Empirical Evidence for Computing Immersion: A Novel Way of Teaching

By Emmanuelle Deaton

ABSTRACT

The author provides historical evidence that pedagogical techniques used in language immersion can provide a pathway to more student participation in computing education and greater retention of those students. Through a review of interdisciplinary literature, the author demonstrates that while some academics during the 1970s and 1980s, including Donald E. Knuth and Francis E. Masat, believed that computing education is as much an art as a science, it ought to be approached first as an art. The author proposes solutions to today's computing education and industry problems that have their basis in French and language immersion and language arts, showing that there is historical evidence that the use of their pedagogical techniques, along with the decentralization of teaching authority from computer science departments, can lead to majority participation of women and increased participation of visible minorities in computing education as well as an excellent, if novice, grasp of a computer language by adults with only a few days of training. As such, it may be easier to train computing teachers than presently believed and the best computing educators may be those trained in teaching language arts and literacy skills. The author examines this possibility by reviewing the historical relationship in computing education between decentralization of teaching authority, immersion and language arts techniques, constructionism, and project-based learning. Evidence is presented that block programming, such as Scratch, is at odds with constructionism, failing to develop requirements-based programming skills in students and posing inherent problems in the development of computational thinking and computational logic. Ultimately, the best computer languages for introductory teaching purposes, and those most aligned with constructionism, may simply be those that are textual with a visual interpretation.

Categories and Subject Descriptors:

• Social and professional topics~History of computing • Social and professional topics~Historical people • Social and professional topics~Computing education • Social and professional topics~Computing education programs • Social and professional topics~Computer science education • Social and professional topics~Computer engineering education • Social and professional topics~Computational science and engineering education • Social and professional topics~Software engineering education • Social and professional topics~Computing literacy • Social and professional topics~K-12 education • Social and professional topics~Adult education • Social and professional topics~Economic impact • Social and professional topics~User characteristics • Social and professional topics~Race and ethnicity • Social and professional topics~Gender • Social and professional topics~Women • Social and professional topics~Age • Social and professional topics~Children • Social and professional topics~Adolescents • Social and professional topics~History of programming languages • Social and professional topics~Computational thinking • Social and professional topics~Informal education • Social and professional topics~Computing and business • Social and professional topics~Employment issues

KEYWORDS

Immersion, French immersion, language immersion, language arts, teaching authority, computer science departments, computer programming, art, science, girls, women, visible minorities, blacks, Hispanics/Latinos, content-based instruction, project-based learning, constructionism, pedagogy, pedagogical techniques, Donald Knuth, Cynthia Solomon, Seymour Papert, Francis E. Masat, Glassboro, Ohio State, computer programming, computational thinking, computational logic, block programming, Scratch

1. INTRODUCTION

> We cannot sustain a society based on innovation unless we have citizens well educated in math, science, and engineering. If we fail at this, we won't be able to compete in the global economy. How strong the country is 20 years from now and how equitable the country is 20 years from now will be largely driven by this issue.
>
> —Bill Gates, *Waiting for Superman, 2010*

This paper addresses two intimately connected issues: a) current instructional methodologies in computer programming, computer science, and software engineering are poorly structured to attract a wide swath of entrants into the field and b) once in the field, the instructional methodologies are particularly poor at preparing learners for building and creating applications in today's economy. Today's world is a world of the 4th Industrial Revolution: a world in which there is rapid movement towards automation and a workplace that needs good software engineers. Given this world, both the top and bottom of the funnel (for training computer programmers) need to be wider. In other words, society writ large needs to bring more people into the field and once in the field, needs to do a much better job training them to be able to meaningfully contribute to the world as computer programmers and software engineers.

A review of the literature provides clues to a potential solution that could handle both issues simultaneously: teach computer programming, computer science, and software engineering more as an art and as a language

with an instructional entry point that has much in common with native or immersion language learning. In other words, make computer programming easily accessible to almost anyone on the planet.

1.1. Background

Although the world's first computer science degree program, the Cambridge Diploma in Computer Science, began at the University of Cambridge Computer Laboratory in 1953, researchers at King's College, London (Wingate 2018), maintain that there is as of yet very little research about how and what to teach in school computing programs (computing being the umbrella term employed to describe everything from keyboarding skills to knowledge of computer science). Henceforth, any reference to Computing Education or Computing in this paper should be understood as a reference to education in, or the fields of, computer science, software engineering, and computer programming.

It is noteworthy that the term computer science is often used in nomenclature as a means of referencing one or all three of the fields, as though they had no distinguishing features. While they are distinct disciplines, an important part of computer science and software engineering is computer programming, otherwise understood to mean knowledge of a computer language that can be used to create computer programs. Through a review of the literature, this paper demonstrates that instructional methodologies in computer science, software engineering, and computer programming are presently failing students, and that while the first is widely discussed and accepted as belonging to the "sciences", it is also aligned with language arts. Consequently, there are parallels to be drawn between the development of skills in computer science, software engineering, computer programming, literacy, and second language development.

While there is research in the field of Computing Education, much work remains to be done so that valid conclusions can be drawn vis-à-vis the pedagogy that ought to be employed at different stages of learning. In contrast, there is sound research and good conclusions about what works in the development of literacy and second language acquisition. Given the parallels that exist between skill development in language arts and computer languages, there is reason to believe that educators should abandon the current instructional paradigm that exists in Computing and adopt instructional methodologies that have been well-studied in language arts and that are considered beneficial to students. In particular, in addition to applying reading and writing strategies to Computing lessons, students would benefit from learning a computer language through content-based instruction immersion (a type of project-based learning) and doing so as soon as they are able to type since the plasticity of young brains facilitates learning languages.

2. METHODOLOGY

Bearing in mind that this paper relies on interdisciplinary studies, literature relevant to it was collected from numerous sources. First, an electronic search for articles pertaining to the successes and failures of core French and French immersion was conducted. Also sought after were articles dealing with matters of equity in French immersion instruction. Second, an electronic search for articles about teaching and learning in computer science and computer programming was done. A more specific search about immersion in computer science followed. Finally, an electronic search dealing with second language acquisition and development was done in order to have a broader context for understanding core and French immersion studies. Electronic searches were done using Google, Google Scholar, and the academic databases JSTOR, ERIC, and ResearchGate. The author's collection of articles and books dealing with child and student development were also consulted.

The author's approach to research was exploratory, hypothesizing that computer programming might have been viewed as a language or an art by other researchers or contributors to the field. In addition, the author hypothesized that if computer programming had previously been viewed in that light, that other researchers would have tried to create an immersion environment for computer programming instruction.

3. THE STATUS QUO IN COMPUTER SCIENCE INSTRUCTION

3.1. Enrollment and Retention Problems

The status quo in Computing instruction is a problematic one. Statistics from various global sources paint an alarming picture, suggesting that there are enrollment and retention problems stemming from instructional methodologies and approaches in the discipline. As a consequence, there are not enough women or visible minorities entering the field. There is also too high a drop-off in enrollment rate for those starting course work.

3.1.1. Girls and Women in Computer Science

In the United States, between 1984 and 2014, the number of women graduating from computer science declined, going from 37% to 18% (Reach Capital 2017, 10). Those statistics must be understood in the broader context of girls' participation in other cognitively

demanding areas of study. The behaviour of young women in Advanced Placement (AP) test-taking in the United States does not support the idea that young women are unable or uninterested in logically rigorous academic studies. Indeed, 55% of AP test-takers overall are girls and girls represent 49% and 52% of the students taking AP tests for Calc AB and Statistics, respectively. In contrast, girls represent only 19% of AP test-takers in Computer Science (Reach Capital 2017, 11). This suggests that something is wrong with the presentation of computer science (more

accurately described as Computing since AP Computer Science is really a programming class) to girls rather than girls being unable to do the work.

3.1.2. Other (Mostly) Problem Statistics

Today, black and Latino/Hispanic people, representing 13.3% and 17.8% of the total American population (U.S. Census Bureau 2016), account for just 8% and 7%, respectively, of those working in Computing (Reach Capital 2017, 8). Online, historic statistics pertaining to their graduation from Computing programs go only as far back as 1991 and were the only ones consulted for this paper. It seems that since 1991, the percentage of black students graduating with Computing degrees in the United States has held steady at about 10-11% while the number of Latino/Hispanic students graduating with Computing degrees has increased from 3.5% to 9% (U.S. Census Bureau 2016). In spite of this, the percentage of black and Latino/Hispanic people actually working in the Computing industry is lower than the percentage graduating from Computing.

Furthermore, while statistics pertaining to drop out rates in post-secondary and K-12 Computing programs are inconsistently available, information coming from Ireland (O'Brien, Humphreys, and McAuliffe 2016) and Norway (Giannakos et al. 2016) respectively suggests that there is regularly a drop-off in student enrollment in computer science programs of 33% to 40%, and that in certain courses in Ireland, the drop-off rate can be as high as 80% (O'Brien, Humphreys, and McAuliffe 2016). In Canada, Ontario's Ministry of Education reports that in the university stream of computer science courses between 2011 and 2016, the drop-off in student enrollment from the Grade 11 to the Grade 12 course was consistently between 52% and 55% (Ministry of Education of Ontario, Course Enrolment). In the community college stream, the drop-off in enrollment was between 78% and 80%. The Ministry of Education of British Columbia reported similar statistics between 2009 and 2013: approximately 63% of students taking Grade 11 computer programming decided not to enroll in the Grade 12 course (Ministry of Education of British Columbia, BC Schools). The global economy needs more workers with Computing backgrounds. This can be accomplished by increasing the size of the funnel for people studying in the field. It can also be accomplished by decreasing the filter effect of the funnel, i.e. changing the approach to education such that more qualified students per capita stay with the field and graduate.

3.2. Computer Science Programs Don't Lead to Industry-Ready Graduates

Other statistics, articles, and sources of evidence pertaining to the status quo of Computing Education give us equal cause for concern. Indeed, on April 20th, 2017, *Mint* (India 2017), the Indian daily business newspaper, ran a headline that said "95% [of] engineers in India unfit for programming jobs: study". The study cited in that newspaper story found that 60% of the 36,000 computer engineering candidates whose skills were assessed could not write code that compiles.

While such a study has not been conducted in Canada, Shopify, a large Canadian business headquartered in Ottawa that relies heavily on "development" (slang for computer programming) and "devs" (meaning, developers or coders who write computer programs), initiated a partnership agreement with Carleton University in Ottawa in 2016. The agreement is for an intensive internship program at Shopify. The students attend university, but engage in on-the-job practice and skill development by doing "full stack development" work (ie: programming in multiple computer languages in order to deliver fully functional web applications). In a 2018 personal conversation with the author, Jean-Michel Lemieux (the SVP Engineering at Shopify) explained that the goal of the internship program is to have a pipeline of industry-ready software engineers; which the company felt it did not have at its disposal prior to commencing the internship program.

A study conducted in Ghana provides further evidence that Computing programs don't lead to industry-ready graduates. Indeed, the authors, Sarpong, Arthur, and Amoako (2013, 27-28) repeatedly assess student proficiency in computer programming at the Institute of Computer Science at Valley View University in Ghana as being weak and confused. Yet, the students at Valley View University are only a few hundred amongst millions of other Computing students having similarly disappointing educational experiences. If the drop-out rate of programs failed to provide complete clarity on the matter, the popularity of books providing guidance and exercises to help software engineers learn how to program is further evidence that Computing programs around the world fail to ready students for the workforce. The book "Learn Python the Hard Way" is one of hundreds of books that teach programming through practice and repetition. Discussions between reputable software engineers about it and other such books abound on Internet forums such as Quora (2016), supporting the contention that Computing programs are failing to teach students in three ways: a) how to write computer programs, b) a computer language, and/or c) how to do development work.

3.3. What We Know From the UK

The United Kingdom instituted a national elementary and secondary school curriculum in Computing Education in 2014. In the years since, other countries, such as Australia, and some provinces and states in Canada and the United States (respectively) have done the same. On November 10, 2017, The Royal Society (2017, 6) published a report calling Computing Education in the UK "patchy and fragile" in no small part due to having a shortage of Computing teachers; a problem that can be remedied with training.

Perhaps unsurprisingly, researchers at King's College, London (Wingate 2018), say that where Computing Education is concerned, there is presently very little research about how and what to teach in schools. They also report that "studies of effective pedagogies in university computing courses have only recently begun to emerge." This is despite the fact that Computing has been an area of academic study in one capacity or another in K-12 schools in the UK, Canada, and the United States since the 1960s.

According to a review of the literature conducted by Jane Waite (2017, 41-53) on behalf of The Royal Society and as an addendum to The Royal Society's computing education report, there seems to be agreement amongst many researchers that having a variety of different instructional methodologies and tools is to the advantage of both students and teachers—though more research is essential. We must remain concerned with finding instructional methodologies that work well for many people given that the task at hand, as evidenced by The Royal Society's report, is not just the education of K-12 students, but also the training of teachers.

4. WHY ENROLLMENT AND RETENTION IN COMPUTER SCIENCE MATTERS

Klaus Schwab (2016, 38) of the World Economic Forum, citing work done at the Oxford Martin School, anticipates that 47% of current jobs will have been destroyed by 2034 and (Cann 2016) that there will be a net loss of over 5 million jobs in 15 developed and emerging economies by 2020 which means that enrollment and retention statistics in Computing programs must be of major concern to us all. The research being done by the World Economic Forum (Cann 2016) suggests strongly that the most important skills for workers in the coming decade are soft skills, Computing skills, and ability in data analysis. The ramifications of declining enrollment of women in the field of computer science, the dismal statistics pertaining to enrollment of visible minorities in the United States, and the atrocious drop-out rates in post-secondary computer science programs must lead to our: a) **resuming** instructional methodologies that lower barriers to entry and retention in the field and, b) further developing some now long-forgotten (or abandoned) instructional approaches (to be explored in more detail below) so that enrollment and retention of women, visible minorities, and people with disabilities is improved.

5. LOW ENROLLMENT, POOR RETENTION—HOW WE GOT HERE

5.1. Computer Programming as an Art

Donald E. Knuth, in his 1974 acceptance speech for the Association of Computing Machinery (ACM) Turing Award (sometimes called the Nobel Prize for computer science) (Brown 2011), spoke eloquently about how art and science complement one another as fields of study. He spoke with some amusement of the historical transformation of computer programming, saying:

> Meanwhile we have actually succeeded in making our discipline a science, and in a remarkably simple way: merely by deciding to call it 'computer science.' Implicit in these remarks is the notion that there is something undesirable about an area of human activity that is classified as an 'art'; it has to be a Science before it has any real stature. On the other hand, I have been working for more than 12 years on a series of books called 'The Art of Computer Programming' (667).

Knuth's remarks are of utmost importance as we seek to understand why there is an exodus of students from computer science programs and why those who graduate from Computing are ill-prepared for jobs in the industry. The question remains: what is happening in Computing Education and how do we solve its problems? As we consider those questions, we must remember that Donald E. Knuth is a distinguished and multi-award winning scholar in the discipline of Computing. He is no intellectual lightweight. His remarks about the discipline and the direction it was taking in 1974 may help us to understand why the field has had ongoing enrollment and retention problems since the mid-1980s.

In his remarks, Knuth (1974) appeals to his colleagues that they should consider computer programming both an art and a science and take pleasure in the dual nature of the discipline. Most of his remarks are devoted to expressing the idea that he and others are motivated to write computer programs because of the creative process. He builds a persuasive argument that by simply writing many short programs an individual becomes an able computer programmer. While he believes that a scientific approach to computer programming has value, he also expresses what seems to be urgent concern that science not overtake the creative learning process—and the creative drive.

5.2. An Accident of History Deliberately Joins Computer Programming to Mathematics

Ultimately, as we consider that computer science departments began to be established in universities across the United States and Canada only in the mid-1960s in the midst of the nuclear arms race, we should linger over Knuth's (1974) observation that "[activity] has to be a Science before it has any real stature" (667). The politics involved in establishing a new university and college department, and the conversations surrounding the chosen title, would not have escaped him.

It is an accident of the Cold War era that computer programming has come to be associated with mathematics. The politics behind the selection of the title

"computer science" for the emerging discipline of Computing and the power of the title's nomenclature are indirectly addressed by a contemporary of Knuth's, Francis E. Masat, who wrote in his 1981 book titled "Computer Literacy in Higher Education" that

> The place of computer literacy in the curriculum depends on whether it is viewed as general education, a basic skill, or just another "math" requirement. In the past, social relevance was used to support the addition of computer science to the mathematics requirement that exists at many major universities today. This rationale, besides assuming that practically anything can be justified on the grounds of social relevance, relies on the popular and problematic assumption that computer science, or even computer literacy, is mathematics. It is not. Although mathematics is useful to someone using a computer, language and logic are more useful (18).

In the above paragraph, Masat reveals his belief that social pressure joined computer science to mathematics. Joining mathematics and Computing together may have been done because of the social association between mathematics and science, thus lending "real stature" (Knuth 1974, 667) to Computing and justifying the creation of computer science departments at universities and colleges. It is also possible that mathematics departments were viewed as the best fit for the discipline given its dependance on numeric representation.

Still, Masat, who was first and foremost a mathematician, contends (1981, 19) that spoken language and logic are more foundational to Computing than is mathematics. (Note: mathematics is a language in the same way that computer languages are—they are all dependent on a spoken language and dependent on that language's culturally accepted logical presentation. Logic is critical to all forms of language.) His assertion that Computing is not mathematics is important and perhaps unusual given that the term "Computing" comes from the discipline's reliance on numbers. He says (1981, 19) that "many in higher education believe that computer literacy and language literacy can be combined since they are fundamental, intellectually similar, and mutually reinforceable forms of communication." Ultimately, Masat is explaining that computer programming (Computing) relies on spoken language communication even if at the machine level it reposes on mathematics. The result, in Masat's view, is that mathematics is not the only field of study that can develop in students the logic needed to do Computing.

Indeed, Masat (like Knuth), is clearly in favour of treating Computing as an art and a language and so his approach to situating it in the curriculum is sophisticated. He denies (1981, 21) that computer science is mathematics, and while he advocates for the study of mathematics for students doing advanced Computing in second, and third programming courses he is clear on the point that having students undertake mathematical studies serves the singular purpose of developing in them the ability to reason sequentially and chronologically.

Masat (1981, 21) is not opposed to the development of these skills by other means, being at pains to distinguish between mathematics, language, and logic since he knows that language, like maths, can develop logical ability. He understands that advanced logic can be developed through the study of mathematics because he is firstly a mathematician. A professor of English literature or political theory could similarly attest to the fact that logic can be developed through reading, writing, and debate. In Masat's case, because computer science was initially introduced to his college by adding it to the mathematics department—perhaps (as Knuth argues) to legitimize the new discipline—Masat was tasked with teaching computer science and his personal means of developing logical aptitude in his students was, naturally, mathematics (Masat 1981, 24). So it would have been with his peers at other colleges and universities.

5.3. Lost Findings from the 1970s and early 1980s

Francis E. Masat's 1982 "Journal of Educational Technology Systems" article, *An Immersion Course in BASIC*, proposed an instructional methodology for computer science (though all activity described therein seems related to computer programming) that should be seen as a full immersion program. It is a program that could potentially be used to successfully teach computer programming and to solve the shortcomings of present-day instructional methodologies. As of this writing, its findings seem to remain unchallenged and don't seem to have been reproduced—at least, not for research purposes. The article reports the results of a study conducted between 1979 and 1981. The study tracked the progress of 103 students at Glassboro State College in New Jersey (renamed Rowan University in 1996) who participated in a standard course in computer programming (referred to as computer science in the article) covering the same content over either a 12 day or 6 week period to learn the computer language BASIC, comparing their academic results to those of 49 students in the regular 16 week semester classes. The study pertained to the activity of 152 students in total (Masat 1982, 327).

According to Grabe and Stoller (1997, 15), where languages are concerned, immersion programs use the second language being studied and a subject of study (eg: history, geography, maths) in order to motivate students in second language learning and to deepen their understanding of the second language (this is content-based instruction immersion). Ultimately, this is project-based learning. It is this procedural paradigm that makes it possible for immersion students to spend their days

thinking and using the subject being studied to develop their linguistic abilities. By doing project-based learning with a subject like history or geography, students have more opportunities to practice speaking a second language, e.g. Spanish, in a real-world context. This model of learning closely approximates the way native language speakers learn to communicate in their native tongue, including both content-learning and language learning activities in order to facilitate linguistic development (Grabe and Stoller 1997, 6-7).

Although short, Masat's 12 day course of study was an immersion program since immersion in a language can be for a short period of time. What matters is the structure of the learning. Yet Masat's results (1982, 327) might be considered surprising since, as will be discussed fully in Section 5.4, he held variables pertaining to evaluation and assessment constant in his courses and the course with the best results (albeit not statistically significant) was also the shortest course (i.e.: the 12 day course).

Unlike today's Computing student demographics, there was a nearly even split between women and men in the classes whose activities and results Masat studied. More striking is the fact that women were a majority in the study, totaling 53% of all participants (1982, 327). That said, Francis E. Masat himself does not seem to believe that this is the norm as he says in his 1981 book "Computer Literacy in Higher Education" that "Minorities, women, and the physically handicapped continue to be underrepresented in the profession" (9). Women participated most in Masat's (1982, 327) 6 week immersion course in BASIC: in that course they were 57% of all participants. Of further interest in the Glassboro study is the fact that none of the participants seem to be majoring in computer science. To the contrary, Masat documents (1982, 327) that 36% of the students were in Management (18% were women), 16% in Accounting (7% were women), 7% in Marketing (5% were women), 11% in Math and Science Education (7% were women), and 30% were listed as majoring in "Bio., Psych., Soc., etc." (16% were women). Clearly, in the early 1980s, Computing—at the very least, computer programming—was still drawing in a wide variety of different people with different skills and backgrounds. While the field of computer science was integrated into the mathematics department at Glassboro, the fact that most students in Masat's study weren't in mathematics or in Computing speaks to the reality that this latter area of study (or— more specifically, and again—computer programming as a subset of computer science), was not yet entrenched as a field considered best suited to mathematicians. In fact, at that time, it was believed that small amounts of computer programming would be done by a variety of different professional classes (Masat 1981, 15-16). Indeed, Masat (1981) wrote that

> Computer literacy courses are not computer science or computer programming per se, although a first course in computer literacy will usually include simple programming experiences. In fact, learning a computer language, if only the rudiments of one as simple as BASIC, prepares one for new and expansive learning experiences (16-17).

A course proposal published, rather startingly, as a journal article in the December 1985 issue of "North American Colleges and Teachers of Agriculture" by John R. Fiske, Marvin T. Batte, and Reed D. Taylor of (what was then) the Agricultural Economics and Rural Sociology Department at Ohio State University attests to their agreement with Francis E. Masat's stance (1985, 6). Their work supports Masat's belief that computer literacy includes knowing a computer language for the purpose of creating computer programs. In their course proposal, the professors write that

> The computer literate individual would not be expected to know computer architecture or how to configure hardware, although he or she should know the functions of each major class of hardware. The computer literate individual would not be expected to have the ability to write any but the simplest algorithms although he should understand what an algorithm is and its importance to data processing. Computer literacy does not imply the ability to design and code efficient, general purpose programs, although it should imply an understanding of when such standard procedures as looping or branching are appropriate (4).

We could read the statement "Computer literacy does not imply the ability to design and code efficient, general purpose programs" (Fiske, Batte, and Taylor 1985, 4) as an assertion that computer literate individuals don't need to learn a computer language or be able to do simple programming. However, this would be a mischaracterization of the authors' intent, which is to align themselves with Francis E. Masat (whom they cite) and to have students in the Agricultural Economics and Rural Sociology Department learn a computer language well enough to write some simple programs. This is made clear by their next assertion:

> In the context of graduates of colleges of agriculture, computer literacy should mean: (1) The ability to identify and understand the functions of the hardware components of a computer system. (2) An understanding of the impact, both current and expected, of computer technology on agriculture. (3) The ability to conceptualize a computer solution to typical agricultural problems such as farm record keeping, feed ration analysis, and budgeting. (4) The ability to write simple computer programs that contain read and write commands and branching and looping procedures (4-5).

Given the above paragraph, and given their explanation that the computer literate individual would not be expected to know computer architecture or hardware configuration, we should surmise that their comments about designing and coding efficient, general purpose programs refer to advanced computer programming skills. Indeed, they also write that "The ability to write algorithms to solve simple problems and to do elementary programming in a language such as BASIC or Pascal is consistent with the characteristics of the computer literate student" (1985, 6). In other words: everyone should be comfortable with some simple computer programming in the same way that everyone should be comfortable with reading the newspaper—while not everyone needs to be able to write a novel.

5.4. Computing Immersion: More About the Glassboro Study

Francis E. Masat's idea of Computing immersion was to have an accelerated learning experience by requiring that students engage in both problem-solving and project creation. The class environment he described was busy, engaging, and productive:

> Generally, the first part of each class is question and answer, followed by student chalkboard work of instructor generated and text exercises. Many times, as many as nine to ten students will be producing programs at the boards. This provides experience, alternate views, and methods for programming the same problem, and facility in reading and correcting programs. . . many times a student takes over the computer keyboard to make the changes or additions suggested by the instructor or class. The exchange between computer, student, and instructor becomes dynamic, stimulating, and exciting (1982, 323).

What is most interesting about the scenario is that all this activity occurred in an introductory computer science course that Masat says "was designed to contain more programming content, more hands on experience, and academic measures of achievement" (1982, 322). Presumably, the students had had no previous exposure to Computing. Yet their lack of knowledge does not seem to have held them back. Masat's immersion approach required collaboration and an ongoing exchange of ideas. It also clearly demanded that students engage in a serious amount of trial and error (or cause and effect) programming. In fact, the students were encouraged to spend 3.5 hours a day in the computer lab in order to create programs and learn how to debug with the assistance of a student lab supervisor. In addition to which, the students needed to formally turn in six computer programs (projects) every two days and a dozen daily homework problems (1982, 323-324).

This Computing immersion program, which was reported on in Masat's Glassboro study (1982), was instituted between 1979 and 1981 "As the College's introductory computer course was over-subscribed (an understatement)" (321). Put differently, Masat started the different length courses as a means of meeting the enormous student demand for introductory computer science. Without making use of intersessional weeks, Glassboro simply could not meet the student demand for that course. In an attempt to meet student demand, the aforementioned 12 day and 6 week immersion programs were established and student progress was tracked. The objective was to teach the same curriculum in an accelerated manner by focusing first and foremost on having students practice programming skills. Clearly articulated at numerous points throughout the study is the goal that the integrity of academic standards be maintained in spite of the short length of the program of study. In the end, students who registered in Glassboro's immersive 12 day and 6 week program did as well as, or better than, their peers registered in the regular 16 week long semester classes. In every case, including the 12 day immersion program, grades hovered around the 81% mark but, of the groups examined, the 12 day immersion group of 1980 did best: the average grade was 85%. Overall, the 12 day immersion group had the best results with a combined average (mean) of 83% compared to 81% in the 6 week immersion group and 81% in the regular 16 week semester class group. The results are meaningful because, as Masat explains, in order to maintain the academic integrity of the courses of study and of the research being done,

> Each twelve-day and six-week class was told at the beginning of the respective course that only the time frame was different: content, programming, homework, and testing were to be considered to be the same as that offered in a semester course. As a match, the same homework problems, exams, and programs were given to two sixteen week semester classes (different semesters), and two six-week summer session classes, 1979 and 1980. All the exams and programs for the courses were graded by the instructor using the same criteria and scales (1982, 326).

In 1985, Fiske, Batte, and Taylor of Ohio State University's Department of Agricultural Economics and Rural Sociology not only published the aforementioned course proposal (Section 5.3 above) detailing their belief that students could successfully study computer programming within non-computer science departments, they also co-authored an empirical study based on the activity of 172 students who learned computer programming through a course in their department titled "Agricultural Economics 250" (AE 250). This empirical

study was published in the journal "North Central Journal of Agricultural Economics".

The study tracked student performance in three areas of learning including computer literacy—the term used by the researchers to mean competency in computer programming (in this case, with the computer language BASIC)—giving a separate percentage grade for all three areas. Student work and activity was similar to that described by Francis E. Masat. Of the 172 course participants, only 29 had previously had any exposure to computer programming (127). The computer literacy (computer programming) average for students with prior exposure to computer programming was 84.23% while the average for those 143 students novice to computer programming was 84.57% (126). Of the participants, 66% were male while 34% were female. 28% were from a commercial farm while an additional 22% came from rural areas other than commercial farms. The rest of the students came from small towns and cities (121). This empirical study is important as it echoes Francis E. Masat's findings with his own non-computer science cohort of students. It also provides evidence to substantiate Fiske, Batte, and Taylor's claim that students can successfully be taught computer programming outside computer science departments and by professors whose own expertise lies elsewhere. Moreover, it serves to illustrate that by centralizing teaching authority for computer programming to computer science departments, academics may have inadvertently decreased the number of women learning computer programming and later on participating in the Computing industry.

6. WHAT WENT WRONG IN COMPUTING EDUCATION?

Based on the Glassboro study, it would seem that there existed between 1979 and 1981 a program of study that appealed (a bit) more to women than to men, that facilitated the development of excellent computer programming skills, and which was accessible to all and even primarily, to non-maths majors. Based on the Ohio State Study, we also know that many women in agricultural economics and rural sociology were also interested in computer programming and that students in AE 250 had excellent results learning computer programming outside Ohio State's Department of Computer Science. So why don't such teaching and learning opportunities exist at today's post-secondary institutions?

6.1. Historic Limitations in Communication and Academic Pressures

We can answer the above questions through inference. The Glassboro study and its significance most likely failed to take root because the era of publication was pre-internet and pre-email, therefore impeding the flow of information and discourse. It is also possible, given the academic interest in making computer programming into a science (as explained by Donald E. Knuth), that academics in the field were most focused on dealing with transformative, abstract, ideas in computational thinking and computational logic—at the expense of understanding the art of teaching computer programming. That is to say, historically, the pedagogy of Computing Education has not been of central importance to the growth of computer science as a field.

Indeed, as Knuth (1974, 673-674) indicated in his ACM Turing Award lecture, academics at the time were particularly interested in the development of new computer languages, operating systems, and programming efficiency (e.g.: computer architecture). It is possible that even in introductory courses to Computing, the academics of the era wound up focusing on what they themselves were most interested in: computer architecture, data, semantics, syntax; the theory that would help make computer programming into a *science*. In the process, they may have inadvertently set a course for student exodus of the field.

Impossible to overlook as a driver for women's exodus (specifically) from Computing is the possibility that women's ability to access introductory computer science courses was cut off or dramatically reduced when teaching authority for Computing was restricted to computer science departments in and around 1985. The fact that Francis E. Masat's (1982, 327) students seem to have all majored in areas other than computer science and to have been successful in a very hands-on, non-theoretical, computer programming course lends credibility to the position held by Fiske, Batte, and Taylor: that almost anyone from any discipline can teach and learn computer programming when theory is not the focal point.

This is the position that they present in their 1985 course proposal and which they substantiate with their empirical study. In the proposal they explain that there is tension between professors who want to emphasize the scientific part of Computing and professors such as themselves who think that without a focus on theory, computer programming instruction can be integrated into other subject areas such as biology or agriculture (1985, 5). That Fiske, Batte, and Taylor published their course proposal in a journal serves to illustrate the depth of their feeling that teaching authority for computer literacy—including computer programming—should not be centralized in, or restricted to, computer science departments. We can surmise that they were fighting an academic and pedagogical turf war.

In their course proposal, they quote their colleague, Dr. Bruce W. Weide, a professor of computer science. They cite his words from the March 1985 issue of *The Chronicle of Higher Education* where he says that "'There are good academic reasons why computer science ought to be taught by computer scientists. There is some theory about computing, some intellectual content to the science'" (5). Importantly, Donald E. Knuth's explanation of the

transformation of computer programming into a "science" and the way in which Fiske, Batte, and Taylor structure their arguments, lead to the conclusion that while Weide says "computer science" he is also thinking of computer programming: certainly, Fiske, Batte, and Taylor believe that to be the case. Fiske, Batte, and Taylor, as well as another Ohio State colleague, Dr. Russel V. Skavaril, a professor of genetics, disagree with Weide's assessment, arguing in the Ohio State course proposal of 1985 that they should have the right to teach computer literacy, including computer programming, in their own departments (5). Unlike Weide, they were uninterested in teaching Computing theory and did not conflate computer programming with computer science.

When seen as part of a timeline of important events and insights, Donald E. Knuth's 1974 comments about the transformation of computer programming into a *science* help shed light on the development of problems in Computing and Computing Education: we go from 1974 comments about the fact that computer programming is partially becoming a science (while also remaining an art), to departmental and pedagogical turf wars in the mid-1980s about who should be able to teach computer programming, to the knowledge, in 2018, that while computer *scientists* have been able to centralize (or restrict) teaching authority of computer programming, they have failed, to some degree, to be effective in Computing Education since there are now serious enrollment and retention problems in Computing programs and most students graduating from a Computing degree are unable to code.

Academics like Donald E. Knuth and Francis E. Masat spoke eloquently about the relationship between computer programming and language, proposing that strong logic and linguistic ability along with ongoing practice (Knuth, 672) develop computer programming skills. However, it is clear that they had colleagues who thought that theory was far more important. In fact, Donald E. Knuth references colleagues working in artificial intelligence who, already in 1974, perceived computer programming as an artifact: little more than a relic that would soon be taken over by machines (1974, 669). Given that view of computer programming and the integral role of computer programming within computer science, it is perhaps unsurprising that little attention has been given to the pedagogy of Computing Education. Indeed, since, as Donald E. Knuth (1974, 669-672) explains, the *science* of computing was new in the 1970s, it is to be expected that it continues to go through an extraordinary growth phase even today. As a result, while the teaching authority for Computing lies squarely in the hands of computer scientists, there has been little impetus to pay attention to successful experiments in teaching computer programming (such as Francis E. Masat's) and even less reason to try repeating them: the scale and rapidity of change have been the focal points in Computing.

6.2. Industry Pressures

The immense pressures of industry have also contributed to the failures of Computing Education. Beyond an evidenced desire from academia to see computer programming made into a *science*, lies the fact that in Computing in the 1970s and 80s, industry needed to build operating systems in order to accelerate and streamline data processing. This is substantiated by some of Masat's observations in the Glassboro immersion study. Toward the end of the Glassboro article, Masat writes (1982, 328) that

> The twelve-day design compares favorably to that used by Colorado College and several industrial firms. However, in terms of course content and emphasis, the Glassboro State twelve-day course appears to differ significantly from the more data processing-oriented course offered by Colorado College and information science companies.

With this statement, Masat shows his awareness that Colorado College and information science companies were focused more on data processing-oriented courses and that they too were using an immersion approach to teaching Computing. Thus, Masat is aware in 1981 that while industry employed immersion approaches in teaching, the curricular content and emphasis at Glassboro differed significantly from that of information science companies and at least one other academic institution: Colorado College. Whereas his course content was focused on computer programming, Colorado College and private industry were interested in the issues of retrieving, transforming, and classifying information—in other words: computer science.

The genuine shape and structure of Colorado College's and industry's immersion programs is presently unknown, as are the type and rate of success they experienced. Yet, we do know that they were focused more on data processing—and consequently, computer science and, with it, advanced mathematics. Given that this is the predominant type of instruction that exists in computer science programs today, we can infer that industry pressure had a significant impact on the creation of a program of study that is predominantly designed for students who excel at advanced mathematics. Yet Francis E. Masat (1981), for whom neither computer science nor computer literacy—including computer programming—repose on mathematics (19), had a different pedagogical approach: he introduced to students some small amount of fact-based learning of a computer language (BASIC) followed immediately by large amounts of practice: albeit over as little as 12 days (1982, 322).

We must consider that Francis E. Masat's hands-on approach to learning computer programming went unadopted by computer science departments because they were focused on teaching skills required to build operating systems and to create new computer languages. The

development of those skills at the college and university level is far less hands-on and based far more on learning and understanding abstract ideas. We can speculate that this approach to teaching Computing was then brought to K-12 schools by teachers who graduated from this type of information science, data processing-oriented computer science degree. More realistically, K-12 Computing teachers have probably taken but one or two computer science courses.

Moreover, the rapid evolution of computer languages and their jockeying for dominance will have also negatively impacted the instruction of computer programming as classroom teachers will not have had any of the resources (of time, material, or professional development) required to keep up or to understand what and how to teach. As Masat (1981) writes, "The potential for change in curricular development is enormous" (44).

The alignment of Colorado College's data processing-oriented course with the activities of information science companies speaks to the impact of industry on the work of academics. To be sure, there is a connection between industry's need for better data processing and the academic focus on computer science and its development: building operating systems for use in data processing requires significant ability in abstract thought, advanced mathematics, and physics.

6.3. The Association of Mathematics with Computing Changed the Instructional Approach

In coming to terms with why Francis E. Masat's successful pedagogical approaches weren't widely adopted we've dealt with three factors: historic limitations in communicating, academic pressures, and industry pressures. In the end, the latter two contributing factors created a close association between mathematics and computing that Masat himself rejected.

Masat distinguished clearly between logic, mathematics, and languages in order to facilitate understanding what is possible in computer programming instruction. While Masat was a mathematician, he turned to immersion—an instructional methodology used for the development of second languages—as a means of quickly developing computer programming skills in his students because he viewed programming as an *art*, both practical and hands-on. The instructional approaches used to teach computer programming appear to be **significantly** impacted by whether or not we accept and adopt the belief that computer programming isn't the same thing as mathematics—Masat's clearly stated position. In 1981, he wrote "Although mathematics is useful to someone using a computer, language and logic are more useful" (8). Indeed, in a post-1985 world where computing is entrenched as a science, novices to computer programming (in high school and post-secondary computer science classes) are expected to learn what amounts to be the theory of computer programming before they ever get to experience the act. In fact, they take classes where the content favours students who are already expected to function at a high level of mathematical abstraction; no surprise given how industry shaped computer science. Knuth and Masat, on the other hand, having approached computer programming as an art, were clearly advocating for *creating* in the discipline rather than focusing on theory or *science*. They understood that by writing small computer programs, students develop awareness and knowledge of a computer language in much the same way they do a spoken language. Indeed, Masat (1981) states that

> The task of programming a computer becomes a linguistic one: analysis, synthesis, semantics, logic, sequential reasoning, and punctuation. Computers either understand you or they do not. Cause and effect take on dynamic and immediate meaning, what you do makes a difference. Clarity and precision are necessary when you are communicating with a computer; rigid adherence to syntax is the rule. In fact, some authors claim that a person's experience with computers will transfer to his or her use of grammatical rules. Thus, computer programming, and computer literacy in general, is not a hallowed area reserved only for scientists or mathematicians. It may benefit anyone capable of learning it (17).

In assessing the reasons why computer science is today taught the way it is, we also need to remember that historical and social forces associated computer programming with mathematics and deemed computing a *science* in order to give it stature. Given the comments made by Knuth and Masat, we have reason to believe that the deliberate association of computer programming with mathematics, which was viewed as an elite field of study due to its close rapport to the nuclear arms race (Dean, 2007), served to transform the art of computer programming into a *science*—with a focus on abstract theory. In the process, computer science was legitimized as a field of study. The more computer science relied on mathematics as the basis for work—in data processing and developing operating systems—the more it came to be associated (conflated) with maths. Conversations about the relationship between language, logic, and computer programming stopped because they became irrelevant to a discipline now disproportionately associated with theory and mathematics. The Ohio State battle for integration of computer programming into separate courses of study showcases how the discipline came to be housed in computer science departments: how studying abstract theory won the battle against studying practical application. The result of tying the study of computer science to abstract mathematics has resulted in both a smaller number of students entering the field, and also, in

an ongoing focus on abstract theory, taking away a path that is more useful to the outside world: learning computer programming for practical and project-based work.

7. INSTRUCTIONAL METHODOLOGIES THAT DEVELOP(ED) COMPUTER PROGRAMMING ABILITIES

7.1. Constructionism

Interestingly, Knuth and Masat's defence of computer programming as an art form, an activity, and a skill set accessible to all is aligned with the constructionist theory of learning developed by Seymour Papert at MIT. The constructionist theory of learning holds that students learn best when they are able to use knowledge so as to construct—that is, to create for authentic purposes (Papert, 1986). As such, it is aligned with language immersion practices that give students some exposure to facts, but encourage them to use these facts in a creative and hands-on capacity. In doing so, the students inadvertently practice skills. This is what Papert (1986) calls "learning without curriculum" (30).

Papert, along with Cynthia Solomon and Wally Feurzeig, created the first computer language deliberately designed to teach programming to children: Logo (Solomon, 2018). The team conceptualized the idea in 1966 and had the language ready by 1967. Field work was done and the language subsequently totally redesigned. Between 1968 and 1969, Papert and Solomon used it to teach a class of Grade 7 students; turtles, robots that could be programmed using Logo, were added at the end of that year. According to The Royal Society's report on the state of Computing Education in the United Kingdom (2017, 26), 17% of primary teachers responding to their survey said that they used Logo to teach Computing while 5% of secondary teachers said they used it (28). Within the report itself, Logo is identified as a block-programming "language" (26), but this is false. Logo allows for the full expression of computational thought and logic and uses syntax and vocabulary, whereas block-programming does not. Cynthia Solomon has explained (2018) that Logo was created to provide visual outputs to help students develop abstraction. This was done on the understanding that there is usually a relationship between age and an individual's ability in abstraction—a relationship currently being studied in the hopes of mapping what researchers are calling "levels of abstraction" in computational thinking (Waite et al., 2018; Waite 2017, 89).

While there is no specific mention of Logo in Masat's writings, he is clearly familiar with Papert's work in constructionism and with children as he discusses Papert's 1980 book "Mindstorms, Children, Computers, and Powerful Ideas" in his own educational treatise (1981), saying that

> Seymour Papert of M.I.T. conducted computer-learning experiments with elementary school children and found they could use the computer to solve complex problems in physics, geometry, and physiology and that they also were capable of generating music and poetry (15).

The above excerpt must be read with the knowledge that in her first-hand account about the development of Logo (hosted on the now defunct Wikispaces), Cynthia Solomon (2018) says that the work she, Papert, and Feurzeig did in the late 1960s was the foundation for Papert's 1980 "Mindstorms" book, already referenced above. Since Masat is familiar with the book, and it thoroughly discusses Logo and how children were learning computer programming with it, it is curious that Masat doesn't talk about it himself. The inference here is that Masat—like Knuth, Papert, Solomon, and Feurzeig—is in favour of constructionism, but is not yet prepared to draw conclusions about which computer language(s) should be used to teach children. He does say that there is a "need to consider a computer literacy curriculum that spans elementary through college levels . . . national in scope [and in the process] criteria need to be developed for each level" (1981, 16). He also says that "there is no consensus on precisely what constitutes a basic course in computer science, nor in computer literacy" (1981, 16).

In computer programming, the founding principle of constructionism—furthering knowledge and understanding by creating something meaningful, recognizable, and based in the real world (Papert, 1986)—is best achieved by having students apply their knowledge in a creative capacity as quickly and as much as possible. Indeed, Knuth says "When we teach programming nowadays, it is a curious fact that we rarely capture the heart of a student for computer science until he has taken a course which allows 'hands on' experience with a minicomputer" (1974, 671). This position is echoed in the research presented by Sarpong, Arthur, and Amoako of Valley View University in Ghana. Their research (2013, 30) found that 88% of the students surveyed agreed that writing programs and applying concepts learned from their teachers was the best way of learning computer programming. In addition, 74% of the students surveyed said that the second best way of learning computer programming was to complete lots of projects. According to the article, the students believe that completing lots of projects "enhances their understanding of concepts and sharpens their skills in the course" (30). This opinion is shared by many professional computer programmers (Quora, 2016).

7.2. Immersion

Masat's Computing immersion study at Glassboro came at a time when immersion, which can be considered concurrently as an approach, a framework, and a methodology (Stryker and Leaver 1997, 5), was being

heralded as a very successful means of developing second language abilities in young and old students alike (Grabe and Stoller 1997, 6). Spoken immersion was first developed in Canada during the 1960s as a means of teaching English to native French speakers in Quebec (Paikin, 2016). It was subsequently introduced in Ontario during the 1970s and to other Canadian provinces and territories in order to teach French to speakers of other native tongues, including English (Paikin, 2016). Extensive studies on the success of French immersion (as it is known in Canada) and the failures of core French (in Canada, the more common form of French as a Second Language instruction) affirm that French immersion is a successful means of developing fluency in all students (Cummins, 2014) particularly if explicit language learning activities are used to support content-learning activities in the classroom (Grabe and Stoller 1997, 6).

French immersion has its roots in an instructional approach called content-based instruction (CBI), that Stryker and Leaver (1997) say

> ...can be at once a philosophical orientation, a methodological system, a syllabus design for a single course, or a framework for an entire program of instruction. CBI implies the total integration of language learning and content learning. It represents a significant departure from traditional foreign language teaching methods in that language proficiency is achieved by shifting the focus of instruction from the learning of language *per se* to the learning of language through the study of subject matter (5).

In other words, the more the learning environment facilitates practicing the language in real-life scenarios, the more CBI is successful. Needing to discuss geography in French—either in a geography class or in a social setting—is an example of a real-life opportunity to practice speaking French (Stryker and Leaver 1997, 288-290). This methodology is so successful at developing second language proficiency that it has been the methodology of choice employed by the US Department of State's Foreign Service Institute (Stryker and Leaver 1997, 31-33). Additionally, if the success of a methodology is determined by how many people are positively impacted, then we should also bear in mind that a review of the literature published in 2007 in *The Canadian Modern Language Review* references a body of work dating back to the 1970s that demonstrates that, "below-average students in early immersion scored just as well as average and above-average early-immersion students on speaking and listening tests" (Genesee 2007, 659).

Francis E. Masat's Glassboro study is one of just two deliberately designed immersion studies in Computing that were discovered in the research process for this paper, although some of the instructional strategies that he employs are also used in non-immersion settings (Waite 2017, 8). The other deliberately designed Computing immersion study, discovered through research, was undertaken by Miguel Velez-Rubio. His study formed the foundation of his doctoral dissertation (2013). Unfortunately, the dissertation fails to share some much-needed information, notably, how many students participated by sex and how many of those students were part of a visible minority group. Velez-Rubio does report that all of the participants were first year computer science majors, approximately ⅓ of whom dropped the course (2013, 128). In Masat's (1982) immersion study, the female-male split was 53% to 47% while there was "minority enrollment" of 17%, 17.4%, and 15.3% in the 12 day, 6 week, and 16 week programs, respectively (326). Here too, it would be helpful to have more accurate information about the "minority enrollment". Masat does not report on drop-out at all, but the reason for establishing the immersion programs was to deal with the overwhelming popularity of the computer programming course (321).

Most of the Glassboro study findings have already been reported, but not yet compared to the K-12 French immersion instructional setting and instructional approaches. Such a comparison is absolutely vital if we accept, as Masat and Knuth do, that computer programming is ultimately a literacy skill and, as such, that it should be developed at a young age in order to take advantage of the plasticity of young students' brains (Eliot 1999, 364). Only time and research will tell us if adopting such an approach helps to improve enrollment and retention rates in post-secondary Computing programs, but there is good evidence in language arts studies showing that early literacy development is beneficial to students (Bakken, Brown, and Downing 2007, 265-268; Jones, Reutzel, and Fargo 2010, 334-338).

The Glassboro study lends itself well to comparison with K-12 French immersion programs because of the cross-section of students majoring in different subject areas (Masat 1982, 327), the fact that the 12 day and 6 week immersion groups had classes every day (323-325) in the same way that students in French immersion programs have French immersion classes daily, and because of the detailed account of type and quantity of work produced (323-325) which paints a clear picture of the pattern "learn facts, then practice by creating a lot": hallmarks of instruction both in French immersion and in constructionism (Lapkin et al. 2009, 10; Papert 1986, 6).

The study conducted by Velez-Rubio, on the other hand, does not give the impression of being an immersion program that aligns itself well with what happens in K-12 classrooms—for three reasons. First, all of the participants in his study were computer science majors (Velez-Rubio 2013, 2), which is not reflective of the diverse cross-section of students in a French immersion program. Second, Velez-Rubio does not report on whether or not students had ongoing daily exposure to computer programming: in K-12 settings, daily practice is an element that frequently is a part of spoken immersion programs. Note however,

that in general, linguistic immersion is defined by environment and type of activity—which tends to be constructionist (i.e.: project-based; content-based instruction) and to be bolstered by explicit language activities—rather than by length so that outside the K-12 environment, immersion could happen once a week (or more often) for ongoing weeks (Stryker and Leaver 1997, 190). Third, Velez-Rubio insufficiently reports on the type and quantity of work given to students and on the deadlines, which makes it impossible to know if the work was project-based (i.e.: content-based instruction; constructionist) and to have a genuine understanding of the extent to which students were successful. Indeed, a full assessment of pedagogical value cannot be provided when there is no clarity on the types of assignments given to students, their number, or the amount of time given for completion.

Thus, for the purpose of his dissertation, Velez-Rubio created a learning experience that he deemed immersion, but failed to clearly illustrate why it should be considered immersion. Unfortunately, as few demographic details were included, the participants were all computer science majors, and there is a dearth of details about how learning was done, what was built or created that was new, and how students were assessed, Velez-Rubio's dissertation cannot be said to contribute in a meaningful way to our understanding of what is possible in Computing Education and specifically, in immersion. For similar reasons, the study that inspired Velez-Rubio's dissertation, "Immersion language theory meets CS" (Harper 2006, 85-91), fails to find an alignment with K-12 French immersion programs although Harper (2006) does attest to the fact that using principles of language instruction allowed him to better understand his students' needs (88) and says that using the principles of language immersion created a richer Computing Education experience for his students (90).

7.2.1. Reading Strategies for Literacy Development

An important part of learning any language is the ability to read, but in computer programming this is especially true since the languages used to program are written and, barring special accommodations, never spoken. Masat's Glassboro study is interesting because it produced excellent results in an accelerated period of time, but also because he gives us an excellent sense of the environment in which students learned and the methodologies used to teach them. Masat's study is important for three reasons: first, his study helps to define what a Computing immersion program includes from an instructional standpoint, second, he shows that immersion achieves excellent results, and third, he shows that the same results can be achieved in either 12 days or 6 weeks. While Harper and Velez-Rubio's work do not have much in common with K-12 French immersion programs they, along with Masat, did use some of the same reading strategies that school teachers use to develop literacy skills in their students.

According to the documents "A Guide to Effective Instruction in Reading, Kindergarten to Grade 3" , "A Guide to Effective Literacy Instruction, Grades 4-6", and "Think Literacy: Cross-Curricular Approaches, Grades 7-12", all published by the Ministry of Education of Ontario, reading strategies that should be used and taught by school teachers to develop student literacy skills include: previewing a text, analyzing the features of a text, finding organizational patterns, using an anticipation guide, finding signal words, using context to find meaning, making inferences, summarizing, questioning, predicting, synthesizing, sorting ideas, using a concept map, visualizing, making notes, drawing conclusions, making judgements, guided reading, shared reading (which includes paired reading), and independent reading (2003; 2006; n.d.).

Variations of these reading strategies were variously used by Masat, Harper, and Velez-Rubio. Masat testifies to the success of these strategies when he writes that

> The second half of the class session usually is devoted to new concepts and material. Examples are demonstrated on one of the TRS-80 microcomputers that is connected to two large TV monitors in the classroom. **[previewing a text]** The concept of linking computing concepts and BASIC commands to a visual demonstration has been effective and efficient beyond the author's original expectations **[analyzing the features of a text; finding organizational patterns; guided reading; shared reading]**. Students are able to see, hear, and use new commands and processes immediately. Moreover, the technique allows students to amend and change the programs generated **[drawing conclusions; making judgements; independent reading; synthesizing; sorting ideas]** (1982, 323).

Harper (2006) and Velez-Rubio (2013) used similar instructional methodologies with their students (Harper 2006, 86-87, 89; Velez-Rubio 2013, 19, 58-59). For them, these instructional methodologies seem to be the foundations of immersion, whereas they are in fact reading and writing strategies that can be employed in both immersion and non-immersion settings.

7.2.2. Writing to Develop Literacy Skills

A joint position statement issued by the *International Reading Association* and the *National Association for the Education of Young Children* (1999) says that "writing challenges children to actively think about print. As young authors struggle to express themselves, they come to grips with different forms, syntactic patterns, and themes" (7). Study after study corroborates the position (Bakken, Brown, and Downing, 2017; Hall et al., 2015; Jones, Reutzel, and Fargo 2010, 334-338).

The Glassboro immersion study, which relied on now well-researched and well-endorsed reading strategies to help students learn at an accelerated pace, also relied on some writing strategies to further develop literacy skills—notably, group and shared writing (Stahl, 2014). Indeed, this is what was happening when students "[took] over the computer keyboard to make the changes or additions suggested by the instructor or class" (Masat 1982, 323). Furthermore, in the Glassboro study, both shared and independent writing opportunities abounded in the daily computer lab sessions that students were expected to engage in. Both in the class setting and in the lab, the students seem to have had the freedom to work alone, in pairs, or in groups. In both settings, students learned that "the power of writing is expressing one's own ideas in ways that can be understood by others" (National Association for the Education of Young Children 1998, 7; Masat 1982, 323). The fact that Masat had his immersion students doing computer lab projects daily—where they were reading and writing—mirrors educational recommendations that students engage in daily reading and writing activities in order to develop literacy skills (Ministry of Education of Ontario, 2003; 2006; n.d.; Jones, Reutzel, and Fargo 2010, 334-338; Stahl 2015, 263-265). Indeed, literacy and fluency seem to depend on immersion.

Given that educational research shows that forms, syntactic patterns, and themes are developed by writing, it should perhaps come as no surprise that amongst aspiring computer programmers, one of the most popular means of learning the art of computer programming is an exercise known as Type-What-You-See. This sort of exercise has long been the go-to both for independent, autodidacts learning computer languages and people who simply need to learn something quickly:

> Auriel Fournier had no choice but to learn programming. The ecology PhD student wanted to use a complex set of calculations to estimate migratory populations from field observations, and doing so efficiently required a software package that ran in the programming language R. Her principal investigator (PI) did not know the language. Neither did anyone else in her lab at the University of Arkansas in Fayetteville. "My PI said, 'Figure it out'," says Fournier. She began googling online tutorials, mastered the package and now helps other researchers to make sense of R and similar tools (Baker 2017, 563).

Similar stories are reported by many Computing students and people who are good programmers who indicate that they've learned computer languages independently and through repetitive exercises because they needed to figure it out (Quora, 2016).

Type-What-You-See was introduced to the general public in a popular instructional tome: "Learn Python the Hard Way". The title is the author's tongue in cheek way of saying that learning a computer language happens by doing, notably by reading and re-typing lines of code in order to develop understanding of forms, syntactic patterns, and themes (Shaw, 2015).

Accordingly, proficiency in computational thinking and computational logic is developed by creating projects through a trial and error process. This echoes "Research [which] indicates that seeing a word in print, imagining how it is spelled, and copying new words is an effective way of acquiring spellings" (National Association for the the Education of Young Children 1998, 7). Consequently, it may be true that reading followed by writing helps in the development of literacy in a computer language just as much as in a spoken language.

8. SOME CONSIDERATIONS: FORMS TEACHING, IMMERSION/CONSTRUCTIONISM, & INTENSITY

Given the evidence presented in the previous sections of this paper, it seems that there is a strong case for treating computer programming as an art rather than a science. When Masat did so, he used instructional methodologies that are commonly used in language arts today—and with great results. Yet, many of the instructional practices that are used in Computing programs today are reminiscent of *forms* (or *focus on forms*) teaching which is "discrete-point grammar teaching [. . .] in which classes spend most of their time working on isolated linguistic structures in a sequence predetermined externally by a syllabus or textbook writer" (Long 2000, 179).

Harper (2006, 86) and Velez-Rubio's (2013, 6) reasons for experimenting with pedagogical techniques in Computing Education are absolutely reminiscent of this. A *focus on forms* style of teaching also seems to typify the instructional practices at Valley View University (Sarpong, Arthur, and Amoako 2013, 31). In a 2009 analysis by Sheard et al. about the teaching and learning of computer programming, the use of K-12 reading and writing strategies such as paired reading and writing are considered a novelty in Computing instruction (99-100). Jane Waite's work (2017) suggests that this still holds true (37-39).

According to Wu Yakun (2006) of Liaoning University, he and other professors

> Usually [begin their] lectures with the introduction of the syntax of a particular programming construct. Then, it is demonstrated in isolation and later incorporated into a larger program that solves a particular problem. Students are able to understand the construct in isolation and recognise it in the sample program but

> are unable to transfer this knowledge to their own programming (64).

Yakun's description of how Computing instruction occurs at his university accords with aforementioned accounts. It also parallels what used to happen in core French classes in Ontario: direct and teacher-centred instruction, shows of understanding in isolated contexts, subsequently followed by a failure to transfer skills to broader and more meaningful communication contexts (Lapkin, Mady, and Arnott 2009, 19-22). Indeed, unlike French immersion, we know that the traditional *forms focus* methodology (Long 2000, 182) of core French (in which students focus on exercises that teach grammatical structure) has been unsuccessful in developing second language abilities in students because in the past: a) only 3% of Ontario students who began the program in Grade 4 stayed with it until the end of Grade 12, signalling a retention problem, and b) of those students, most graduated with little ability to speak or understand the language, signalling a failure to achieve the objective—fluency in a second language—even after 9 years of French lessons (Cummins, 2014).

In order to move away from a *forms focus* methodology in core French, the Province of Ontario adopted a version of the Common European Framework of Reference for Languages (CEFR) in 2013. A report commissioned by the province in 2017 shows that the CEFR approach is methodologically related to French immersion as its focus is on speaking French for "authentic, everyday uses" (Rehner 2017, 23) and that after training, teachers are moving away from the old paradigm (Rehner 2017, 23). Indeed, in language arts, teaching methodologies and environments that facilitate student-centred learning and the use of language for authentic communication are constructionist, or project-based, because by their very nature, they encourage student creation. In fact, Papert (1986) uses speaking French while visiting France as an example of constructionism (6). Furthermore, in a review of the literature (2009) dealing with core French teaching and learning, Lapkin, Mady, and Arnott say that studies of core French students who have spent an intensive amount of time in a French learning environment (which they define as existing when 60 to 75 percent of instruction occurs in French), have shown that this environment "allows 'a language arts approach to teaching FSL'" (17) by which they mean that there is greater focus on "communication (oral and written), literacy, interaction with others, and project-based pedagogical principles" (17). This, of course, has clear echoes of the Glassboro study both in design and result.

9. FUTURE STUDIES: BLOCK PROGRAMMING

The consensus in the field of linguistics is that because synapses close off as the brain matures, the younger individuals are when they begin learning a language, the better (Eliot 1999, 364). In light of that information, it may be advisable to start learning computer languages at a young age. Today, students who are not yet able to type can begin learning some of the principles of computational thinking and computational logic with block based programming languages such as Scratch. While the advantages of this early exposure may parallel the advantages of early literacy experienced in spoken languages, there may also be unfortunate problems that arise from this specific type of Computing instruction.

Notably, the nature of block programming is such that it hides real lines of code and does not give students the chance to engage in the type of learning that happens in Type-What-You-See exercises. Block programming also limits opportunities to apply reading and writing strategies that would help develop proficiency in students. More research needs to be done to understand the extent to which this masking effect is either a hindrance to understanding Computing or an on-ramp to typed computer languages (Waite et al., 2018).

9.1. Student Performance with Scratch

As a contribution to understanding how block programming fits into the taxonomy of Computing Education, Meerbaum-Salan, Armoni, and Ben-Ari (2013) examined how well a group of Israeli middle school students could learn Computing with the block programming language, Scratch. The researchers determined that the group failed to understand key concepts such as repeated execution, functions, and concurrency (Meerbaum-Salan, Armoni, and Ben-Ari 2013, 73-75). The research team based their assessment of student understanding of Computing concepts on the students' ability to define said concepts, as opposed to basing their assessment on the students' ability to create computer programs.

While the team acknowledges that creating is considered a pedagogically more important proof of understanding than is the definition of terms, they can't quite seem to accept it as they say "Creating is considered to be much more complex than Understanding, but can we really say that creating a simple project – whose goal is to move one sprite from one point to another – is cognitively [more] complex than fully understanding the concept of concurrency?" (71). Perhaps the researchers, themselves graduates of computer science programs that have a bias toward theory which is reflected in teaching and assessment practices, cannot accept that the skills needed to create computer programs require a different pedagogical approach for their development and that there is also a need for different assessment practices to understand and evaluate student learning. Their aforementioned rhetorical question is reminiscent of Donald E. Knuth's 1974 opinion that for many of his colleagues in the field of artificial intelligence, computer programming is an artifact (669). Knuth also believes that whatever discoveries are made in the science of

Computing serve to create better art (669), but art demands an outlet for expression that is not constrained by definition.

While the 2013 study by Meerbaum-Salan, Armoni, and Ben-Ari shows that the middle school students they followed had difficulty defining certain Computing concepts, the study also shows that students had a lot of difficulty creating a functional computer program with Scratch. Unfortunately, in the final analysis, the researchers deem student performance quite disappointing since the mean grade for creating was only 32.8 (73). It seems that the students were unable to code a Scratch program from this requirement,

> Construct an animation with two sprites. The sprites will be placed in two corners of the stage facing the center of the stage. Pick one sprite whose task will be to broadcast the message switch to the other sprite. After the message is received the two sprites will change their places using the instruction glide 1 secs to x: 0 y: 0. During the process of changing places, the sprites will say something to each other when they meet (73).

What stands out is that by all the metrics for success established by this team of researchers, the students seem to have done poorly. The results outlined above speak to some of the failures. However, the researchers had also previously established that students could be considered to have successfully learned some Computing if they could give good definitions of Computing concepts (71). While student failure in that regard was less abject than it was in the creation of computer programs, the results were still weak: the report shows that in the post test, students correctly answered only 7.5% of questions about Multistructural Understanding, only 37.5% of questions about Multistructural Applying, and only 62.5% of questions about Relational Applying (75).

The research team of Meerbaum-Salan, Armoni, and Ben-Ari published a follow-up study in 2015. The 2015 study included students who had learned Scratch (a visual language) during the first, 2013, study. The 2015 study focused on student and teacher experiences in high school as students moving from middle school had their first exposure to a textual computer language: either Java or C#. The results of this study highlight the limitations of block programming; more specifically, Scratch.

First, the researchers found that students who had previously learned Scratch (i.e.: "Scratch students") recognized some Computing concepts while those who had had no prior exposure, did not (Meerbaum-Salan, Armoni, and Ben-Ari 2015, 9). Second, they also found that experienced computer science teachers ,who were teaching Scratch students, reported being able to teach concepts in Java or C# more quickly than in years prior (9). This is a hopeful bit of information, yet without further study, we cannot know how much exposure in middle school to a computer language of any kind would make for faster conceptual explanations to those same students in following years. Third, they found that on interim test results measuring student ability to create a computer program in Java or C#, there was no significant difference in knowledge and understanding between the Scratch students and those students with no Scratch coding experience (8). In the final analysis, they say that the Scratch students had final test scores for program creation that were better in a statistically significant way (8). However, the final test question for computer program creation "(e) [*Relational creating*] Write a program segment that gives the same results but that uses only one loop" (13), is a question that can be answered with logic learned in Scratch that is supported in Java and C#, that can compile when run, but which would be assessed as inelegant, qualitatively poor, and simply wouldn't have been taught to students studying Java or C#. For example,

while (1 == 1) { LOOP }

a) will compile, b) is similar to an unbounded or "forever" loop in Scratch, and c) is poor form when more straightforward, bounded, loops like "for … next" are available.

In other words, Scratch students might have answered the final test question using unbounded loops, while non-Scratch students would have attempted to answer with bounded loop logic because that was the only logic they learned. It is possible that the Scratch students received points on their final test for being technically correct in their work: the report does not explain what logic was used to answer the question and does not look at whether or not there was a trend amongst the two different groups of students. Additionally, there is no reportage on what interim test questions (for program creation) were asked, making it difficult to know why the interim test results showed no significant differences between the groups of students, while the final test question for program creation had Scratch students pulling ahead.

There remains another outstanding question for this research team: Why was there a jump in Scratch student ability to write code that compiles between 2013 and 2015? Given that the test results for program creation in the 2013 study were atrocious while Scratch student test results in 2015 were not, this question must be answered. A possible answer is that having a visual language provides a valuable, concrete, starting point for teaching and learning abstract thought, but that block programming languages inhibit the expression of computational thinking and computational logic to the point where coding in a block programming language is very difficult to do: requirements-based programming becomes slow and laborious.

In an article published in January 2018, Jane Waite et al. discuss the fact that novice teachers of the block programming language Scratch, feel that they have been learning computer programming through trial and error (Waite et al., 2018). The researchers show that teachers

participating in the study don't clearly understand the concept of algorithms and have difficulty pointing out the algorithms created in Scratch (Waite et al., 2018). Yet in constructionism and in French immersion, trial and error is a key component of eventual success in learning. If block programming languages such as Scratch fail to allow coherent development of computer programs and coherent understanding of a computer program's component parts, it may not be due to a failure in the theory of constructionism, but rather due to a failure in the pedagogical approach of block programming.

This possibility is addressed by Cynthia Solomon (2015), who pioneered Logo (a visual programming language) with Seymour Papert and Wally Feurzeig, and who says in a lecture preserved on YouTube, that while she admires the community that Scratch has built, in Scratch the "code goes on and on and on and on" which means that there is little room to practice valuable computer programming skills: procedures, sub-procedures, and recursion—a very common type of algorithm (Solomon, 2015). When users of Scratch, such as the aforementioned teachers in Waite et al.'s 2018 study, have difficulty understanding algorithms, it would appear that the masking nature of block programming languages hinders a trial and error learning process. In addition, while content-based instruction represents the bulk of the work done in French immersion, there is evidence that students benefit from the introduction of direct learning language activities to clarify concepts discovered or used in project-based learning (Grabe and Stoller 1997, 6). This type of activity is difficult to do in block programming languages because they mask syntax and vocabulary and truncate the full expression of computational thinking and computational logic.

10. CONCLUSION: LITERACY, SOFT SKILLS, & COMPUTER PROGRAMMING AS AN ART

As the world moves ever faster towards full automation and genuine artificial intelligence we face the prospect of enormous economic changes. We also see that the calls made in the 1970s and 80s for national curricula in Computing are finally being heeded. Fundamentally, the curricula should be based on research. Yet, at present, we are far away from having any definitive answers about what instructional approaches and methodologies work best. It will be years, maybe even decades, before we have excellent information. In the meantime, educational researchers can look outside the field of computing education for guidance in instructional methodologies and approaches.

This research paper has focused on drawing parallels between spoken language development and the development of Computing skills. Of distinct importance is the evidence brought to bear that computer programming is an art as much as it is a science. An analysis of some of the literature published by Donald E. Knuth and Francis E. Masat in the 1970s and 80s shows that computer programming education did not always fail students. Masat's work in particular sheds some light on instructional methodologies that worked—and most of those are rooted in constructionism, reading and writing strategies, and a learning environment similar to that of French immersion. Ultimately, educators who are trained in teaching language arts and literacy skills may make the best computer programming teachers. While more research does need to be done, there is reason to believe that a Computing immersion program which (through a creative trial and error process) engages critical and creative thinking competencies in K-12 students can prepare them for the new world. In the final analysis, treating computer programming as an art would allow educators to simultaneously: a) develop literacy skills in either a native or a second spoken language, b) develop computer programming skills and, c) use collaborative reading and writing strategies to develop students' soft skills so that they do well in the years to come.

REFERENCES

Baker, Monya. 2017. "Scientific Computing: Code alert." *Nature* 541, no.7638 (January): 563-565. https://doi.org/10.1038/nj7638563a.

Bakken, Linda, Nola Brown, and Barry Downing. 2007. "Early Childhood Education: The Long-Term Benefits." *Journal of Research in Childhood Education* 31, no.2 (February): 255-269. https://doi.org/10.1080/02568543.2016.1273285.

Brown, Bob. 2011. "Why there's no Nobel Prize in Computing." *Network World*: Data Centers, June 6, 2011. https://www.networkworld.com/article/2177705/data-center/data-center-why-there-s-no-Nobel-prize-in-computing.html.

Cann, Oliver. 2016. "Five Million Jobs by 2020: the Real Challenge of the Fourth Industrial Revolution." *The World Economic Forum* (January). https://www.weforum.org/press/2016/01/five-million-jobs-by-2020-the-real-challenge-of-the-fourth-industrial-revolution/.

Cummins, Jim. 2014. "To What Extent are Canadian Second Language Policies Evidence-Based? Reflections on the intersections of research and policy." *Frontiers in Psychology* 5 (May): Article 358. https://doi.org/10.3389/fpsyg.2014.00358.

Dean, Cornelia. 2007. "When Science Suddenly Mattered, in Space and in Class." *The New York Times*: Science, September 25, 2007. https://www.nytimes.com/2007/09/25/science/space/25educ.html.

Eliot, Lise. 1999. *What's Going on in There?: How the Brain and Mind Develop in the First Five Years of Life.* 1st ed. New York: Bantam Books.

Fiske, John R., Marvin T. Batte, and Reed D. Taylor. 1985. "A Computer Literacy Course for the College of Agriculture: A Survey of Student Attitudes, Evaluation and Performance." *North Central Journal of Agricultural Economics* (July): 119-128. https://kb.osu.edu/bitstream/handle/1811/65725/CFAES_ESO_1170.pdf?sequence=1&isAllowed=y.

— — —. 1985. "Course Proposal: Providing Computer Literacy." *North American Colleges and Teachers of Agriculture Journal* (December). https://www.nactateachers.org/attachments/article/1207/Fiske_NACTA_Journal_December_1985.pdf.

Genesee, Fred. 2007. "French Immersion and At-Risk Students: A Review of Research Evidence." *Canadian Modern Language Review* 63, no.5 (August): 655-687. https://doi.org/10.3138/cmlr.63.5.655.

Giannakos, Michail N., Ilias O. Pappas, Letizia Jaccheri, and Demetrios G. Sampson. 2016. "Understanding student retention in computer science education: The role of environment, gains, barriers and usefulness." *Education and Information Technologies* 22, no.5 (October). https://doi.org/10.1007/s10639-016-9538-1.

Grabe, William, and Fredericka L. Stoller. 1997. "Content-Based Instruction: Research Foundations." In *The Content-based Classroom: Perspectives on integrating language and content*, edited by Marguerite Ann Snow and Donna Brinton, 5-21. London: Longman Publishing. http://www.univie.ac.at/Anglistik/Dalton/SE08%20clil/Stoller&Grabe970001.pdf.

Hall, Anna H., Amber Simpson, Ying Guo, and Shanshan Wang. 2015. "Examining the Effects of Preschool Writing Instruction on Emergent Literacy Skills: A Systematic Review of the Literature." *Literacy Research and Instruction* 54, no.2 (January): 115-134. https://doi.org/10.1080/19388071.2014.991883.

Harper, Steve. 2006. "Immersion language theory meets CS." *Journal of Computing in Small Colleges* 22, no. 2 (December): 85-91.

India. 2017. "95% engineers in India unfit for programming jobs: Study." *Live Mint: Industry* April 20, 2017. https://www.livemint.com/Industry/cFUpp8wN9sXhXBVaBXRHlM/95-engineers-in-India-unfit-for-software-development-jobs.html.

Jones, Cindy D'On, D. Ray Reutzel, and Jamison D. Fargo. 2010. "Comparing Two Methods of Writing Instruction: Effects on Kindergarten Students' Reading Skills." *The Journal of Educational Research* 103, no.5 (August): 327-341. https://doi.org:10.1080/00220670903383119.

Knuth, Donald E. 1974. "1974 ACM Turing Award Lecture: Computer Programming as an Art." In *Communications of the ACM* 17, no. 12. (December): 667-673. New York: NY: ACM. https://doi.org/10.1145/1283920.1283939.

Lapkin Sharon, Callie Mady, and Stephanie Arnott. 2009. "Research perspectives on Core French: A Literature Review." *Canadian Journal of Applied Linguistics* 12, no.2. 6–30. https://journals.lib.unb.ca/index.php/CJAL/article/view/19936/21811.

Long, Michael H. 2000. "Focus on Form in Task-Based Language Teaching." In *Language policy and pedagogy: Essays in honor of A. Ronald Walton*, edited by R. D. Lambert, E. Shohamy, and A. R. Walton, 179-192. Philadelphia: Benjamins.

Masat, Francis E. 1981. *Computer Literacy in Higher Education.* Washington, D.C.: American Association for Higher Education.

— — —. 1982. "An Immersion Course in Basic." *Journal of Educational Technology Systems* 10, no. 4 (June): 321-329. https://doi.org/10.2190/cl7h-bdxq-t03f-hpbh.

Meerbaum-Salan, Orni, Michal Armoni, and Mordechai Ben-Ari. 2013. "Learning Computer Science Concepts with Scratch." *ICER '10 Proceedings of the sixth international workshop on Computing education research workshop.* (August): 69-76. Aarhus, Denmark: ACM. https://doi.org/10.1145/1839594.1839607

— — —. 2015. "From Scratch to "Real" Programming." *ACM Transactions on Computing Education* 14, no. 4 (February): Article No. 25. https://doi.org/10.1145/2677087.

Ministry of Education of British Columbia. "BC Schools - Enrolment and Completion by School." Catalogue.data.gov.bc.ca. https://catalogue.data.gov.bc.ca/dataset/bc-schools-course-enrolment-and-completion-by-school (accessed April 12, 2018).

Ministry of Education of Ontario. 2003. "A Guide to Effective Instruction in Reading, Kindergarten to Grade 3: Ontario Early Reading Strategy." http://www.eworkshop.on.ca/edu/resources/guides/Reading_K_3_English.pdf.

— — —. 2006. “A Guide to Effective Literacy Instruction, Grades 4 to 6.” *Foundations of Literacy Instruction for the Junior Learner* 1, no.1 http://www.eworkshop.on.ca/edu/resources/guides/guide_lit_456_vol_1_pt1_junior_learner.pdf.

— — —. “Think Literacy: Cross-Curricular Approaches, Grades 7-12.” http://www.edu.gov.on.ca/eng/studentsuccess/thinkliteracy/files/Reading.pdf.

— — —. “Course enrolment in secondary schools.” Ontario.ca. https://www.ontario.ca/data/course-enrolment-secondary-schools (accessed April 8, 2018).

National Association for the the Education of Young Children. 1998. “Learning to Read and Write: Developmentally Appropriate Practices for Young Children: A joint position statement of the International Reading Association and the National Association for the Education of Young Children.” *Young Children* 53, no.4 (July): 30-46. https://www.naeyc.org/sites/default/files/globally-shared/downloads/PDFs/resources/position-statements/PSREAD98.PDF.

O'Brien, Carl, Joe Humphreys, and Nora Ide McAuliffe. 2016. “Concern over drop-out rates in computer science courses.” *The Irish Times: Education* January 11, 2016. https://www.irishtimes.com/news/education/concern-over-drop-out-rates-in-computer-science-courses-1.2491751.

Paikin, Steve. 2016. “The History of French Immersion.” Toronto: TVO. Transcript. https://tvo.org/transcript/2391707/video/programs/the-agenda-with-steve-paikin/the-history-of-french-immersion.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books.

— — —. 1986. *Constructionism: A new opportunity for elementary science education.* Massachusetts: Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group.

Quora. 2016. “How good is Learn Python the Hard Way for learning Python and coding.” (May) https://www.quora.com/How-good-is-Learn-Python-the-Hard-Way-for-learning-Python-and-coding (accessed March 25, 2018).

Reach Capital. 2017. “Field Report on K12 CS.” (March). https://drive.google.com/file/d/0B2eCjHNmaBGZeGpwTGlRTUJKZlU/view.

Rehner, Katherine. 2017. “The CEFR in Ontario: Transforming Classroom Practice.” Curriculum Services Canada. https://transformingfsl.ca/wp-content/uploads/2017/12/LGY769-DELF.pdf.

Sarpong, Kofi Adu-Manu, John Kingsley Arthur, and Prince Yaw Owusu Amoako. 2013. “Causes of Failure of Students in Computer Programming Courses: The Teacher Learner Perspective.” *International Journal of Computer Applications* 77, no.12 (September): 27-32. https://doi.org/10.5120/13448-1311.

Schwab, Klaus. 2016. *The Fourth Industrial Revolution*. New York: Crown Business.

Shaw, Zed. 2015. *Learn Python the Hard Way: A very simple introduction to the terrifyingly beautiful world of computers and code*. Upper Saddle River, NJ: Addison-Wesley.

Sheard, Judy, S. Simon, Margaret Hamilton, and Jan Lönnberg. 2009. “Analysis of research into the teaching and learning of programming.” *ICER ‘o9 Proceedings of the fifth international workshop on Computing education research workshop*. (August): 93-104. Berkeley, CA: ACM. https://doi.org/10.1145/1584322.1584334.

Solomon, Cynthia. 2015. “Oct 2015 - Cynthia Solomon - Computer Cultures: Logo, Scratch and Beyond.” Uploaded on Oct 9, 2015. YouTube, 1:12. https://www.youtube.com/watch?time_continue=90&v=6OsmkUVWZY0.

— — —. 2018. *Logothings*. https://logothings.wikispaces.com/. This website, Cynthia Solomon’s first hand account of her work with Papert and Feurzeig, will be defunct as of July 31, 2018. As much of it as possible is being preserved.

Stahl, Katherine A. Dougherty. 2014. “New Insights About Letter Learning.” *The Reading Teacher* 68, no. 4 (November): 261-265. https://doi.org/10.1002/trtr.1320.

Stryker, Stephen B., and Betty Lou Leaver. 1997. *Content-Based Instruction in Foreign Language Education: Models and Methods*. Washington, D.C.: Georgetown University Press.

The Royal Society. 2017. *After the reboot: computing education in UK schools*. London: The Royal Society. https://royalsociety.org/news/2017/11/invest-tenfold-in-computing-at-schools/.

U.S. Census Bureau. 2016. *U.S. Census Bureau QuickFacts: United States*. https://www.census.gov/quickfacts/fact/table/US/PST045216

Velez-Rubio, Miguel. 2013. “Introductory Computer Programming Course Teaching Improvement Using Immersion Language, Extreme Programming, and Education Theories.” https://dl.acm.org/citation.cfm?id=2574995.

Waite, Jane. 2017. “Pedagogy in teaching Computer Science in schools: A Literature Review.” In *After the reboot: computing education in UK schools*. London: The Royal Society. https://royalsociety.org/~/media/policy/projects/computing-education/literature-review-pedagogy-in-teaching.pdf

Waite, Jane Lisa, Paul Curzon, William Marsh, Sue Sentance, and Alex Hadwen-Bennett. 2018. “Abstraction in action: K-5 teachers uses of levels of abstraction, particularly the design level, in teaching programming.” *International Journal of Computer Science Education in Schools* 2, no. 1 (January): 14-40. https://doi.org/10.21585/ijcses.v2i1.23.

Wingate, Richard. 2018. *New Interdisciplinary Computing Education Research Centre (CERC) at KCL*. London: King’s College. https://blogs.kcl.ac.uk/cser/2018/01/12/new-interdisciplinary-computing-education-research-centre-cerc-at-kcl/

Yakun, Wu. 2006. “Applying a hybrid problem-based learning method to the teaching of computer programming.” *The China Papers: Tertiary Science and Mathematics Teaching for the 21st Century* (November): 63-66. https://web.archive.org/web/20180411182505/http://science.uniserve.edu.au/pubs/china/vol6/IT6.pdf.