2-26-2020

Introduction to computational thinking: A new high school curriculum using codeworld

Fernando Alegre
Louisiana State University

John Underwoood
Louisiana State University

Juana Moreno
Louisiana State University

Recommended Citation

Alegre, F., Underwoood, J., & Moreno, J. (2020). Introduction to computational thinking: A new high school curriculum using codeworld. *Annual Conference on Innovation and Technology in Computer Science Education, ITiCSE*, 992-998. https://doi.org/10.1145/3328778.3366960

Introduction to Computational Thinking: A New High School Curriculum using CodeWorld

Fernando Alegre
Gordon A. Cain Center for STEM Literacy
Louisiana State University
Baton Rouge, LA, USA

John Underwoood
School of Education
Louisiana State University
Baton Rouge, LA, USA

Juana Moreno
Department of Physics & Astronomy and
Center for Computation & Technology
Louisiana State University
Baton Rouge, LA, USA

Mario Alegre
Department of Physics
Pennsylvania State University
University Park, PA, USA

ABSTRACT

The Louisiana Department of Education partnered with the Gordon A. Cain Center at LSU to pilot a Computing High School Graduation Pathway. The first course in the pathway, Introduction to Computational Thinking (ICT), is designed to teach programming and reinforce mathematical practice skills of nine-grade students, with an emphasis on promoting higher order thinking. In 2017-18, about 200 students and five teachers participated in the pilot, in 2018-2019 the participation increased to 400 students, and in the current 2019-2020 year about 800 students in 11 schools are involved. After describing the course content and the teacher training, we briefly discuss the data we have collected in the last two years. The overall student reception of the course has been positive, but the course was categorized by most students as hard. The pre-post test content assessments show that students have learned not only the language, but also general principles of programming. Lessons learned during the pilot phase have motivated changes, such as emphasizing during Professional Development the need to provide timely feedback to students, provide detailed rubrics for the projects and reorganize the lessons to increase the initial engagement with the material. After two years of running pilots, the course is becoming student-centered, where most of the code and image samples provided in the lessons are based on code created by previous students.

KEYWORDS

computational thinking, K-12, high school, course, introductory programming, functional language, Haskell, CodeWorld, professional development, teacher training, computer science education

SIGCSE '20, March 11–14, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6793-6/20/03…$15.00
https://doi.org/10.1145/3328778.3366960

ACM Reference Format:
Fernando Alegre, John Underwoood, Juana Moreno, and Mario Alegre. 2020. Introduction to Computational Thinking: A New High School Curriculum using CodeWorld. In *The 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20), March 11–14, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3328778.3366960

1 INTRODUCTION

Our project started in 2015, when we were contacted by the East Baton Rouge Parish School System (EBRPSS) to help develop computer science curricula for a new STEM magnet high school, to offer new opportunities to the under-served population of the district, which consists of 85% minority and 75% economically disadvantaged students. We were tasked with creating the curriculum, including its assessment and the delivery of the summer teacher training. Additionally, the curriculum had to be designed in such a way that teachers of other academic subjects could quickly learn it, since there were no computer science teachers available in the area. The first course in this set is Introduction to Computational Thinking (ICT), an introductory programming course offered to eighth or ninth graders who are concurrently taking an Algebra I course. The course teaches the conceptual foundations of coding in a language syntax and semantics that follow closely the language of algebra. It is not intended to be a math remediation course, but rather to highlight the connections to algebra, geometry and science modeling.

During the 2016-2017 academic year, we conducted several three-month pilot tests of the course and developed an assessment instrument, the *Conceptual Foundations of Coding* (CFC) test, which was vetted with about 100 students. A full-course pilot was deployed in the 2017-2018 academic year. At that time, the Louisiana Department of Education (DoE) became interested in the ICT curriculum and partnered with the Cain Center to create and pilot a Computing High School Graduation Pathway, following the model pioneered by the EBRPSS STEM magnet high school. The Pathway offers a hybrid curriculum that prepares students both for college and to enter the workforce after graduation.

During 2017-2018, the course was taken by more than 200 ninth grade students in four different schools. Approximately 400 students in ten schools in eight school districts were enrolled in ICT for the 2018-2019 academic year, and there are 800 students enrolled in the 2019-2020 academic year.

Table 1: ICT Course outline.

Unit	Content
The Software Development Cycle	Students learn how to use an IDE, how to draw basic shapes, how to overlay several pictures and move them around the screen. They also learn about design techniques, such as creating prototypes and using pseudo-code to plan a program, and practice collaboration with pair programming and a collaborative creation of a scene, where each team member is in charge of a character or prop.
Abstraction and Decomposition	Students learn to map expressions to syntax trees, handle order of operations, and use trees to represent other aspects of code, such as dependencies between variables and organization of layout into nested layers. They also learn how to use an object dimension as a unit of measurement for other objects (e.g., 2.5 smileys wide) and to combine rotations, translations and scalings to create complex mosaics or quilt patterns.
Patterns and Regularity	Students use repetition to create regular polygons, regular stars and create recursive patterns. They also learn about generating random patterns and irregular grids, and use them to generate a procedural map of a neighborhood.
Data and Calculations	Students learn to process lists to create bar charts and pie charts from scratch, create itemized bills including taxes and discounts, calculate weighted averages and compute areas of complex settings, such as the area occupied by chairs and tables in a dining hall.
Models in Space and Time	Students create simple games (rock, paper, scissors; dice rolling games; tic-tac-toe) and simple animations (characters performing repetitive circular or linear motion; see-saws; slide shows; marquee messages)

In the summer of 2017, we conducted our first Professional Development program, which is an intensive five-week summer institute. In 2017 we trained eight teachers, with an additional nine teachers in 2018, and most recently nine more teachers. The teachers were absolute novices with respect to programming. They were placed into student roles as the first part of their training, where they completed all the programming assignments, presented them to their peers, and modified their code according to the feedback received. The teachers were additionally instructed in pedagogical techniques and lesson design. At the end of the summer Professional Development, the teachers felt comfortable enough to teach the course and to modify the assignments to meet their school's unique cultures and needs. The majority of the teachers participating were certified in either secondary math or science, but, typically, each year two social studies teachers also participate.

All the activities are programmed in CodeWorld [4], a web-based integrated development environment initially designed for middle school students, that uses a simplified variant of the Haskell language. The lessons are organized in units that follow the concepts of Computational Thinking, with the syntax of the language being presented at the beginning of each semester. However, very little emphasis is placed on teaching the language, whose features are introduced only when needed. In the first semester, only expressions, variables and functions are used. No conditionals, looping constructs or data structures are needed for the programming assignments. In the second semester, lists and tuples are the only new syntactic features needed, and looping constructs are based on a second-order function, called `foreach`, which is a regular function with no special syntax.

2 DEVELOPMENT PROCESS

2.1 Foundational Stages

Frameworks to introduce computer science in K-12 education, either in isolation or integrated with other core subjects, are emerging in many countries [10]. These new frameworks aim to present the subject of computer science as a living discipline with connections to the real world and to other STEM subjects [2]. Learning computer science is neither about the specifics of a programming language nor the commands and techniques on how to program. The term *Computational Thinking* (CT) was introduced in education to describe the process of problem solving using computational techniques within the context of other disciplines [1, 8, 9, 20, 21, 24]. Although programming is the means and not the end goal, it can be difficult to understand CT fully without exposure to programming [5, 8].

In our training materials, we try to provide insight about the meaning of computational thinking. In our view, when someone is thinking computationally they do the following:

(1) Use introspection to observe their own thought process as if it were performed by a machine and express their thoughts explicitly and without any ambiguity.
(2) Imagine in their head a computer running a given program and anticipate the outcome without actually running the program.
(3) Reason constructively: the purpose of computing is to construct a solution. Computing works under a closed world assumption, where only those entities explicitly built are assumed to exist.
(4) Invent a process to solve a problem as a series of mechanical steps, where each step requires no intelligence to be performed. The intelligence contained in a program is an emergent feature and cannot be pinpointed to any particular line in the code.
(5) Think in terms of causality. A function is not just a relationship between an input and an output, as it would be in mathematics. It is also a process that causes the computer to produce an output when the given input is consumed. This process occurs in time, and so the input must exist before the output can exist. Computations change the world.
(6) Reason by proxy: Distinguish between what a concept is and how it is represented. For example, represent a polygon as a list of pairs of coordinates.

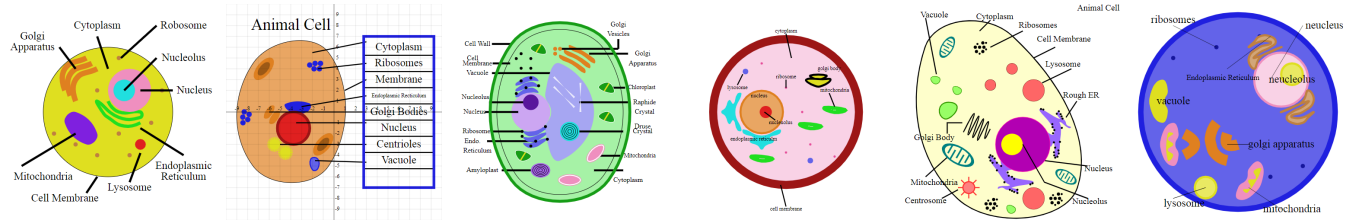

Figure 1: Six examples of cells designed by different students in the 2017-2018 academic year.

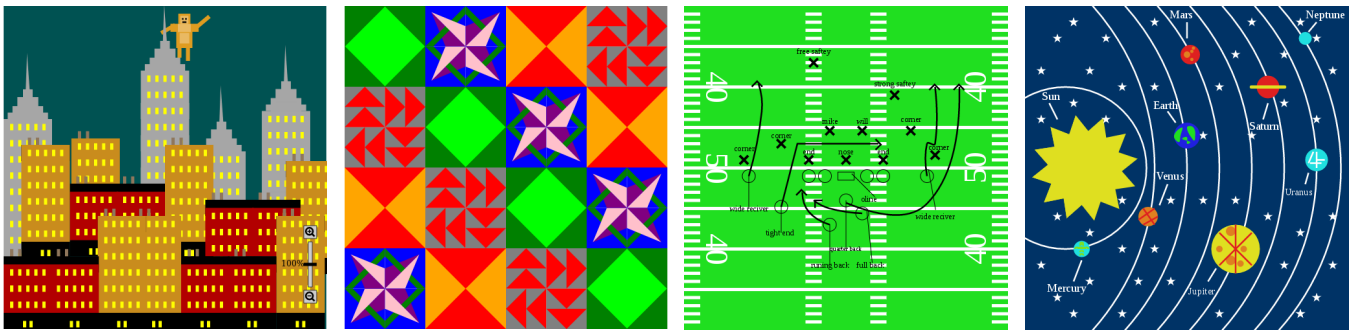

Figure 2: Four examples of end-of-unit projects made by different students.

(7) Establish relationships between concepts by writing equations between their corresponding representations. For example, move a polygon horizontally by adding the same number to each X coordinate in the corresponding list.

Computational thinking is about expressing thoughts formally, in a way that is actionable by an automated system. Programming languages are not the only possible formal systems in which computational thinking can be expressed, but they are the most accessible and prone to automation. Thus, using programming as a vehicle for computational thinking is a natural choice.

Unfortunately, in many elementary and middle school settings, the term CT has become synonymous with either *computing with no programming* or *block-based programming*. This interpretation omits the central tenet of Computational Thinking, which is the *building of high-level abstractions that can be executed by a computer* [25]. Currently, there is a need to have a high school course that introduces CT with substantial amounts of programming and connections to math and science. This CT course should depend as little as possible on the extensive knowledge of a particular language or technology. The CT course would be a natural progression for students to take along with Exploring Computer Science (ECS) and Computer Science Principles (CSP).

For the most part, ECS, CSP and block-based programming courses rely on the teachers to act as facilitators of instruction provided by an online system. This instructional model is based on the idea that students will learn even if they are not being directly instructed by their teacher. However, a flaw of the model is the fact that many concepts, such as abstraction, are only developed through higher-order learning [15, 19].

A recognizable factor for why this model is on the rise is due to a current scarcity of teachers who know how to program. However, students need direct interaction with a teacher to master higher order thinking concepts. There is great value in having a teacher who can evaluate the students work, reflect on the student's progress, offer guidance to the student on ways that they may correct habits, and examine unique work products from the perspective of the student's intellectual evolution. These attributes have proven difficult to evaluate effectively in an automated way.

2.2 Learning Objectives

ICT is an elective course in Louisiana, where a majority of the students have historically demonstrated weak mathematical skills for all grade levels. In designing the course, additional attention was given to ways to help students improve their math skills as they learn computer science. This course was not intended to be strictly a math remediation or math intervention course, but rather an integrated component of a STEM elective pathway. The learning objectives were established and designed to be recurring throughout each of the units, and not isolated to specific lessons. The learning objectives include:

- Develop a procedural understanding of the pillars of Computational Thinking: recognize patterns and regularity, decompose problems into smaller parts, formulate and solve simplified problems, generalize solutions and encapsulate solutions.
- Acquire experience with algebraic manipulation of complex expressions.
- Use mathematical functions to model artifacts, such as diagrams or animations.
- Transform many data items as if they were a single entity.
- Organize data hierarchically.

- Calculate totals, averages and quantities using rates, such as taxes and discounts.
- Use random sampling to explore instances of relationships and find the general case.

2.3 Conceptual Framework

The idea of using coding to help students learn mathematics and science has a long history. Early attempts to use coding as a tool were based on unguided discovery [13]. This approach proved to be ineffective for transfer [12]. Over the years, it has become clear that transfer between programming and mathematics is difficult to initiate, and whether it occurs or not depends strongly on the teaching methodology used [3, 11, 14]. Recent attempts to establish the link between programming and mathematics have been based on a modern framework of computational thinking [8, 24, 25] and supported by modern theories such as convergent cognition [16].

One of the few cases in which a project targeted the learning of mathematics with coding and showed promising results is Bootstrap, a 17-hour curriculum designed to be used either standalone or embedded in a computer science or mathematics course. It is one of the few documented instances of transfer between programming and algebra. Schanzer et al. [17, 18] attribute the favorable results of their intervention to their use of a functional language as the medium and to the absence of distracting features.

Our approach is inspired by the works of Felleisen et al. [7] and Bootstrap [17, 18], due to their promising results concerning transfer between programming and mathematics [17, 18]. They introduced the *design recipe*, which is a series of steps for guiding students when they are trying to create a function: write a definition in English, then describe the inputs and outputs, then provide at least 3 examples, then look at what is common in those examples (the template) and what changes from example to example (the variables), and finally give names to those variables.

However, we differ from Bootstrap in several ways. We have developed a full-year curriculum centered on CT instead of a 17-hour intervention focused on math word problems. Our use of Haskell makes writing function definitions very lightweight, so students are encouraged to create lots of functions. Also, the lazy evaluation model relieves us from the need to have special syntax for program control. We have also extended the design recipe with the introduction of random variables, so that students create random samples of uses of a function after (or instead of) providing examples with fixed numbers. Finally, we put more emphasis on modeling techniques and using the software development cycle rather than on guided exercises based on code templates.

2.4 CodeWorld activities

All the activities are programmed in CodeWorld [4], which uses a very limited set of graphical primitives to draw circles, rectangles, and text. It is then possible to apply translations, rotations, scalings and colors to them. Smaller elements can be combined into more complex shapes via the overlay operator (denoted by &). Animations are represented as functions that depend on a parameter, namely the time in seconds since the animation started. The language follows a syntax very similar to mathematical notation, and the evaluation semantics follows exactly the same rules as algebra.

Here is a complete CodeWorld program to draw a house:

```
program  = drawingOf(house(red,yellow)
         & coordinatePlane)
house(rcolor,fcolor) =
  colored(roof,rcolor)
  & windows & door
  & colored(facade,fcolor)
  & pathway
roof     = solidPolygon([ (-4,4),(4,4),(0,6) ])
windows  = floor2 & floor3
floor2   = translated(window,-2,1)
         & translated(window,2,1)
floor3   = translated(floor2,0,2)
window   = solidRectangle(1,1)
door     = translated(solidRectangle(1,2),0,-1)
facade   = translated(solidRectangle(8,6),0,1)
pathway  = overlays(tile,8)
tile(n) = translated(stone,-(n-1)/2,-1.5-(n+1)/2)
stone = colored(oval,translucent(grey(0.2)))
oval = scaled(solidCircle(0.5),2,1)
```

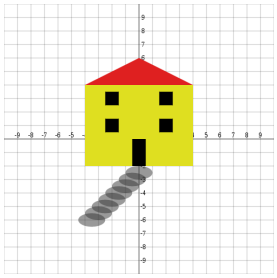

Figure 3: A drawing of a house

Practically all the syntax of the language is illustrated in the previous program, and all programs are written in exactly the same format (a list of lines that read `head = body`) with `program` being the starting point of the execution of the program. Functions are defined in the same way as variables, but the head includes parameters. No special constructions for loops or conditionals are necessary. Definite loops are provided by library functions, such as `overlays`, which works as follows: the expression `overlays(f,n)` is equivalent to `f(1) & f(2) & ... & f(n)`. Indefinite loops are created by recursive definitions. Conditionals are produced by having functions with special cases, which are created by adding a vertical bar and a condition to their definition. For example, the absolute value would be defined by the following two lines:

```
absoluteValue(x) | x <  0  = -x
absoluteValue(x) | x >= 0  =  x
```

In the second semester, lists and tuples are introduced. Basic list usage needs 3 additional symbols: `[`, `]` and `#`, to build a literal list, and to access the n^{th} element, respectively.

The simplified version of Haskell we are using stops here. No advanced features of the language (such as typeclasses, IO or monads) are exposed to students. In a sense, our use of Haskell provides the same affordances that a block-based language would, because the key features of block-based languages are their simple, bare-bones

syntax, as opposed to regular programming languages, and the avoidance of errors due to misspelling or misuse of variables and constructs placed in the wrong spot [22, 23]. Haskell shares both of these features, because in addition to the simple syntax explained above, the advanced type-inference features of Haskell catch practically all misspellings and misuses of variables and functions.

2.5 Curriculum content

The ICT curriculum comprises five units, where the first two units take approximately eight weeks each, and the last three units take approximately five weeks each. Table 1 lists the title and a brief summary of the content of each unit. In the 2019-2020 version of the curriculum, all the activities in the first four units include samples created by students who took the course in the previous two years. See Fig. 1 for samples of student work in an activity where students are asked to create a diagram of a cell.

Assessment is based on a project at the end of each unit, plus a midterm project and a final project. See Fig. 2 for samples of such projects. Our team developed analytical rubrics that present the criteria and levels of performance for each assignment. The rubrics are tiered from minimal, to lower, to mid-level, and finally high attainment. Each tier contained descriptors with point values. Each attribute was aligned to the learning objectives, which are stated at the start of each lesson and integrated in the activities that build to the project. Table 2 has examples of high and low attainment in a project to create an analog clock face.

2.6 Technical considerations

Our choice of programming environment was also influenced by the following properties: 1) The programming language should make it easy for students to build high level abstractions; 2) The language should also have a syntax and semantics as similar to algebra as possible; 3) No prior or additional knowledge of coding or software should be needed by teachers to produce code for the lessons; 4) Execution of any component of the system should not depend on any third party service or product; and 5) The programming paradigm should preferably be functional.

One technical restriction on our choice of programming environment was due to the fact that many Louisiana schools have policies concerning which software can be run on their computers. Often the computing environment is optimized for use on standardized testing platforms, which can prevent root access and local installation of software. In addition, policies in many schools prevent students from being required to register with third-party organizations or submit their work to external web sites. Given these conditions, we elected to use a Web-based environment that required no local installation and could be used without restrictions and without the need for students to register or provide any personal information. The next requirement our team faced was that we were required by the Louisiana Department of Education, which partially funded our project, to rely upon fully open source software. Our final requirement was that as a team we wanted to do graphics-based programs rather than text-based programs. Given all of the aforementioned requirements the number of possibilities we considered was limited. For example, at the project's onset there was no fully open source, fully online version of Python for graphics programming.

Table 2: Clock Face Example Rubric

High Attainment	Low Attainment
Used expressions with variables	Used magic numbers
Created a function to draw clock hands	Repeated code for hour hand and minute hand
Created different nested layers for hour ticks, minute ticks, and so on	Used a flat layout
Handled ranges properly	Printed redundant elements, e.g. printing 12 o'clock twice
Used local variables	Only used global variables
Followed good practices when naming, indenting, and grouping code	Did not follow good practices
Calculations to convert hours and minutes to rotations were correct	Calculations were not correct
Output shows analog clock with all elements placed appropriately	Elements misplaced or missing

3 IMPACT

3.1 Results and Analysis

In 2017-2018 the course was deployed at four schools with five teachers and 208 students. Ten schools, 13 teachers and 395 students participated during the 2018-2019 academic year. Unfortunately, due to time constraints near the end of the school year, post-assessments were not collected for many students, so we only have matched data for 325 students, or about 54% of the total enrollment.

The *Conceptual Foundations of Coding* (CFC) test is a multiple-choice test with four choices given on most questions. It was developed by the authors due to the lack of suitable tests designed to measure computational thinking as we define it. Fig. 4 displays histograms comparing the distribution in scores of the pre- and post- tests, showing that the post-test results are on average 24% higher, although the distribution is also significantly wider. The initial average score was $29.5 \pm 0.6\%$, which increased to $53.8 \pm 1.1\%$ at the end of the course. The difference between the distributions was found to be statistically significant ($p < 2.2 \cdot 10^{-16}$), as determined by the Wilcoxon signed rank test.

Fig. 5 displays the pre- and post-test results for the four categories included in the CFC test: variables, expressions and functions, `CodeWorld` specifics, mathematical modeling, and logic and programming. The average of the post-results is higher than the average of the pre-results in all four categories. The difference between the pre- and post- distributions is significant ($p < 10^{-12}$) for all the categories.

In the analysis of the Computing Attitudes Survey (CAS) [6] data, we found that the student attitudes did not significantly change ($p > 0.07$) after completing the course, as can be determined the minuscule shift in attitudes (0.02 ± 0.02), and the fact that the pre- and post-test results were very strongly correlated ($R = 0.54$, $p = 1.3 \cdot 10^{-10}$).

Additionally, we found a positive correlation between the post-test CAS results and the shifts in CFC scores ($R = 0.29$, $p = 0.001$), indicating that those students who had a more positive attitude at the end of the course also tended to have the higher gains in learning.

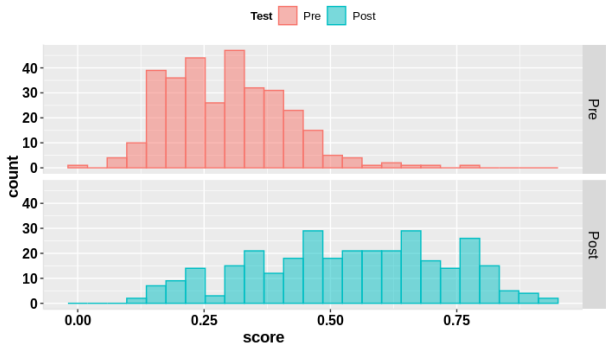

Figure 4: Histograms comparing the results of the CFC pre- and post-tests.

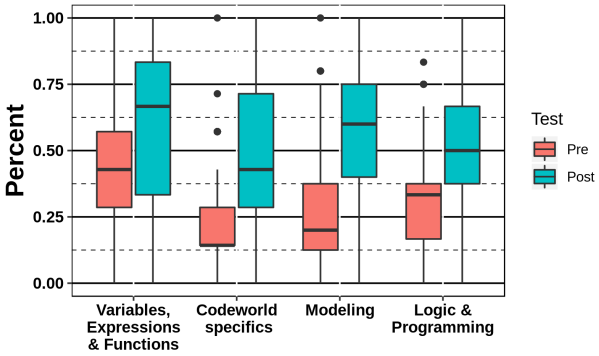

Figure 5: Box plots comparing results of the CFC Test by categories. Boxes stretch from the 25th to 75th percentile of the distribution.

The overall reception of the course has been positive, but the course was uniformly categorized by most students as hard. Our qualitative in-class observations found that students were not accustomed to having to use more than one mathematical idea in a single problem, and were also unsettled by the fact that the same image could be generated in many different ways (i.e., that there is no canonically correct way to write code to produce a certain output). Nevertheless, the Computing Attitude Survey analysis indicates that the difficulty of the course did not demotivate the students. The pre-post test analysis shows that the students learned not only the language, but also general principles of programming, logic and modeling, as well as use of variables, expressions and functions.

3.2 Future work

The results of the preliminary analysis seem encouraging, but further data collection and analysis will be required to determine exactly how effective the ICT curriculum is. Our plans for future years include increased emphasis on collecting post-assessments so matched data can be obtained for a larger percentage of the student body, more extensive validation of the CFC test so it can be published for the benefit of the wider computer science education community, and identification (or if no suitable test exists, development) of an assessment that can be used to measure if students' math skills have changed significantly between the beginning and end of the ICT course.

Additionally, as the ICT course continues to be adopted across Louisiana, a growing number of students will have taken ICT in eighth or ninth grade. This large pool of student participants could be tracked longitudinally, and compared with control groups from the same schools, to determine if participation in the ICT course has had an effect on their performance in subsequent math courses.

Furthermore, as more math and science teachers are trained to teach ICT, we will be able to develop a bank of coding activities for math and science courses, which could then be made available to teachers via an online portal. The teachers would then be able to adapt and incorporate these coding activities into their regular lessons. Using this population of students and teachers trained in our ICT course, we will be able to study whether the incorporation of functional programming into math and science curricula promotes deeper understanding of those subjects.

3.3 Conclusion

While the need for teaching computational thinking is already well established, there is still controversy about whether programming should be included or not, or, as Denning [5] calls it, the clash between Traditional CT and New CT. Courses such as ECS or CSP are examples of New CT, but there is not much available in terms of courses that focus on Traditional CT. Due to its capacity for automation and formalization, programming is a natural vehicle for learning computational thinking. While Python and JavaScript courses are relatively available, they do not usually focus on CT. Instead, they follow traditional syntax-oriented approaches to teaching computing, with few connections to math and science. Those courses are more useful for students aspiring to be software developers than for the general student population. We have presented an alternative approach.

We have described the design and implementation of a secondary Computational Thinking course based on programming with connections to science and math. This course provides a proof of concept for curricula halfway between traditional programming language courses and recent computational thinking courses with limited programming content. This course addresses the need for computational thinking courses intended not only for future software developers but for all students no matter what they do later in their lives. We find that a focus on programming content does not need to be discouraging to students. Our approach is highly student-centered, and has been proven to be suitable for traditionally under-served populations. We also build on Bootstrap ideas and techniques and have opened a way to investigate many interesting connections between the learning of programming and the learning of mathematics and science, and we are excited to delve into them.

Acknowledgment. This work is partly supported by NSF award CNS-1923573.

REFERENCES

[1] Charoula Angeli, Joke Voogt, Andrew Fluck, Mary Webb, Margaret Cox, Joyce Malyn-Smith, and Jason Zagami. 2016. A K-6 computational thinking curriculum framework: implications for teacher knowledge. *Journal of Educational Technology & Society* 19, 3 (2016), 47.

[2] Austin Cory Bart, Eli Tilevich, Simin Hall, Tony Allevato, and Clifford A Shaffer. 2014. Transforming introductory computer science projects via real-time web data. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 289–294.

[3] Earl C Butterfield and Gregory D Nelson. 1989. Theory and practice of teaching for transfer. *Educational Technology Research and Development* 37, 3 (1989), 5–38.

[4] Codeworld. 2019. An educational environment using Haskell. https://github.com/google/codeworld.

[5] Peter J Denning. 2017. Remaining trouble spots with computational thinking. *Commun. ACM* 60, 6 (2017), 33–39.

[6] Brian Dorn and Allison Elliott Tew. 2015. Empirical validation and application of the computing attitudes survey. *Computer Science Education* 25, 1 (2015), 1–36.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2004. The structure and interpretation of the computer science curriculum. *Journal of Functional Programming* 14, 4 (2004), 365–378.

[8] Shuchi Grover and Roy Pea. 2013. Computational Thinking in K–12: A Review of the State of the Field. *Educational Researcher* 42, 1 (2013), 38–43.

[9] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education* 25, 2 (2015), 199–237.

[10] Fredrik Heintz, Linda Mannila, and Tommy Färnqvist. 2016. A review of models for introducing computational thinking, computer science and computing in K-12 education. In *2016 IEEE Frontiers in Education conference (FIE)*. IEEE, 1–9.

[11] D Midian Kurland, Roy D Pea, Catherine Clement, and Ronald Mawby. 1986. A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research* 2, 4 (1986), 429–458.

[12] Richard E Mayer. 2004. Should there be a three-strikes rule against pure discovery learning? *American Psychologist* 59, 1 (2004), 14.

[13] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc., New York, NY, USA.

[14] Roy D. Pea. 1983. Logo Programming and Problem Solving. [Technical Report No. 12].

[15] Chaitanya Ramineni and David Williamson. 2018. Understanding Mean Score Differences Between the e-rater® Automated Scoring Engine and Humans for Demographically Based Groups in the GRE® General Test. *ETS Research Report Series* 2018, 1 (2018), 1–31.

[16] Peter J. Rich, Keith R. Leatham, and Geoffrey A. Wright. 2013. Convergent cognition. *Instructional Science* 41, 2 (2013), 431–453.

[17] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2013. *Bootstrap: Going Beyond Programming in After-School Computer Science*. SPLASH-E (Education track of the OOPSLA/SPLASH conference).

[18] Emmanuel Schanzer, Kathi Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. Transferring Skills at Solving Word Problems from Computing to Algebra Through Bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 616–621.

[19] T. Feathers. 2019, August 20. Flawed algorithms are grading millions of students essays wrong. https://www.vice.com.

[20] Joke Voogt, Petra Fisser, Jon Good, Punya Mishra, and Aman Yadav. 2015. Computational thinking in compulsory education: Towards an agenda for research and practice. *Education and Information Technologies* 20, 4 (2015), 715–728.

[21] David Weintrop, Elham Beheshti, Michael Horn, Kai Orton, Kemi Jona, Laura Trouille, and Uri Wilensky. 2016. Defining computational thinking for mathematics and science classrooms. *Journal of Science Education and Technology* 25, 1 (2016), 127–147.

[22] David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. ACM, New York, NY, USA, 199–208.

[23] David Weintrop and Uri Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *11th Annual ACM Conference on International Computing Education Research, ICER 2015*. Association for Computing Machinery, Inc.

[24] Jeannette M Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.

[25] Jeannette M Wing. 2008. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366, 1881 (2008), 3717–3725.