged and anecdote-laden discussion.

For example, some people with a contrary view, provoked by the article, raised questions about how important it is for an 8-year-old to be prepared for a specific job, pointing out that it may be more valuable to let children enjoy childhood without the pressure of preparing for a specific career, and that working with computers too much takes them away from living in the physical world and developing one's imagination.

Such issues are also being raised in the design of new curricula, where programming is being introduced in primary schools as a formal requirement [1]. To address these properly, we must define what we mean by "programming", and consider the cognitive abilities of young children, as well as

[1] `http://venturebeat.com/2013/04/12/why-your-8-year-old-should-be-coding/`

the ability of our education systems to deliver and teach programming meaningfully to a typical student. Such discussions must also be culturally situated; in some countries programming is already taught by qualified teachers in primary schools, whereas in others teachers with no computing qualifications are having to teach this unfamiliar subject.

Gross and Powers [9] review the evaluations of about two dozen programming environments for novices, and they reported that robust assessments are hard to find. They observed that the assessment of a tool is most often done by those who wrote it, and the assessment can be opportunistic (around a self-selected group trying the new tool). They also find that details can be missing from descriptions of evaluations that make them hard to scrutinize. These evaluations can help us to determine *if* a particular tool is effective, but the question we are more concerned with here is *at what age* the tool should be used, and what other factors affect its usefulness (such as teacher confidence).

Many programming environments for novices have been developed, and it's important to acknowledge that these are very creative innovations that are simple for students to use, and have often seen very widespread adoption which in itself is a testimony to their usefulness.

In this paper we catalogue the many issues around teaching young students to "code," survey the kinds of environments that have been developed to support teaching programming to young students, and draw out the issues that need to be considered when deciding if students of a particular age should have lessons in programming.

2. BACKGROUND

Papert's "Mindstorms" reports early efforts to engage with school students in the late 1960s, and his Logo language was a landmark in taking programming into the classroom where "children of almost any age could learn to program in Logo under good conditions with plenty of time and powerful research computers" [25]. The caveats at the end of the sentence indicate the issues that can come up; access to suitable computers has improved with time, but providing "good conditions" and "plenty of time" remain a challenge for school systems, and this raises the concern that teaching programming without the right conditions may be worse than not teaching it at all. Similar issues were considered by Alan Kay [15], who ran trials with SmallTalk around 1973 with middle school children in Palo Alto. They had some very positive results, but in a 1993 paper he observes that "in part, what we were seeing was the 'hacker phenomenon', that, for any given pursuit, a particular 5% of the population will jump into it naturally, while the 80% or so who can learn it over time do not find it at all natural', and that "real pedagogy has to work in much less idealistic settings". Successful experiences in the classroom can be encouraging, but we do need to check that they generalise to the typical classroom run by typical teachers.

2.1 Cultural Factors

The best age for teaching programming also depends on the educational system, culture, and personal experiences. Comments in the media are often based on personal experience; one person may have had an inspiring teacher who included programming in a class, while another may have had very dull "ICT" lessons. Anecdotal stories of individuals who have done particularly well at a young age don't help us to form a view of what should be offered to *typical* students.

Culture also affects attitudes to learning and careers. In this paper we will mainly focus on western culture, as the shortage of qualified software engineers and computer scientists, and the gender bias associated with the disciplines, is largely a western phenomenon. In countries such as India and China, students are more motivated to train in this area which has strong employment prospects, and they will seek opportunities rather than needing a curriculum that "sells" the career to them.

In the west, the gender imbalance in CS is also an issue, and student numbers would increase substantially if as many women went into a CS career as men. However, in other countries (e.g., Malaysia), computing is already seen as a strong career path, and gender balance is not the issue that it is in countries such as the US and UK [22].

2.2 Changing School Curricula

Several western countries are in the middle of changes to school curricula as they see the value of introducing topics such as programming, CS and computational thinking. For example, the changes in both the UK and Australian curricula that are being introduced in 2014/2015 include introductory programming experiences in primary schools. Such changes stimulate public debate. A key driver for updating the school programmes has been articulated by Rushkoff: "We teach kids how to use software to write, but not how to write software. This means they have access to the capabilities given to them by others, but not the power to determine the value-creating capabilities of these technologies for themselves" [29].

Against this, there are concerns that programming may not be a required skill in the future, either because it is somehow automated, or is outsourced overseas. A related argument is that programming is a sweatshop activity, raising images of training students to be cogs in a large industrial wheel. History doesn't support any of these ideas, and the balance for this argument is the value of *computational thinking* and understanding how technology works. It is almost certain that today's students will be interacting with technology throughout their working lives, regardless of what career paths they choose, so having an understanding of technology and its computational limitations will serve them well.

Clubs and computer camps are also available, which provide de-facto curricula either where the school system doesn't provide it, or as an extension for interested students. Because they are not part of a compulsory system, and students in the clubs are largely self-selected, many of the issues that we raise won't be as serious in these environments. Students who have already discovered a passion for programming can benefit from clubs and camps where they are able to get suitable mentoring and encouragement.

3. WHAT IS CODING?

The word "coding" is very widely used to describe the skill needed for computer programming. The term is widely used by organisations that promote learning programming, such as code.org, Made with Code, Code Club, CoderDojo, Black Girls Code, Codecademy, Code Avengers, CodeHS and MotherCoders. Before we unpack the question of when students should have the opportunity to learn to "code" we need to explore what we mean by "coding".

In the world of software the word "coding" is regularly

used interchangeably with the word "programming", and is often considered to mean the same thing. In the field of CS, the term "code" is used with a variety of meanings: in relation to data analysis, encryption, compression, error correction (source coding), machine code, even binary "codes." In fact, "source coding" refers to Shannon's work, and is quite different from "source code." In the context of programming, traditionally "coding" would only refer to the last stage of the process of programming, translating a designed program into programming expressions and typing/entering these into a computer.

So what is the difference between coding and programming? Are they simply different words for the same thing, or is the former a tool used to accomplish the later? Both uses are common, and in the context of education there are advantages and disadvantages to using these words as synonyms. The main advantage of using this word is that it captures interest. "Code" is a popular buzz word in today's technology driven world, and it also provides an element of mystery (there are hints of a secret code), and achievement (cracking the code). As mentioned previously, it is a term already used by many groups associated with learning programming.

The disadvantage, however, is that it can cause confusion among students and teachers. For example, if we use the meaning that "coding" is only entering programming expressions, students may only be taught to take existing code or pseudo-code and enter it themselves. Many existing tutorials currently follow this model, with some only asking students to make small changes to values in the program. While this may reveal to them how a program works, it could also lead to them becoming overconfident as they may believe this is all there is to programming when it really also involves many other skills, such as getting specifications, planning and debugging.

From a constructivist point of view, having students write new code and wrestle with requirements has more educational value, but there is also value in having a student actually see what "code" looks like, type it in, and have it execute; this may inspire them to get over the barrier that coding is some magic incantation that is beyond them. An argument in favour of the latter is that they are learning to read before they write, which would seem an obvious approach, although there are also arguments that learning to write is a good way to learn to read. Lopez *et al* show evidence that the ability to write has a correlation with the ability to read code [17].

An experience of teaching young students by having them work through sample programs is reported by Smith *et al.* in the context of the UK "Code Club" [31]. They report that a common comment from survey respondents was that "children were able to follow the early, guided instructions but were unable to apply any knowledge to unguided challenges." The survey also revealed that from a list of several concepts, the area that students were least confident with was debugging, an essential skill for creating one's own programs. This has prompted an effort to develop new pedagogy to address these concerns.

So an important question we must tackle now is what is programming? The definitions of "programming", "programmers" and "programming languages" have changed and developed over time as software, hardware and usage of computers has changed [5]. In 2004 Guzdial observed "Perhaps we don't know yet what programming really is or what it could be" [11]. As the technology industry is ever-changing it can be expected that these definitions will remain fluid in the future. Blackwell [5] observes that people refer to programming a computer, a video recorder or a microwave oven, but not a word processor document. The common feature of things that are programmed is that the user is *not* engaging in direct manipulation; that is, the sequence of actions is specified in advance, and then the programmer must effectively wait to see if the sequence of actions produces the desired result. From this point of view, beginner programming (for example, with Scratch) is close to direct manipulation (in fact, students can double click on a "Forward" command to execute it). However, it provides a transition away from this as long as the teacher encourages students to do so. A trial-and-error approach to programming, combined with a very simple repertoire of control commands, can lead to what some teachers have referred to as a "Spritefest", where students simply "program" sprites to move around, and make do with how the program happens to work if they don't understand how to achieve the effect they were after.

For the purposes of the discussion here, we will use the term "programming" for the broader activity of analysing a problem and implementing a program that solves it, which isn't always in a speaker's mind when he or she says "coding." The analysis or design of the program is challenging in its own right, particularly if the requirements are inflexible. Typically this requires students to come up with an algorithm expressed as pseudo-code or some other general notation that will then be coded into a program. In the New Zealand NCEA standards for high school programming, the two tasks of design and implementation can even be assessed separately; it is possible that a student can demonstrate the ability to implement a program from pseudo-code, but not get credit for the design of the program — or vice-versa [3]. Implementation skills need to go beyond simply translating pseudo-code to code, as there is also testing and debugging the program, which can include devising suitable test cases.

4. YOUNG STUDENTS PROGRAMMING

Several studies have been made of how young students use coding Initial Learning Environments (ILEs), and these studies provide clues about the kind of programming they are doing, the concepts that students pick up from the experience, and any changes of attitudes they form towards programming and CS.

Meerbaum-Salant *et al.* [20] looked at the habits of middle-school students programming in Scratch, and found that they invariably used a bottom-up development process (selecting commands, and then combining them for a desired effect), and focused on "extremely fine grained programming" (having many very small scripts that made the logic of the program difficult to follow). They point out that this is at odds with good programming habits. Furthermore, they observed that control structures were often used incorrectly, and that students avoided the fundamental structures of conditional execution and bounded loops. Their concern was that these students are developing *habits*, which would need to be broken later, although they conclude with the dilemma that there is value in making it "easy" for students to get into programming, and that sound programming techniques are not easy to learn. Also, the version of Scratch (in 2011) being used would not have supported the

writing of functions, and these may turn out to encourage better top-down design habits.

Kaucic and Asic [14] evaluated the use of Scratch with 32 students in primary school in Slovenia and found that (for this small sample at least) students who had used Scratch continued to use it more than those who had learned a conventional programming language. This could be taken to be an indicator of the value of such languages for forming long-term opinions about the enjoyment of programming.

Werner *et al.* [33] investigated work done by middle-school students programming games using Storytelling Alice (a variant of the Alice ILE). They found that 13 out of the 23 programs developed met at least 4 criteria for being a game, such as having rules, a goal, and an un-predetermined outcome. Having this high proportion of students produce a game with these elements indicates an ability to design a program as well as implement it. A later more detailed investigation found that, while writing games, middle-school students were able to work successfully with several abstract constructs, but the students "struggle with variable initialization, looping, conditionals, pointers, and recursion" [33]. This gives an indication that there might be a limit on the level of abstraction that students of this age can naturally work with, at least without a different pedagogical approach.

The Code Club survey mentioned earlier [31] also found that while students were comfortable with several concepts, being able to design and debug their own programs could be a challenge (although there were some who were able to build their own projects confidently).

The value of teaching programming to young students is generally supported for one of two reasons: enabling students to understand what programming is all about, and the general value of *computational thinking* (CT) which will be of use regardless of a student's career.

The CT argument is stronger since it applies to more of the population. Resnick *et al.* [27], when describing the reason for developing the Scratch ILE, say that their "primary goal is not to prepare people for careers as professional programmers but to nurture a new generation of creative, systematic thinkers comfortable using programming to express their ideas".

Of course, CT involves more than learning to program, but many of the concepts in CT are exercised during programming. Some ILEs encourage CT. While even the simplest of them introduces sequencing (i.e., giving instructions without direct manipulation), some enforce the need for concepts like decomposition and abstraction by placing limitations on the commands available to achieve a goal.

In addition to tools for teaching programming, there is also a wide variety of off-line material for teaching ideas from programming (e.g., [4]), to inspire young students to engage in CT. Lack of space prevents us from listing such material here, but we note that this kind of teaching can be a valuable supplement to working on a computer, and much of it is suitable for young students.

5. STUDENTS' AGE AND PROGRAMMING

To think about the question of what age a student should be learning programming, we look at it from several angles, including ideas from developmental psychology, the relationship between age and gender issues, the nexus with other disciplines, and the opinions of practitioners (including those designing school curricula).

5.1 Developmental Psychology

Developmental psychology helps us to work out what kind of concepts students are able to cope with at various ages. Piaget's four stages of development [26] provided age-based milestones, with boundaries at 2-, 7-, 11-, and 16-years of age. These boundaries are no longer regarded as universal, but they do characterise commonly observed transitions for typical children as they gain an awareness of a world outside their own, and eventually develop abstract and logical thinking. In Piaget's model, the age from 7 to 11 is important as the phase where they become physically dexterous as well as gaining the mental ability to understand the world around them better, and think logically. However, the formal operational stage (which includes working with symbols and using logical reasoning) would seem to map well to introductory computer programming, which Piaget pigeon-holed as 11 to 16 years. Neo-Piagetian theories [23] have established that assigning specific ages to cognitive development isn't that simple, and can depend on the area of learning and the individual. For example, there is evidence that tertiary students go through neo-Piagetian stages for programming [6], while it is also clear that some programming concepts can be taught to students as young as 5 to 7 years old [7, 16] as long as the language uses concepts accessible to that age group (e.g., not requiring large numbers).

Seiter and Foreman [30] introduce their "Progression of Early Computational Thinking" (PECT) model which is intended to provide research-based underpinnings for the design of curricula for primary grades (Grades 1 to 6). This was based on the patterns used by students programming in Scratch. For example, they found that Grade 1 students were able to work with patterns called *Animate Looks* (changing the look of a sprite) and *Conversate* (conversations between sprites), and somewhat with *Animate Motion* (changing location with possible syncronization), but not with concepts that involved selection or controlled repetition (*Collide*, *User Interaction*, *Maintain Score*). The *Maintain Score* pattern, which includes variables and boolean expressions, was achieved only in grades 4 and up. Seiter and Foreman note that "this data seems to indicate that certain patterns are best suited for certain grades".

Specific results with young students using sequencing have been reported: Kazakoff [16] reports success with kindergarten aged students learning sequencing through robotics programming, and an experiment with programming for primary schools in Slovakia with students aged 8 to 10 years old has reported success with students working with a custom designed environment in which students move from command-by-command control of an object to move it to a target in 2D space, to giving the commands in advance [10].

5.2 Gender Issues

The lack of female students studying CS at a tertiary level can be traced back to relatively early in their schooling, and increasing the number of women in CS would have a large impact on overall numbers, as well as improving the quality of software by increasing diversity [2]. Margolis and Fisher observed that the middle school years (ages around 12 years) are a critical time when it comes to encouraging female students interest in CS and programming [19]. During High school and puberty the majority of female students experience a drop in their self-esteem and confidence in computing, but previous experience (i.e., in primary years) can assist them in maintaining their confidence in CS.

Positive school experiences with computing curricula increase the likelihood that female students will study CS in the future. However negative experiences, such as poor teaching, also have a very strong and lasting impact on their interest in CS careers [19]. This suggests that if a teacher does not wish to, or does not feel confident teaching programming then it may be better if programming is not taught extensively in school, and instead students are encouraged to pursue it in their own time or through a club.

Riegle-Crumb *et al.* [28] evaluate the balance between learning that inspires students, and learning that gives them skills to achieve well, and conclude that "an educational system that focuses on increasing achievement without some degree of attention to whether students are engaged and having positive experiences is unlikely to produce greater numbers of future scientists, especially female ones." They note that career aspirations become apparent before students enter high school, which supports the idea that learning at primary school is valuable, but also should be a positive experience.

5.3 Nexus With Other Disciplines

Many concepts in programming build on concepts from other subjects, such as mathematics and language, so one factor in considering what students can learn is how strong they will be in related subjects. For example, Scratch programming can involve working with a coordinate system and negative numbers, which may be difficult for a younger student to understand. Many computer-based games incorporate simple computational models involving possibilities of motion and achieving states, or scoring points. The mental models required to understand these games are often abstractions that can be compared with mathematical ideas (e.g., functions – "I put in this, and that comes out.") This illustrates that the pedagogy can work both ways; computer programming can be used as a vehicle for learning concepts for other subjects[2], but nevertheless it would be unreasonable to expect a young programmer to have to grapple with a very advanced concept from another subject in order to progress.

5.4 Learning Natural Languages

Another clue about teaching programming can be drawn from what we know about learning natural languages, and often an analogy is drawn between learning a new natural language and learning a new programming language. The "critical period effect" in natural language learning is the idea that a language is best learnt when one is young, and there is good evidence that learning a new language beyond puberty is more difficult [13]. This establishes the beginning of puberty (typically around 10 to 12 years old) as a critical milestone for young students, and the time immediately up to this is a key time for learning. Our own experience teaching to various age groups has been that the 10 to 12 year old age group is particularly ready to absorb new ideas in CS. While younger students can also be enthusiastic and open to learning, they have fewer social and academic skills to build on, making progress slower, and beyond 12 years old they can be less open to new ideas. One of our long-term goals is to quantify this effect which is currently based only on anecdotal experience.

Of course, although learning models and observed norms offer predictions, there will be outliers, and often these are the ones visible in the media. Many of the industry leaders (e.g., Gates, Jobs, Zuckerberg) were exceptional as children, yet are often used as examples to argue how a student's career might progress. The reality is that the majority of people who are hired by companies (including those owned by the leaders just mentioned) would normally have completed a full K-12 schooling programme and a university degree.

5.5 Opinions of Practitioners

Another benchmark for the age at which programming can be taught is the opinions of those who have been working with students.

One manifestation of these views are curricula that have been developed. For example, in the 2011 CSTA K-12 curriculum[3] activities for programming are suggested at all levels; Level 1 (K–6) focuses on sequencing, while Level 2 (grades 6–9) *collaborative* programming is suggested without strong guidance on the constructs to be used; and at Level 3 algorithmic problem solving (i.e., programming) is explicitly suggested, with conditions, variables, operators and other general constructs mentioned.

In 2014/2015 both the UK and Australia are adopting new curricula where primary school students will be learning elements of programming. For example, the UK curriculum[4] specifies "create and debug simple programs" at Key Stage 1 (KS1, about 5 to 7 years old), and "sequence, selection and repetition" at Key Stage 2 (about 7 to 11 years old). Supplementary materials would indicate that systems like "Beebot" are expected at KS 1 (i.e., sequencing only), and languages like Scratch are expected at KS2. The Australian curriculum[5] specifies following sequences in years F–2 (up to 8 years old), the use of branching and user input in years 3–4 (8 to 10 years old), and iteration is added at years 5–6 (10 to 12 years old), all as "simple visual programs" i.e., using a language such as Scratch. Thus both of these curricula expect the full range of programming commands to be covered by the end of primary school.

Another view from professionals is a poll made by the makers of Tynker, who asked teachers at ISTE "How early should computer programming skills be introduced to kids?"[6] Elementary (primary) school was the main response (75%), although the report noted that some teachers "fervently disagreed" with starting that young, and others felt that it should be optional.

Another angle on age and learning to program is provided by an informal study by Neil Fraser, who surveyed colleagues at Google, asking them at which age they could first program, and a simple evaluation of their current programming skills[7]. The staff most likely to become good programmers had reported starting programming between grades 3 and 7 (about 8 to 13 years old). Of course, we

[2]For example, see `http://scratched.media.mit.edu/resources/investigation-using-scratch-teach-ks3-mathematics`, `http://computerbasedmath.org/`, and `http://www.codebymath.com/`

[3]Available from `http://csta.acm.org/`

[4]`http://www.computingatschool.org.uk/data/uploads/primary_national_curriculum_-_computing.pdf`

[5]`http://www.australiancurriculum.edu.au/technologies/digital-technologies/Curriculum/F-10`

[6]`http://www.tynker.com/blog/articles/stem-education/at-what-age-should-kids-start-learning-programming/`

[7]`https://neil.fraser.name/news/2012/07/01/`

can't assume that this correlation implies causation, and since the results come from a school system where computing hasn't been taught well for those who are interested in it, it's inevitable that the results reflect self-selection. However, it does add some weight to the value of students being exposed to programming before they leave primary school.

6. TEACHING AND LEARNING TOOLS

We have collected a list of 47 tools that can be used for teaching programming and programming concepts to children. These tools range from novice programming environments (for educational and also industry level languages), to games and challenges which require programming or programming concepts to complete, game building environments which involve programming, and online "learn to program" courses. We refer to these tools as Initial Learning Environments (ILEs). To fit into this category a tool must allow, or be designed for, a novice to interact with the tool and have the opportunity to learn programming concepts. These ILEs can be used independently of teacher guidance, but many of them can be used with additional teacher support and instruction.

The 47 tools we have examined for this paper do not cover all available ILEs as there are a huge number of these. It does however include the majority of the most well known ones. Taking in to account that 31 of the 47 ILEs we examined were released later than 2010 (including several which are yet to be fully released) and that public interest in "coding" curricula is growing, many more of these ILEs are likely to be developed and released in the future.

With so many of these ILEs available it is inevitable that some will be better suited to teaching programming concepts to a target age group and ability level than others. While having a large number of ILEs will give learners many choices and opportunities to learn, there is a risk that students and teachers may be overwhelmed by the range of choices. We have attempted to classify these so that similar ILEs can be examined as a group to assist educators who are choosing tools and deciding how to use them effectively.

There are many different criteria against which these can be classified. Many come with recommended age ranges, although it is not always clear if these are based simply on the age group who enjoy the ILE the most rather than the suitability of the concepts taught. They can be accessed in numerous ways including through a browser, downloads for Linux, Windows and OS X, apps for Android, iPhone, iPod touch and iPad, and some are available on XBox and Playstation. They vary greatly in cost, although a large number are free.

The majority of ILEs fall into one of two groups based on whether their interfaces and/or the programming language use Drag-and-Drop or Text-based input. Text-based environments involve the user entering text to form programming expressions in order to interact with the environment. This text is either a commonly used programming language, e.g., Python, Java, or JavaScript, or in some cases it is a novel language that only exists in the ILE. Drag-and-Drop environments on the other hand do not require users to manually enter programming expressions; instead they provide the user with a selection of 'blocks' that represent programming expressions.These blocks can then be used by the learner to interact with the ILE, most commonly by dragging and dropping the blocks into an area where they can build a program, although several simply have the user click the blocks and do not have an area for 'building' programs. Drag-and-Drop environments have become very popular for teaching programming to young children as they do not require knowledge of programming syntax and provide an environment where compile-time errors are nonexistent. For example, in the Drag-and-Drop language Scratch, any program a user builds with the Scratch blocks will run (rather than giving a build or compile error) because it is impossible to join two blocks together if they are not *allowed* to be joined. This prevents novices from encountering confusing error messages, which can be very discouraging to learners. However, as Drag-and-Drop languages tend to limit exactly how much you can create, and are not widely used for professional software development, it is useful that text-based ILEs do exist for older and/or more experienced learners.

Ease of adoption is also an important consideration for schools who wish to use an ILE in the classroom. A free-to-use ILE that can run in a browser on any computer will be much easier for a school to acquire and set up than, for example, an ILE that must be purchased and requires a specific device such as an iPad or an XBox to be used. ILEs that need to be installed on a computer can also be challenging for schools to adopt as downloading, installing and updating software is often a complicated process for school network administrators who have a large number of machines to look after and are constrained by school policies for new software.

For the purposes of our research we have created a set of heuristics that can be used to classify an ILE according to a 'Level' between 0 and 4. These levels are intended to give an idea of the approximate age group, ability level and learning outcomes that each ILE is best suited for. The heuristics we used are as follows:

Level 0 — Age range 2-7 years. Drag-and-Drop or simpler. Teaches planning (sequence) only. Requires no abstraction. Contains no significant use of: functions, variables, iteration, indexed data structures, conditional execution.

Level 1 — Age range 5-10 years. Drag-and-Drop. Requires no abstraction (or small amounts). Contains none or few of: functions, variables, iteration, indexed data structures, conditional execution.

Level 2 — Age range 8-14 years. Drag-and-Drop or text-based. Includes some abstraction. Contains some or most of: functions, variables, iteration, indexed data structures, conditional execution.

Level 3 — Ages 12 years and up. Drag-and-Drop or text-based. Includes abstraction. Contains all of: functions, variables, iteration, indexed data structures, conditional execution.

Level 4 — Ages 14 years and up. Teaches an industry-level Turing-complete programming language. Advanced, with extensions available. Contains all of: functions, variables, iteration, indexed data structures, conditional execution.

Table 1 shows a list of ILEs surveyed giving the approach they use, their cost, and their level classification. The age range of an ILE was generally determined by the recommended age provided by the creators, although in some

Table 1: Initial Learning Environments for young students (prices are in $US)

Name	Type	Cost	Level
Bee-Bot app	Drag and Drop	Free	0
Cato's Hike	Drag and Drop	$4.99	0
Daisy the Dinosaur	Drag and Drop	Free	0
Lightbot jnr	Drag and Drop	$2.99	0 1
My Robot friend	Drag and Drop	$3.99	0 1
ScratchJnr	Drag and Drop	Not available yet	0 1
Beta	Text	Free Lite, $10 Full	1
Hakitzu Elite: Robot Hackers	Javascript	Free, in app purchases	1
Kodable	Drag and Drop	Free app, $6.99 Pro, $2.99 per student Sync	1
Lightbot	Drag and Drop	$2.99	1
Lightbot lite	Drag and Drop	Free	1
Robo Logic	Drag and Drop	Free Lite, $0.99 Full	1
Robo Logic 2HD	Drag and Drop	Free Lite, $2.59 Full	1
Cargo-Bot	Drag and Drop	Free	1 2
Code Kingdoms	Drag and Drop, some Text	Free	1 2
Kodu	Drag and Drop	Free on PC, $5 on Xbox	1 2
Move the Turtle	Drag and Drop	$2.99	1 2
Turtle Academy	Logo	Free	1 2
Blockly	Drag and Drop	Free	1 2 3
HopScotch	Drag and Drop	Free	1 2 3
Scratch	Drag and Drop	Free	1 2 3
Tynker	Drag and Drop	$50 per course	1 2 3
Code HS	Karel, JavaScript, HTML, CSS	Free, $25, $75, $400 depending on subscription	1 2 3 4
Codemancer	Drag and Drop	Not available yet	2
CodeSpells	JavaScript	Free	2
i-Logo	Logo	$2.99	2
NetLogo	Logo	Free	2
StarLogo TNG	Drag and Drop	Free	2
12blocks	Drag and Drop	$49 Lite, $99 Standard	2 3
Alice	Drag and Drop	Free	2 3
AppInventor	Drag and Drop	Free	2 3
Code Combat	JavaScript	Free	2 3
Greenfoot	Drag and Drop + Java	Free	2 3
Hackety Hack	Ruby	Free	2 3
Looking Glass	Drag and Drop	Free	2 3
Snap! (BYOB)	Drag and Drop	Free	2 3
KidsRuby	Ruby	Free	3
StarLogo	Logo	Free	3
StarLogo Nova	Drag and Drop	Free	3
BlueJ	Java	Free	3 4
Code avengers	JavaScript, HTML, CSS	Free	3 4
Codecademy	Python, Ruby, Javascript, HTML, CSS, PHP	Free	3 4
Minecraft	JavaScript	$26.95	3 4
Squeak	Small Talk	Free	3 4
Codea	Text	$9.99	4
Khan academy	JavaScript	Free	4
Simduino	Arduino C	$1.99	4

cases this was not available or did not match the age range of similar ILEs. Not all ILEs fit these heuristics exactly. For example Hakitzu Elite, Robot Hackers uses the Industry level language JavaScript, and the recommended age is higher than the range for level 1; however, it doesn't include creating functions, loops, conditionals etc., which lowers its classification. Several ILEs are shown spanning several levels. In the cases of game or challenge based ILEs this is generally due to different game-levels requiring different levels of programming concepts. Several of the ILEs, e.g., Codecademy and Code HS, have tutorials for different languages and ranges of ability and so span multiple levels. Others that span multiple levels generally contain the functionality of the highest level they have been classified at, but do not force this on the user and so can be used for lower ability levels as well. Other features that were not included in these heuristics but have still been identified as significant are "share and remix" abilities (students can share their work in an online community and view other users' projects and explore and learn by 'remixing' them), and the availability of teaching resources.

The challenge of an ILE is sometimes described as trying to offer a low floor (minimal barriers to starting), a high ceiling (not putting limits on how far a project can be taken), and (ideally) wide walls (can be used for a wide variety of projects) [27].

The Level 0 ILEs are targeted at very young users and focus only on sequences of operations. They aim to teach learners the concept of planning a program and having the actions they have selected execute sequentially. They are highly visual and involve the manipulation of objects with Turtle-graphics-like commands; for example the buttons on the BeeBot instruct the Bee Bot to move forward or turn. This provides young learners with scaffolding as their actions have a clear and direct impact on the objects within the app. These ILEs are very limited however and if a learner has an idea for something he or she would like to program, it's unlikely one could do this with these applications.

The Level 1 ILEs are still generally drag-and-drop environments with clear manipulation of objects, but offer more functionality than the Level 0 tools as they tend to include more "blocks" (e.g., functions or conditionals). They tend to focus on teaching slightly more than the planning of the Level 0 ILEs.

The Level 2 ILEs have another increase in functionality and complexity. At this stage we begin to include text-based ILEs, including several that use industry level programming languages. (These do not allow full use of the language, however, and so the are distinct from the Level 4 ILEs.) Each of these contains some level of abstraction, most often in the form of variables and functions.

At Level 3 an ILE will involve a language that is Turing complete, or very close to this. Most of these will have much 'wider walls', and learners will find it easier to take their ideas and implement them. This is significant as it will allow learners to express their creativity and learn in a constructivist manner.

Level 4 ILEs are for advanced school students who have a strong interest in programming or have already become tired of lower level ILEs. They would not be necessary for a typical school curriculum, but could still be used in high school if students and teachers felt up to the challenge. The fact that these ILEs exist is important because they offer advanced students the opportunity to extend their skills. However they also pose the risk of school students exhausting the available learning content and becoming bored in the future when they run out of possible extensions.

Based on the reviews above, for primary school aged students the ideal levels to use for learning would be 1, 2 and 3. However, each of these ILEs has individual strengths and limitations that need to be taken into account. Some ILEs have very structured environments that guide learners through specific concepts in a predetermined order. For example, Lightbot introduces students to the concepts of sequencing and planning actions, and later it introduces and forces the use of functions to complete challenges. The advantage of this strategy is that it ensures that students are introduced to each of the concepts the ILE aims to teach, which could make it particularly useful to students learning in their own time or to teachers who are not yet confident in teaching programming. The downside of this, however, is that once a student has completed the tutorial or challenges he or she is then unable to continue exploring.

The ILEs that do not follow this method, for example Scratch, Alice and Hopscotch, offer a programming environment with much more freedom. Students are able to experiment with different "blocks" and concepts in any way they please, and while tutorials and challenges do exist for many of these environments, they are not forced upon the learner. The drawback of these environments, however, is the risk that students come away without having learnt many (or any) programming, CS or CT concepts, or they may develop an incorrect model of what programming is. We have referred to this previously as a "Spritefest", where students focus on the simple animation or turtle-movement functions and don't extend themselves to more abstract concepts such as conditionals. However, with proper teacher guidance these environments can be used to great benefit for teaching programming skills and abstract CS and CT concepts [32]. Their richer functionality allows students more opportunities to explore programming.

7. PREPARING TEACHERS

Having good programming tools and capable learners isn't sufficient for teaching programming in schools, as teachers need to be able to deliver the material confidently in the context they find themselves in. For example, Goode et al. report this observation in a high school programme [8], and Meerbaum-Salant et al. [21] found that Scratch is suitable for teaching fundamental programming concepts but only if teachers have sufficient guidance on good didactic approaches, which in turn takes time. Yadav *et al.* [34] compared attitudes towards CT between teachers who had had specific training in the area and those in a control group. They found that those in the control group, who didn't have training, had misunderstandings about what CT is and how it could be taught. Ni and Guzdial [24] highlight the importance of teachers' sense of identity so that they can be committed and effective teachers

One exception, where students' success was independent of teacher's experience is at the Kindergarten level [16]. There, the material being covered uses only sequencing. For this level sequences are not peculiar to programming, and a teacher is likely to be familiar with the ideas of giving instructions in sequence!

A teacher's attitude and pedagogical knowledge will be related. For example, Makris [18] reported on two groups

of 12 to 15 year old students in Greece who were taught Scratch programming. One group had programming demonstrated to them, while another group had an "encouraging style," where the demonstrator encouraged them as they worked out for themselves how to achieve the goal. The latter group showed a considerably more positive attitude to programming afterwards. Confident teachers are better placed to convey a positive attitude to students, and in the opposite situation, teaching programming to young students may do more harm than good if the teacher is confused or frustrated.

8. CONCLUSIONS

Teaching programming to young students has both costs and benefits. The benefits include enabling students to become fluent in programming while they are at an age when they can learn quickly, shaping attitudes to programming before it is too late, and support learning outside of just programming (particularly maths and CT). The costs include equipping teachers to deliver programming in their classes, and the time taken away from other subjects. There are also risks: if a student is taught programming by a teacher who lacks confidence, there is the possibility that the student will create a negative impression of the subject.

There isn't a simple answer to the question that the title of this paper poses, although it is clear from a variety of evidence that some exposure to programming before about 12 years old is both worthwhile and feasible. However, it's more important to ask the right questions, and avoid jumping to conclusions based on exceptional cases, bad experiences, or opinions in the press. Based on the issues raised above, we have synthesised the following list of considerations that need to be explored when considering what kind of "coding" should be taught to a particular age group, and in a particular environment. Many of the issues raised here are those covered by the Darmstadt model which considers the factors involved in the delivery of a curriculum in computing [12].

- Can teachers be supported to be confident and competent teaching programming? If not, requiring programming could have a negative influence on students' attitudes. This is particularly important for female students as building confidence in programming while at primary school is more likely to translate to long-term confidence in their ability and help address the shortage of females in computer science.
- We need to be clear what is meant by programming: at younger ages just sequencing is a useful concept to master; for older students programming should also include design, implementation, debugging and testing.
- When selecting which programming concepts to teach and which ILE to use consider the following
  - The age and maturity of the students
  - Their mathematical ability and language skills
  - The purpose of teaching programming; in some situations it is to capture student's enthusiasm, at more senior levels it could be to develop skills that will be useful in a career.
  - Are there good teaching resources (such as books, tutorials, and web sites) available for the ILE?
  - The ILE should have a "low floor", and where possible, "wide walls" and a "high ceiling" [27].
  - There needs to be a pathway beyond an initial experience if students become enthusiastic.
- The risk of teaching programming depends on the context: there are higher stakes at school where results may impact grades and qualifications, whereas in lower grades and clubs students are more able to experiment.
- With limited space in the curriculum, existing material may have to be removed to make space, although programming could also be used to support parts of the existing curriculum, particularly math.
- If a student is already enthusiastic then it is important to maintain their passion, and a club or mentoring arrangement may be best.

Teaching programming to younger students is clearly possible, and it has benefits. The appropriate aspects to teach depend on many factors other than just a student's age, including the context and teacher confidence. We have identified many of the issues involved with the intention of informing discussions and planning around teaching programming to younger students. We have also uncovered areas that need more research before we can draw strong conclusions.

9. REFERENCES

[1] B. M. Armoni. Designing a K-12 computing curriculum. *ACM Inroads*, 4(2):34–35, 2013.

[2] C. Ashcraft and A. Breitzman. *Who Invents IT? An Analysis of Women's Participation in Information Technology Patenting, 2012 Update*. National Centre for Women & Information Technology, 2012.

[3] T. Bell, P. Andreae, and A. Robins. A case study of the Introduction of Computer Science in NZ schools. *ACM Trans. Computing Educ. (TOCE)*, page to appear, 2014.

[4] T. Bell, P. Curzon, Q. Cutts, V. Dagiene, and B. Haberman. Overcoming Obstacles to CS Educ. by using Non-Programming Outreach Programmes. In *Proceedings of Informatics in Schools: Situation, Evolution and Perspectives (ISSEP) 26, Bratislava*, LNCS 7013, page to appear, Oct. 2011.

[5] A. Blackwell. What is programming. In *14th workshop of the Psychology of Programming Interest Group*, pages 204–218, 2002.

[6] M. Corney, D. Teague, A. Ahadi, and R. Lister. Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. In *Proc. Fourteenth Australasian Computing Educ. Conference-Volume 123*, pages 77–86. Australian Computer Society, Inc., 2012.

[7] L. P. Flannery, B. Silverman, E. R. Kazakoff, M. U. Bers, P. Bontá, and M. Resnick. Designing ScratchJr: Support for Early Childhood Learning Through Computer Programming. In *Proc. 12th Int'l. Conf. Interaction Design and Children*, IDC '13, pages 1–10, New York, NY, USA, 2013. ACM.

[8] J. Goode, J. Margolis, and G. Chapman. Curriculum is Not Enough: The Educational Theory and Research Foundation of the Exploring Computer

Science Professional Development Model. In *Proc. 45th ACM Technical Symposium on Computer Science Educ.*, SIGCSE '14, pages 493–498, New York, NY, USA, 2014. ACM.

[9] P. Gross and K. Powers. Evaluating assessments of novice programming environments. In *Proc. 2005 Int'l. workshop on Computing Educ. research - ICER '05*, pages 99–110, New York, New York, USA, Oct. 2005. ACM Press.

[10] M. Gujberova and I. Kalas. Designing productive gradations of tasks in primary programming education. In *Proc. 8th Workshop in Primary and Secondary Computing Education*, WiPSE '13, pages 108–117, New York, NY, USA, 2013. ACM.

[11] M. Guzdial. Programming environments for novices. *Computer science Educ. research*, 2004:127–154, 2004.

[12] P. Hubwieser, M. Armoni, T. Brinda, V. Dagiene, I. Diethelm, M. N. Giannakos, M. Knobelsdorf, J. Magenheim, R. Mittermeir, and S. Schubert. Computer science/informatics in secondary education. In *Proc. 16th annual conference reports on Innovation and technology in computer science education-working group reports*, pages 19–38. ACM, 2011.

[13] J. S. Johnson and E. L. Newport. Critical period effects in second language learning: The influence of maturational state on the acquisition of English as a second language. *Cognitive Psychology*, 21(1):60–99, Jan. 1989.

[14] B. Kaucic and T. Asic. Improving introductory programming with Scratch? *2011 Proc. 34th Int'l. Convention MIPRO*, pages 1095–1100, 2011.

[15] A. C. Kay. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conf. History of Programming Languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM.

[16] E. Kazakoff and M. Bers. Programming in a robotics context in the kindergarten classroom: The impact on sequencing skills. *Journal of Educational Multimedia and Hypermedia*, 21(4):371–391, 2012.

[17] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proc. Fourth Int'l. Workshop on Computing Education Research*, ICER '08, pages 101–112, New York, NY, USA, 2008. ACM.

[18] D. Makris, K. Euaggelopoulos, K. Chorianopoulos, and M. N. Giannakos. Could you help me to change the variables? Comparing instruction to encouragement for teaching programming. In *Proc. 8th Workshop in Primary and Secondary Computing Educ. on - WiPSE '13*, pages 79–82, New York, New York, USA, Nov. 2013. ACM Press.

[19] J. Margolis and A. Fisher. *Unlocking the clubhouse: Women in computing*. MIT press, 2003.

[20] O. Meerbaum-Salant, M. Armoni, and M. Ben-Ari. Habits of programming in scratch. In *Proc. 16th annual joint Conf. Innovation and technology in computer science education*, ITiCSE '11, pages 168–172, New York, NY, USA, 2011. ACM.

[21] O. Meerbaum-Salant, M. Armoni, and M. M. Ben-Ari. Learning computer science concepts with scratch. In *Proc. Sixth Int'l. workshop on Computing Education Research*, ICER '10, pages 69–76, New York, NY, USA, 2010. ACM.

[22] U. Mellström. The intersection of gender, race and cultural boundaries, or why is computer science in Malaysia dominated by women? *Social Studies of Science*, 39(6):885–907, 2009.

[23] S. Morra, C. Gobbo, Z. Marini, and R. Sheese. *Cognitive development: neo-Piagetian perspectives*. Psychology Press, 2007.

[24] L. Ni and M. Guzdial. Who AM I? Understanding High School Computer Science Teachers' Professional Identity. In *Proc. 43rd ACM technical symposium on Computer Science Educ., Raleigh, NC, USA*, pages 499–504, 2012.

[25] S. Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, Jan. 1980.

[26] J. Piaget and B. Inhelder. *The psychology of the child*. Basic Books, 1969.

[27] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Others. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[28] C. Riegle-Crumb, C. Moore, and A. Ramos-Wada. Who wants to have a career in science or math? exploring adolescents' future aspirations by gender and race/ethnicity. *Science Educ.*, 95(3):458–476, May 2011.

[29] D. Rushkoff. *Program or be programmed: Ten commands for a digital age*. Or Books, 2010.

[30] L. Seiter and B. Foreman. Modeling the learning progressions of computational thinking of primary grade students. In *Proc. Ninth annual Int'l. ACM Conf. on Computing Educ. Research - ICER '13*, page 59, New York, New York, USA, Aug. 2013. ACM Press.

[31] N. Smith, C. Sutcliffe, and L. Sandvik. Code club: Bringing programming to uk primary schools through scratch. In *Proc. 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 517–522, New York, NY, USA, 2014. ACM.

[32] B. Ward, T. Bell, D. Marghitu, and L. Lambert. Teaching Computer Science Concepts in Scratch and Alice. *The Journal of Computing Sciences in Colleges*, 26(2):173–180, Dec. 2010.

[33] L. Werner, S. Campe, and J. Denner. Children Learning Computer Science Concepts via Alice Game-programming. In *Proc. 43rd ACM Technical Symposium on Computer Science Educ.*, SIGCSE '12, pages 427–432, New York, NY, USA, 2012. ACM.

[34] A. Yadav, C. Mayfield, N. Zhou, S. Hambrusch, and J. T. Korb. Computational Thinking in Elementary and Secondary Teacher Educ. *ACM Trans. Computing Educ.*, 14(1):1–16, Mar. 2014.