Genost: A System for Introductory Computer Science Education with a Focus on Computational Thinking

by

Garret Walliman

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Masters of Science

Approved April 2015 by the
Graduate Supervisory Committee:

Robert Atkinson, Co-Chair
Yinong Chen, Co-Chair
Yann-Hang Lee

ARIZONA STATE UNIVERSITY

May 2015

ABSTRACT

Computational thinking, the creative thought process behind algorithmic design and programming, is a crucial introductory skill for both computer scientists and the population in general. In this thesis I perform an investigation into introductory computer science education in the United States and find that computational thinking is not effectively taught at either the high school or the college level. To remedy this, I present a new educational system intended to teach computational thinking called Genost. Genost consists of a software tool and a curriculum based on teaching computational thinking through fundamental programming structures and algorithm design. Genost's software design is informed by a review of eight major computer science educational software systems. Genost's curriculum is informed by a review of major literature on computational thinking. In two educational tests of Genost utilizing both college and high school students, Genost was shown to significantly increase computational thinking ability with a large effect size.

ACKNOWLEDGEMENTS

Special thanks to the following individuals:

My thesis advisors Dr. Yinong Chen, Dr. Robert Atkinson, and Dr. Yann-Hang Lee for opening many doors and providing an enormous amount of feedback.

Active Capstone team members Rizwan Ahmad, Garth Bjerk, Tracey Heath, David Humphries, Corey Jallen, Ian Plumley, Stephen Pluta, Randy Queen, and Matt Rechia, for helping to create both the Genost design and the Genost software and robot.

Tracy Ryan and the staff of the Arizona School for the Arts, for helping to organize and run the ASA test.

The Fall 2014 FSE100 professors Jadiel de Armas, Refika Koseler Emre, Ryan Meuth, Phillip Miller, Tahora Nazer, Yoalli Hidalgo Pontet, Satyabrata Sharma, and Chao Zhang, for assistance with the FSE100 test.

Monica Dugan, Theresa Chai and the rest of the Brickyard staff, for their constant willingness to provide materials and rooms for my work and testing.
And my wife, Naomi Walliman, for her support, feedback and considerable patience

TABLE OF CONTENTS

Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

Table Page

LIST OF FIGURES

Figure Page

Figure

Page

1. INTRODUCTION

The United States and many other countries around the world are currently experiencing a revolution in STEM (Carnevale, STEM: Science, Technology, Engineering, Math, 2011). STEM is an acronym that stands for "Science, Technology, Engineering and Math", and the STEM revolution refers to the explosion in the first world job market for careers requiring STEM skills. The STEM revolution is not affecting only the job market; in fact, STEM skills, and the products and services that these skills provide, are now integrated into the life of the average first world citizen, in some ways deeply so (Carnevale, STEM: Science, Technology, Engineering, Math, 2011) (Cowen, 2013).

This thesis is about STEM, and its creation was prompted by the elevation of STEM abilities to their current position of importance in people's lives. In this introduction, we will begin by reflecting on the current status of STEM. Our reflection will show that the STEM revolution has prompted a great need for students to be taught a skill (or perhaps better put, a "paradigm") known as computational thinking. We will define what computational thinking is, why it is important, and why we ought to be teaching it to all students. We will conclude this introduction with an outline of the rest of the thesis.

1.1. THE STATUS OF STEM

STEM skills are currently in very high demand (Carnevale, STEM: Science, Technology, Engineering, Math, 2011) (BurningGlass, 2014). However, it has been noted by many that this demand is not being filled, prompting many to warn of an impending "STEM crisis" (CSTA Curriculum Improvement Task Force, 2006) (President's Council of

Advisors on Science and Technology, 2012). The root of this putative crisis is the upcoming high demand for STEM jobs, paired with a worrying lack of qualified graduates to fill these jobs.

In this section we will investigate the claims above. We will also argue that the "STEM crisis" is more properly called a "computer science crisis", as the impending high demand for STEM skills, as well as some broader predicted trends, are deeply intertwined with computer science.

1.1.1. The Demand for STEM

It has been argued by many that there is currently a great demand for STEM skills, which will only increase in the near future. In a report by Georgetown University's Center on Education and the Workforce, United States jobs requiring STEM skills are projected to grow at a 17% rate through the year 2018, compared to a 10% growth rate for US occupations as a whole throughout that time period (Carnevale, STEM: Science, Technology, Engineering, Math, 2011). Another report by Burning Glass Technologies, a company focusing on workforce development and trend prediction, states that for every new graduate with a 4-year STEM degree there are 2.5 entry-level job postings that this graduate could fill, with 5.7 million postings in all for 2013. This creates a 26% salary premium for STEM degree holders over non-STEM degree holders (BurningGlass, 2014). These statistics, and others like them, lead many to conclude that the demand for STEM degree holders will continue for the foreseeable future.

1.1.2. The STEM Crisis

Despite the large upcoming demand for STEM degree holders, current educational trends have led some to declare that this demand will not be filled. This phenomenon has been called the “STEM crisis.”

There are two aspects to the STEM crisis: first, that there are not enough STEM majors being produced (CSTA Curriculum Improvement Task Force, 2006) (President's Council of Advisors on Science and Technology, 2012) (CBI, 2013) (Rothwell, 2014), and second, that those STEM majors who are being produced do not have adequate skills to perform the jobs they will be asked to fill (House of Lords Select Committee on Science and Technology, 2012) (Gross, 2014) (CBI, 2013).

Many sources claim that there are not enough STEM majors to fill the large upcoming demand. For example, the Computer Science Teachers Association (CSTA) noted in 2005 that there would be an anticipated shortage of 1.5 million qualified candidates for CS and IT jobs by 2012 (CSTA Curriculum Improvement Task Force, 2006). This prediction was somewhat borne out; a 2012 executive report prepared by the United States President’s Council of Advisors on Science and Technology declared that, based on current economic projections and graduation rates, there would be a shortfall of 1 million STEM degree holders over the next decade (President's Council of Advisors on Science and Technology, 2012).

Many of the businesses that wish to hire STEM majors report trouble doing so. In the CBI / Pearson Education and Skills Survey for 2013, which surveyed 294 firms, 39% of

firms surveyed reported having difficulty recruiting workers with the STEM skills they needed (CBI, 2013). Furthermore, a report by the Brookings Institute declares that STEM job positions take nearly twice as long to fill as positions that do not require STEM skills (Rothwell, 2014).

In addition to the above problems, individuals pursuing or graduating with STEM majors are reported to not possess the skills adequate to perform 21st century STEM jobs. An oft-cited 2012 report on STEM education by the United Kingdom's House of Lords notes that students entering college do not possess the skills they need to succeed in STEM subjects – for example, these students do not possess the math skills needed for first-year college math courses (House of Lords Select Committee on Science and Technology, 2012). If students are entering college unprepared to handle STEM challenges, it does not appear that colleges are providing them this preparedness; a 2014 paper presented at IEEE's Global Engineering Education Conference notes that STEM graduates "lack the numeracy skills needed to succeed in the workplace" (Gross, 2014). The effect of this can be seen in another statistic reported by the aforementioned CBI / Pearson report: 48% of firms report having their workers undergo "basic remedial training" in literacy, numeracy, and technical skills (CBI, 2013).

These statistics, and many more like them, contribute to the widespread belief that the United States and other countries are entering a "STEM crisis", and that immediate action must be taken to avert this.

1.1.3. The STEM Crisis is a CS Crisis

As mentioned above, STEM stands for "Science, Technology, Engineering and Math". Career fields designated as "STEM jobs" have been noted to include "medicine and dentistry; … biological sciences; veterinary science, agriculture and related subjects; physical sciences; mathematical sciences; computer science; engineering; technologies; and architecture, building and planning" (House of Lords Select Committee on Science and Technology, 2012).

It might be believed that the "STEM crisis" affects all these fields, and to some extent it does. However, inspection of the data reveals that the STEM crisis is really a computer science crisis. The job growth, current and upcoming, is overwhelmingly in CS-related jobs; there are broad trends that are bringing a need for CS abilities to jobs across the market (not just STEM!); and despite all this, CS enrollment and graduation are on the decline.

Let us first consider the aforementioned explosive job growth in STEM fields. According to a report by the Bureau of Labor Statistics, between 2010 and 2020 62% of all newly created jobs (not just newly created STEM jobs, but *all* newly created jobs) will require some CS skills. This same report also notes that by 2020, 50% of all STEM jobs will be CS related[1] (K-12 Computer Science Education: Unlocking the Future of Students, 2012). This trend is reiterated by Stanford University's Elizabeth Stark, who notes that

[1] The BLS report cited here notes that their definition of STEM excludes jobs in the medical profession; if these jobs were included, these number might be lower. However, it could also be argued that, with the increasing computerization of the medical fields, CS skills will be just as important in medicine as elsewhere.

"[b]y 2018, there will be nearly three times as many job openings requiring computer science knowledge [as] qualified applicants" (Stark, 2013). The CS flavor of the STEM crisis is summarized by Code.org's Hadi Partovi who, relying on data from the BLS and NSF, declares that "[c]omputer science is the only STEM field where there are more jobs than students" (Partovi, 2014).

Looking beyond STEM jobs, we can find trends and forces that are quickly making computer science skills an important factor – perhaps even a necessity – across all industries. As contended by Carnevale's STEM report, as well as Jeannette Wing, in her seminal Computational Thinking article, computers have invaded every industry, not just the hard sciences but the social sciences and even the humanities; we now need individuals to work with these computers (Carnevale, STEM: Science, Technology, Engineering, Math, 2011) (Wing J. M., Computational Thinking, 2006). This demand for STEM skills across the workforce is further evinced by the fact that, in the UK, almost half of all STEM graduates end up taking a job in what is considered a non-STEM field (House of Lords Select Committee on Science and Technology, 2012).

Tyler Cowen, an economist and professor nominated by *The Economist* as one of the most influential economists in the last decade[2], wrote *Average is Over* in 2013 on precisely this subject. In this book he concludes that the future of all industries will involve massive integration with computers (he goes further and predicts that we will see similar integration with our personal and social lives) (Cowen, 2013). *Qua* Cowen, these

[2] http://www.economist.com/blogs/freeexchange/2011/02/economics

trends are inevitable, and he declares that "[w]hether we will remain a middle class society or not depends firstly on how many people will prove to be effective working with intelligent machines" (Cowen, 2013).

From this information we can conclude that the upcoming need for STEM is predominantly a need for CS. Yet despite these upcoming needs, computer science education has been on the decline. Multiple sources report that CS enrollment is decreasing (Hu, 2011) (Carter, 2006). Other sources indicate that students are losing interest in pursuing CS majors (Wagstaff, 2012) (Blum, 2007).

Taken together, the high demand for CS skills, paired with the decrease in student pursuit of CS as a major, leads to the conclusion that the STEM crisis is in reality a CS crisis, and that – while we should not ignore the other STEM fields – CS is the one we may want to focus on addressing the most.

1.1.4. A Response to the STEM Crisis

It is important to acknowledge that not everyone accepts the existence or severity of the STEM crisis. In an influential article for the IEEE, Robert Charette argues that misinterpretations of job data, incorrect or inaccurate estimations, optimistic predictions and other errors are causing many to believe in a crisis when there is none (Charette). According to Charette – and others who argue among similar lines – there is no impending shortfall of engineers which requires drastic action to correct.

Despite not believing that the future will be rife with unfilled STEM positions, Charette does share common ground with many of those cited above when he declares that "everyone needs a solid grounding in science, engineering, and math. In that sense, there is indeed a shortage – a STEM knowledge shortage" (Charette).

Whether one agrees with the CS shortfall predictions or not, there does seem to be broad agreement that certain aspects of STEM – which, as we argue above, are really certain aspects of CS – are becoming necessary skills for all individuals, whether they work in a STEM field or not.

In the next section we will argue that these skills overlap considerably with a set of skills currently rising to prominence in CS education, which are collectively known as "computational thinking".

1.2. COMPUTATIONAL THINKING

Computational thinking refers to a skill, a set of skills, or even an entire paradigm of thought, depending on the source and the way in which the term is being used. The term "computational thinking" and the ideas it represents are not new, but it has risen to considerable prominence following a 2006 IEEE article by Jeannette Wing, the President's Professor of Computer Science at Carnage Mellon University. This article, appropriately entitled "Computational Thinking", launched a discussion within the computer science community which continues to this day.

In this section we will analyze many aspects of this discussion, beginning with an attempt to define exactly what computational thinking is. We will consider some specific components of computational thinking, and review why computational thinking is important both for CS students and for everyone. We will conclude this section with a discussion of the difference between computational thinking and programming.

1.2.1. The Definition of Computational Thinking

In order to discuss computational thinking, we must first get an idea of what it specifically is. Jeannette Wing is responsible for beginning the ongoing discussion of computational thinking, and the most concise definition of computational thinking she provides is as follows: "Computational Thinking is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing unit" (Wing J. M., Computational Thinking, 2006) (Wing J. M., Computational Thinking: What and Why?, 2010). By this definition, computational thinking is a critical thinking and heuristic reasoning skill, but one specifically involved with formulating both problems and solutions in a computable form.

Despite the name, computational thinking does not literally mean "thinking like a computer" (Wing J. M., Computational Thinking: What and Why?, 2010) (Gouws, 2013) . Wing and others note that "thinking like a computer" implies *mechanization of thought*, while computational thinking is *heuristic*, and perhaps even *creative*, in its approach to transforming and solving problems (Wing J. M., Computational Thinking, 2006).

Computational thinking does rely on the fact that a problem, ultimately, must be implemented in a mechanical setting (a computer), and the solution must be output from this mechanical setting; but between the problem statement and its implementation in a program lies a wealth of heuristics, creative thought and transformation, and it is that ground between problem statement and compiled program which computational thinking covers.

1.2.2. Components of Computational Thinking

In the previous section we provided a brief high-level definition of computational thinking. In this section we will describe four specific (but still somewhat high-level) components of computational thinking. These are not the only components of computational thinking, but their frequent inclusion in computational thinking related literature suggests that they are the most important ones.

The four components are as follows:

1. **Ability to Read and Understand Algorithms**

   As computational thinking is fundamentally involved with the creation and evaluation of algorithms, the ability to read and understand algorithms is a crucial prerequisite of computational thinking. This skill involves understanding an algorithm *qua* algorithm – understanding it as an entity separate from the specific context that it operates in.

2. **Ability to Engage in Abstraction**

   This is commonly cited as by far the most fundamental skill involved with computational thinking, and it is at the base of most if not all other skills involved. The ability to engage in abstraction involves (but is not limited to) the ability to generalize, compartmentalize, move between levels of abstraction, and understand and apply recursion.

3. **Ability to Decompose a Problem into Solvable Processes**

   This skill may be roughly thought of abstraction applied to the problem domain. It involves the ability to identify the pertinent aspects of a problem, the ability to transform a problem from one domain to another, and the ability break down a problem into subproblems. It also involves the ability to identify which computable processes can solve problem components, and the ability to combine these processes to create algorithms.

4. **Ability to Evaluate the Quality of a Solution**

   If computational thinking deals with formatting and transforming problems, and creating solutions, that are computable, then the ability to evaluate the quality of the resulting algorithm ought to also be part of the skillset. This component involves not just evaluating the correctness of an algorithm, but also its elegance, cleanliness, optimization, generality, and reliability, among other things.

These items are described variously in (Hu, 2011) (Wing J. , 2006) (Wing J. M., Computational Thinking: What and Why?, 2010) (James J. Lu, 2009). A more comprehensive treatment of these items will be provided in Section 2.

1.2.3. Computational Thinking is not Programming

In a list of misconceptions about computational science compiled by the CSTA, the very first misconception is the idea that computer science should equal programming (CSTA Curriculum Improvement Task Force, 2006). If computational thinking is at the root of computer science, then it should be similarly believed that computational thinking and programming are two different skillsets. This point is forcefully made by Wing, who declares quite simply that "[c]omputational thinking is not computer programming" (Wing J. M., Computational Thinking, 2006).

Before we proceed, we ought to properly define programming as precisely as we have defined computational thinking. In this thesis, "programming" refers to the act of transforming an algorithm from an idea into some symbolic notation which can be executed on a computer. This can include writing code in Java or C, or it can include writing an algorithm in a visual programming environment such as Scratch[3] or Microsoft VPL[4]. In a more abstract sense, it can even refer to writing natural language step-by-step instructions for use by a human "computer". The important part is that the development of the algorithm precludes the practice of "programming" – put simply, programming is

[3] http://scratch.mit.edu/

[4] https://msdn.microsoft.com/en-us/library/bb483088.aspx

algorithm transcription. Others refer to this act as “coding”; in this thesis, “coding” and “programming” will refer to the same act that we have just defined.

If computational thinking and programming are two separate skillsets, how do they relate? It was noted previously that computational thinking is the ground between a problem definition and the implementation of an algorithm. Programming is involved only in the implementation stage – in other words, programming begins where computational thinking ends. Programming, when thought of this way, is a *rote*, or *mechanical* process; this contrasts with computational thinking, which is a *creative, heuristic* process (Wing J. M., Computational Thinking, 2006). Instead of thinking that they are equivalent, computational thinking ought to be thought of as the “parent” of programming – computational thinking skills produce a computable algorithm, while programming implements the algorithm in solid form.

The misconception that computational thinking and programming are the same thing can be harmful to student education. Wing warns that in teaching computer science “we do not want people to come away thinking they understood the concepts because they are adept at using the tool” – in other words, only teaching students how to program, combined with the misconception that programming and computational thinking are the same thing, can result in students who believe they have mastered skills that they really haven’t (Wing J. M., Computational Thinking and Thinking about Computing, 2008). Learning how to use a programming language is like learning how to use a tool; learning

computational thinking, on the other hand, creates students who are "not merely tool users but tool builders" (Valerie Barr, 2011).

This is an important point to make and will be used to support evaluations of computer science education, as well as decisions in creating a new computational thinking-based educational system, later in the thesis.

1.2.4. Computational Thinking is Important for CS Students

Having defined computational thinking, we may now consider its utility for computer science education. Unsurprisingly, this utility is argued to be quite high.

A joint report by the ACM and CSTA notes that "[c]omputer science education is strongly based upon the higher tiers of Bloom's cognitive taxonomy, as it involves design, creativity, problem solving, analyzing a variety of possible solutions to a problem, collaboration, and presentation skills" (Wilson, 2010). Design, creativity, and problem solving are exactly what computational thinking teaches; if this is true, then a strong foundation in computational thinking skills will prepare students to learn the higher levels of computer science. This is corroborated by Mohtadi, who notes that when computational thinking skills are taught systematically, students are able to internalize and gain a deeper understanding of mathematical and programming / engineering concepts very quickly (Mohtadi, 2013). Additionally, James Lu, a professor at Emory University's Mathematics & Computer Science department, declares that computational thinking education prepares students to learn programming (James J. Lu, 2009).

To sum it up, computational thinking skills are simply "indispensable in [the] modern engineering practice" (Mohtadi, 2013), and are therefore highly important for all students of that practice to learn.

1.2.5. Computational Thinking is Important for Everybody

While computational thinking has considerable utility for computer science students, the benefits of computational thinking are usually argued to be much broader: Wing notes that "[i]f computational thinking will be used everywhere, then it will touch everyone directly or indirectly" (Wing J. M., Computational Thinking and Thinking about Computing, 2008). In other words, these skills are not just beneficial for computer scientists, they are beneficial for everybody.

A commonly argued point is that computational thinking education teaches problem solving skills and critical thinking skills that are invaluable to the modern world (Stark, 2013) (Carey, 2010) (Wilson, 2010) (Wing J. , 2006). Steve Jobs describes it as a liberal art that teaches people how to think (Jobs, 1995). Jeannette Wing and others have gone so far as to declare it the "4th R", along with the traditional "3 R's" of reading, writing, and arithmetic (David Barr, 2011) (Wing J. M., Computational Thinking, 2006). These arguments all hold that computational thinking is a fundamental skill that everyone ought to have familiarity with.

Many of the articles that examine computational thinking often investigate its utility in non-STEM career fields. The idea that the fundamental skills of computer science could

be valuable in career fields outside of computing is not a new one; Donald Knuth made the argument in 1985 that computational thinking overlaps with thought patterns used in other careers (Knuth D. , 1985).

The most common career area said to benefit from computational thinking skills is the sciences, a claim that is not surprising when one considers the enormous role that computers now play in scientific work. Wing notes that computational thinking has, in recent years, "become the 'third pillar'" of scientific research, "along with theory and experimentation" (Wing J. M., Computational Thinking: What and Why?, 2010). However, the sciences are not the only career fields said to utilize and benefit from computational thinking.

Alan Bundy notes that computational thinking is influencing research across all disciplines – not just the sciences, but also the humanities (Bundy, 2007). Barr and Stephenson show that computational thinking concepts can be utilized even in social studies and language arts fields (mostly having to do with big data) (Valerie Barr, 2011). In her Computational Thinking paper, Wing lists many of the specific careers that computational thinking already influences:

> Computational thinking has also begun to influence disciplines and professions beyond science and engineering. For example, areas of active study include algorithmic medicine, computational archaeology, computational economics, computational finance, computation and journalism, computational law, computational social science, and digital humanities. Data analytics is used in training Army recruits, spam and credit card fraud detection, recommendation and reputation services, and personalizing coupons at supermarket checkout. (Wing J. M., Computational Thinking: What and Why?, 2010)

This section and the one preceding it make the argument that computational thinking is a highly valuable skill not just for computer scientists but for everyone. If this is the case, then it follows that computational thinking is a subject that ought to be broadly and universally taught.

1.3. THESIS OUTLINE

In Section 1.2 we established the importance of introductory computational thinking education. Our recognition of this importance motivated us to create a new educational system to teach introductory computational thinking. Our new system is made up of a software tool and a curriculum, both of which were designed by utilizing lessons learned through the evaluation of other educational systems and methods of teaching introductory computer science. This new system is called “Genost”. This thesis is dedicated to introducing Genost, explaining the thought that went into its development, and describing our testing to determine whether Genost effectively teaches computational thinking. We will divide the remainder of this thesis up into five sections.

In Section 2, we will perform a review of introductory computer science education and the systems that are used as part of introductory CS education, in order to determine the effectiveness of both traditional and newer educational systems in teaching computational thinking skills. This review will be divided up into four major sections.

First, in Sections 2.1 and 2.2, we will perform an expanded investigation into computational thinking in education, and the four specific components of computational thinking that we mentioned earlier. Second, in Section 2.3, we will review current

practices in introductory computer science education in the United States. Third, in Section 2.4, we will review some of the new computer science educational software systems and investigate their usability, educational value, and the degree to which they teach computational thinking. Finally, at the end of Section 2.4, we will review the lessons learned from the previous sections, and based on these lessons describe the qualities that an ideal computer science educational software system might have.

In Section 3 we will describe the software and curriculum that comprise our new Genost system. In this section we will describe each part of Genost, along with our goals in developing this part, the ways we attempted to implement these goals, and our justifications for the goals and our implementation of them.

In Section 4 we will describe the two tests that we performed of the Genost software and its ability to teach computational thinking skills. We will describe the recruitment criteria, time allotted, test environment and the data collected from each test.

In Section 5 we will present the results of the two tests, and our analysis of these results.

Finally, in Section 6 we will conclude the thesis, and present our ideas on future improvements to the Genost software.

2. REVIEW OF COMPUTER SCIENCE EDUCATION

In this section we will perform a review of introductory computer science education, focusing on the quality of both traditional and newer forms and methods of introductory education, and the degree to which these systems teach computational thinking. This section will be divided into four subsections.

- The first section will be a deeper investigation into computational thinking, focusing first on whether computational thinking skills are broadly teachable. Following this, we will perform an expanded look at the four major components of computational thinking described earlier and an investigation into how these four components ought to be taught as part of introductory computer science education.

- Second, we will perform a review of introductory computer science education in American high schools and colleges, focusing on the degree to which these classes teach computational thinking. We will also look at the general poor performance of these educational programs, and consider some of the reasons that these programs may be performing poorly.

- Third, we will review many newer educational software systems, like Alice[5] or Scratch, which are designed to be used in introductory computer science

[5] http://www.alice.org/index.php

education. This review will consider the general quality of these systems, their pros and cons, and what takeaways we can glean from our evaluation of these systems.

- Finally, we will reflect on the lessons learned from the preceding sections, and using these lessons, discuss the composition of the ideal introductory computer science educational system focused on teaching computational thinking.

2.1. EDUCATIONAL GOALS OF INTRODUCTORY COMPUTATIONAL THINKING EDUCATION

As we have argued in Section 1.2, computational thinking is an important skill for both computer scientists and individuals in general. Computational thinking is, however, a very rich subject, and it is not realistic, or necessary, to teach every aspect of it in full depth as part of introductory education. For introductory education, only the most fundamental and important concepts ought to be taught. This section will investigate the educational objectives that might be involved in teaching introductory computational thinking education.

2.1.1. Is Computational Thinking Teachable?

Before beginning to consider specific educational goals in introductory computational thinking education, we ought to first consider whether computational thinking skills are in fact able to be effectively taught in the first place. Most authors that write about computational thinking and education, such as Jeannette Wing, Valerie Barr or Chris

Stephenson, appear to assume that computational thinking skills are in fact teachable. However, a relatively well-cited paper by Saeed Dehnadi and Richard Bornat appears to bring this assumption into question.

Bornat and Dehnadi's paper, written in 2006, describes an attempt to devise a test by which students could be separated into two groups: those who had an "aptitude" for programming and those who did not (Dehnadi, 2006).This paper claims to have discovered such a test, which consists of providing an exam full of simple programming questions to students, and then grading these not based on actual correctness, but the degree to which student answers display a "consistent mental model" (to use a phrase from the paper). This paper defines a consistent mental model as a model of how a program should execute that is consistent across multiple programming questions (utilizing different programs per question). Dehnadi and Bornat directly correlate the degree to which this consistency is displayed with ultimate performance in introductory computer science classes.

This result is notable for us because the "consistent mental model" discussed by Dehnadi and Bornat sounds very much like a computational thinking skill. Dehnadi and Bornat argue that students displaying a "consistent mental model" have internalized the crucial idea that computers are literal machines, that they execute their algorithms the same way every time, regardless of the input or context. This appears to relate very closely to items 1 (Ability to Read and Understand Algorithms) and 2 (Ability to Engage in Abstraction) discussed in Section 1.2.2.

It is worth noting that Dehnadi and Bornat's paper was not published in a peer-reviewed journal[6]; despite this, it garnered a respectable number of citations in its draft form[7], many of them supportive (see (Robins, 2010) for an example of this). This reaction may be explained by considering the following notion: if the "programming aptitude test" is viable, then this means that the computational thinking skills required to learn and work as an effective computer scientist are, to some degree, innate, or at least not teachable.

What are we to make of this? Is computational thinking, to some degree, not teachable? This conclusion, and the paper that ventures it, is treated with skepticism by Alan Kay, who posits that Dehnadi and Bornat "could be right, but there is nothing in the paper that substantiates it" (Kay, 2008). Kay describes similar work in introductory science classes which found that, despite a pretest having the apparent ability to predict students grades in these classes, students could be taught skills to improve their performance on this pretest that also resulted in higher performance in the class itself (Kay, 2008).

More notable than Kay's commentary, however, is the fact that Bornat has retracted the paper (Bornat, 2014). In this retraction Bornat declares that, upon further investigation, the "aptitude test" could to some degree predict pass/fail in Bornat's programming class, but could not predict performance beyond that; more importantly, he notes that this predictive phenomenon does not divide students into those who can and those who

[6] http://retractionwatch.com/2014/07/18/the-camel-doesnt-have-two-humps-programming-aptitude-test-canned-for-overzealous-conclusion/

[7] https://scholar.google.com/scholar?cites=887892586020755938

cannot program. Putting it quite bluntly, he declares “Dehnadi didn’t discover a programming aptitude test” (Bornat, 2014).

The overwhelming assumption is that computational thinking skills are teachable. Bornat’s retraction removes a possible challenge to this consensus. We will proceed from here in agreement with this assumption, that computational thinking is a teachable skillset. The next question that must be confronted, then, is what skills must be taught.

The next four sections will each be dedicated to discussing the four aforementioned skills that are part of and fundamental to the computational thinking skillset. To review, these skills are:

1. Ability to Read and Understand Algorithms
2. Ability to Engage in Abstraction
3. Ability to Decompose a Problem into Solvable Processes
4. Ability to Identify the Quality of a Solution

As previously mentioned, these skills are not necessarily the only skills involved in computational thinking, though we argue that they are the most important ones. These skills are also not perfectly discrete or separate from one another: they intertwine, and are involved with one another. It may be better to think of these as facets of computational thinking, different ways of looking at a unified whole.

For each skill or facet, we will investigate what the ability is, why it is important to computational thinking, some specific instances in which this skill is exercised, and any other pertinent notes.

2.1.2. Ability to Read and Understand Algorithms

The ability to read and understand algorithms is defined as follows: students possessing this skill are able to read an algorithm encoded in some form (plain English, pseudocode, or some programming language) and "translate" the algorithm from its encoded, contextual form into a more general, cognitive set of processes. In order for students to properly claim they have understood an algorithm, they must mentally grasp the "idea" behind it.

This is an important skill to possess and is absolutely fundamental to computational thinking (James J. Lu, 2009). If computational thinking focuses on *creating* algorithms, then in order to learn computational thinking students first must be able to *read* algorithms. In this way, reading algorithms is the basic literacy of computational thinking, and serves as a prerequisite for all other computational thinking skills. This ability does not come naturally to everybody – the CSTA notes that novice students tend to not understand algorithms, or to treat them as inscrutable standalone processes that magically work "right" (CSTA Curriculum Improvement Task Force, 2006). In order for students to ever create their own algorithms, they must have a strong ability to read existing algorithms and understand both the general "idea" behind the whole algorithm, as well as the "point" of each step.

Important components of this skill are listed below.

1. Students must have a firm grasp of what an algorithm actually is – and what it is not. The formal, scientific definition of "algorithm" is an open question (Buss, 2001); nevertheless students may be (and routinely are) taught a practical definition which consists of the following characteristics:

   - An algorithm is *finite* – it has a finite number of steps.
   - Each step of an algorithm is *clearly and precisely defined*.
   - An algorithm takes zero or more items as *input* and produces some *output*. The output produced is *directly related* to the input provided.
   - The steps of an algorithm are *effective* – they can be performed by the human brain in finite time.

   The definition above is taken from (Knuth, 1997). Using this definition, students should be able to understand the characteristics of an algorithm and identify steps in algorithms they are reading that violate the above conditions.

2. Students must have a solid, immutable understanding of the *literality* of algorithms and their execution. This is implied in the definition of an algorithm, but is important – and misunderstood – enough to warrant its own object. Students

absolutely must understand that algorithms do exactly what they say they do, and nothing more whatsoever. Authorial intent does not matter; only the code written down matters.

3. Students must understand the way in which algorithms execute – starting from a precise entrance point and proceeding step by step. Steps are never skipped, unless the algorithm skips them in a well-defined manner; steps are never repeated, unless the algorithm repeats them in a well-defined manner. Again, this is implied in the definition of an algorithm, but is commonly misunderstood and leads to much confusion.

4. Students should be able to read the symbolic representation of an algorithm and understand each individual step. In other words, whatever language represents the algorithm (English, pseudocode or code proper) students must be able to read that language. This literacy need not be exhaustive, but before being asked to utilize a specific concept in their own algorithms, students ought to be able to read that concept when it is encoded in a language.

5. Given the general understanding of the characteristics of an algorithm reported in 1, 2 and 3, and the ability to read a specific encoding of an algorithm in 4, students should be able to combine these skills and understand the "idea" behind a specific instance of an algorithm. This understanding should be achieved on many levels – students should understand the idea and purpose of each individual step,

the idea and purpose behind certain well-defined groups of steps, and the idea and purpose behind the algorithm as a whole.

Jeannette Wing notes that the understanding of algorithms is the most basic form of abstraction (Wing J. M., Computational Thinking: What and Why?, 2010). The abstraction she refers to is the ability to, among other things, separate the general idea of an algorithm from its context – for example, to see that a sorting algorithm will sort whatever data it is given, no matter the size or content, so long as this data is of the type the algorithm will accept.

2.1.3. Ability to Engage in Abstraction

The ability to engage in abstraction is a skill which, appropriately, must be understood somewhat abstractly. The most concise definition may be that given by Lu, who notes that abstraction is generalizing information and principles from specifics (James J. Lu, 2009). While this definition is formally accurate, it hides the richness of abstraction in computational thinking. Wing helps show this richness when she states that "[a]bstraction is used in defining patterns, generalizing from instances, and parameterization. It is used to let one object stand for many. It is used to capture essential properties common to a set of objects while hiding irrelevant distinctions among them" (Wing J. M., Computational Thinking: What and Why?, 2010).

The importance of abstraction to computational thinking is hard to overstate. Wing calls this the "most important and high level thought process" in computational thinking (Wing

J. M., Computational Thinking: What and Why?, 2010), and declares "[t]he essence of computational thinking is abstraction." (Wing J. M., Computational Thinking and Thinking about Computing, 2008). This importance stems from the fact that abstraction is, to some degree, present in virtually all other skills and modes of thinking which fall under the definition of "computational thinking." It is required to read and understand algorithms, and it is required at every step of algorithm creation.

Computational thinking is an inherently layered paradigm (Wing J. M., Computational Thinking and Thinking about Computing, 2008). Abstraction comes into play at all of these layers. Specific uses of abstraction at these different layers are listed below.

1. Abstraction is involved in the basic creation and understanding of what an algorithm is. To even understand a particular algorithm we must abstract, and realize that an algorithm is a set of instructions independent of the real world circumstances in which it executes. These instructions gain context and meaning once they are applied to the real world; but the algorithm itself need not depend on this context. (Wing J. M., Computational Thinking and Thinking about Computing, 2008) (Wing J. M., Computational Thinking: What and Why?, 2010)

2. Abstraction is required to understand the fundamental structures of a programming language outside of their specific implementations. For example, we must use abstraction to understand the idea of "loop" outside of any specific implementation thereof. It is clear that when encountering and solving a problem,

CS students must know what tools are available to them, how they can be combined; this is the abstraction we are talking about. (Gouws, 2013)

3. Abstraction is involved in the analysis of a problem that we wish to solve with an algorithm. We must abstract a problem out of its real world circumstances to some degree in order to begin breaking it down and solving it (Wing J. M., Computational Thinking: What and Why?, 2010).

4. Once a problem has been abstracted out of its circumstances, we must abstract still further when breaking this problem down into processes. At each point we abstract away the irrelevant parts of a problem and encapsulate what remains into a subproblem. This abstracting process is repeated, often multiple times, in the breakdown process. Abstraction is also used in building a program up into a complete algorithm, as we solve subproblems and combine these solutions into larger solutions, and eventually into a single algorithm (Wing J. M., Computational Thinking: What and Why?, 2010).

These are only a few of the layers that abstraction is used on. We believe that they are some of the most fundamental ones and the ones that introductory students ought to be explicitly taught to work on.

There are other concerns involved with the abstraction skill. Aside from simply being able to abstract, Wing notes, students should also know how to identify which abstraction

is best out of multiple options (Wing J. M., Computational Thinking and Thinking about Computing, 2008). Furthermore, in addition to being able to properly identify which information is important and generalizing it, students should also have the skill to identify information that is not as important, which can be ignored and abstracted away (Wing J. M., Computational Thinking: What and Why?, 2010).

2.1.4. Ability to Decompose a Problem into Solvable Processes

The ability to decompose a problem into solvable processes is, in essence, the general approach that one takes when designing an algorithm based on a problem statement. There are two important components to this skill: first, the decomposition ("break down") of the problem into subproblems, and second, the creation of solution processes and the combination ("build up") of these processes into a final algorithm. Both of these processes are iterative – one breaks down a problem into subproblems (abstracting along the way), and then breaks those subproblems down further, until one reaches a point where each subproblem may be easily modeled and solved. One then iterates back up, combining the solution processes into a final algorithm. Every step of this ability involves modeling: Hu notes that "[a] model allows transforming data from one representation to another to make the data better understood or more "easily" manipulated", and this is what we are doing as we both break down and build up (Hu, 2011).

The skill to decompose problems and solve subproblems with processes is fundamental to computational thinking. Wing notes in her initial computational thinking paper that "[c]omputational thinking is reformulating a seemingly difficult problem into one we

know how to solve, perhaps by reduction, embedding, transformation, or simulation" (Wing J. M., Computational Thinking, 2006). The modeling aspect of this is also important: Dave Moursund considers "developing models…of problems that one is trying to study and solve [as the] underlying idea" of computational thinking (Moursund, 2013). Michael Resnick notes that this iterative modeling and transformative process is central to creative thinking (Resnick, 2007). Finally, Mohtadi notes that this skill is important not just in computational thinking but in all engineering disciplines, declaring that the skills of "reformulating seemingly difficult problems, reduction, embedding, [and] transformation…are indispensable in modern engineering practice" (Mohtadi, 2013).

Like the ability to read and understand algorithms, or the ability to abstract, this is a general skill which must be taught (Hu, 2011). The items below are some specific ways that this skill must be taught.

1. Students must be able to initially model a problem: that is, to abstract it out of its specific circumstances and generalize it to the extent that it can begin to be broken down. A large part of this is simply choosing an appropriate model, which Hu notes is crucial for learning this skill (Hu, 2011).

2. After modeling the problem, students must transform or decompose it to the point where it is solvable using simple processes, using techniques Wing describes like "reduction, embedding, transformation, or simulation" (Wing J. M., Computational Thinking, 2006).

There are two separate classes of techniques that deserve further elaboration, which we will describe as “transformation” and “decomposition”:

a. *Transformation* refers here to converting or narrowing a problem from one form to another. It is a one-to-one process: one problem is *transformed* into a newer problem, usually by noting that if we can solve the transformed problem, then we can solve the original (in other words, we abstract the “real problem” away from the less important details). Challenges here include not just transforming the problem but transforming it *correctly*.

b. *Decomposition* refers to breaking a problem down into subproblem. This tends to involve identifying ‘submodels’ that, when put together, make up a single problem model. This is a one-to-many process: problems are *broken down* and one problem is turned into multiple, smaller problems which, when solved, can have their solutions put together to create a solution for the original problem. Challenges here include breaking down the problem in an intelligent, clean and well thought out manner.

3. Students must have the ability to select and build a solution process for a subproblem, once the subproblem is small enough to have a discrete solution applied to it.

4. Students must have the ability to combine two or more solution processes to make a “superprocess” – that is, a single algorithm that implements both subprocesses. This is very similar to the decomposition process and in practice is often a direct reverse.

This skill, like the others, is really a special form of abstraction; in this case, it is abstraction as relates to problems and problem spaces.

2.1.5. Ability to Identify the Quality of a Solution

The final skill we are considering brings the element of quality evaluation into the computational thinking skillset. Individuals possessing this ability are able to accurately evaluate various facets of quality in regards to the steps of the computational thinking process. It has been noted by Lu and others that this is most definitely a computational thinking skill (James J. Lu, 2009) (Orni Meerbaum-Salant, Habits of Programming in Scratch, 2011).

The importance of judging computational thinking products for quality is fairly straightforward: Donald Knuth declares on the seventh page of the first volume of his seminal Art of Computer Programming that "[i]n practice, we not only want algorithms, we want good algorithms” (Knuth D. , 1985). Wing also weighs in on the matter, noting that after solving a problem, we ought to ask: is our solution good enough? She goes on to describe that computational thinking involves “judging a problem not just for

correctness and efficiency but for aesthetics, and a system's design for simplicity and elegance" (Wing J. M., Computational Thinking, 2006).

One fundamental part of this ability is the recognition that a problem may have multiple, and sometimes infinite, solutions. Much like the understanding of what an algorithm actually is, the understanding that algorithms may have multiple solutions is very basic, yet also may not be understood by novices. Stephenson and Barr note this as a highly important skill for computer scientists (Valerie Barr, 2011).

After understanding that algorithms may have multiple solutions, the next question is: which of these solutions are the best (and how do we judge this)? This question accounts for the remainder of the ability to identify the quality of a solution.

There are many different parts of the algorithm creation process involved in computational thinking that can be judged for quality. These parts include (among others):

- The initial modeling of the problem statement.
- The breakdown of the problem into subproblems
- The solution processes created to solve these subproblems
- The combination of these subprocesses into a single solution algorithm.

Furthermore, there are many different kinds of quality that we can evaluate. These include (among others):

- The ability to determine whether a solution does, in fact, solve the problem.

- The ability to gauge whether a solution is optimized – is our implementation the best possible one for this solution?

- The ability to evaluate the cleanliness of the solution – is this solution free from extraneous steps, unnecessary actions, etc.? Is it easily understood by others?

- The ability to evaluate the generality of a solution – could it be easily extended, if needed? Could it be easily adapted to solve a similar problem?

- The ability to gauge the reliability of a solution – does it depend on many different assumptions? Can it break easily?

It is important to note that this skill does not ask for students to be able to judge quality objectively, or assign a cardinal value to the quality of a solution – such things are impossible. Students with this ability should simply be able to cogently and persuasively argue for the quality of a particular solution and its ordinal superiority (or inferiority, or equivalency) to other solutions.

2.1.6. Introductory Computational Educational Goals – Conclusion

The above four subsections 2.1.2 – 2.1.5 described some of the specifics of the four computational thinking goals which we and others have identified as important for introductory computational science education. We note again that these goals intertwine with one another and cannot properly be taught as separate, discrete ideas – instead, it might be more effective to teach them as part of a unified curriculum focusing on algorithm development.

In Section 2.1 we have provided a rough sketch of what fundamental computational thinking education should consist of. In the next section, we will briefly justify why computational thinking ought to be taught as an introductory class – that is, as the very first computer science class that a computer science student (or anyone) should take. Following this we will present a review of both traditional and newer introductory computer science education and analyze the degree to which these offer an education like the one described in section 2.1.

2.2. THE NEED FOR INTRODUCTORY COMPUTATIONAL THINKING EDUCATION

Up to this point we have spoken of introductory education – but the question may be asked: why ought the computational thinking skills we have described be taught in a student's first computer science class? This section will make the argument for dedicating a student's first computer science class to computational thinking – and only computational thinking.

2.2.1. Introductory Computer Science Education Ought to Involve Computational Thinking

As has been argued many times above, computational thinking is a foundational concept (Wilson, 2010) which is fundamental for learning higher topics in computer science – for example, programming (James J. Lu, 2009). If this is the case, then it ought to be introduced as being among the first topics a student learns in their CS career.

The National Academy of Engineering study "Educating the Engineer of 2020: Adapting Engineering Education to the New Century" that the skills that are a part of computational thinking should be taught as the first thing in the curriculum (Committee on the Engineer of 2020, Phase II, Committee on Engineering Education, National Academy of Engineering, 2005). Wing concurs, declaring that an introductory computational thinking course ought to be taught to all college freshmen, both computer science and non-computer science (Wing J. M., Computational Thinking, 2006).

Some argue that computational thinking ought to be introduced even earlier. Stephenson and Barr declare:

> It is no longer sufficient to wait until students are in college to introduce these concepts. All of today's students will go on to live a life heavily influenced by computing, and many will work in fields that involve or are influenced by computing. They must begin to work with algorithmic problem solving and computational methods and tools in K-12. (Valerie Barr, 2011)

Even if one does not believe that high school students *should* be introduced to computational thinking principles (whether they are interested in CS or not), if one accepts the arguments above then it follows that high school students who do take

computer science classes should be learning computational thinking, just as freshmen college students do.

There are additional reasons to support introductory CS courses containing computational thinking education. For example, the previously mentioned Executive Report to the United States President notes that "high-performing students frequently cite uninspiring introductory courses as a factor in their choice to switch majors" (President's Council of Advisors on Science and Technology, 2012). We will argue in section 2.3 that most existing introductory computer science education is focused mostly on learning programming languages; bringing computational thinking into these introductory courses may make these courses far more exciting and inspiring for students, solving to some degree the problem identified by the executive report.

These are the arguments for bringing computational thinking education into introductory computer science courses. The next section will argue that not only should these ideas be present in introductory education: they should be the only ideas present.

2.2.2. Introductory Computer Science Education Ought to ONLY Involve Computational Thinking (and not Formal Syntax)

In this section we will argue that introductory computer science education should consist only of computational thinking education, teaching skills such as the four described above. This section is in truth an argument against the current introductory education focus on learning a formal programming language like Java or C (for evidence of this

heavy focus, see section 2.3). There are many reasons to exclude learning a formal programming language in introductory computer science courses, which we will discuss in this section.

We are not arguing here that nothing which may be described as a "programming language" ought to be taught in introductory computer science – our own Genost solution, which we will describe below, as well as Alice, Scratch, or other newer educational software, feature simplified programming languages which can and perhaps should be used in introductory education. We are arguing against the use of formal, complex, text-based languages such as Java or C, which have traditionally been used in introductory education.

As has been noted in Section 1.2.3, computational thinking is not programming. These are two separate subjects. James Lu has argued that computational thinking education ought to come before programming education. He argues this by noting that programming serves a role in computer science similar to the role that proofs play in mathematics – that is, programming is a skill that opens the door to higher topics. Before students learn to write formal proofs in mathematics, they learn a host of simpler, more fundamental skills, including arithmetic and logic. By this analogy, programming ought to come only after students have learned the fundamental skills of computational thinking. Lu states that programming is therefore very important to computer science students – but it ought not to serve as their "first encounter" with the field (James J. Lu, 2009).

Malan states that teaching programming in introductory education may actually be harmful to student education. He states:

> In the first weeks of an introductory course (for majors or non-majors), too often do semicolons and their syntactical cousins delay, if not downright discourage, students' appreciation and mastery of more fundamental programmatic constructs (e.g. conditions, loops, variables, etc.) as well as logic itself. We daresay that languages like Java challenge students to master programmatic overhead before programming itself: students must become masters of syntax before solvers of problems. (David J. Malan, 2007).

The arguments presented above show that introductory computer science education ought to focus on computational thinking alone, and not attempt to teach a formal programming language. This argument may be taken further by noting that if computational thinking is to be treated as a general skill and taught to everyone, then introductory computer science education has even more reasons to be free from learning a programming language. Non-CS students taking an introductory computer science class will have a different set of needs, both short-term and long-term, to CS students. These non-CS students will have little or no need to learn Java. While they may need a skill that may be called "programming", this skill won't look like traditional programming (CSTA Curriculum Improvement Task Force, 2006). It may look instead more like the visual programming of Scratch or VPL, or something else entirely. This upcoming paradigm shift may already be seen in the advent of service-oriented programming, in which programmers, instead of coding a program from scratch, cobble an algorithm together using preexisting services (Yinong Chen, 2014) (W.T. Tsai, 2008). In fact, visual "drag and drop" software like

Oracle SOA Suite[8] already exist and are being used in the industry to allow individuals without traditional CS skills to build service-oriented algorithms (Yinong Chen, 2014). Individuals using the SOA suite do not need the ability to code in traditional languages, but they still do require computational thinking abilities.

In light of this, we declare that the purpose of teaching non-CS students computational thinking is not to teach them to "think like a computer scientist". Rather, as stated by Barr and Stephenson, "the ultimate goal should [be] … to teach them to apply these common elements to solve problems and discover new questions that can be explored within and across all disciplines" (Valerie Barr, 2011).

The statements presented in this section argue that introductory computer science education should consist of computational thinking, and nothing else. In the next two sections, we will investigate both traditional and newer forms of introductory computer science education, with a focus on the degree to which they offer computational thinking. The traditional systems will be found to offer almost no computational thinking education; and as a result, we will show, they have very poor results. The newer systems are mixed in their usefulness for teaching computational thinking education: some approaches are good, others are not so good. We will investigate these approaches and describe them accordingly.

[8] http://www.oracle.com/us/products/middleware/soa/suite/overview/index.html

2.3. REVIEW OF TRADITIONAL INTRODUCTORY COMPUTER SCIENCE EDUCATION

In “Computational Thinking - What it Might Mean and What we Might Do About It”, Chenglie Hu notes that “[w]e seem confident that whatever we teach in computing promotes computational thinking. But why is this true? We struggle to answer this question” (Hu, 2011). Do our current methods of teaching computer science in college and high school promote computational thinking? This section will examine that question.

In order to determine whether current introductory computer science education in the United States teaches computational thinking, we have performed a review of both introductory college computer science education and high school computer science education offered in the US. We will first present these reviews, describing our methodology and results. We will then argue that these reviews show that current methods of introductory computer science education do not effectively teach computational thinking skills.

Following this, we will describe the current poor results of computer science education. We will argue that these poor effects are attributable to lack of computational thinking content in introductory content, among other things.

2.3.1. Review of Introductory Computer Science Education in United States Colleges

In order to determine the degree to which introductory computer science education in college teaches computational thinking skills, we performed a review of introductory computer science classes taught at United States colleges. For this review, we investigated the curricula of these introductory classes for both the top 25 colleges overall, and the top 25 colleges for computer science, as ranked by US News[9].

2.3.1.1. Course Selection Criteria

For each college on both of these lists, we selected the courses that served as the introductory computer science course for this college's computer science major. Our selection criteria were as follows:

1. The course should be part of the track for a *computer science (CS)* degree. We did not consider introductory education for other computer-related degrees, such as computer science engineering (CSE), computer engineering (CE), computer information science (CIS), etc., if the introductory education for those degrees differed from the computer science degree.

[9] The listing for the top 25 overall colleges can be found at http://colleges.usnews.rankingsandreviews.com/best-colleges/rankings/national-universities/data, and was accessed on 11/29/2014. The listing for the top 25 colleges for computer science can be found at http://www.usnews.com/education/best-global-universities/search?country=united-states&subject=computer-science, and was accessed on 11/29/2014

2. The course should be part of the *bachelors of science* track. We did not consider introductory education for minor tracks, master's degrees, doctoral degrees, certificates, etc., if the introductory education for these tracks differed from the BS track.

3. The course should be *required*. We did not consider optional courses or electives.

4. The course should be taken in the *first year,* i.e. it should be what one would call a 100-level class (note that the designation "100-level class" does not necessarily mean the course number was itself between 100 and 199!) We did not consider higher level courses.

5. The course should be presented as a *fundamentals* course. We did not consider courses which were presented as specialized courses.

6. When colleges had multiple tracks for a bachelor's degree in computer science, we tried to choose courses that were required for all tracks. Usually, the same first year introductory course was taught between all the tracks. When it was not, we either investigated all the introductory courses, or chose the course that appeared to be the most general-purpose.

The selection of courses was done by reviewing information publically available on the college's website. Most colleges had between one and three courses selected for review.

For each course, we attempted to find a curriculum, syllabus, or course calendar available online. If these resources were not available online, or they were out of date (all resources used were from 2010 or later; most were from the most recent provision of the course, in Fall 2014) we contacted the professor responsible for the most recent provision of the course, and requested access to a syllabus or curriculum. In cases where we did not hear back from the professor, we skipped the review of that course.

2.3.1.2. Course Information / Data Collected

For each course that we were able to retrieve a syllabus, curriculum or calendar for, we reviewed the information provided in this syllabus and recorded the following:

1. A brief description of the course overall
2. A brief description of how the course taught computational thinking skills
3. The primary tool or language used in the course (e.g. Java, Scratch, etc.)
4. A numeric rating, ranging from 0 – 5, of the degree to which the course taught computational thinking skills.

The numeric rating described above used a scale of our own design. The scale description is as follows:

0) A zero rating indicates that the course is a survey of many different subjects or fields in computer science. Courses with this ratings attempt to provide an

"overview of the computer science career field", and does not attempt to teach either a language or computational thinking ideas.

1) A one rating indicates that the course is focused entirely on teaching a formal programming language, like Java or C. Computational thinking concepts are not specifically discussed.

2) A two rating indicates that the course is focused mostly on teaching a formal programming language, like Java or C. Some computational thinking ideas are discussed, but they are discussed solely within the context of the programming language being taught, and not as separate, language-independent concepts.

3) A three rating indicates that the course is a mixture of learning a formal programming language, and learning computational thinking ideas in abstract. Newer software, such as Scratch or Alice, may be used.

4) A four rating indicates that the course has the majority of its focus on learning computational thinking ideas in abstract. A formal language may be involved, but learning this language is not the focus of the course. Newer software, such as Scratch or Alice, may be used.

5) A five rating indicates that the course is entirely focused on teaching computational thinking ideas, and students are not asked to learn any formal

programming language syntax. Newer software, such as Scratch or Alice, may be used.

Some of the data collected, most importantly the numeric rating, may be found in Appendix A. We have also downloaded copies of all curricula that we utilized as part of this study, and these are available upon request.

2.3.1.3. Course Review Results

The results of this survey can be seen in the table below.

Table 1

Review of introductory computer science education in US colleges

	Top 25 Colleges Overall	**Top 25 Colleges for Computer Science**
Number of Colleges Reviewed	26	25
Number of Courses Selected	50	42
Number of Courses Selected	43	38
Number / Percent of Courses Rated 0	1 / 2.33%	0 / 0%
Number / Percent of Courses Rated 1	16 / 37.21%	11 / 28.95%
Number / Percent of Courses Rated 2	19 / 44.19%	18 / 47.37%
Number / Percent of Courses Rated 3	6 / 13.95%	8 / 21.05%
Number / Percent of Courses Rated 4	0 / 0%	0 / 0%
Number / Percent of Courses Rated 5	1 / 2.33%	1 / 2.63%
Average Rating	1.79	2

We actually reviewed 26 individual colleges off of the top 25 overall list, because two colleges were tied for 25th place. From these 26 colleges, 51 courses were selected and reviewed, with a rounded average of 2 courses selected per college. Of those 51 courses, we were unable to retrieve recent curricula for 7 of them.

There were no ties for the top 25 colleges for CS, and so only 25 colleges were reviewed. From these 25 colleges, 43 courses were selected and reviewed, with an average of ~1.6 courses selected per college. Of those 43 courses, we were unable to retrieve recent curricula for 4 of them.

The average rating for the top 25 colleges overall was 1.79. The average rating for the top 25 colleges in computer science was 2.

The following table summarizes the tools that were used by these classes. Note that some courses used multiple tools.

Table 2

The tools used by the reviewed colleges

	Top 25 Colleges Overall	Top 25 Colleges for Computer Science
Formal Languages		
Java	14	14
Python	12	10
C++	7	8
C	4	2
HTML / CSS / JavaScript	3	3
LISP / Racket	3	0
PHP	2	2
OCAML	2	0
MATLAB	1	1
SCALA	1	0
Newer Educational Systems		
Scratch	2	2
Karel the Robot	1	1

2.3.1.4. College Review Conclusion

These results indicate three things.

1. **These colleges do not effectively teach computational thinking in their introductory computer science education.**

   The average ratings for both colleges were on the low end of the scale – 1.79 for the top 25 overall colleges, and 2 for the top 25 computer science colleges. The overwhelming majority of courses were rated as 0, 1 or 2, which indicates that computational thinking is at best taught within the context of a programming language (a 2 rating), or not taught at all (0 or 1). For the top 25 colleges overall,

83.73% of courses surveyed were rated with a 0, 1 or 2; for the top 25 computer science colleges, 76.32% of courses surveyed were rated with a 0, 1 or 2.

2. **Colleges overwhelmingly use formal programming languages in their introductory computer science education.**

   As can be seen in Table 2, the vast majority of tools used in these introductory courses are formal programming languages. Only 3 courses from either the top 25 overall colleges or the top 25 computer science colleges use tools that are not formal programming languages – two courses use Scratch, and one course uses Karel the Robot[10]. Of the languages used, Java is used the most among both sets of colleges, followed by Python and C++.

3. **The top 25 computer science colleges are only slightly better at teaching computational thinking than the top 25 overall colleges.**

   The average rating for the top 25 computer science colleges (2) is slightly higher than the average rating for the top 25 colleges overall (1.79) – but only slightly. The percentage of courses rated 1 is lower for the top 25 CS colleges than in the top 25 overall colleges (by roughly 8 percentage points), while the percentage of courses rated 2 and 3 is higher (by roughly 3 percentage points and 7 percentage

[10] http://karel.sourceforge.net/

points respectively). However, the number of courses rated 4 and 5 are exactly the same in both groups.

The tools used do not significantly differ between the two groups of colleges. There are some slight variations – for example, the top 25 overall colleges have more courses using C, while the top 25 computer science colleges have more courses using C++ – but the number of courses using the newer tools is exactly the same.

This evidence indicates that, for the top 25 colleges both overall and for computer science in the United States, computational thinking is not effectively taught. It is not effectively taught as an independent subject (if it is mentioned at all), and instead students are taught the syntax and rules of a formal programming language. Assuming that the top 25 schools are as good as or better than the remainder of colleges in the United States, this review indicates that there is a serious failure to teach effective computational thinking skills in college introductory computer science education is the US.

2.3.2. Review of Introductory Computer Science Education in United States High Schools

Unlike the review of introductory computer science education in college, we are not able to easily perform a formal review of all introductory high school computer science courses. These courses – and the institutions that offer them – are vastly more numerous than introductory college courses. Furthermore, virtually all high school computer science education would be classified as “introductory”, since by definition these courses

are likely to be the first computer science course given to the students taking them. The pool of individual courses to be reviewed is much wider.

In lieu of performing an exhaustive review ourselves, we will turn to the literature which has already, to some degree, done this. We will also review one of the more widely offered high school computer science courses: AP Computer Science. After looking at both these sources, we will again conclude that high school computer science education does not adequately teach computational thinking skills.

2.3.2.1. Review of Literature on High School Computer Science Education

We begin by noting that very little computer science is taught in US public high schools in the first place. According to Partovi, as of 2014 90% of public schools in the United States do not teach any computer science (Partovi, 2014). Even in schools that do teach it, students have little incentive to take it beyond personal interest – only 9 states count computer science as a mathematics credit, and only 1 state counts it as a science credit (6 states allow the district to determine what it will count as) (Wilson, 2010). Therefore, even before analyzing the course content itself, we can see that computer science education in high school is not at all widely taught.

We will now investigate the actual content of the little computer science education that is offered in American high schools. It is widely reported in sources such as (Wagstaff, 2012) that high school computer education focuses more on "skill-based aspects" of computing, such as keyboarding, OS-specific operational skills, word processing, spreadsheets, etc., as opposed to the algorithm creation skills of computational thinking.

To further investigate the contents of high school computer science education, we turn to the ACM's 2010 report *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age*.

This report describes three different levels of computer science education standards that the ACM and the CSTA have created. These three levels are intended for three different age groups, and are described as follows:

1. The Level I standards are intended for the K-8 age group. This set of standards focuses mainly on computer operation and awareness, and includes such items as "use standard input and output devices to successfully operate computers and related technology", "create developmentally appropriate multimedia products", "discuss basic issues related to the responsible use of technology and information", "exhibit legal and ethical behaviors when using information and technology", "develop, publish and present products using technology resource", etc. These Level I standards are focused much more on teaching computer operation than computational thinking skills, though there are some standards that offer very basic computational thinking, such as "develop a simple understanding of an algorithm, such as text compression, search, or network routing, using computer free exercises" (Wilson, 2010).

2. The Level II standards are intended for the 9th/ 10th grade age group. This set of standards focuses on introductory computer engineering and science concepts, and asks students to understand "principles of computer organization and the major components", "the basic steps in algorithmic problem solving", "the basic components of computer networks", "the notion of hierarchy and abstraction in computing", etc. Computational thinking skills are well represented in these standards (Wilson, 2010).

3. The Level III Standards are intended for 11th or 12th grade age groups. This set of standards continues the focus on computer engineering and science concepts, and includes standards such as "fundamental ideas about the process of program design and problem solving", "simple data structures", "fundamentals of hardware design", "the limits of computing", etc. Again, computational thinking is well represented here (Wilson, 2010).

The ACM uses these standards in their report and investigates the degree to which these standards are adopted by high schools across the United States. They report that, on average, each state has adopted 70% of the "skill-based" Level I standards. In contrast to this, the higher level standards – which focus far more on computational thinking skills – are adopted at much lower rates. Only 35% of Level II standards and 30% of Level III standards are adopted by a state on average (Wilson, 2010).

These are average ratings. Looking at the actual number of standards adopted by the states, we find that "there are 16 states with no model curriculum standards adopted at Level II and 22 states with no model curriculum standards adopted at Level III". Only 10 states adopt all Level II standards, and only 9 states adopt all Level III standards (Wilson, 2010). In other words, only 1/5 of the states have adopted 100% of the standards focusing on computational thinking, and there are more states that haven't adopted *any* computational thinking standards than those that have adopted all of them. Both these numbers and the average adoption rates described above indicate that high school CS standards mostly focus on computer operation instead of computational thinking.

In addition to considering the adoption of the standards by level, the ACM also considers the adoption of the standards by category. The report defines three separate categories, and divides the standards between them. The three categories are defined in the text as follows:

> *Concepts*: emphasize one of the 10 basic ideas that, at a high level, define modern computers, networks, and information...[e]xamples include computer organization, information systems, networks, digital representation of information, information organization, modeling and abstraction, algorithmic thinking and programming, universality, limitations of information technology, and societal impact of information technology.
>
> *Capabilities:* emphasize one of the 10 fundamental abilities for using computing to solve a problem…[e]xamples include the ability to engage in sustained reasoning, manage complexity, test a solution, manage faulty systems and software, organize and navigate information structures and evaluate information, collaborate, communicate to other audiences, expect the unexpected, anticipate changing technologies, and think abstractly about IT.
>
> *Skills:* "emphasize one of the 10 abilities to use today's computer applications in one's own work…[e]xamples include the ability to set up a personal computer; use basic operating system features; use a word processor and create a document; use a graphics or artwork package to create illustrations, slides, and images;

> connect a computer to a network; use the Internet to find information and resources; use a computer to communicate with others; use a spreadsheet to model simple processes or financial tables; use a database system to set up and access information; and use instructional materials to learn about new applications or features. (Wilson, 2010)

Organizing the standards along these categories allows us to see the problem even more clearly. First, consider the "concepts" standards, which overlap very strongly with computational thinking skills. There are 19 total "concepts" standards, but only 16 states have adopted more than half of them. Fewer than half of the states – 22 – have adopted even ¼ of the total number of "concepts" standards. 11 states have adopted only one standard, and 9 states have adopted only two standards.

Compare this to the "capabilities" standard, which does include computational thinking skills to a small degree, but mostly focuses on the reasoning, thinking and skills expected of a computer operator or IT manager. There are 19 "capabilities" standards, and only 13 states have failed to adopt at least half of these standards. 21 states have adopted at least ¾ of the standards, and 10 have adopted every single one.

Finally, consider the "skills" standards, which are entirely based on computer operation and do not contain any computational thinking. These skills are the most widely adopted of all: like the "capabilities" standards, only 13 states have not adopted at least half of the "skills" standards, 30 states have adopted at least ¾ of the standards, and 23 – nearly half – have adopted every single one (Wilson, 2010).

The ACM concludes that these findings indicate that high school computer science education is "focused almost exclusively on skill-based aspects of computing…and [has] few standards on the conceptual aspects of computer science that lay the foundation for innovation and deeper study in the field" (Wilson, 2010). In other words, computational thinking is very poorly represented in these curricula, and computer operation skills are the main focus.

2.3.2.2. Review of AP Computer Science

A traditional way in which US high schools have offered computer science education has been through the Advanced Placement or AP Computer Science course[11]. Of the aforementioned 10% of US schools that do teach CS, half of them teach AP Computer Science (Microsoft). The AP Computer Science course has a well-defined curriculum, and therefore considering its prominence in US high school computer education can serve as a useful indicator of the content of this education.

A review of the AP Computer Science curriculum makes it clear that the major focus of the course is learning Java. This curriculum rates a 2 on the scale described in Section 2.3.1.2 – the course's major focus is learning a formal programming language, and while some computational thinking concepts are discussed, they are discussed entirely within the context of Java, and not as abstract concepts in their own right. The National Science Foundation appears to concur with this assessment, declaring that the AP Computer

[11] http://media.collegeboard.com/digitalServices/pdf/ap/ap-computer-science-a-course-description-2014.pdf

Science course simply "focuses on programming skills. The course teaches students how to code in a specific language (Java)" (NSF, 2014).

Perhaps the best indicator of the degree to which computational thinking is absent from AP Computer Science can be found in the release – and the reaction to – a new AP course, "AP Computer Science Principles", which is explicitly stated to be designed as a computational thinking-centric alternative to the traditional AP Computer Science course (NSF, 2014). In the National Science Foundation's press release on the new AP course, Jan Cuny, the program director at the NSF for Computer Science Education and Workforce Development, states that "[t]his new course will broaden the appeal of computing to a wider group of students by focusing on the creative aspects of computing and computational thinking practices that enable students to be creators, not just users, of technology" (NSF, 2014). Elsewhere, Cuny notes that "[a]lthough [AP Computer Science Principles] does include programming, the course isn't programming-centric. Instead, it focuses on the underlying principles of computation including problem solving, abstraction, algorithms, data and knowledge creation, and programming" (Cuny, 2011).

The development of a new course to provide computational thinking education seems a clear indication that the existing course does not provide this. The AP Computer Science Principles course will launch in Fall 2016; until then, many US high school students are limited to the Java-centric AP Computer Science course. The unsuitability of this existing course to the needs of computer science students is perhaps indicated by the falling amount of students taking AP CS. The percentage of high school students earning credits

in AP Computer Science has declined from 1990 (25%) to 2009 (19%) (K-12 Computer Science Education: Unlocking the Future of Students, 2012).

2.3.2.3. High School Review Conclusion

The results from both the overview of the literature related to US high school computer science education, and a review of the AP Computer Science course, indicates that high school computer science education is rarely and inconsistently offered to high school students, and when it is offered, is primarily based on teaching computing skills such as keyboarding, OS-specific operational skills, word processing, spreadsheets, etc., as opposed to computational thinking skills. We can conclude from this that computational thinking is very poorly represented in US high schools.

This concludes our review of the degree to which computational thinking is taught in existing educational contexts. We will conclude Section 2.3 by reviewing the results of computer science education in the United States – that is, the degree to which students are successful in pursuing computer science education – and will argue that these results are quite poor. We will then argue that this poor state is attributable to the lack of computational thinking concepts being taught in introductory education.

2.3.3. The Poor State of Current Introductory Computer Science Education

Introductory computer science in the United States produces many poor results. This section will briefly review this poor state by examining three performance metrics from college introductory computer science. These metrics are: failure rates, attrition rates, and student ability to program.

2.3.3.1. Failure Rates

It is a commonly accepted idea that introductory computer science education has a high failure rate. However, only two papers appear to have actually attempted to investigate this assertion. These two papers find that failure rates are nontrivial, although they may not be as high as some think.

In 2007 Bennedsen and Caspersen solicited responses from the SIGCSE mailing list for information on the participants' school's pass / fail rates for introductory computer science education. A small number of schools responded (63) and reported a failure rate of 33%. The authors noted that due to the low response rate, as well as the potential motivation for those schools with low rates to not respond, this number may not be representative (Bennedsen, 2007).

A second attempt was made to consider the failure rates in 2014. Watson and Li, drawing inspiration from Bennedsen and Caspersen's paper, performed a very similar analysis, this time with 161 schools participating. Interestingly, the authors found an identical

failure rate of 33%. The authors also performed a study of failure rates historically, and have found that since 1969 the rates have not significantly decreased (Watson, 2014).

Both papers state that, in light of common claims that failure rates in introductory computer science education are astronomically high, a 33% failure rate is not that bad. However, it still represents 1 in 3 students failing their very first programming class – and this, paired with the observed failure of this rate to decrease over time, is cause for concern.

2.3.3.2. Attrition Rates

The next metric we will investigate are attrition rates for computer science, which are also popularly believed to be quite high. In this case, the popular belief appears to be correct.

Drew notes that 40% of all engineering students either change their major away from engineering, or drop out entirely (Drew). These numbers are corroborated by Beaubouef, who notes that computer science has an attrition rate of 30% to 40%. Beaubouef also notes that most of this attrition happens during students' freshman or sophomore years – that is to say, during their introductory education. Interestingly, Beaubouef hypothesizes that one major reason for these attrition rates and patterns may be low problem solving abilities in incoming computer science students, and notes that this deficiency is mostly in their "inability to form [an] algorithm in the first place" (Beaubouef, 2005).

Finally, we note that when compared to other subjects, computer science has one of the highest – if not the highest – drop out rates, at 27% (Computer science courses get highest drop outs - study, 2010).

2.3.3.3. Student Ability to Program

The final metric we will investigate is student ability to program. We note that this metric only applies to people who pass introductory programming – that is, make it through the relatively high failure and attrition rates – and therefore the numbers here represent the small amount of "successful" CS degree-seekers.

Despite introductory computer science education traditionally focusing on teaching students a programming language (as described in Section 2.3.1), students emerging from their first CS class are often not able to actually program in this language. The famous McCracken group study is the traditional example of this; published in 2001, it describes a simple test administered to 216 first year computing students across four universities. This test – which asks students to evaluate arithmetic expressions read in from a text file in either postfix, prefix or infix notation – is challenging, but should not be too difficult for a novice programmer. McCracken found that students scored an average of 21% on this test, a considerably low score. He also notes that many of the participants did not even finishing designing an algorithm, much less implementing it (Michael McCracken, 2001).

10 years later, Guzdial reports that these low numbers are still present. He describes an even simpler programming test administered to students at Yale in 1983, which resulted in a pass rate of only 14%. Guzdial reports that follow up studies and repetitions of this test (the latest repetition coming in 2009 when his article was published) have found similar results (Guzdial, Education: From Science to Engineering, 2011). These results indicate that, despite ostensibly being taught how to program in their first class, emerging students often cannot program at all.

From these three metrics, we can see that introductory computer science is in fact in a poor state. In the next section, we will argue that this poor state is attributable to a curriculum that is heavy on syntax and light on computational thinking.

2.3.4. The Reasons for this Poor State

We have established in sections 2.3.1 and 2.3.2 that introductory computer science education focuses mostly on teaching either computer operation, or the syntax of a formal programming language. This conclusion is shared by many other individuals. Malan, for example, has declared that learning syntax is the major focus of introductory computer science (David J. Malan, 2007). These sources, and our own studies above, lead us to the conclusion that computational thinking is not effectively taught in introductory education.

In section 1.2.4, we argued that computational thinking is highly important for computer science students, and that it should be considered a "prerequisite" to programming. Given this, its failure to be taught is questionable. A generous explanation for this failure might

be the belief that students already possess computational thinking skills. The CSTA notes that "educators may be inclined to believe that [basic concepts such as algorithm design] are trivial and therefore easy to understand" (CSTA Curriculum Improvement Task Force, 2006). If this is the case, then introductory computer science need not waste time teaching something students already know.

It is apparent, however, that computational thinking is not a trivial or natural belief. Cynthia Selby has argued that computational thinking does not come naturally to students (Selby, 2012). This same argument is made by Donald Knuth, who has argued as early as 1985 that, based on his experiences in teaching, only 2% of CS students naturally have the ability to "think algorithmically" (Knuth D. , 1985). One may be tempted to declare that, with the rise of widespread home computer use, Knuth's observation may no longer be valid, or at least his percentage may be higher. Despite popular beliefs, however, it is apparent that the "digital natives" also do not naturally possess computational thinking skills (Mitchel Resnick, 2009).

With these facts in mind, the poor state does not look surprising. Students are asked to utilize skills which, by and large, they do not possess, and are not taught. Malan notes that syntax-heavy classes are asking students to "master programmatic overhead before programming itself" (David J. Malan, 2007); Resnick notes that the syntax students are asked to learn is too difficult for them (Mitchel Resnick, 2009); both Wagstaff and the ACM's *Running on Empty* report states that programming classes are often far too

advanced for students (Wagstaff, 2012) (Wilson, 2010). This difficulty, then, understandably leads to frustration, forfeiture and failure.

Even students who make it through their introductory computer science course may still suffer from its failure to teach computational thinking. Aside from the continuing struggle to learn in an environment where prerequisite skills are not taught, the very fact that these skills are not taught may set up the wrong expectations for people. Malan and McCracken both note that the heavy focus on programming leads students to believe that these are the only skills they need to succeed in computer science (David J. Malan, 2007) (Michael McCracken, 2001). Failure, then, begets failure.

It is not necessarily the case that the lack of computational thinking is the *only* reason for the poor state of computer science. Other reasons have been proposed – for example, Resnick has noted that introductory projects are often very unengaging or uninteresting for students (consider the classic "Hello World" – a valuable first project, but also a very boring one for students expecting something more) (Mitchel Resnick, 2009). From what we have established, however, it seems likely that the lack of explicit computational thinking education is one of the primary reasons for the poor state of computer science education.

2.4. REVIEW OF NEWER INTRODUCTORY COMPUTER SCIENCE EDUCATION SOFTWARE

Mitchel Resnick, the leader of the MIT team which developed and maintains the Scratch software, declares that, traditionally, "programming was introduced in contexts where no one could provide guidance when things went wrong – or encourage deeper exploration when things went right" (Mitchel Resnick, 2009). In other words, traditional introductory programming education is poorly designed and ineffective. In an attempt to mitigate this, he and many others have developed a number of "newer" educational software systems which attempt to teach introductory computer science education in a simpler, easier, clearer, more guided manner, deeper, and often more fun manner.

These systems often state a goal or intent of providing computational thinking education. Many create their own language, or forego formal text-based languages altogether, in an attempt to shift the focus away from learning syntax and instead towards learning the deeper ideas of computer science.

In this section we will perform a review of many of the newer systems. For each system, we will provide a brief description of the system and at whom it is targeted at, a list of good features of the system, and a list of problems with the system. For each system we will also list takeaways or "lessons learned", and at the end of this section we will collect and summarize these takeaways. After investigating all of the newer systems, we will conclude this section with a collective review of the lessons learned and takeaways from

each system, and in doing so present the features that we believe ought to be present in an ideal introductory computer science educational system.

2.4.1. Methods and Themes in the Review of Newer Introductory CS Software

In the course of performing this review, we will examine each system and describe both the positive benefits of this system, as well as its weaknesses and problems. Our judgments, both positive and negative, will be informed by the field of literature on these systems, but also by our own evaluations with appeals to, among other things, our definition of computational thinking in section 2.1, as well as common sense.

Some common themes will appear in our evaluations. Before starting the evaluations proper, we will briefly describe each of these themes.

1. **Ease of Use**

   A frequent area that we will evaluate will be the ease of use of a particular system. Common sense tells us that a system ought to be as easy to use as possible for both the students using it, and the teachers teaching it, so long as this can be achieved without undue sacrifice of the system's educational value. If a system is difficult to use, then students may lose motivation and must spend more time learning to operate the specific system as opposed to the more general ideas the system is trying to teach.

Furthermore, a system that is highly complicated may very well turn away teachers who are not themselves very technical. With this in mind, the ease of use of a particular system is an area that we will frequently evaluate.

2. **Fun**

It has long been recognized in educational psychology that fun is an intrinsic motivator (Ormrod, 2012). When a student has fun working with a system, the student is motivated, pursues the task under his own initiative, stays engaged in the task, pays closer attention, shows creativity, and ultimately learns more from that system than he would were he not having fun (Ormrod, 2012). Because we are here concerned with introductory programming education, fun is especially important for us since the students we are working with may often be relatively young. The amount of fun that students have, or are likely to have, with a system is therefore an important item to evaluation.

3. **Adaptability**

Classrooms come in all shapes and sizes, and have different resources. Due to this, a system ought to be adaptable such that it can be used in as many classrooms as possible. This means, among other things, having a low price (or at least different levels of pricing), and not requiring specialized equipment or software (though this does not mean the system cannot optionally integrate specialized technologies!) The degree to which a system can be adapted for use in different classrooms is an area that we will often talk about.

This theme also includes the degree to which end users are able to customize the software and curriculum. A system that teaches a single set of lessons is not very adaptable; one in which the end user can customize which lesson it teaches is more adaptable, all things being equal.

4. Computational Thinking

A major concern in our review will be the degree to which a system is able to teach computational thinking skills. We will generally look at this theme in two different ways.

First, we will look at the software itself, and the degree to which this software has the *potential* to be used to effectively teach computational thinking. Note that the above three themes feed into this fourth theme – a system that is easier to use, more fun, and more adaptable is, all things being equal, more suited for computational thinking education than one that is difficult to use, dull, and rigid.

This being said, the *software* is not the source of the computational thinking education – the *curriculum* that utilizes the software is. Therefore, we would also attempt to evaluate the curriculum for these systems and determine whether it effectively teaches computational thinking. There is, however, a difficulty with performing this observation.

The difficulty with evaluating the curriculum for these newer systems is that most of the systems we are evaluating do not *have* curricula – at least, not "official" ones. Instead, many of these systems are released as software-only, and third party organizations – sometimes affiliated with the software developers, sometimes not – release curricula for them. To our evaluation, this is less desirable than the original developers creating and pairing an official curriculum with the software. A full system that includes both a curriculum and the software that will be used to teach it can be developed such that the curriculum needs influence the software design, something that is not possible when the software is developed in isolation. For these reasons, a system that contains an official computational thinking curriculum will be considered better than one that does not.

5. Play / Storytelling / Competition

As mentioned above, fun is a major motivator for student learning and will be considered a benefit in our reviews. However, many systems attempt to achieve fun in ways which we will not consider beneficial, and in some cases will consider problematic.

Many systems attempt to foster enjoyment or motivation in students by focusing on student play, storytelling, or competition, on the theory that this will increase student learning. These systems will, to some degree, eschew traditional teaching which attempts to directly convey a concept to a student, and instead give the

student freedom to play with the system, perhaps with some direction or help when they get stuck. Proponents of this technique claim that it will benefit student education. (Resnick, 2007).

It may be that undirected play is beneficial to student learning in some circumstances. However, the canon of educational research tells us that, for students to learn, basic familiarity with the topic they are learning about is required. Ausubel has noted that students require "anchoring ideas" to properly orient themselves on a topic such that new knowledge can be retained (Ausubel, 1968). If this is accurate, then it follows that play can only assist student learning when the student already has a set of basic "anchoring ideas" to orient themselves with. Play without this introduction, according to this theory, will not produce any meaningful long term knowledge.

Since we are concerned with creating an introductory educational tool here, the students participating in it will by definition have little to no orientation on the topic. With this in mind, we should be wary of organizing our system around creative play, and make sure that our curriculum always establishes the orienting ideas first before turning students loose to discover knowledge on their own.

For these reasons, we will consider systems that focus heavily on play, storytelling and competition as being weaker than those systems that concentrate on explicit computational thinking education first and foremost.

Having described some of our methods for evaluating the newer systems, as well as some of the general themes that will appear in our review, we will next describe the categories into which the systems we will review will be organized.

2.4.2. Overview of Newer System Categories

In order to provide some organization to the large list of newer systems we will be reviewing, we will classify these systems on two different metrics.

The first metric (which we will call "input") will describe the way in which programs are input into the system, and has two values: code-based, and drag and drop. The second metric (which we will call "testing") will describe the method and environment in which student programs are run, and again has two values: virtual worlds, and robots. Below, we will provide a brief description of what each of these four values mean. Additionally, because three of the four metrics (drag and drop, virtual worlds and robots) are often described as good or valuable features of educational systems, we will also recount general arguments for the educational benefits of these three characteristics.

1. **Code Based**

   This is the first value of the "input" metric. This value will apply when a system requires traditional text-based code to be typed out to create the program. This value is often found in some of the older systems that we will recount here. Some systems with this feature utilize a simplified version of an existing programming

language, while others create their own language entirely; this is done in an attempt to simplify the cognitive load of learning complicated grammar and syntax rules.

Because code-based input is the "default" for educational systems (as it is the same as the methods used in traditional systems) we will make no arguments for its educational benefits.

2. Drag and Drop

This is the second value of the "input" metric. This value will apply to systems in which students utilize a GUI to build a program by selecting, dragging, placing, and connecting discrete graphics representing parts of a program in a 2D plane. Most newer programs feature this feature. For obvious reasons, all drag and drop systems utilize a custom graphical language.

Drag and drop has many putative benefits. The major one is that drag and drop GUIs eliminate the possibility of syntax errors – the method of input prevents them by preventing the graphics to be arranged in a syntactically incorrect way. This removes the need for students to learn complex syntactic rules (Wanda Dann, 2009) (Karin Johnsgard, 2008) (David J. Malan, 2007) (Mitchel Resnick, 2009). Another commonly argued benefit is the potential guidance that drag and drop programs can provide – by making different graphics have different shapes, colors or other physical properties, the program can easily communicate through

sight alone which blocks are able to go together (Mitchel Resnick, 2009). For example, a graphic representing a logical comparison can be shaped in such a way that it fits inside a graphic representing a loop test, while a graphic representing a variable assignment statement can be shaped in a different way, visually indicating that the variable assignment does not go inside of the loop test.

3. Virtual Worlds

Virtual worlds is one value of the "testing" metric. This value will applies to the method in which students test their programs – virtual worlds systems are ones in which student programs are simulated through activities in a virtual world. This can involve an animation representing discrete objects being manipulated in accordance with the input program's commands.

The virtual worlds feature is a fundamental aspect of some newer systems, appearing in many of them, and it is often argued to be a primary benefit of these newer systems. There are many putative benefits to virtual worlds. These are listed below.

A primary benefit – and one of the most commonly argued – is that virtual worlds provide a "low floor, wide walls, [and] high ceiling". This phrase, which comes from Seymour Papert's *Mindstorms*, refers to three separate but related ideals for an educational system: "low floor" means that the technology should be easy to learn and get started with. "Wide walls" means that a large range of activities

should be possible within the technology. “High ceiling” means that the technology, while capable of very simple interactions, should also support highly complex interactions, and all those in between (Papert, 1993). Systems implementing virtual worlds commonly assert that they implement all three of these ideals (Mitchel Resnick, 2009).

A second, and perhaps related benefit is the abstraction that is capable within virtual worlds. These virtual worlds can be configured to involve only those factors that are pertinent to a particular example. For example, in a virtual world based on driving a robot around, one need not worry about keeping the robot driving straight, or the accuracy of its turns – as the robot is simulated, it will drive straight and turn accurately every time. Guziman notes that students learning computer science “don’t want or need to deal with the subtle shades of correctness” (Guzdial, Programming Environments for Novices, 2004) – virtual worlds allow students to avoid worrying about the tiny details in favor of the big picture.

There are other benefits to virtual worlds which are much more practical. Virtual worlds are easy to build, easy to configure, are not resource intensive, and are cheap (Thomas R. Flowers, 2002). Given a single engine for designing virtual worlds, a very wide array of challenges can be built without requiring the purchase of additional equipment. This is a large benefit for cash-strapped schools.

Finally, virtual worlds, it is argued, are simply more interesting and fun for students than more traditional methods of testing code. The challenges that can be provided in a virtual world – drive from point A to point B, move object X to position Y, etc. – are both more interesting to students vs. a typical exercise in traditional programming education, such as number sorting, as well as more directly relatable to a student (it is far easier to envision what a robot driving would look like than a sorting algorithm!) (Guzdial, Programming Environments for Novices, 2004).

4. Robots

Robots are the other possible value for the "testing" metric. As one might expect, using a robot to test a program involves the student loading a program onto a physical robot and seeing the robot execute the program in real space, and real time.

Robotic systems are argued to have many of the benefits that virtual worlds do. Like virtual worlds, it is argued that robot programming is much more interesting for students than traditional programming exercises (like Hello World or sorting lists), and that robotic programming provides significant motivation and engagement to students (Maja J Mataric, 2007) (Tom Lauwers, 2009) (Barry Fagin, 2003) (McGill, 2012). In addition to the fun factor of robot programming, it has also been noted that robot exercises offer student a very clear model of execution, something that traditional CS problems do not (Paul, 2012). Students

can easily understand what a successful robot test should look like, and can identify a success or failure without trouble. This modeling benefit is also present in virtual worlds, but the supporters of robots often assert that the model is stronger and more concrete, due to it existing in the real world as opposed to on a computer screen (Wanda Dann, 2009) (Tucker Balch, 2008) (Tom Lauwers, 2009) (Thomas R. Flowers, 2002).

These are the four characteristics that we will use to divide up the systems. Because these characteristics are divided up into two scales with two possible characteristics each, we will present four subsections total: code based virtual worlds, code based robots, drag and drop virtual worlds, and finally, drag and drop robots.

The first set of systems we will review are the code-based systems. There are two permutations that we will review: code based systems utilizing virtual worlds, and code based systems utilizing robots.

2.4.3. Code Based Virtual World Systems

Code-based virtual world systems are among the first "newer" systems to have been created, starting with the Logo programming language[12], developed by Wally Feurzeig and Seymour Papert in the 1960s. The Logo programming language is a Lisp dialect created to teach students the fundamentals of programming. Logo was originally a robot-based system and controlled a turtle-shaped robot, but it is today best remembered for the

[12] http://el.media.mit.edu/logo-foundation/logo/programming.html

simulated Logo turtle that could be programmed to move around in a virtual world (What Is Logo?, 2011). A screenshot of Logo may be seen in Figure 1.

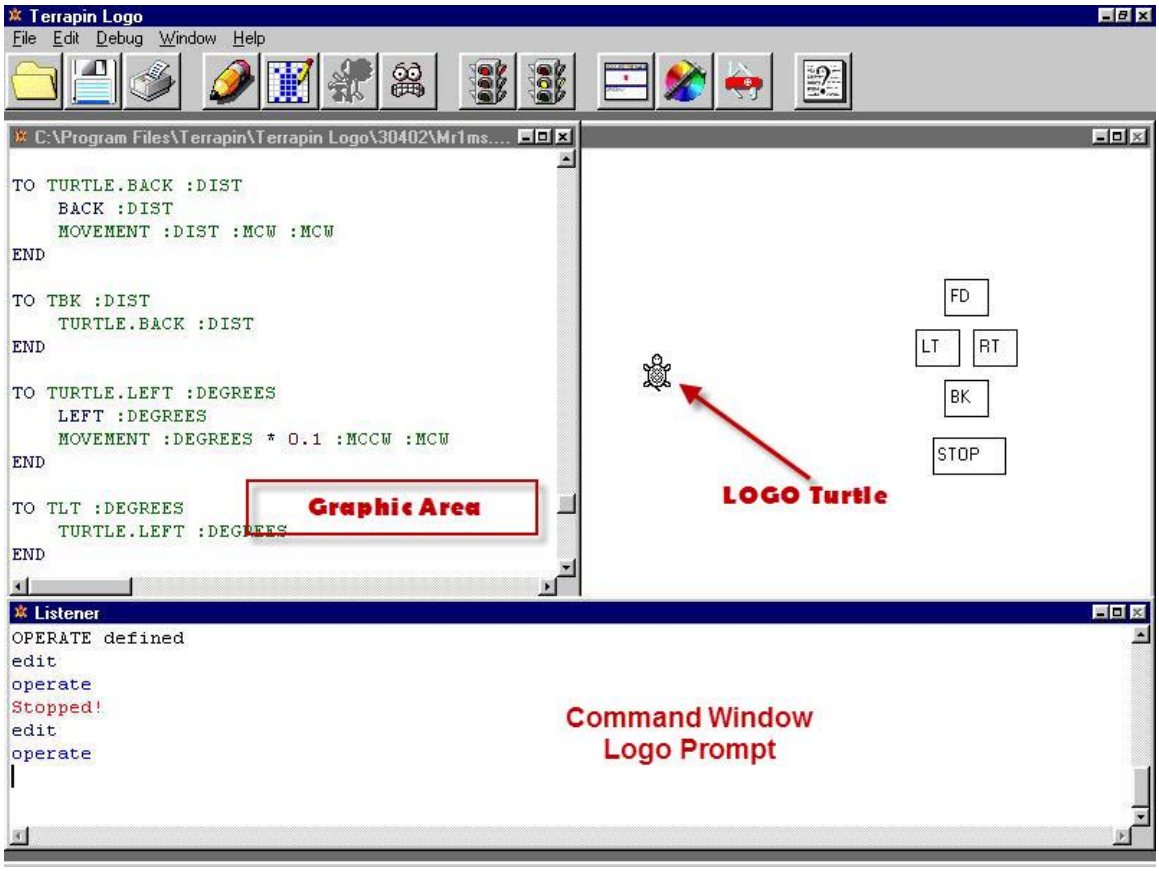


Figure 1. The Logo software. Screenshot credit: http://www.techibuzz.com/logo-programming-language-software-for-kids/

Another code-based virtual world system is Robocode[13] (previously called IBM Robocode), in which students write Java or .NET code to create a simple AI system for a virtual battle tank. Students using Robocode are able to compete with one another, pitting their programmed tanks against one another in a virtual battlefield (Larsen, 2013). Robocode was first developed in 2000, was adopted by IBM in 2001, was released as

[13] http://robocode.sourceforge.net/

open source in 2005, and continues to be developed to this day. Figure 2 shows the Robocode system.

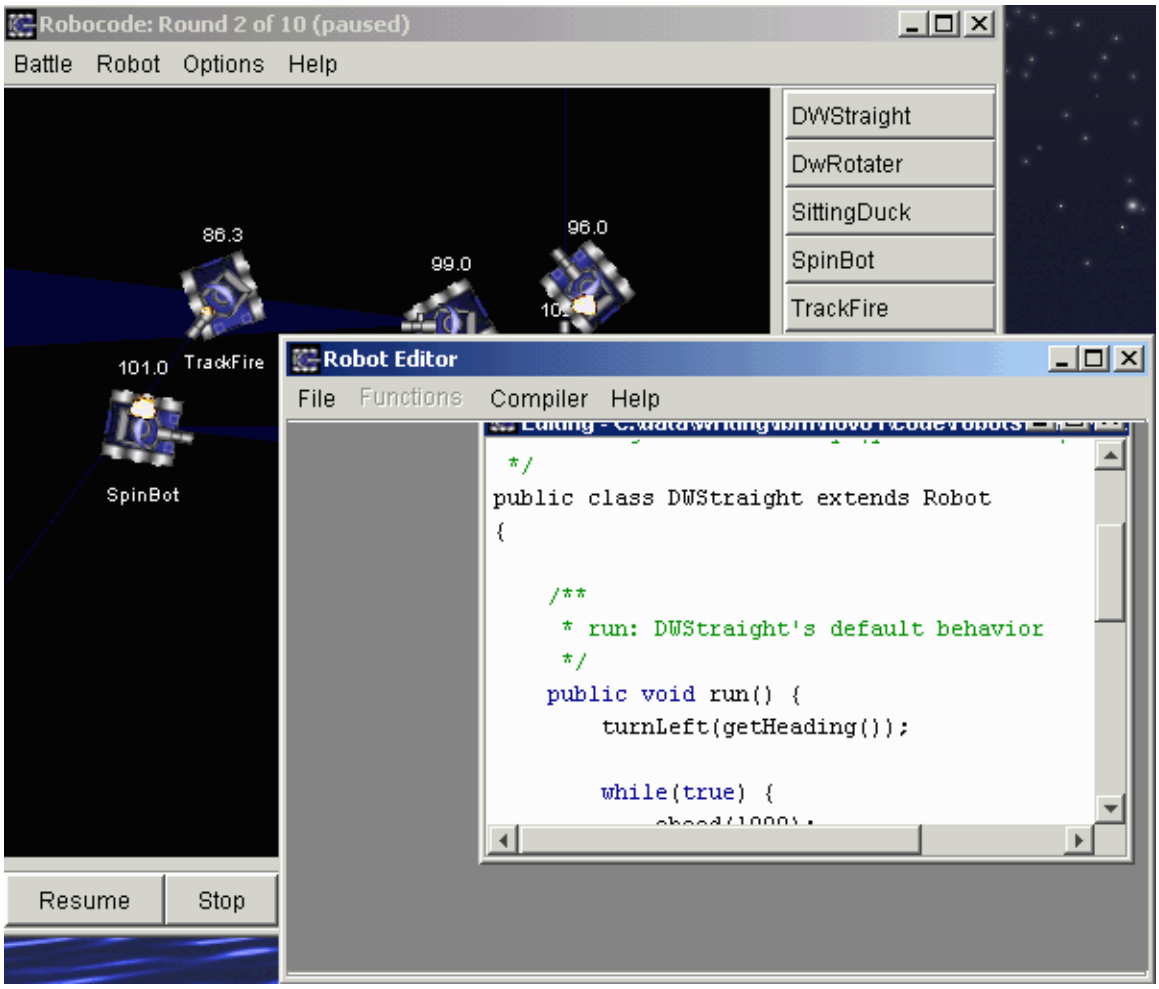


Figure 2. A screenshot of Robocode in action. The robots may be seen in the background. Screenshot credit: http://www.ibm.com/developerworks/library/j-robocode/

Both of these systems appear to be targeted towards any student who wishes to learn them. Additionally, both systems have gone through numerous iterations and have many different variants. This is especially true for Logo, which has been offered and promoted in many different ways, and in many different environments, in the 40+ years since its initial creation.

Because these two systems are both very early attempts at creating an alternative to traditional programming education, we will consider the benefits and downsides of these systems together as opposed to separately.

2.4.3.1. Benefits of Code Based Virtual World Systems

The primary benefits argued for these systems tend to be in the context of the virtual worlds. Most of the virtual world benefits mentioned earlier are applicable here. Papert's "low floor, wide walls, high ceiling" ideal is stated to have informed Logo's design and Logo carries its benefits (What Is Logo?, 2011). Robocode has similar benefits; it also benefits from its competitive design, which must certainly interest and motivate certain programmers more than traditional programming exercises (Jackie O'Kelly, 2006).

2.4.3.2. Problems of Code Based Virtual World Systems

The primary downsides to these code-based systems are, predictably, the formal code. Logo is a Lisp dialect designed for simplicity (What Is Logo?, 2011), and Robocode is straight Java or .NET (Larsen, 2013). The problems with teaching introductory computer science using formal languages have been discussed above; the primary difficulty is students needing to learn complex and difficult syntax. Robocode, as it uses unaltered, unsimplified formal languages certainly suffers from this problem, but even the simplified Logo language has been noted to have a high enough complexity to scare away students or teachers (Tucker Balch, 2008), and dampen the fun for students using it (Long, 2007).

2.4.3.3. Takeaways of Code Based Virtual World Systems

The most widely recognized lesson of early systems like Logo was the value of simplified programming languages and virtual worlds. Perhaps the more important lesson to learn, however, is the reason that programs like Logo did not spread further. As stated in the previous section, the reason for this failure to "catch on" this would seem to be the complexity of its system, especially its programming language. For an introductory educational system to be effective, then, it must be vastly simplified so as not to scare off or discourage both students and teachers (Tucker Balch, 2008).

2.4.4. Code Based Robotic Systems

Code-based robotic systems are focused on teaching students computer science by developing programs to control a physical robot, as opposed the manipulating of objects within a virtual world. As the name indicates, the programs in these systems are developed using formal programming languages.

We will here investigate two code based robotic systems: the FIRST Robotics Competition[14] and Myro[15].

2.4.4.1. FIRST Robotics Competition

FIRST Robotics Competition is a popular extracurricular organization centered on student robotics competitions. Participating students form teams and work together to design,

[14] http://www.usfirst.org/roboticsprograms/frc

[15] http://wiki.roboteducation.org/Myro_Development

build and program a robot. The robot may then be brought to FRC tournaments around the country and compete in various events against other FRC teams' robots (Welcome to the FIRST Robotics Competition, n.d.).

FRC is an extracurricular program – coaches may form teams through a school or on their own. The age range for students to participate in FRC is 14 – 18. These students are entirely responsible for the design and construction of the robot – the only limitations are related to certain forbidden parts and an overall budget cap. All participating teams start with a common set of parts, but may (within certain budgetary restrictions) purchase or build additional ones. After building the robot, students then program it to compete using special FRC variants of common programming languages: in 2015, FRC variants of C++, Java and LabVIEW were available (2015 FRC Control System, 2015).

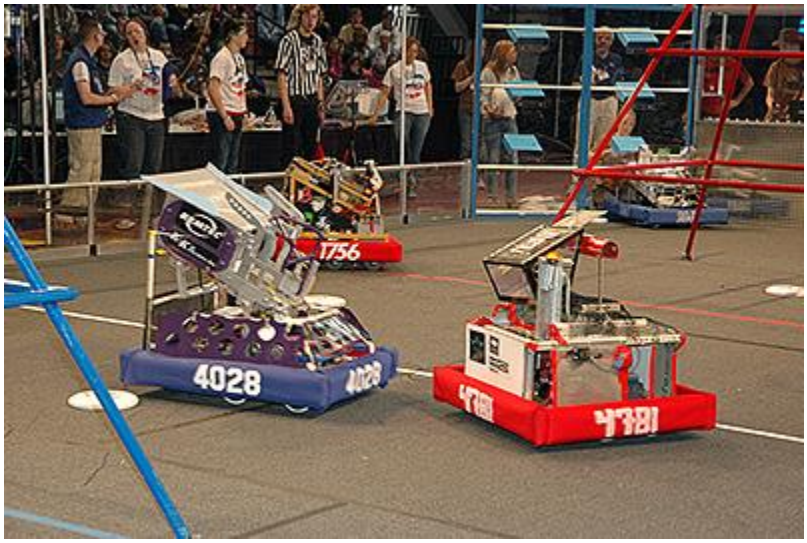


Figure 3. A photo of teams competing in the FIRST Robotics Competition. Photo Credit: http://www.rose-hulman.edu/offices-and-services/first-robotics-regional/first-faqs/first%C2%AE-robotics-competition-faq.aspx

FRC is a popular program. In 2015, 75,000 high school-aged students participated in 3,000 teams. 56 regional events, 5 regional championships and 1 grand championship event were held (The FIRST Robotics Competition: HOW IT WORKS, 2014).

2.4.4.1.1. Benefits of First Robotics Competition

FRC is commonly promoted as being a fun way to teach students engineering. The FIRST organization states that FRC provides "[r]eal-world engineering experience", "[t]echnological literacy", and has a "proven positive impact on student interest in engineering" (The FIRST Robotics Competition: HOW IT WORKS, 2014) (The FIRST Robotics Competition: OVERVIEW, 2014).

Other stated benefits include a positive impact on a student's academic success (The FIRST Robotics Competition: SUCCESS, 2013), and the imparting of real-world technological skills (The FIRST Robotics Competition: CAREERS, 2014). 89% of FRC students report an "[i]ncreased understanding of [s]cience & [t]echnology", and 90% report "[l]earning new practical and work-related skills" (The FIRST Robotics Competition: EVALUATION, 2013).

As with other competitive educational systems, the competition itself has been stated to increase student motivation and enjoyment (Long, 2007).

2.4.4.1.2. Problems of First Robotics Competition

FRC is, as mentioned above, a popular engineering program, and is often cited for its value in both mechanical engineering and computer science. Academic reviews of the FIRST program, however, have found that the former tends to be far more of a focus for an FRC team than the latter. Buckhaults, a professor at the University of South Carolina and FRC coach, notes that in a typical FRC six-week build sprint the majority of student time is spent engineering and building the robot itself. She notes that little time is spent actually programming the robot. This problem is one that is common to robotic-based educational systems in which one actually builds the robot (as we will see in Section 2.4.7): the primary focus is often the building instead of the programming, the mechanical engineering as opposed to the computational thinking (Buckhaults, 2009) (Delden, 2008).

This can be seen further in the program's stated results on major choices. Buckhaults notes that FRC alumni "major in engineering about seven times…the rate for high school graduates", and notes further that the program also produces computer science majors at "two times the rate for high school graduates" (Buckhaults, 2009). A 100% increase in computer science graduation is nothing to be scoffed at – but these numbers still make it clear that the FRC program produces far more mechanical engineers than it does computer scientists.

2.4.4.1.3. Takeaways of First Robotics Competition

The fact that the FIRST competition involves building robots in addition to programming them leads participants focusing more on mechanical engineering than computer science and computational thinking is a theme that will be repeated with other robotic solutions of this nature. The takeaway from this is that, if one wishes to use robots to teach computational thinking, students should spend minimal or no time building these robots. It may be better for the program overall if the robots come prebuilt.

2.4.4.2. Myro

Myro is a project of the Institute for Personal Robots in Education[16], which is itself a collaboration between Georgia Tech and Bryn Mawr College (Institute for Personal Robots in Education, 2008). The Myro project involves using a prebuilt, simple robot to teach students programming skills: students can use programming languages such as CPython[17], IDLE[18] and Tkinter[19] to program the robots (Myro Development, 2009).

The Myro robot is relatively simple and the project appears to be targeted at younger learners. A screenshot of the Myro software may be seen in Figure 4.

[16] http://www.roboteducation.org/

[17] http://cython.org/

[18] https://docs.python.org/2/library/idle.html

[19] https://docs.python.org/2/library/tkinter.html

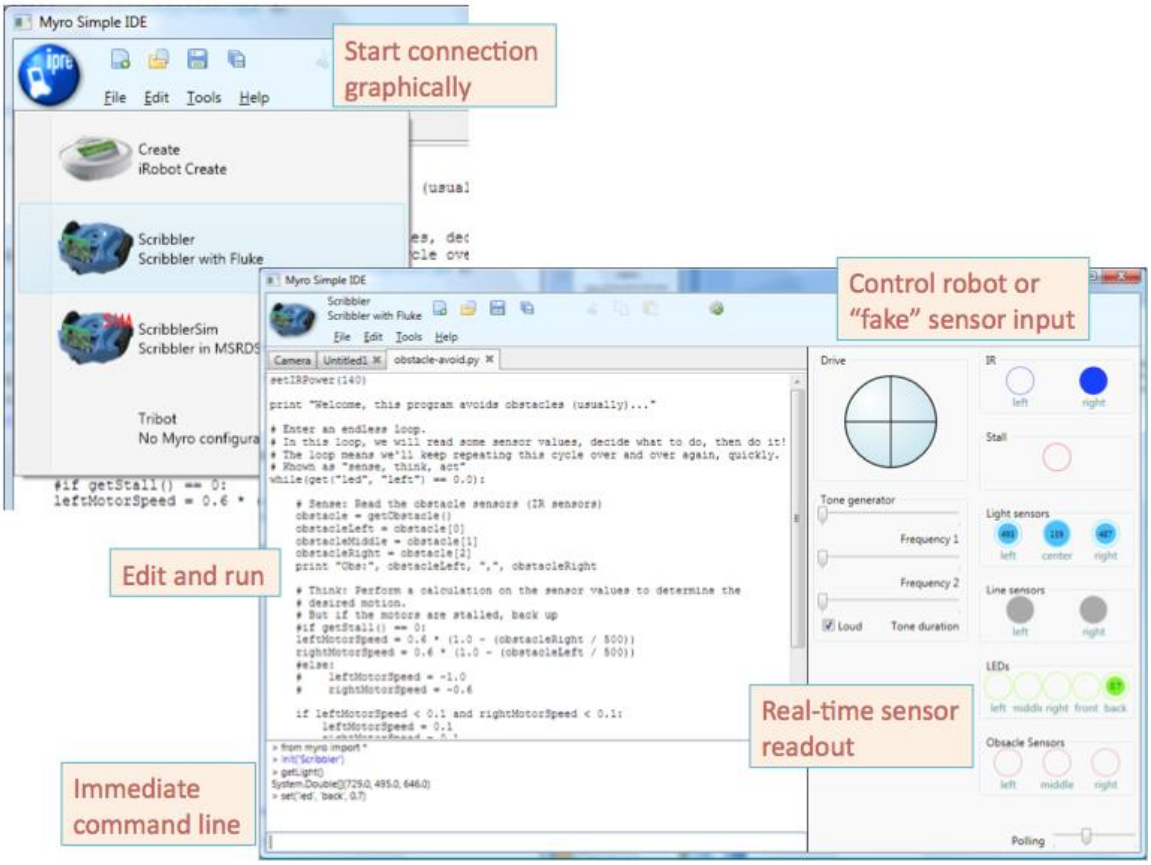


Figure 4. The Myro software. Screenshot credit: http://www.cc.gatech.edu/~richard/oldsite/Myro3/

2.4.4.2.1. Benefits of Myro

The Myro system has many interesting features that other robot-based systems do not have. One of the most interesting features is the fact that the programs developed to control the robot do not actually execute on the robot itself – instead, the programs are executed on a desktop computer and commands are transmitted to the robot as it runs. This feature allows for many benefits: the robot can be vastly simpler (requiring only a receiver and some basic translation software as opposed to a full computer), and the program can be debugged, stepped through or otherwise manipulated on the computer as the robot runs (Tucker Balch, 2008).

Another major benefit of the Myro system, and one that is not often seen, is the bundling with Myro of a curriculum specifically designed for it (Deepak Kumar, 2008). Many other systems, such as Scratch or Alice, are released without curricula, and users must rely on other organizations to create these. The IPRE, on the other hand, has designed both the Myro robot and the curriculum with the other in mind, "let[ting] the needs of the curriculum drive the design of the robot" and vice versa (Tucker Balch, 2008). This, it is argued, leads to a superior system.

It is notable that the Myro robots are technically quite simple. Mohtadi notes that the "biggest barrier" to using hardware-based testbeds – which includes robots – is the high technical complexity, which can severely hinder their adoption in the classroom (Mohtadi, 2013). Myro avoids this by using a very simple prebuilt robot design.

Finally, it has been noted in at least one study that, compared with a traditional Java class, students who took a Myro-centered class had more fun on at least one project, and emerged from the class feeling more confident in their knowledge about computers than their Java companions (Harms, 2013).

2.4.4.2.2. Problems of Myro

Despite its benefits, Myro suffers from a few weaknesses. The primary weakness is one that we will examine in more depth when examining Scratch: this weakness comes as part of the "low floor, wide walls, high ceiling" philosophy, and specifically, the high ceiling. Myro is very scalable, and can be programmed to do some very advanced things.

In order to enable this high scalability, however, Myro is necessarily very technologically complex. In this way, the "high ceiling" conflicts with the "low floor", and Myro's technical complexity has been reported to be too great for some students (Deepak Kumar, 2008).

A similar problem comes from the Myro curriculum. The topics that the curriculum goes over are very advanced for an introductory programming class - while starting out with common introductory topics, the curriculum continues on to ask students to create computer vision algorithms, or artificial intelligence (Deepak Kumar, 2008). While justified in terms of providing an overview of the computing field, these topics may be too advanced for a proper introductory curriculum – either in terms of their difficulty, or in terms of their pertinence.

2.4.4.2.3. Takeaways of Myro

There are two major positive takeaways for Myro. The first is the benefit of executing a student algorithm on a computer instead of the robot itself. The ability to step through, set breakpoints, and debug are common features in standard programming IDEs and are quite helpful for program development. These features cannot be used if a student must send his algorithm to a robot for remote execution. By executing the algorithm on the local desktop computer and transmitting instructions to the robot, the debug features can be utilized, to the student's educational benefit. An additional benefit of this feature is that the robot need not have hardware for compiling or executing code, and may therefore be less expensive and technically complex.

The second major takeaway is more general: this is the benefit of pairing educational software with a curriculum. For an educational system to be successful, it must have both good software and a good curriculum. By pairing the development of these, the system becomes stronger overall.

There is a negative takeaway for the Myro system, which relates to the curriculum as well: the curriculum for an introductory computer science course must stay at a basic level. It should not involve topics which are too advanced, and should instead focus on the basic skills involved with computational thinking.

2.4.5. Summary of Code Based Systems

Having reviewed four separate code-based systems across two categories, we will now perform a brief review of the takeaways and lessons learned from these systems in general.

The positives takeaways of code-based systems are as follows:

- A system ought to feature either a virtual world or a robot, in order to make the problem domain more comprehensible and visual.

- A system should reduce the complexity of its language as much as practically possible.

- A system ought to, if possible, develop and pair a curriculum along with the system. In this way the system's design may affect the curriculum's design, and the curriculum's design may affect the system's design.

- Robot code should run on a computer instead of the robot itself, to allow for active debugging, step-through, and other benefits. This also reduces the cost of the robot.

The negatives takeaways of code-based systems are as follows:

- A system should not require that students learn complex syntax and grammar rules in order to work with the system. This can lead to a focus on syntax at the expense of computational thinking skills.

- A robot based system should not allow the focus to be on building or engineering the robot, if one wishes to focus on computational thinking education.

- A robot-based system ought to have a platform that is as technically simple as practical.

- A curriculum ought to not be inappropriately advanced – curricula for computational thinking courses should limit themselves to basic computational thinking topics, and not advanced subjects such as AI or computer vision.

With these lessons in mind, we will now move on to the drag and drop systems. Once again we have two permutations: drag and drop systems utilizing virtual worlds, and drag and drop systems utilizing robots.

2.4.6. Drag and Drop Virtual World Systems

The drag and drop systems are in many ways more modern than the code based systems. By and large they are newer developments, and have more notoriety. Because of this, they also have much more related research. This reflects the drag and drop input system's popularity due to its simplicity, and the inability to make syntax errors using a properly designed drag and drop programming system.

The two drag and drop systems utilizing virtual worlds that we will here review are Alice and Scratch, both of which are quite popular and well-known systems. We will, as with the code-based systems above, discuss what these systems do well, what problems or criticism they face, and the takeaways from these systems.

2.4.6.1. Alice

Alice is a graphical educational programming tool developed by Carnegie Mellon, with which students can write programs to manipulate 3D characters in a virtual world. Alice's website describes the software as follows:

> Alice is an innovative 3D programming environment that makes it easy to create an animation for telling a story, playing an interactive game, or a video to share on the web. Alice is a freely available teaching tool designed to be a student's first exposure to object-oriented programming. It allows students to learn fundamental programming concepts in the context of creating animated movies and simple

> video games. In Alice, 3-D objects (e.g., people, animals, and vehicles) populate a virtual world and students create a program to animate the objects.
>
> In Alice's interactive interface, students drag and drop graphic tiles to create a program, where the instructions correspond to standard statements in a production oriented programming language, such as Java, C++, and C#. Alice allows students to immediately see how their animation programs run, enabling them to easily understand the relationship between the programming statements and the behavior of objects in their animation. By manipulating the objects in their virtual world, students gain experience with all the programming constructs typically taught in an introductory programming course. (What is Alice?, 2015)

Alice uses a visual drag-and-drop programming language. In the Alice GUI, students are able to select blocks, drag them into a class or function, drop them, and pair them together to create programs. Most program actions consist of moving a character model in a certain way – students are able to click on a part of a 3D character, or the entire character itself, and drag different blocks into their program to command that character to move in certain ways. As the Alice description notes, Alice's visual language has an object-oriented design – individual characters are treated as objects, their different movable parts (arms, legs, etc.) as attribute objects of the character, and students call object methods to move or otherwise manipulate these objects.

Alice attempts to motivate students to learn by asking them to create a story or video game using the software's 3D models and virtual worlds (What is Alice?, 2015). A variant of Alice, called Storytelling Alice[20], takes this even further, providing changes to the basic Alice software to increase the focus on creating a story using the software (Kelleher, 2007).

[20] http://www.alice.org/kelleher/storytelling/index.html

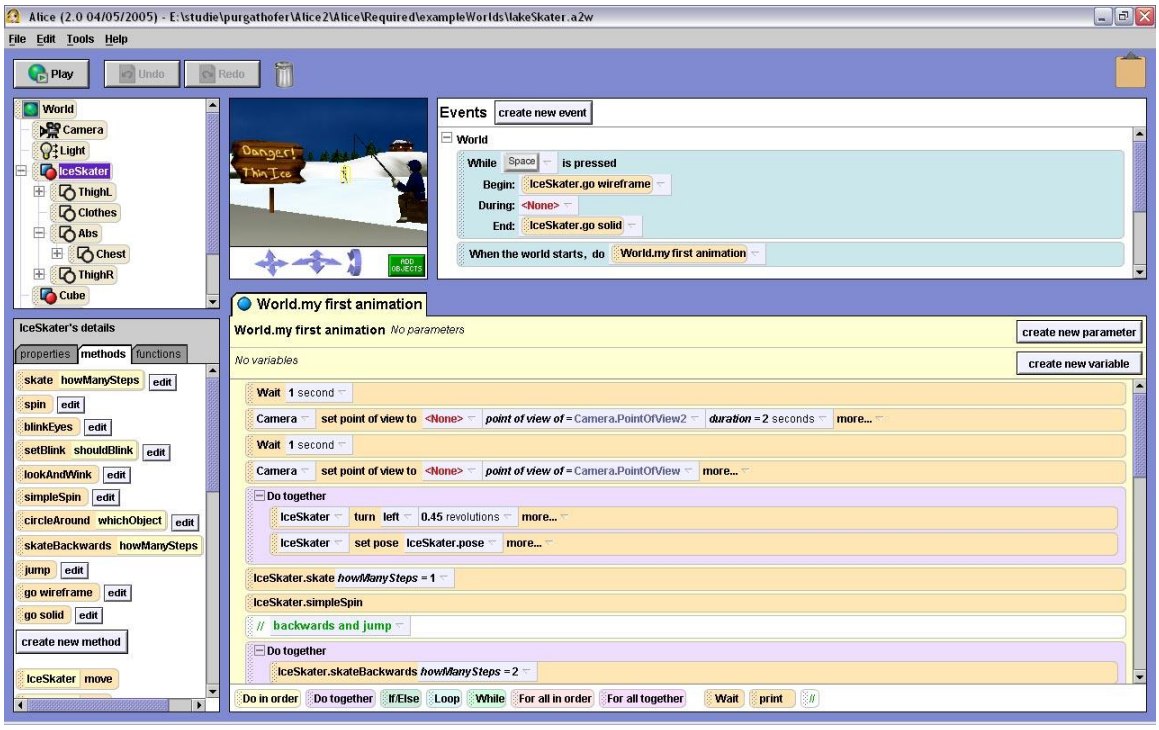


Figure 5. Screenshot of the Alice IDE. Screenshot credit: http://en.wikipedia.org/wiki/Alice_(software)

Alice is used in a wide array of schools and with a wide array of students. As can be seen on the Alice Testimonials page, high school students, college students, and young elementary-age students have all used the software (Testimonials from Alice users, 2015).

2.4.6.1.1. Benefits of Alice

Alice, with its drag-and-drop programming interfaces, features all the standard benefits of a system of this nature: it makes programming "gentler" (Paul Mullins, 2009), more enjoyable (Karin Johnsgard, 2008), and does not suffer from syntax errors (Wanda Dann, 2009) (Karin Johnsgard, 2008).

Alice is designed around the concept of a “head fake” – students are asked to learn a fundamental concept of computer science while they think they are learning something else. Alice’s creator argues that this results in superior learning results (Wanda Dann, 2009). The storytelling aspect of Alice is also considered quite important to motivate and foster student learning (Caitlin Kelleher, 2007).

Certain aspects of Alice’s implementation have been praised. Alice’s methods for moving characters are presented at a high level – instead of having to design methods to turn a robot’s wheels for a certain amount of time, students using Alice can simply command their characters to “walk”. Storytelling Alice is designed to abstract these methods even further (Caitlin Kelleher, 2007). Another argued benefit is the way Alice’s code executes – as characters move around during program execution, students can directly follow along in the code. In this way the student’s program is directly tied to the on-screen results (Ian Utting, 2010).

In various tests, Alice has been found to increase student enrollment in computer science, increase the retention of those students who are enrolled, and increase the success of these students in their computer science classes (Ryan Garlick, 2010) (Paul Mullins, 2009) (Karin Johnsgard, 2008).

Finally, it is notable that Alice is released along with an official curriculum. This curriculum is developed by Carnegie Mellon, the developers of Alice itself (About the Alice 3 Instructional Materials, 2015).

2.4.6.1.2. Problems of Alice

Despite Alice's benefits, there are also criticisms, some of which directly challenge the cited benefits.

One criticism is that Alice's storytelling focus, while sounding nice in theory and showing some increase the amount of student interaction with the program (Kelleher, 2007), represents *play* and not serious learning. Mullins reports a test of Alice he conducted in which he found that students focus far more on the story they are creating than the concepts being learned. He notes further that students often base their story around what their program is doing, and not the other way around – in other words, students will make their program do something – *anything* – and then tell a story about it, incorporating bugs, mistakes or random behavior into the tale (Paul Mullins, 2009).

Other criticisms focus on Alice's practical effects. Garlick notes that students trained in Alice had difficulty transitioning to a formal programming language, and that these students had lower grades in traditional programming education, directly contradicting the aforementioned reports of Alice *increasing* success (Ryan Garlick, 2010). Finally, Garlick and Mullins note that in their tests students complained that Alice was not "real programming" (Ryan Garlick, 2010) (Paul Mullins, 2009) – this would seem to contradict the claim that Alice always results in increased student motivation.

2.4.6.1.3. Takeaways of Alice

Alice shows that drag and drop programming interfaces can be very successful. Furthermore, there are certain aspects of Alice that deserve note.

Alice keeps things simple by keeping its basic actions at a high level. A character can be instructed to "walk", and the requisite animation plays automatically – the student need not program the movement of every limb. This high level abstraction reduces cognitive load on students and allows them to "make things happen" without considerable effort. Additionally, Alice's close ties between the program and its on-screen execution is certainly valuable for students learning to track algorithm execution.

The criticisms of Alice, however, show that certain features ought to be avoided. There is contention as to whether Alice's storytelling aspects are beneficial or not. This indicates that one ought to be cautious about using play to motivate students – it appears that while it may have benefits, it is very easy for the play to become the main focus of the activity (this will be discussed further later). Additionally, the difficulty students have with moving from Alice to a formal language indicates a possible weakness of the drag and drop interface: despite having the same concepts, Alice code does not look or feel like "real" code. A possible takeaway is that visual GUIs ought not to look too radically different from real code in their feel and structure.

2.4.6.2. Scratch

Scratch is another graphical programming tool designed to teach students the fundamentals of programming. Developed by MIT and targeted at 8 – 16 year olds, Scratch allows students to create and manipulate graphics on a 2D plane through the use of drag and drop programming blocks. Students are able to use Scratch to create animations, stories and games (For Parents, n.d.). Specific topics that Scratch focuses on teaching include "mathematical and computational ideas", the "process of design", and computer fluency (Learning with Scratch).

Scratch's programming language is graphical – blocks may be dragged, dropped and attached together to create programs. Scratch's programming paradigm is mostly procedural, though it contains some object-oriented features (Object-Oriented Programming, 2014). A notable feature of Scratch is the design of the blocks – each block is color coded and has a physical shape indicative of which blocks it can attach to (David J. Malan, 2007) (Mitchel Resnick, 2009).

Scratch integrates with a specially-designed social networking platform that allows students to upload and share their programs. Other users may comment on a student's program, or even download the program and edit it, something that Scratch calls "remixing" (Mitchel Resnick, 2009). Scratch is used in many universities, high schools, and even elementary schools in more than 150 different countries (About Scratch, n.d.).

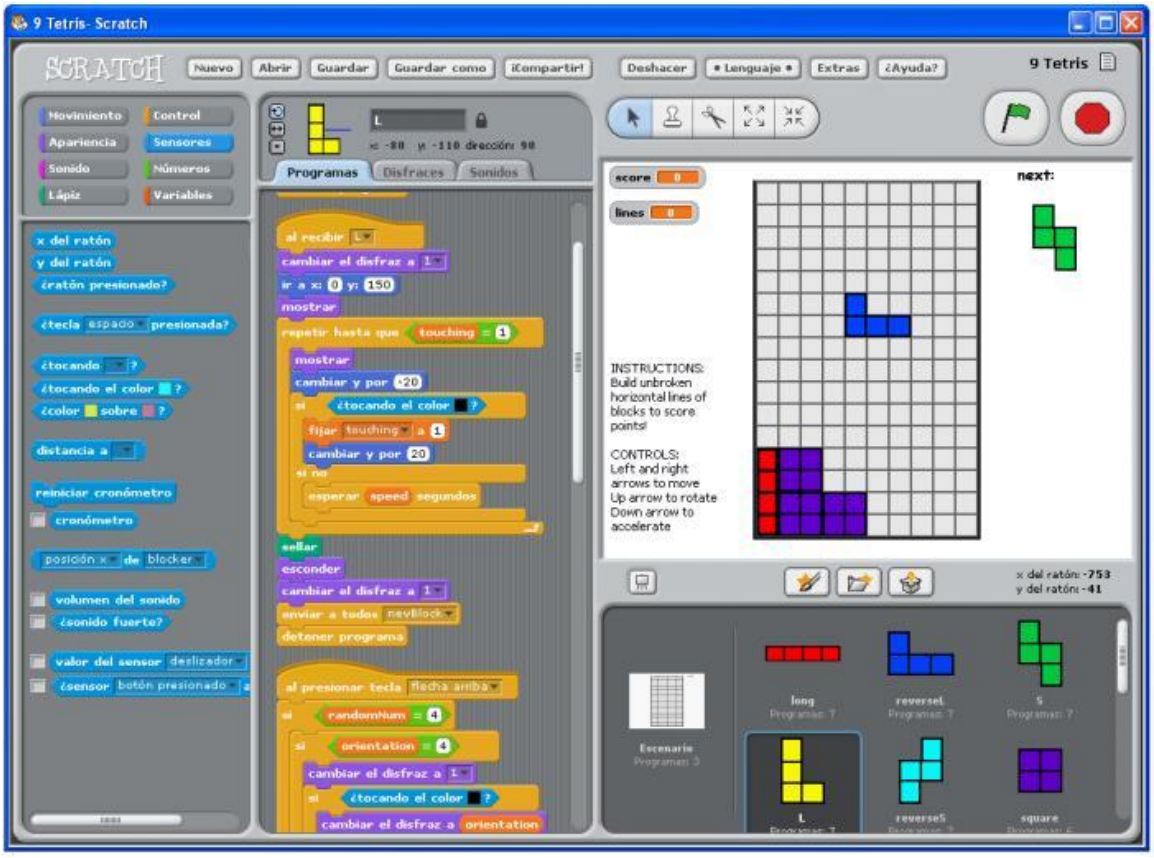


Figure 6. Screenshot of the Scratch IDE. Screenshot credit: http://mit-scratch.softonic.com/

2.4.6.2.1. Benefits of Scratch

As with Alice, Scratch features all the benefits of drag-and-drop programming: syntax errors are impossible to make, and the visual design of the language itself contributes to student understanding of how to use it (David J. Malan, 2007) (Mitchel Resnick, 2009). Mitchel Resnick, the leader of MIT's Scratch team, notes that the Scratch language is designed to be clear and precise, readable even to students with little or no programming experience (Mitchel Resnick, 2009).

Resnick and others have described Scratch as attempting to embody Papert's "low floor, wide walls, high ceiling" (David J. Malan, 2007): it is designed to be a language that is not just easy to use, but also fun; a language not just for formal learning, but for "tinkering" (Mitchel Resnick, 2009). Scratch's free-for-all project design allows students to create "personally meaningful" projects, contributing to motivation (Mitchel Resnick, 2009).

Resnick and others have noted that Scratch's social network as a very important benefit of the program. Putative benefits of this network include increased motivation for students, and the ability for students to share projects and help each other learn (Mitchel Resnick, 2009).

Like Alice, Scratch directly ties program statements to their on-screen results: as students execute their program, they can track the code as the computer steps through it (Ian Utting, 2010).

Finally, Scratch is stated to have proven educational benefits. Some studies have found that Scratch has increased student knowledge and internalization of computer science concepts (David J. Malan, 2007) (Diana Franklin, 2013). Other studies have found that students exposed to Scratch have had an easier transition to formal languages like Java or C (Ursula Wolz, 2009). Finally, it has been noted that using Scratch helps students

overcome anxiety and increases self-esteem and confidence in relation to programming (Orni Meerbaum-Salant, Learning Computer Science Concepts with Scratch, 2013) (David J. Malan, 2007).

2.4.6.2.2. Problems of Scratch

The problems that we have identified with Scratch has mostly to do with one of its stated benefits – its "wide walls" design goal of enabling – and even encouraging – creative exploration and "tinkering". Scratch is essentially undirected – students are able to create any kind of algorithm they want within the software. Multiple sources have noted that this playful, undirected design may interfere with computational thinking education (Orni Meerbaum-Salant, Habits of Programming in Scratch, 2011) (Orni Meerbaum-Salant, Learning Computer Science Concepts with Scratch, 2013) (Maloney, 2008).

The difficulty may be stated as such: Scratch, *per* Resnick, emphasizes "bottom-up tinkering" versus "top-down planning" when building algorithms (Mitchel Resnick, 2009). However, as we have noted in above sections, "top-down planning" is a fundamental and crucial part of computational thinking.

At least two studies have found that Scratch's focus on creative exploration and play causes difficulty in getting certain computational thinking concepts across to students. One study noted that in one class, 21% of the students participating did not create any actual programs – they instead engaged in "media manipulation", that is, activities such as "drawing [or] playing music" (Orni Meerbaum-Salant, Learning Computer Science

Concepts with Scratch, 2013) (Maloney, 2008). A second study notes that Scratch's "creativity aspect" and bottom-up design actively interferes with attempts to teach planning and design principles (Orni Meerbaum-Salant, Habits of Programming in Scratch, 2011).

A final problem with the Scratch system is that it does not have an "official" curriculum. There are many online curricula for Scratch; one major curriculum, developed by Harvard and linked to on the Scratch website, can be found at (Scratch Curriculum Guide, 2014). Despite the fact that the Harvard curriculum appears to have a strong computational thinking foundation (An Introductory Computing Curriculum Using Scratch, n.d.), it is a fundamentally separate development from Scratch itself. It is also notable that the Harvard curriculum was released in 2014, a full 7 years after Scratch's online release (Scratch Curriculum Guide, 2014).

2.4.6.2.3. Takeaways of Scratch

Many of Alice's takeaways are also found in Scratch. Scratch's programming language design – clear, precise, and easy to use – contains many valuable features that should be appreciated. Furthermore, Scratch's ability for students to link program code with onscreen actions is valuable feature.

Scratch's social media aspect is also an item which should be considered in future systems, due to its increase in student motivation and other benefits.

The potential problems with Scratch's bottom-up, wide-walls design should be taken with a grain of salt, but they also should not be dismissed entirely. Like with Alice, while introducing creative "play" elements into the design of an educational system is something to strive for, one should also be aware of the detriments that these elements may have to computational thinking education, and be careful that one does not lose sight of the true goal of teaching computational thinking skills.

2.4.7. Drag and Drop Robotic Systems

The last type of system we will investigate are drag and drop systems that utilize physical robots instead of virtual worlds. Once again, the two systems that we will review – Microsoft Robotics Developer Studio (also referred to as VPL in some sources)[21] and Lego Mindstorms (also referred to as NXT or EV3)[22] – are fairly well known and widely used in educational contexts.

2.4.7.1. Lego Mindstorms

The Lego Mindstorms product is developed by The Lego Group, and consists of a buildable robot and software to program their robot. Mindstorms is a generic name for the product family, while terms like NXT or EV3 refer to specific generations of the product (About EV3, 2015); we will use the Mindstorms name for this thesis. Students build their robots around a central computer (usually referred to as the "brick") and can

[21] https://msdn.microsoft.com/en-us/library/dd939239.aspx

[22] http://www.lego.com/en-us/mindstorms/

attach motors, various sensors, and other common Lego pieces. While Lego provides plans to build many different models of robots, the modularity of the Mindstorms kits allows students to design and build their own original robots as well (Build a Robot, 2015).

After building a robot, students can use various languages and softwares to program it. Some of these languages are traditional formal languages, like LeJOS (which uses Java)[23] or brickOS (which uses C/C++)[24], while others are drag and drop, like NXT-G[25] or LabVIEW for Lego Mindstorms[26]. NXT-G is the most common language used with Lego Mindstorms (NXT-G, 2014), and that is what we will be focusing on here. NXT-G is a drag-and-drop model, and students are able to place and connect blocks to write programs for their robot. The software divides its blocks up into five categories: Action (to control motors and lights), Flow (to enable looping and conditional statements), Sensor (to retrieve data from the various sensors), Data Operation (to operate on variables) and Advanced (allowing access to files, Bluetooth connections, and other features) (Learn to Program, 2015).

The Lego Education website states that Mindstorms is targeted at middle-school students; despite this, the product is used in educational contexts for students of all ages, from

[23] http://www.lejos.org/

[24] http://brickos.sourceforge.net/

[25] http://www.legoengineering.com/program/nxt-g/

[26] http://www.ni.com/academic/mindstorms/

elementary school to college (Lego MINDSTORMS Education EV3, 2014). It is also used in the FIRST robotics competition (FIRST, 2015).

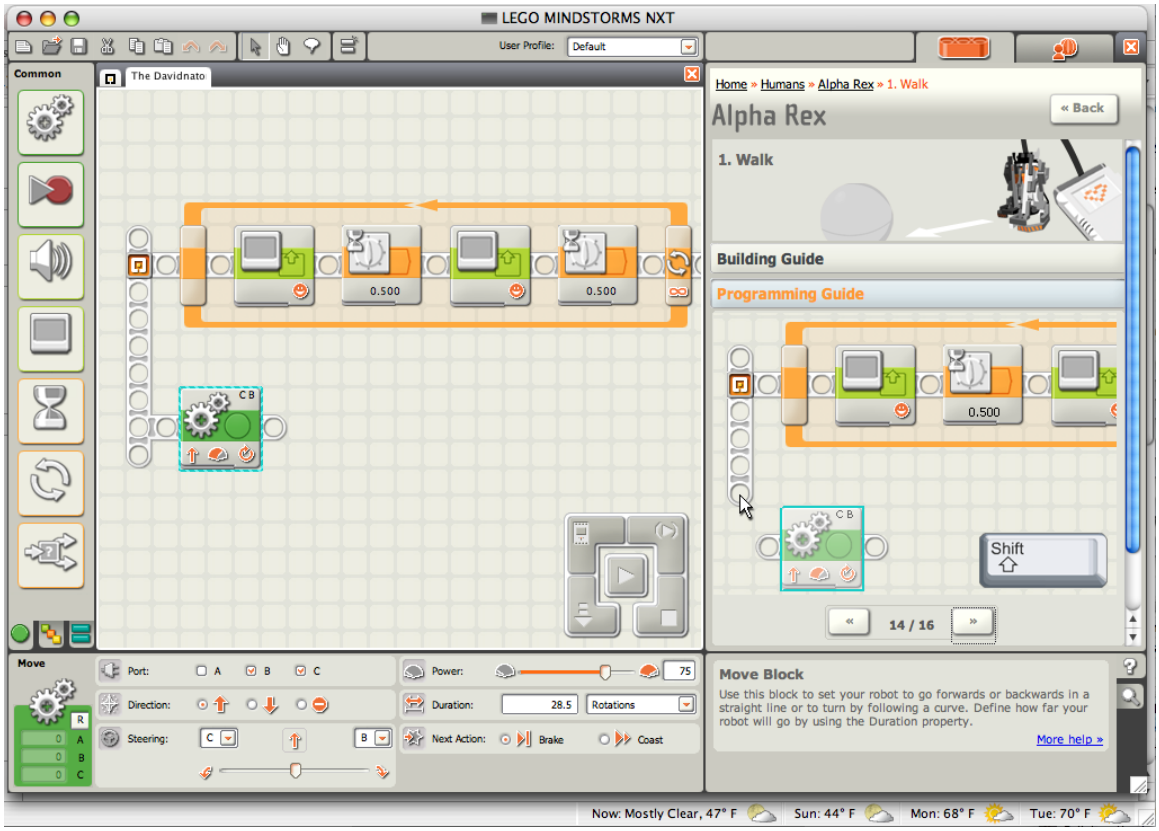


Figure 7. The Lego Mindstorms IDE. Screenshot credit: http://spectrum.ieee.org/automaton/robotics/robotics-software/review_lego_mindstorms_nxt_1

Note that the Mindstorms robots are able to be programmed using other software, such as Microsoft's VPL (Lego Mindstorms NXT, 2015). For the purpose of this section we will ignore these integrations and will only consider the Mindstorms NXT-G.

2.4.7.1.1. Benefits of Lego Mindstorms

Once again, we find that the Lego Mindstorms software has a clear GUI that is easy for students to work with. Because NXT-G is a drag-and-drop GUI, syntax errors are impossible to make.

Mindstorms robotics kits are very customizable and therefore can be built for many different situations or scenarios. This allows great flexibility in the use of these kits (Get Started (LEGO MINDSTORMS), n.d.). Additionally, it has been noted that students very much enjoy interacting with the Mindstorms robots, and are motivated to learn because of them (Maja J Mataric, 2007) (Barry Fagin, 2003).

2.4.7.1.2. Problems of Lego Mindstorms

The most commonly cited problem with the Lego Mindstorms software is its expense (Maja J Mataric, 2007). At time of writing, a single Mindstorms EV3 kit costs $349.99USD[27]. In a classroom setting, multiple robots will be required, which rapidly becomes quite costly.

Mindstorms also suffers from the problem common to robotics systems in which students build their own robots – the danger of focusing far more on the engineering challenge of building the robot than the computational challenge of programming it. It has been noted that a robot-based curriculum must be very carefully designed to avoid an "overabundance of robotics related material" (Delden, 2008) (Buckhaults, 2009). The phenomenon of students focusing heavily on building the robots, to the expense of

[27] http://shop.lego.com/en-US/LEGO-MINDSTORMS-EV3-31313

programming them, is something that I have anecdotally witnessed both as an undergrad in an introductory programming class which utilized Mindstorms, and as a teacher when using Mindstorms to teach computer science.

It is notable that the Mindstorms programming language, while clear and easy to understand, looks very dissimilar to a formal textual programming language. We have noted previously that drag and drop languages that are dissimilar to formal languages have resulted in difficulty transferring to a formal language (Ryan Garlick, 2010). Furthermore, in our experience of using the software, we found that the unique and highly simplified language design often made it difficult for relatively complex problems to be created.

Finally, we note that the Lego Mindstorms system does not appear to have an official, comprehensive curriculum centered on computational thinking. It does appear that Lego has built a curriculum based on engineering and exploration, which is offered on its website, (All About EV3 - Curriculum & Tools, 2014) – but this is more of a guide to building specific robots vs. a general curriculum for computational thinking.

2.4.7.1.3. Takeaways of Lego Mindstorms

Mindstorms is a fun system, and its robots greatly increase student enjoyment and motivation. The popularity of Mindstorms shows the degree to which robots can produce student motivation.

However, Mindstorms has many drawbacks. Its interface offers an important lesson: while simplification in drag and drop programming is important, *oversimplification* can work against the ultimate goal of computational thinking.

The expense of the Mindstorms kit has been reported in many works as hindering its adoption and usability. The lesson, then, is that cost must be taken into account when producing a robotic platform, and should be minimized.

Finally, we can once again see the problems of having students build their own robots, and we can conclude from this that prebuilt robots may be superior in regards to computational thinking focus.

2.4.7.2. Microsoft Robotics Developer Studio / VPL

The last system that we will review is Microsoft's Robotics Developer Studio. MRDS is a graphical IDE designed "for hobbyist, academic and commercial developers to create robotics applications for a variety of hardware platforms" (Microsoft Robotics - Overview, 2012).

MRDS refers to the program as a whole – the graphical language used within MRDS is called Visual Programming Language, or VPL. VPL, like other languages discussed here, allows students to select blocks, drag them around and place them in a 2D canvas. VPL is a bit different in its approach: instead of attaching the blocks directly to one another, the

blocks are connected using thin lines, meant to resemble a workflow diagram (VPL Introduction, 2012).

VPL has two categories of blocks. The first category is Basic Activities, which contains the standard control and data features common to all procedural and object-oriented languages, such as Variable, If, etc. The second category is Services, which consists of blocks representing input, output, actions, and other system features. These Services are in fact independent programs following Microsoft's DSS Protocol, thereby allowing them to be compiled together with a VPL program and invoked in a service-oriented manner (VPL Introduction, 2012) (Visual Programming Language - Using Services, 2012).

MRDS bundles together a VPL editor and a large number of robotics DSS services. Both platform specific and generic services are included, allowing a MRDS program to compile for many different robot platforms, including Lego Mindstorms, iRobot, and Roomba (Supported Robots, 2015). Once compiled, the program runs on the robot itself.

MRDS is used in many educational settings, including Arizona State University, where up until 2013 it was used in the introductory programming course FSE100.

2.4.7.2.1. Benefits of Microsoft Robotics Developer Studio / VPL

As with all drag and drop systems, VPL is incapable of syntax errors. However, VPL does not feature many of the other benefits of standard drag and drop languages, as we will discuss in the next subsection.

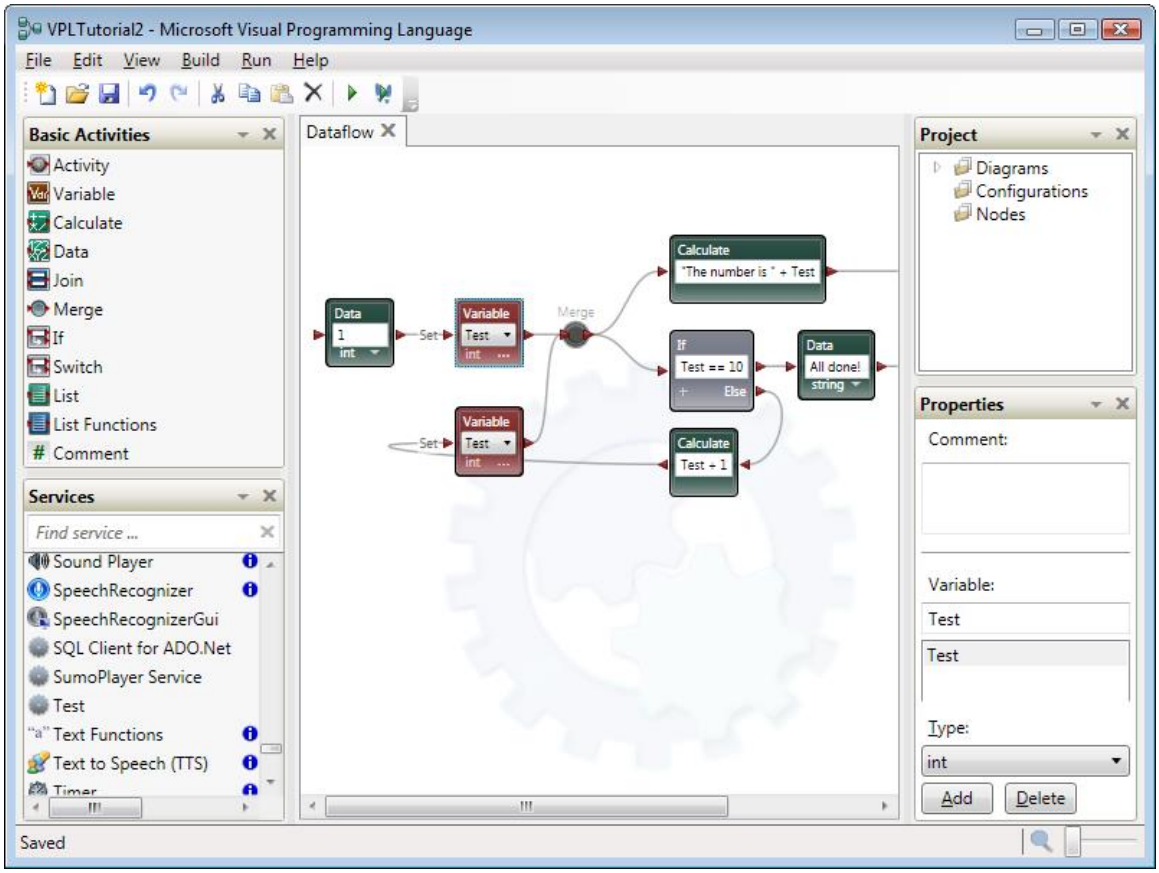


Figure 8. The Microsoft VPL IDE. Screenshot credit: https://msdn.microsoft.com/en-us/library/bb483088.aspx

MRDS, in addition to compiling programs and sending them to a robot, also has a simulator on which programs can be tested. This simulator has been reported to be very accuracy and simulates the physics of robot execution to a high degree – this has been stated to be a benefit for teams testing out robot designs where this accuracy is needed (for example, in competition) (Buckhaults, 2009).

As mentioned above, MRDS programs can be compiled and run on many different robots without changing the VPL code (Supported Robots, 2015). This allows schools different options for which robots they purchase and use. Furthermore, MRDS's DSS services may be extended, or new services may be written entirely, adding further flexibility to the program (Creating DSS Service Projects, 2012).

2.4.7.2.2. Problems of Microsoft Robotics Developer Studio / VPL

Perhaps the primary criticism of MRDS is its high technical complexity (Tucker Balch, 2008). VPL is a good example of a program with a "high ceiling" – it is written such that it can be used not just by novices but by advanced users as well, and its simulator is realistic enough to be useful for real-world applications (VPL Introduction, 2012). This high technical ability comes at a price: VPL is a very general language and is somewhat complex, making it difficult for students to learn and use (Tucker Balch, 2008). This difficulty is something that I have anecdotally observed as well in my interactions with the software as both a student and teacher.

VPL is a graphical language without many of the benefits that the graphical languages often feature. Unlike Alice or Scratch, its blocks feature almost no indication of how they can be placed together, and nontrivial configuration is often required when connecting two blocks.

Like Mindstorms code, VPL code does not physically resemble formal program code. Visual dissimilarity can result in difficulty for students when they transfer from VPL to a formal language, as we have noted earlier (Ryan Garlick, 2010).

Finally, note that like Lego Mindstorms, MRDS does not feature an official curriculum with which students can learn computational thinking. A Microsoft blog post from 2007, written one year after the initial release of MRDS, notes that Microsoft did develop and release a robotics curriculum – however, this curriculum is apparently no longer available, and based on the post, the content of the course was mostly focused on learning robot engineering and robot-specific programming, instead of more general computational thinking skills (Thompson, 2007).

2.4.7.2.3. Takeaways of Microsoft Robotics Developer Studio / VPL

MRDS's generality (its ability to run on many different robots) and easy extendibility are laudable. However, most of the takeaways for this system are negative.

The VPL programming language is a difficult language to learn and use; the reasons for this difficulty (high generality and lack of indication for how the blocks should go together) are things that should be avoided in future graphical languages.

More generally, the design of VPL, in trying to be both an educational language and a useful language for real-world applications, results in high technical complexity which

makes it difficult to learn. This is one example of a “high ceiling” inadvertently raising the floor, and is a good lesson in what not to do.

2.4.8. Summary of Drag and Drop Systems

Having reviewed four separate drag and drop systems across two categories, we will now perform a brief review of the takeaways and lessons learned from these systems in general.

The positive takeaways of the drag-and-drop systems are as follows:

- The drag and drop language should be vastly simplified from a formal programming language, and made to be both clear and precise.

- The drag and drop language should be designed such that syntax errors are impossible.

- The drag and drop language should be designed such that the physical design of the program blocks (color, shape, etc.) should indicate how those blocks can go together.

- The drag and drop language should abstract the actions that the user can command to a high enough level to allow practical programming without significant work.

- The program should closely tie the program's steps with the on-screen execution, so students can easily follow what each step does as it executes.
- Either a virtual world or a robotic system should be included if possible, as these increase student motivation and enjoyment. The robotic system especially has been shown to increase these things.

- A virtual world should be designed to be extendable and wide, so that many different tests, tasks and problems may be set up and solved within it.

- A robotic system should be set up such that the software works with multiple robots.

- A system can benefit from integrating with a social media sharing platform.

- A system should allow for easy extension and customization.

The negative takeaways of the drag-and-drop systems are as follows:

- A system should be careful not to focus more on creative "play" or storytelling to the detriment of computational thinking education.

- A drag-and-drop language's structure and visual appearance should not be completely visually dissimilar to a formal text-based programming language.

- A drag-and-drop language should not drastically *oversimplify* its language and should allow for some complex algorithms to be built.

- A drag-and-drop language should not attempt to be useful for both educational purposes and industrial work, as this can result in a language becoming overcomplicated. An educational language should limit its focus to education.

- A robotic system should not be extraordinarily expensive.

- A robotic system should not allow the focus to be on engineering (building the robot) to the detriment of computer science (programming the robot).

We have now reviewed eight different systems across four different categories, and have identified many useful lessons and takeaways. We will now conclude Section 2 by attempting to synthesize the lessons learned into the description of an ideal introductory computer science educational system.

2.4.9. Creating the Ideal Introductory Computer Science Educational System

In this section, we synthesize the information collected in the review performed above and create a description of the "ideal" computer science educational system. This "ideal"

system features the benefits identified in the eight systems reviewed above, and avoids the problems that these suffer.

Our description will be divided up into 5 major parts – with each part containing both the features we wish to include, and the problems we wish to avoid, as pertains to that part.

1. **Drag and Drop Style Programming**

   Drag and drop programming in an introductory computer science educational system is an absolute must. As shown above, these languages allow students to dispense with learning complicated syntax and instead focus on the *idea* behind programming. A proper drag and drop language ought to make syntax errors impossible by not allowing programming blocks to be arranged in invalid ways. It also ought to design the blocks to reflect their use in their physical shape, color, or in other attributes, thereby giving students an indication of how these blocks properly go together. Finally, the actual language ought to be simplified and abstracted to a high level, to allow effective programs to be written without requiring students to fill in all the details. For example, a language designed around students driving a robot ought to have a single "drive" block, instead of forcing students to set up multiple blocks to turn individual wheels in a specific manner.

   The language should not be complicated, and should not require learning any more syntax or grammar rules than are absolutely necessary. On the other hand, it

also should not be drastically *oversimplified*, and should allow students to write programs of some depth and complexity. Most or all advanced techniques in programming such as nesting, recursion, etc. should be present.

Finally, a drag and drop language should attempt to resemble, at least in some ways, the structure of an actual program written in a formal language. Examples of how this can be achieved include ensuring that code reads roughly in one physical direction (versus, say, VPL's design that allows code blocks to be placed anywhere), and by separating different "levels" of code into different groupings (for example, a loop's body code may be grouped together in some way). In light of the finding that visually dissimilar languages can hinder student transfer to formal languages (Ryan Garlick, 2010), we believe that making a language visually similar to formal languages should counteract or even reverse this effect.

2. Virtual Worlds

A virtual world ought to be featured in an ideal educational system. The virtual world allows students to simulate their algorithms in an abstracted, simplified world which is beneficial to computational thinking education (David Barr, 2011). Papert's goal of "Low floor, wide walls, high ceiling" can be implemented in these virtual worlds. Finally, virtual worlds are inexpensive for classrooms, and can be designed such that they are easy to customize and extend (Thomas R. Flowers, 2002).

Because the execution of an algorithm in a virtual world happens on a computer screen, students may view both the executing algorithm and their program code at the same time. Furthermore, because the executing algorithm usually involves graphics moving around on screen (for example, Logo's turtle graphics, or Alice's character movement) these algorithms usually execute slowly enough for students to easily follow the algorithm code as it executes, thereby "pairing" the execution with the code. A really good virtual world should go one step further and explicitly highlight the code blocks currently being executed, to allow for easy tracking.

Finally, the virtual world should be designed so that interacting with it is fun for the student, in order to increase motivation and enjoyment.

3. Robots

In addition to a virtual world, a good educational system should feature a robot. Robots have been shown to vastly increase student fun, motivation, interest and engagement (Maja J Mataric, 2007) (Tom Lauwers, 2009) (Barry Fagin, 2003) (McGill, 2012). Furthermore, executing an algorithm on a physical robot in the real world adds a level of concreteness that virtual worlds do not achieve – the robot acts as a well-defined model executing in a familiar context, which increases student learning (Paul, 2012) (Wanda Dann, 2009) (Tucker Balch, 2008) (Tom Lauwers, 2009) (Thomas R. Flowers, 2002). For these reasons, a robot should be included alongside the virtual world. These two different media

of execution can be used in different scenarios where their different advantages can be maximized.

Because robots can be expensive, and because classrooms have different needs, an ideal system should be designed such that its algorithms can execute on different robot models. This will allow classrooms to choose a robot model that best fits their needs and their budget.

Furthermore, if possible, the system should be designed such that the code is *executed* on a local computer and *transmitted* to the robot, instead of having the code execute on the robot itself. This allows robots to be built in a simpler and less expensive manner, since they do not require expensive hardware to execute programs. Furthermore, by executing the algorithm on a computer, students can debug the algorithm as it executes on the robot.

Things to avoid when building the robot include making the robot prohibitively expensive, or too technically complex. The robot should be prebuilt if possible, to avoid requiring teachers or students to build it – this can scare less technically competent users away from the system, and can also shift the focus of the system away from computer science and towards mechanical engineering, as was seen in the FIRST Robotics Competition and Mindstorms sections above.

4. Curriculum

An educational system without a paired curriculum is no more than a tool. This tool may be used properly, or it may be used poorly, depending on the teacher utilizing it. Similarly, curricula may be made by third party users, and these may be good or bad. A better solution is for the developers of the tool to create an "official" curriculum and pair it with the tool, thereby making it a full educational system. In doing so, the curriculum design may be informed by the software design, and the software design may be informed by the curriculum design.

Our reviews above have identified some problems with systems that are heavily focused on competition (such as FIRST), creative "play" (such as Scratch) or storytelling (such as Alice). For some students, these focuses were distracting or hindering to their computer science education. It is not necessarily the case that this will be true for all students, and these focuses may be motivating or helpful for some students. In light of the fact that these things can be distracting, however, an ideal system should ensure that first and foremost it focuses on computational thinking education. Play, storytelling and competition must be secondary to this primary goal.

It may be that undirected play is most useful for students who have some foundation to work with, some basic orientation to point them in the right direction in which to learn. Ausubel has noted that students require "anchoring ideas" to properly orient themselves on a topic such that new knowledge can be retained (Ausubel, 1968). If this is accurate, then it follows that undirected play

coming after introductory orienting education can be quite educational – but play without this introduction will not produce any meaningful long term knowledge. Since we are concerned with creating an introductory educational tool here, we should be wary of organizing it around creative play, and make sure that our curriculum always establishes the orienting ideas first before turning students loose to discover knowledge on their own.

5. Other

The following items are design goals for the ideal curriculum that do not fit into the above four groups.

An ideal system should be extendable and customizable. This allows both easy additions to it by its maintainer, but it also allows end users (teachers or parents) to customize it to their student's needs. For example, by allowing the language to be customizable, teachers are able to add new programming blocks containing concepts that they wish to teach. By allowing the virtual world to be customizable, teachers may add new graphics or goals for the student to play with. In this way teachers can add lessons or concepts to the paired curriculum.

A beneficial feature, as we have seen with Scratch, is an integration with social media. Students should be able to upload their algorithms and share them with friends. Scratch has found that this increases student motivation, and also allows students to learn from each other's work (Mitchel Resnick, 2009).

> Finally, a system should limit its purpose to education. Systems such as VPL that attempt to be useful for both education and for industrial or specialized use run a high risk of being too technically complicated for easy student use – or, put another way, by raising the ceiling too high, they also raise the floor.

This concludes our review of newer educational systems. A chart comparing the newer educational systems discussed here, using the ideal features identified above as criteria, may be found in Appendix G.

Using this review, we have developed a new system for teaching introductory computer science and computational thinking. We have attempted to implement the features identified in this review as desirable, and we have attempted to avoid all the features identified as problematic. This system will be fully described in the next section.

3. DESCRIPTION OF GENOST

In Section 1, we described computational thinking and established the need for this subject to be effectively taught. In Section 2, we showed that traditional introductory computer science education does not effectively teach computational thinking, and that "newer" systems, while in many ways superior to the traditional systems in teaching computational thinking, also have many flaws. This brings us to the main subject of this thesis: the introduction and description of a new educational system designed using our

analysis of an "ideal" system above, and intended to effectively teach computational thinking. We have named this new system "Genost"[28].

Genost is a full educational system, including both an educational *tool*, consisting of a GUI, simulator and robot, and a *curriculum* (along with a third component, an administrative website for managing, customizing and tracking data from the system). In this section we will describe the Genost system, provide an overview of each part, and state our goals for its development, our justification for these goals, and how we implemented those goals.

We wish to acknowledge the contributions of the undergraduate students that assisted us in the development of the Genost software and robot. These students are: Rizwan Ahmad, Garth Bjerk, Tracey Heath, David Humphries, Corey Jallen, Ian Plumley, Stephen Pluta, Randy Queen, and Matt Rechia.

3.1. GENOST OVERVIEW

Before covering each component of the Genost system in depth, we will here provide a brief summary of the whole system. This will allow us to consider each individual part in context of the whole, and to speak on how these parts interact, without undue elaboration in each subsection.

[28] "Genost" is a truncation and corruption of the Greek adjective *gnostikos*, which translates to "cognitive" or "intellectual." The related noun, *Gnosis*, means "knowledge."

There are six major parts of Genost that deserve discussion. These parts are the **Language**, the **Mazes**, the **GUI,** the **Simulator**, the **Robot**, the **Management Website,** and the **Curriculum.** The way these parts work together is described below.

First, students use the **GUI** to create algorithms utilizing a visual drag-and-drop **Language**. The **Language** has a Turing-complete drag-and-drop design, which involves arranging blocks to create an executable program. Each block type represents a single fundamental programming concept. Students write algorithms in the language using the browser-based **GUI**, which is highly customizable and may be configured for different lessons from the **Curriculum.**

The problems that students create algorithms to solve are all based around moving a robot through a **Maze**. Each **Maze** is associated with a specific lesson in the **Curriculum;** these mazes are highly customizable and may have multiple objectives, such as driving the robot from one end of the maze to another, or collecting all the pickup objects ('coins') in the maze.

The actual **Maze** is implemented in either a *Virtual World*, in which case the algorithm controls a virtual robot moving in a **Simulator**, or it is implemented in the *Real World*, in which case the algorithm controls a physical **Robot** moving in a real-world maze. The ability to execute the algorithm with either a **Simulator** or a **Robot** allows for heavy customization of the Genost experience – classrooms can use the **Simulator** exclusively, which might be good for a small classroom or one with limited resources, or the

Simulator and **Robot** together, in order to get the motivational and physical benefits of using robotic education; a classroom could even use the **Robot** alone if they so choose. Ultimately, our review in Section 2 showed that both virtual world simulations and robots are valuable models to transfer computational thinking knowledge; a system with both simulation and robots allows teachers and learners to have the "best of both worlds".

The above tools may all be used in a **Curriculum** that takes full advantage of the software's capability to teach computational thinking skills. We created a **Curriculum** alongside the software during its development that focuses on teaching the fundamental programming structures and the ability to analyze and break down an algorithm. This **Curriculum** may serve as an "official core" for the Genost system. However, through the use of the **Management Website**, end users may create new lessons to add on to our core **Curriculum,** or create their own entirely. End users can also create class organizations, add students to the organization, and assign one or more curricula to their class – the **GUI** will then interact with the **Management Website** to walk students through their own class's curriculum, and collect data from the students as they go.

This is a brief overview of the Genost system as a whole. Each of the above major parts were designed to meet the goals of an ideal introductory computer science educational system described in Section 2.4.9, as well as to serve the more general goal of teaching computational thinking. In the remainder of Section 3, we will describe the development of each of these six major parts. For each item we will briefly describe the part of the

system, and will then describe the goals we had for its design, our justifications for these goals, and the ways in which we attempted to implement them.

Throughout this review we will make frequent reference to the computational thinking goals described in Section 2.1, the themes described in Section 2.4.1, and the ideal system goals described in Section 2.4.9. We will refer to these in the following manner:

- CG<X> will refer to computational thinking goal X described in Section 2.1.

- T<Y> will refer to theme Y described in Section 2.4.1.

- IG<Z> will refer to ideal system goal Z described in Section 2.4.9.

The first time a specific goal or theme is mentioned, we will briefly summarize it to ease in reading. These goals will be cited as justification for our designs; for justification of the goals themselves, please refer to the sections mentioned above.

3.2. THE LANGUAGE

The language that is used in the Genost software is a procedural drag and drop language that we call “Objective G”. To write algorithms with Objective G in the Genost GUI, students drag virtual blocks from block panels and drop them into the “canvas”. Students may arrange the blocks in different ways to create algorithms.

Objective G is designed to be simple, clear and high-level. The syntax is designed to be very easy to learn and understand. Because Objective G is a drag and drop language, syntax errors are impossible – the software will not allow an algorithm with improper syntax to be created.

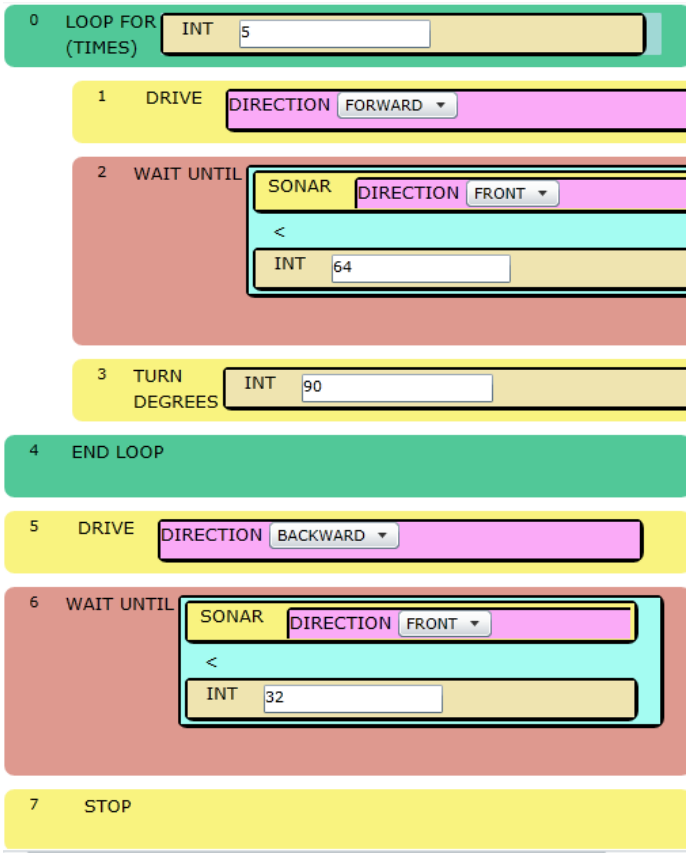


Figure 9. An example program in the Objective G language.

Each block in the Genost language represents a single fundamental programming structure: example blocks include Wait Until and Loop For, as can be seen in Figure 9. Some blocks are “standalone”, requiring no additional data to operate, but most blocks take one or more parameters, in the form of additional blocks. A block that requires a parameter will have a “socket” into which another block must be dropped. When a block

requiring additional data is placed into the canvas, its socket is blank and empty, indicating to students that it must be filled. All of the blocks seen in Figure 9 above require at least one parameter block, as can be seen by the filled sockets. One can also see how some parameter blocks have their own sockets, creating a chain of parameters – for example, the Wait Until block has an < (less than) block in its socket, which in turn has a Sonar block and an Integer block in its own sockets.

Genost code is written unidirectionally, top-to-bottom, and its blocks automatically indent when appropriate, such as in a Loop or If statement body. This can be seen above in Figure 9 – the top block in the algorithm is a Loop For, and the blocks in its body are indented. In this way, Objective G attempts to resemble the look and feel of formal programming languages.

Objective G allows deep nesting – there is no artificial limit on how far one can nest blocks – so that even complex algorithms can be written in Objective G. Other advanced techniques, like recursion, are also possible in Objective G.

Objective G has many different blocks, which are defined by a customizable XML file. We have divided the blocks up into eight different groups, which are defined below.

1. **Action:**

The Action blocks consist of all the blocks that command the robot to physically do something. Most Action blocks either tell the robot to move a certain way, or tell the robot to check a sensor and return a data value. All Action blocks are written at a high level – we have a single "Drive" block, for example, to tell the robot to drive forward, as opposed to requiring students to manipulate the individual wheels.

2. Data and Robot Data

The Data and Robot Data blocks represent different literal values and are inserted into the sockets of other blocks. The Data blocks include traditional primitives like Integer; the Robot Data blocks involve special robot-specific constants, such as Direction. These items are literals in the sense that they are not changeable at runtime, but their values are definable at design time.

3. Loops

The Loop blocks allow certain sections of code to be repeated. There are two loops in Objective G: Loop For, which is equivalent to a Java For loop, and Loop Until, which is equivalent to a Java While loop.

4. Wait Statements

A Wait statement prevents the Genost interpreter from proceeding onto the next line of code for a certain amount of time. While the program is waiting using a Wait block, any robot movement that is already occurring continues (if the robot

is driving, it will continue to drive). There are two Wait statements: Wait Until, which waits until a specified condition becomes true, and Wait For, which waits for a specified amount of time. Wait Statements are useful for writing algorithms that can, for example, drive the robot forward until it detects a wall in front of it.

5. **If Statements**

   If statements are equivalent in nature to their Java companions. Objective G incorporates If, Else If and Else blocks, to allow the creation of "If chains" that allow decisions with an arbitrary number of possible choices.

6. **Variables and Assignments**

   Objective G allows the creation of Variables of types Integer, String and Boolean. Assignment statements may be used to assign values to these Variables.

7. **Logic and Comparison**

   These blocks are used in the sockets of blocks that require a condition – for example, a Loop Until, a Wait Until or an If. Comparison blocks include all mathematical equalities and inequalities (Less Than, Less Than or Equal To, Equal To, Not Equal To, etc.). Logic blocks involve logical relations – AND and OR. Using these Logic and Comparison blocks, arbitrarily complex conditions can be defined.

8. **Methods**

Objective G allows the creation of Methods of type Void, Integer, String or Boolean. These Methods may take arbitrarily defined parameters (which are available as variables within the method body), and may return data of the proper type using a special Return block.

Each block type is colored differently, in an attempt to help students more easily differentiate them.

A full explanation of the blocks in Objective G may be found in Appendix B.

3.2.1. Language Design

Ultimately, our goal with Objective G is to create what Lu calls a "computational thinking language" or CTL. The most general goal of a CTL is to allow students to think about and learn computational thinking ideas without being required to spend considerable time learning the syntax and grammar of the language itself (James J. Lu, 2009). When designing Objective G, we used the following goals to guide us.

3.2.1.1. Goal 1: Language Readability

The idea here is that students should be able to read the language without having to learn much – or ideally, any – syntax and grammar. This is directly from T1 (Ease of Use) and is described specifically in IG1 (Drag and Drop Programming goals), which notes that a language should not be complicated for a student to read. This goal will ultimately help

with general computational thinking education, but especially with CG1 (the ability to read and understand algorithms.)

Our primary attempt at implementing this was in using clear English labels for our programming blocks, and attempting to make the block code itself read like an English sentence. For example, a fully configured Loop For block that repeats 5 times reads left to right, “Loop for 5 times”.

The physical layout of the blocks and their sockets attempts to contribute to the above effort of reading like an English sentence by being organized in a sensible manner. Using the Loop For example from before, the socket to indicate the number of times to loop is located between the text “Loop for” and “times”, hinting that the parameter indicates the number of time the loop should iterate.

We have also colored the blocks differently depending on their types, to assist students in telling them apart.

3.2.1.2. Goal 2: Ease of Programming

The basic idea behind this goal is that developing a program in this language should not require more effort than necessary. Once again, this goal comes directly from T1 and IG1. Furthermore, this goal also serves T2 (Fun) – a language that is easy for students to develop programs in will almost certainly be more fun for them than one that is difficult.

Naturally, the primary way we attempted to make the language easy to program in is by making it a drag and drop language! As noted above, drag and drop languages disallow syntax errors, which reduces the difficulty of programming considerably. Aside from this, efforts to make the language easy to program with include creating a one to one relationship between blocks and concepts – each block represents a single fundamental structure, meaning that students can add in that structure by dragging and dropping a single block. When a concept requires multiple blocks to be complete (such as a Loop For requiring an Integer block to inform it how many times to iterate) we have made the empty socket quite obvious so the learner can easily see that more blocks are needed to complete the structure. Finally, by limiting the system to procedural programming, we are able to implement all of the concepts we wish to implement without considerably complicating the GUI (procedural programming will be discussed more in the next subsection).

As mentioned above, the Objective G blocks are colored differently depending on their type. This is our first attempt at using the physical appearance of the block to indicate what it does, a goal described in IG1. With practice, students can learn that a block colored tan represents a Data parameter, for example. We plan in the future to alter the blocks' and sockets' physical shapes as well to indicate their functions.

3.2.1.3. Goal 3: Procedural Programming

We have decided to limit Objective G to procedural programming, and have not included object or class designs in the language. This section will justify this decision.

Our goal for Genost is to teach *introductory* programming. At the heart of all programming paradigms lies procedural programming concepts – control flow, variables and functions. Therefore it makes sense to focus on these procedural aspects in introductory programming. Furthermore, procedural programming concepts, especially control flow, are at the heart of T4 (computational thinking). Specific skills that procedural programming teaches especially well are CG2 (the ability to engage in abstraction) and CG3 (the ability to decompose a problem into processes). Procedural programming is also fundamentally *easier* to learn than the more advanced paradigms that grow from it. Therefore, focusing on procedural programming helps us fulfill T1 (ease of use).

In order to focus on procedural programming, we limited the blocks available for students to use to the following: basic Actions (driving, turning), Loops, Wait Statements, and If Statements. We also included Variables and Functions. No other block types, such as classes or objects, were implemented; those blocks that were implemented do not feature object-oriented behavior.

We believe that procedural programming allows us to teach all of the computational thinking goals that we wish to teach, and that the benefits in simplicity and clarity makes the decision to focus on procedural programming alone worth it. Adding in features from more advanced paradigms, like OOP, would not seem to gain us any additional advantage

in terms of introductory computational thinking education, but it would complicate the language (and the GUI) quite significantly.

3.2.1.4. Goal 4: Computational Thinking Built Into Language

We wished to design Objective G such that the very act of programming in the language should reinforce certain computational thinking concepts. In other words, even independent of the particular lesson being undertaken, simply writing a program in the language should suggest or reinforce computational concepts, due to the language's design. An example of how this may be achieved (which we implement, as we will describe later) is by visually grouping blocks (using indentation, highlighting, or some other measure) together at a certain level of abstraction. This goal directly benefits T4, and depending on how it is implemented may serve CG1 or CG2.

We have attempted to implement this goal in many ways. As mentioned before, we have made each block represent a single concept, in order to implicitly show the separation and differences between these concepts, and to assist students in learning those differences. A more interesting design choice was to force students to program "outside-in" – that is, when adding a block such as an If statement, students must first place the If block, and only after the block is placed can they add to its body. Another example of this "outside-in" design would be the way a complicated conditional statement – say, ((X > Y) AND Z) – is built. In order to build this in Objective G, students must first place the AND block, then place the > block inside the AND's left socket, and finally place X, Y and Z inside their proper places. This "outside-in" or "top-down" style of programming

pushes students to think of the outer block in context of its own level of abstraction, and the inner blocks that go within the outer block's body as a different, lower abstraction level.

Abstraction and parameterization is further reinforced through certain design choices in the Action blocks. Most Action blocks in Objective G explicitly require a parameter block to be added – for example, the Drive block requires students add a block to tell it to drive either Forward or Backwards. This reinforces abstraction by encouraging students to think of Actions as abstract entities, only becoming concrete when adding a parameter indicating how to perform that action. By choosing to parameterize these Action blocks instead of rolling action and parameter into a single block, these computational thinking ideas are reinforced.

Finally, we allow students to fully explore computational thinking by not oversimplifying Objective G. The language is Turing-complete and programs of arbitrarily high complexity may be written in it. This is powered by, among other things, the inclusion of deep nesting and recursion.

3.2.1.5. Goal 5: Similarity to Formal Programming Language

We want the act of programming in the language to look and feel similar to programming in a formal programming language. We chose this goal for two reasons. The first reason is practicality: eventually students will need to move from Objective G to a formal language like Java or C. We want to make this transition as easy as possible, as stated in IG1 – we believe that making our language look similar to a formal language will assist

in this eventual transition. The second reason has to do once again with computational thinking: we have observed that the structure of real languages often reinforces certain computational thinking concepts, including CG1 and CG2. Examples of this include the inherent unidirectional code flow of real languages (CG1) and the clear "separation of concerns" achieved by the indentation or demarcation of loop, if statement, or function bodies, which reflects the different levels of abstraction in an algorithm (CG2). Since formal programming languages have these benefits, designing our language to look similar to those languages will ideally bring us those benefits as well.

We designed Objective G's look and feel with this goal in mind. Unlike languages such as NXT or VPL, Objective G reads explicitly unidirectionally, top to bottom, just like a formal language. Objective G also automatically indents loop bodies, if bodies, and the like, which is also not the case in NXT or VPL.

We mentioned when discussing Goal 1 that we have attempted to use clear English labels for our blocks. When possible, we tried to use the same labels that are used in actual programming – for example, "If" or "Loop".

3.2.1.6. Goal 6: Design Conflicts

It is inevitable that the five above goals will conflict, and we need some way of resolving this conflict. Due to the fact that teaching computational thinking is our goal above all else, as stated in T4 and IG5, we resolved these conflicts by choosing the solution that was best for computational thinking education.

In our implementation, specific compromises *were* made between some of these goals to support computational thinking. For example, we realized that the "outside-in" programming style may be somewhat confusing to students, and is certainly pedantic in certain cases, violating Goals 1 and 2. However, as we have argued in Goal 4, this practice reinforces computational thinking, and therefore we kept it in.

In a very similar way, if we had rolled Action parameters into the Action blocks themselves, instead of requiring students to explicitly add these parameters to the blocks, we could have saved a step when adding the actions. That we did not do so again violates Goal 2. Once again, however, the computational thinking utility of this practice outweighs the difficulty this introduces into programming.

3.3. THE MAZE

The Genost language, and the curriculum, are centered on mazes. A "maze" in Genost is a 2D plane containing a movable item ("robot") along with obstacles, items that may be picked up ("coins"), and goals. Students control the robot by writing an algorithm in Objective G. Each maze has a specific goal, and it is this goal that students write their algorithms to achieve. Note that the term "maze" does not necessarily imply that any one design is mazelike (for example, it might be a single straight corridor) and similarly, the terms "robot" and "coin" do not imply that the graphics representing those concepts will actually look like a robot or a coin! Figure 10 shows a sample maze, with robot and coins.

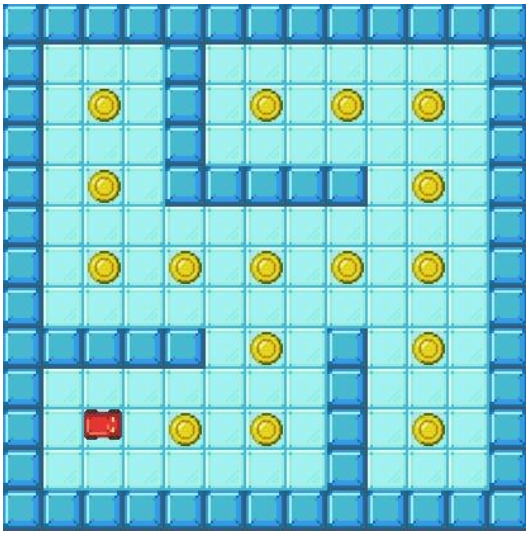


Figure 10. An example of a simulated maze in Genost. The red square is the robot, the dark blue blocks are walls, and the yellow items are coins. This maze's goal is to collect all the coins.

Our mazes are implemented in a simulated virtual world, and can also be implemented in the real world using a real robot. In the simulated mazes, goals are defined according to the maze and success or failure is detected automatically.

In our simulated mazes, there are four possible goals:

1. **Drive to the Finish and Stop**

   In this goal, a student must write a single algorithm which will drive the robot through the maze, reach a special square known as a "finish zone", and complete the algorithm while stopped on the space.

2. **Collect all the Coins and Stop**

   In this goal, a student must write a single algorithm which will drive the robot around the maze and pick up each coin in the maze. After picking up the last coin, the robot algorithm must end with the robot stopped.

3. **Drive to the Finish**

   This is the same as the Drive to the Finish and Stop goal, except the algorithm is not required to end with the robot stopped. Instead, so long as the robot touches the finish zone, the algorithm will be considered a success.

4. **Collect all the Coins**

   This is the same as the Collect all the Coins and Stop goal, except the system does not require the robot to stop and complete its algorithm. Instead, the algorithm will be considered successful the second the robot picks up the last coin.

In a real world maze, the success and failure criteria would be defined and judged by an external observer, such as a teacher, advisor or coach. The above goals could be implemented in the real world maze without considerable trouble.

We use mazes implementing the goals above to teach students computational thinking skills.

3.3.1. Maze Design

Considerable thought was put into deciding to center Genost on maze solving, and once this decision was made, more thought was put into how the mazes should function and the goals of the mazes. The thought processes that led to these decisions will be described here.

3.3.1.1. Goal 1: Teach Computational Thinking

A primary goal was for whatever goal Genost's algorithms were centered on achieving to teach computational thinking effectively. Mazes were ultimately chosen because there are certain features of mazes that make them very effective for teaching computational thinking. A primary computational thinking benefit of maze solving is the fact that a maze solution is an inherently visual one – a student can watch his robot executing each step of his algorithm in real time. Resnick has noted that watching a robot move through a maze is an act of "reflection and evaluation" that is crucial to the learning process (Resnick, 2007).

When watching a robot move through a maze, students can see the algorithm itself operating in real time. Every algorithm step may be seen in the robot's movement. If the algorithm fails, it is immediately obvious that the failure has occurred (i.e. the robot turned the wrong way and crashed into a wall) and students may determine at what point in the algorithm this happened (i.e. immediately after it drove down the third corridor) without considerable difficulty. In this way the ability to watch a robot execute an algorithm helps fulfill CG1, the ability to read and understand algorithms.

We can compare watching the robot's movement to a more traditional way of testing algorithms to further see the benefit. Traditional program testing tends to only show the final output, if such output is generated in the first place. For example, a number sorting program ultimately produces a list of numbers that is either sorted or it is not, assuming the program does not crash. Students cannot easily watch this algorithm execute. CG1 is therefore served far better with maze solving than more opaque algorithmic tasks.

Another benefit of mazes is that a maze is its own model – the problem that a student must solve is clearly visible in 2D form. When solving the maze, it is often useful to focus on solving individual parts of the maze by themselves, and then combine those solutions to make a single algorithm. This is the essence of abstraction (CG2) and problem breakdown (CG3), and we use this technique heavily in our curriculum. Because the maze is its own model, when performing this abstraction and moving between levels, students may literally focus their view on the part of the maze they are working with at the time. Once again, this task is more difficult with more opaque computer tasks such as number sorting, since the different "areas of concern" are entirely invisible. With mazes, they are visible. This assists very strongly with CG2 and CG3 – we believe that mazes help students have an easier time abstracting and breaking down a problem.

When watching a maze algorithm execute, inefficiencies or problems in the algorithm may become very clear. If the algorithm fails, the failure will be visible as a robot crashes or takes a wrong turn. If the algorithm is inefficient, the inefficiency will be visible as the robot traverses the same corridor multiple times, or otherwise performs unnecessary

actions. The visibility of algorithm quality assists students with CG4, learning to evaluate algorithm quality.

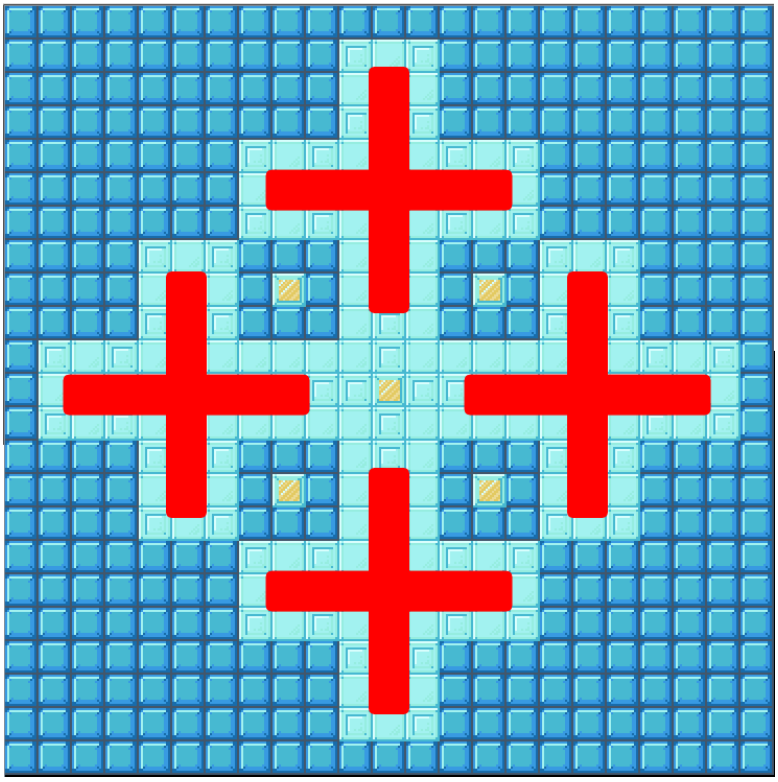


Figure 11. A maze physically "broken down" into similar parts. This breakdown, which can be done visually, helps students with CG2 and CG3.

In this way, mazes can be used to teach all four parts of computational thinking. Our implementation of the mazes, specifically the goals that we chose to include, also help with teaching computational thinking.

For many of our mazes, such as the maze in Figure 11, multiple solutions are possible – this fact is used to help teach CG1, the ability to understand algorithms, since

understanding that the same task may be solved with multiple algorithms is an important step in this computational thinking goal.

The "X and stop" goals introduce an important requirement into the algorithm development process: students must design not just how their algorithm will run but also how it will end. Once again this is a fundamental part of computational thinking and helps round out CG3, the ability to break down a problem and design a solution.

The two main goals – Drive to the Finish, and Collect all the Coins – have different focuses. Drive to the Finish teaches the development of a minimal algorithm, since all that is needed is to get to the end. Meanwhile, Collect all the Coins can be used to teach algorithms focusing on completeness, since the entire maze must be traversed to ensure that every coin is collected. These are again two important elements of computational thinking.

Finally, it is possible to design mazes such that a single algorithm will solve multiple mazes. A lesson, then, may contain multiple mazes that all must be solved by the student with one program. This technique, which we make heavy use of, requires the student to generalize and abstract to a great degree, and is therefore a great way to teach CG2.

These benefits, and more, are the reason that we chose to center Genost on mazes.

3.3.1.2. Goal 2: Simple and Easy to Understand

T1 tells us that our system should be easy to use, and this extends to the mazes. IG2 notes that virtual worlds are superior to traditional algorithm tests because they are easier – as our goal is to make the best system possible, we therefore want to make the mazes as easy to understand as they can be. Finally, Papert's "low floor" is another way of saying that these systems should be easy to understand (Papert, 1993).

Note that "easy to understand" does not necessarily mean "easy to solve". Students should be challenged by the task of creating a high-quality algorithm to solve the maze; the challenge should not come from difficulty in understanding the rules of the virtual world, or the goal that they are being asked to solve.

We have attempted to implement this goal through the use of clear graphical themes and simple to understand goals. The choice of using a 2D world instead of a 3D world was also motivated by this goal: by limiting ourselves to two dimensions, we make the mazes simpler without losing too much richness or potential. We have also limited the number of actions the robot can perform to only those that are necessary, instead of allowing it to perform a very large number of actions that are individually only rarely useful.

3.3.1.3. Goal 3: Rich Interactions

Rich interactions refers to the ability for a maze system to implement many different kinds of tasks, and for each task to have depth to it. The richness of virtual worlds when compared to traditional algorithm tests is a benefit described in IG2 – this is the "wide

walls" feature that Papert describes. Interestingly enough, the richness of the mazes also enables, to some degree, the "high ceiling", as relatively complicated puzzles can be implemented in these mazes (Papert, 1993). This goal also fulfills T3, Adaptability – a maze system with rich interaction is ultimately more adaptable, as more examples and concepts may be taught with it without requiring additional development.

We have tried to make our mazes highly customizable – the obstacles can be arranged in any way, as can the coins and finish zones. The mazes can be any shape or size. In this way a vast array of different mazes can be created. As described above, we have four different possible maze goals, applicable to different situations – these goals may be applied to any maze. We believe that these features taken together make our application very rich, and virtually any concept we wish to teach can be built into a maze.

3.3.1.4. Goal 4: Fun

Fun is one of our major themes (T2). We want all student interaction with our system to be fun – its benefits have been much discussed. We therefore wish to make our mazes fun for the students to solve.

This goal is somewhat solved through the use of mazes themselves, as we believe that the art of solving a maze is itself fun for the students. Other ways we have attempted to make solving the maze fun is through the use of bright graphic and sprite-art themes, which mimic video games. Anecdotally, we have found that students quite enjoy these video game connections.

3.4. THE GUI AND SIMULATOR

Two online systems comprise the core of the Genost software. These systems are the GUI and the simulator. The GUI refers to the software with which students develop their algorithm; the simulator refers to the software with which they can test the algorithm in a virtual world. Both of these systems are web-based.

3.4.1. GUI Description

The GUI is a Microsoft Silverlight[29] program and may be seen in Figure 12 below. It consists of a number of major parts, each of which we will briefly describe.

- **The Canvas**

  The "canvas" is the center part of the Genost GUI, and is where code blocks are dropped to assemble an algorithm. Users can drag the blocks from the block panels to the left to insert new blocks, or they can drag blocks from inside the canvas to new positions to rearrange them.

- **The Block Panels**

  The two panels on the left containing the grouped code blocks are the Block Panels. The top one, the Robot Functions panel, contains blocks related to the robot's actions; the bottom one, the Program Structures panel, contains blocks related to the general programming structures such as Loops or Ifs.

[29] http://www.microsoft.com/silverlight/

These blocks are loaded in from an XML file, which may be customized to add new blocks or alter existing ones. An different XML file is loaded each time a new lesson is loaded.

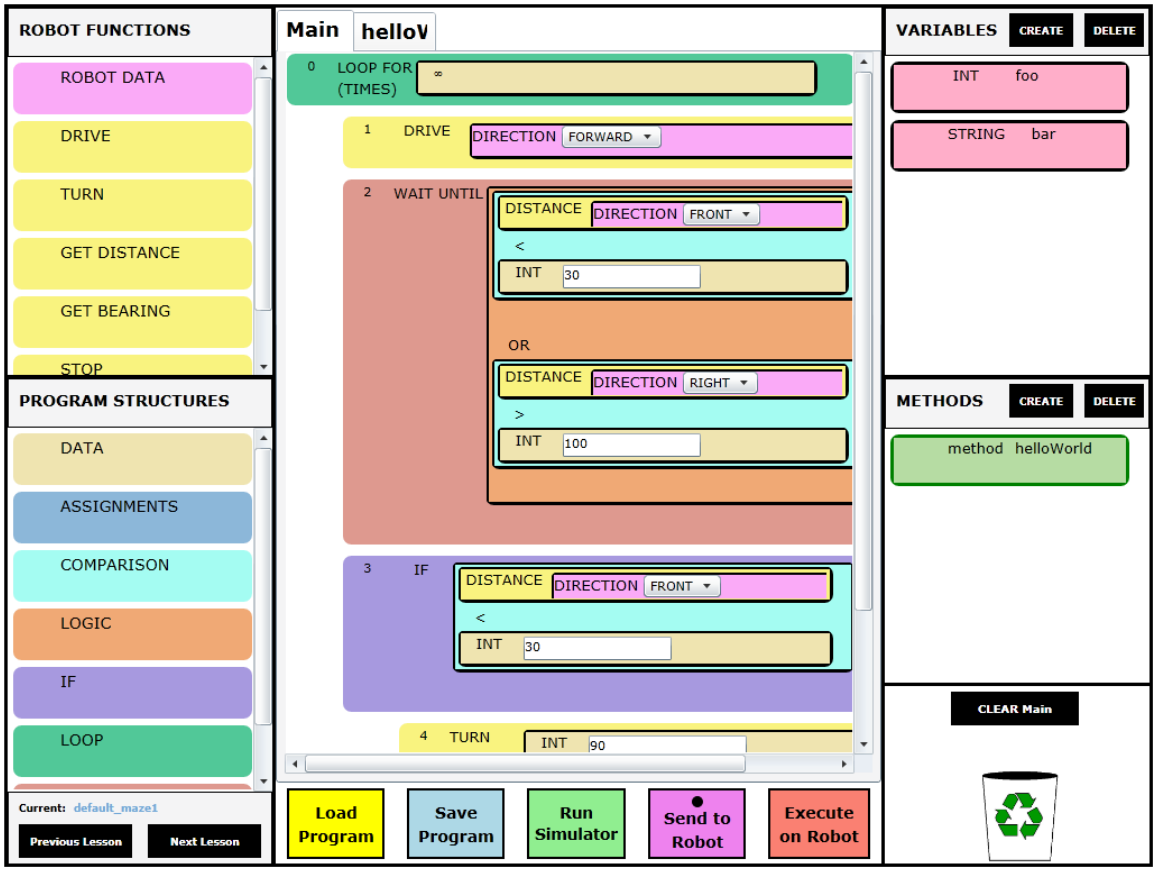


Figure 12. The Genost GUI

- **The Lesson Selection**

  Beneath the Program Structures panel is a pair of black buttons and a “Current” link. These make up the Lesson Selection panel, which allows students to view information about, and change, their current lesson. A Lesson is part of a Curriculum, and contains a block definition file and a maze; when a lesson is

selected, these files are loaded into the GUI. Using the Previous and Next buttons, students may change to the previous or next Lesson in the Curriculum. Using the Current link, they may view an image of the current maze they are solving.

- **The Variables Panel**

  Using the Variables panel on the right, students may define new variables of different types. Once a variable has been defined, it appears in the Variables panel. Users may use the Create button to create new variables, or the Delete button to get rid of old ones.

- **The Methods Panel**

  The Methods panel, like the Variables panel, allows users to create new methods. When a method has been created, a block to call that method appears in the Methods panel, and a tab for that Method appears at the top of the screen. By clicking on the Method tab at the top of the screen, users can access the Methods Definition Screen. Users can Create and Delete methods using the black buttons at the top of the Methods panel.

- **The Trash Panel**

  The bottom right corner of the GUI features the Trash panel. Users may drag blocks to the Trash to delete them, or they may click the Clear button to clear the current canvas.

- **The Method Definition Screen**

  When a method has been defined, clicking the tab at the top of the screen with that method's name allows a user to access the Method Definition Screen. Doing so will change the canvas to the Method's body instead of the Main canvas. On this screen users may define the details of a method's parameters, return type and body. This screen may be seen in Figure 13.

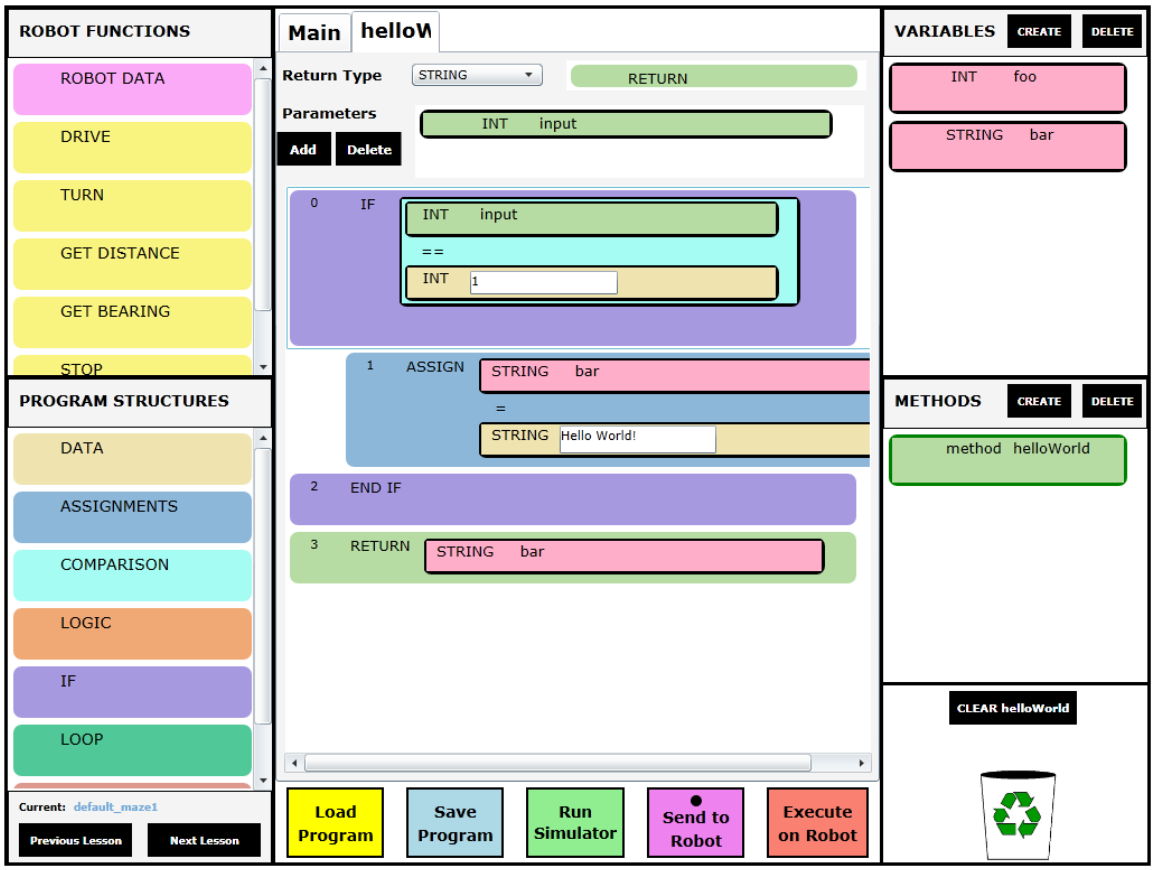


Figure 13. The Method Definition Screen in the Genost GUI

- **The Control Panel**

  The Control Panel is a series of five buttons beneath the canvas. These five buttons are:

  1. The Load Program button, which allows students to load a saved program from a file.
  2. The Save Program button, which outputs the current program to a file.
  3. The Run Simulator button, which will launch the Simulator to run the current program.
  4. The Send to Robot button, which will transmit the current program to a connected robot and make it ready to execute.
  5. The Execute on Robot button, which will send an execute command to a connected robot. When the robot is executing, this button turns into a Stop button to stop the robot.

3.4.2. Simulator Description

The Simulator is a Java applet that is launched when a student clicks the "Run Simulator" in the GUI. When this button is clicked, the GUI sends the code to the Simulator through a web service and opens a new window with the Simulator in it. The Simulator may be seen in Figure 14.

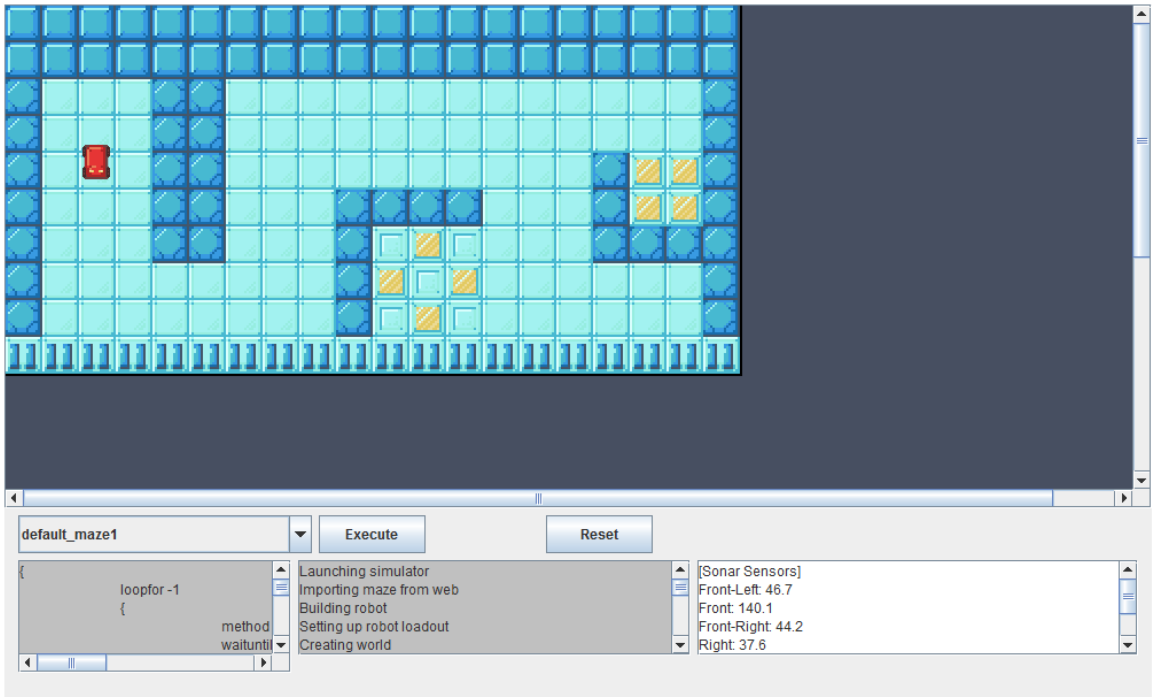


Figure 14. The Simulator

The maze can be seen in the top part of the Simulator. This window is able to scroll if the maze becomes too large.

Students click the "Execute" button to run the algorithm they have developed. The algorithm itself may be seen (in text code form) in the panel in the bottom left hand corner of the Simulator. Students may reset the Simulator at any time by pressing the "Reset" button.

Other notable panels here include the bottom middle panel, which contains output and debug information from the simulator, and the bottom right panel, which provides an accurate readout of the simulated robot's sensor data.

3.4.3. GUI and Simulator Design

The GUI and Simulator are vehicles for delivering the Objective G language and the virtual worlds and mazes – we have already discussed how these items teach computational thinking in Sections 3.2 and 3.3. Therefore, in discussing the design of the GUI and simulator we will focus on their technical design goals instead of any educational design goals, since the latter have already been discussed in the two mentioned sections.

3.4.3.1. Goal 1: Clear, Informative, Intuitive Design

We want our software interfaces to be clean, easy to understand and easy to use. We want these interfaces to provide all the information that a student needs, and none that they do not. We want students to be able to figure out how to use the software without too much trouble. Above all, we do not want the interface to be confusing. This is, of course, justified by T1, Ease of Use, as well as general aesthetics and common sense.

We have attempted to implement this by using a clean design, and making good use of grouping. For example, the various Genost code blocks are grouped in the GUI between two panels – one for general programming blocks, and one for robot-specific blocks. The blocks in these panels are grouped yet again by common theme – for example, the Loop For and Loop Until blocks are grouped under a single Loop header. These headers are collapsible so learners need only see the blocks they are presently concerned with. In this way, the block grouping produces not only a clean interface, but also an informative hierarchy.

Just as with Objective G, we have attempted to use clear English labeling of all parts of the GUI. This labeling serves both to inform and to prevent confusion. We try further to prevent confusion by separating the blocks by different colors.

In the Simulator, we have tried to maintain a clean design, but have also presented information where needed. For example, the sonar sensor output, and the simulator debug console, are both present in order to give students information about the state of the simulated robot, and the algorithm's execution.

We chose to run the simulator in a separate window from the GUI, so that students could have both windows open at the same time. This allows students to follow along with their algorithm as the robot runs it, thereby "pairing" the robot movement and the code execution, a goal explicitly stated in IG2. Currently the GUI does not explicitly highlight code blocks as they are executed, an item we noted would be beneficial; we hope to implement this feature sometime in the future.

Ultimately, this goal is a matter of aesthetics, but we believe the decisions that we have made have resulted in a clean and easy to understand interface.

3.4.3.2. Goal 2: Adaptability and Customizability

We designed the GUI and simulator to run in the web browser with almost no prerequisites or requirements. All that is required to run these items are the Java and Silverlight browser plugins, which are very commonly installed and are free to use. This

makes Genost highly *adaptable*, a goal described in T3, as it can be used on any computer, in any lab – it requires no specialized software or equipment. This design also helps implement T1, ease of use, as teachers and students need not deal with advanced technical complexity to set up and use the software.

Furthermore, as described above, both the Objective G language and the Mazes are designed to be customizable. In order to take advantage of that, the GUI and the simulator are designed to be customizable as well. Both the code blocks in the GUI and the maze in the Simulator are customizable. These items are defined in XML files and are loaded into the software when needed. New mazes to teach new computational thinking skills can be easily created using a maze development kit that we have developed; new code blocks may be created by adding new XML to the block definition file. This has allowed us to easily develop a curriculum to test, and it will allow easy development of new lessons in the future. This goal of customizability is directly stated in T3; it also helps us with T4, the goal of teaching computational thinking, as the customizability of the system allows us to try many ways of teaching the various computational thinking skills to find those that work the best.

3.4.3.3. Goal 3: Management Website Integration

The Management Website, as its name implies, manages the Genost system. The most visible way that this management takes place is by sending data to, and receiving data from, the GUI and simulator. The management website allows teachers to define student logins, which the students actually use to access the GUI. Once logged in, the

management website sends lesson information to the GUI for the student to iterate through. The actual lessons, which contain the mazes and code block definitions described in the previous section on customization, are hosted on the management website and are sent to the GUI and simulator when needed. Finally, the GUI and software send data about the student's performance and interaction with Genost back to the management website for teachers to peruse.

The integration between the management website and the software make this software far easier for teachers to use in teaching a class, which therefore serves T1. This feature also allows the customization we have built into the software to actually be used effectively, and therefore this feature also serves T3.

3.4.4. GUI and Simulator Technology

We have described above the visual and functional design goals for the GUI and simulator. In this subsection we will describe the actual technology that powers the GUI and the simulator. We will describe the technology that powers the GUI and simulator individually, but the way in which these two disparate systems communicate with one another is, in our opinion, the more interesting and novel technology. We will briefly describe this method of communication now.

The main reason for communication between the GUI and the simulator is to allow transfer of the student-developed algorithm from the GUI to the simulator for execution; the algorithm itself must be sent, along with information about what environment (i.e. the

maze) the algorithm ought to be executed in. This is accomplished by first transforming the graphical GUI algorithm into text code and sending it, via RESTful web service, to the simulator. RESTful services are also used to allow the simulator and GUI to communicate with the management website. The fact that these three different systems – written in three different languages – can communicate with each other via the REST protocol is a testament to the language-independence of the service-oriented programming paradigm (Yinong Chen, 2014).

In the subsections below we will describe the GUI and Simulator systems individually in some depth. We will also more deeply describe the technology that facilitates the RESTful transfer of the algorithm from GUI to simulator as described in the previous paragraph. We will finish with a discussion of some of the technical challenges we faced in implementing these items, and how they were overcome.

3.4.4.1. GUI Technology

The GUI, as mentioned previously, is built in Microsoft Silverlight. Silverlight is a plugin for web browsers that is powered by the .NET framework, and enables the development of rich applications such as Genost. Silverlight features a robust drag and drop API that was used in the Genost GUI to enable the dragging of blocks back and forth between the various sockets, panels and canvases. C# is the language used in all .NET frameworks, Silverlight included, and hence it is the language that the Genost GUI was built in. The

use of Silverlight allows the GUI to run across all operating systems and in any browser (so long as it implements a Silverlight plugin), which makes the software very widely usable.

Above, we have described the ability of the GUI to be customized. All programming blocks that may be manipulated within the GUI are loaded and defined at runtime. These blocks are defined in a "toolbox" XML file that contains a definition for each block. Each block definition includes the block name, block color, block type, as well as specifications for where the block may be placed, whether the block has any sockets, and whether it has a body, among other things.

```
196     <Package name="COMPARISON">
197         <Block>
198             <name>COMPARISON-greater</name>
199             <!--<name>></name>-->
200             <type>PLUGIN</type>
201             <contains>
202                 <socket>VARIABLE/METHOD/CONSTANT/PARAMETER</socket>
203                 <string>&gt;</string>
204                 <!--is greater than-->
205                 <socket>VARIABLE/METHOD/CONSTANT/PARAMETER</socket>
206             </contains>
207             <properties>
208                 <color>164 252 242</color>
209                 <socketsMustMatch>true</socketsMustMatch>
210                 <intDisabled>false</intDisabled>
211                 <stringDisabled>true</stringDisabled>
212                 <booleanDisabled>true</booleanDisabled>
213                 <isCondition>true</isCondition>
214                 <hasSocks>true</hasSocks>
215                 <transformer>true</transformer>
216             </properties>
217         </Block>
218         <Block>
219             <name>COMPARISON-less</name>
220             <!--<name>&#60;</name>-->
221             <type>PLUGIN</type>
222             <contains>
223                 <socket>VARIABLE/METHOD/CONSTANT/PARAMETER</socket>
224                 <string>&lt;</string>
225                 <!--is less than-->
226                 <socket>VARIABLE/METHOD/CONSTANT/PARAMETER</socket>
227             </contains>
228             <properties>
229                 <color>164 252 242</color>
230                 <socketsMustMatch>true</socketsMustMatch>
231                 <intDisabled>false</intDisabled>
232                 <stringDisabled>true</stringDisabled>
```

Figure 15: Sample Toolbox Definition XML

The toolbox file defines all the block interactions that are capable within the GUI. This fact allows us very precise control over what kind of algorithm a student may build within a single lesson.

The GUI utilizes many RESTful services to communicate with the management website. When a student first accesses the GUI, he is asked to enter his username and password. When the student clicks the "login" button after entering this login information, these credentials are sent to the management website via a RESTful call. The management website validates the student's username and password and, if they are valid, returns the ID of the current lesson the student is working on. The GUI may then call other RESTful services, using the lesson ID as a key, to load up the toolbox and other information necessary for configuring the GUI.

After a student has used the drag-and-drop functions of the GUI to build an algorithm, the student may send it to the simulator (or robot). When the student chooses to test his algorithm by sending it to one of these systems, the GUI first transforms the graphical algorithm it into text code. The language that we translate into is a formal, textual version of Objective G. A sample of the formal Objective G text code may be seen in Figure 16.

This algorithm, once transformed into text, is transmitted to the Simulator or Robot. The process by which this transfer is performed will also be discussed in Section 3.4.4.4.

```
 4 {
 5   if ( [ method getSonars  ( int 5 ) > int 128 ] )
 6   {
 7     method turnAngle  ( int -90 );
 8   }
 9   elseif ( [ method getSonars  ( int 3 ) > int 128 ] )
10   {
11     method turnAngle  ( int 90 );
12   }
13   method drive  ( string "f" );
14   waituntil ( [ method getSonars  ( int 1 ) < int 32 ] );
15   method stop ();
16
17
18 }
```

Figure 16. An Objective G algorithm written in formal text code

3.4.4.2. Simulator Technology

The Simulator is built in Java, and runs as a Java applet. Just like the GUI, the Simulator applet runs in any browser and on any operating system, so long the Java plugin is installed. We used Java's Swing library to power the graphics in the mazes. Each individual graphical item in the maze is defined by its own class. Intelligent use of object-oriented concepts such as inheritance and encapsulation allow us to distribute the behavior of the items within the maze (the obstacles, the coins, and most importantly the robot) across several different classes.

When starting the simulator, the actual maze is loaded using, once again, an XML file that is retrieved via a RESTful service from the management website. The maze XML file describes the position, type, and graphic of each individual item within the maze, as well as global settings such as the maze goal. The images themselves are loaded via REST from the management website as well. In this way the actual Simulator applet contains no local XML or image files that must be referenced.

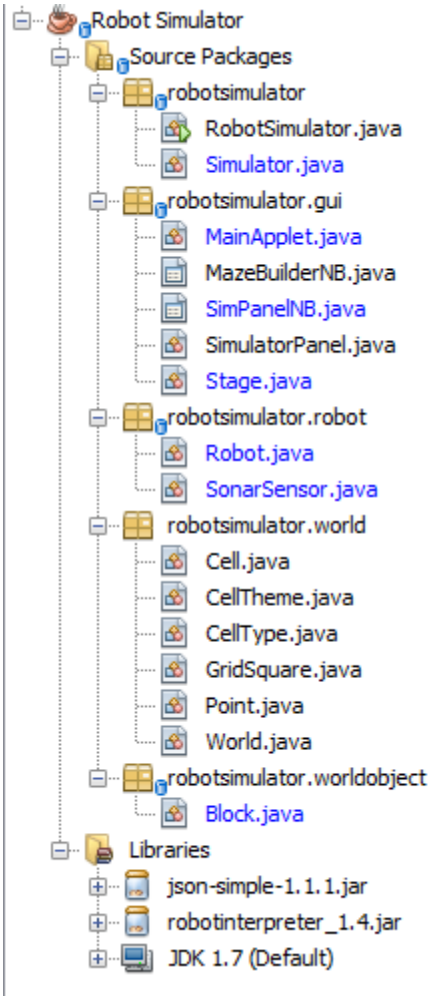


Figure 17. The Simulator classes

In addition to loading the maze at startup, the actual algorithm is loaded at startup as well via REST. In order to execute the algorithm, we send the text code to another Java executable that runs within the simulator which we call the "interpreter". The interpreter will be described in the next section.

```
<stage finishMode="2" guiheight="288" guiwidth="320" theme="loz">
    <world>
        <gridwidth>32</gridwidth>
        <gridheight>32</gridheight>
        <cells>
            <cell>
                <x>192.0</x>
                <y>128.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>224.0</x>
                <y>128.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>64.0</x>
                <y>192.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>96.0</x>
                <y>192.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>224.0</x>
                <y>192.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>192.0</x>
                <y>192.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
            <cell>
                <x>192.0</x>
                <y>160.0</y>
                <a>90.0</a>
                <celltype>loz_wall13</celltype>
            </cell>
```

Figure 18. Snippet of XML from a maze definition file

The Genost simulator is heavily inspired by the "eRobotic" simulator[30], also developed at ASU, which also features a robot navigating a 2D maze. This simulator was created to test the Robot as a Service (RaaS) paradigm (which will be described further in Section

[30] The simulator may be viewed at http://venus.eas.asu.edu/WSRepository/eRobotic/; a screenshot of the simulator may be seen in (Chen Y. H., 2013).

3.5) and does not allow the easy creation and simulation of arbitrary algorithms. However, its physical appearance and robot design are roughly similar (Chen Y. H., 2013).

3.4.4.3. Interpreter Technology

The interpretation and execution of the student algorithm is performed by the Genost Interpreter. The Interpreter is a Java package which must be implemented by another program, such as the Simulator or the Robot core code. The Interpreter defines many events which the implementing system must create handlers for; these handlers are then called by the Interpreter as needed when it executes an input algorithm.

Like the Simulator, the Interpreter is defined in a highly modular way and makes deep use of object-oriented inheritance and other concepts. Each programming structure defined in the Objective G design is implemented as a class with three methods: a constructor to parse the text code of the structure and turn it into an object, a validate function to ensure there are no syntax errors in the structure instance, and an execute function that actually executes the structure when appropriate. The class tree for the Interpreter can be seen in Figure 19 (not all classes are visible).

When the Interpreter receives a text algorithm as input, it immediately begins parsing it line by line. As the Interpreter parses it builds an object tree, turning textual representations of the various program structures that make up the algorithm into objects which may be validated and executed. The parsing is done using a large recursive loop

which creates, in essence, a very large linked-list with the first line of the algorithm at its head. Any typos or syntax errors in the algorithm are identified at this stage; if such a mistake is found, a Java error is thrown from the Interpreter to the system which implements it.

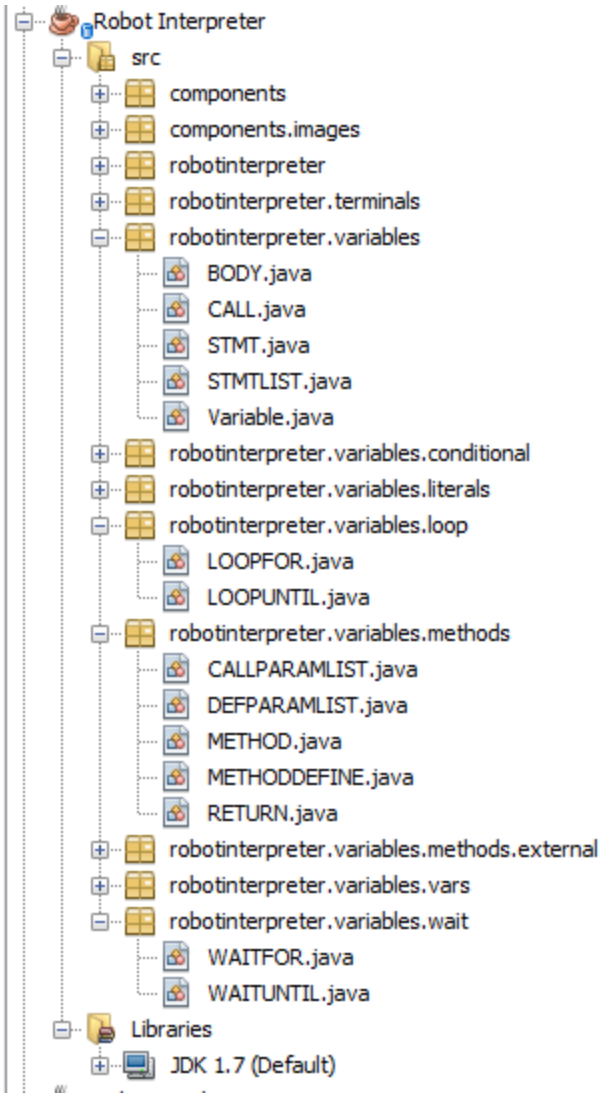


Figure 19. The package structure for the Interpreter

At the end of the parsing step, the algorithm has been transformed into a large linked-list containing program structure objects. This list is recursively validated by calling the validation function of each object in the list, starting at the head. The validation process checks for errors that are not syntactically incorrect but violate the defined rules of the language (such as, for example, attempting to loop for -2 times). If a violation is found, an error is thrown to the implementing system to handle. The validation step also handles setting up any required class members in the program structure objects that could not be set up during the parsing step. For example, a reference to the line of code immediately following the end of an If statement body is found and placed in the If statement object during the validation step.

Once validation has been completed, the algorithm is executed by recursively calling the execution functions of the objects in the list, again starting from the head. Because the object list was built when parsing the text algorithm line-by-line, top to bottom, the list is ordered in the exact same way as the text algorithm, and hence the object execution functions are guaranteed to run in the right order. The execution function for a program structure performs the activity that the structure itself would do in the algorithm – for example, the Loop For execution function calls the execution function of its associated Body object for the specified number of times. The basic structure of a program structure class containing the constructor, validation function and execution function can be seen in Figure 20.

Of particular interest are the special program structure classes that represent actions – for example, the Drive or Turn actions. Because the Interpreter is meant to be implemented inside of a system like the Simulator or the Robot, when executing one of these actions the Interpreter simply throws an event to its parent system. The implementing Simulator (or physical robot) must catch that event and perform the appropriate action (driving, turning, etc.) accordingly. The Interpreter is built in a general fashion – all actions that it can call are defined in their own classes, and any new action may be defined simply by creating a new class that inherits the ExtMethod class. The actions that the Interpreter implements, and the code of one particular action, may be seen in Figure 21. Note that not every external method we have defined in the interpreter is actually used in the GUI.

The general, customizable nature of the Interpreter means that it can be implemented by *any* program, so long as the proper event handlers are defined for it. This means that it can be used to write algorithms to control other things than robots driving in mazes – indeed, it could be repurposed to a completely different goal simply by changing the action methods. More broadly, any change in the Objective G language could easily be affected simply by rewriting the appropriate classes – a whole new program structure could be added just by adding in a new class.

```
21  public class LOOPUNTIL extends Variable
22  {
        private Interpreter interpreter;

        private CONDITIONLIST cl;
        private BODY codeBody;

28      /** public LOOPUNTIL(BODY b, Code c) ...9 lines */
        public LOOPUNTIL(Interpreter in, BODY b, Code c)
38      {...30 lines }

69      /**
         * @return      the code body for this statement
         */
        public BODY getCodeBody()
73      {...3 lines }

77      /** public void print() ...5 lines */
        public void print()
83      {...6 lines }

90      /** public void validate() ...6 lines */
        public void validate()
97      {
            interpreter.writeln("validate", "Validating LOOPUNTIL");
            //VALIDATING CONDITIONLIST
            cl.validate();
            //VALIDATING BODY
            codeBody.validate();
        }

105     /** public Object execute(Object[] args) ...13 lines */
        public Object execute(Object[] args)
119     {
            //EXECUTING CONDITIONLIST (first run)
            boolean go = (Boolean) cl.execute(null);

            while(!go)
            {
                //EXECUTING BODY
                codeBody.execute(null);

                //EXECUTING CONDITIONLIST
                go = (Boolean) cl.execute(null);
            }
```

Figure 20. A snippet of code showing the Loop Until class and its implemented functions.

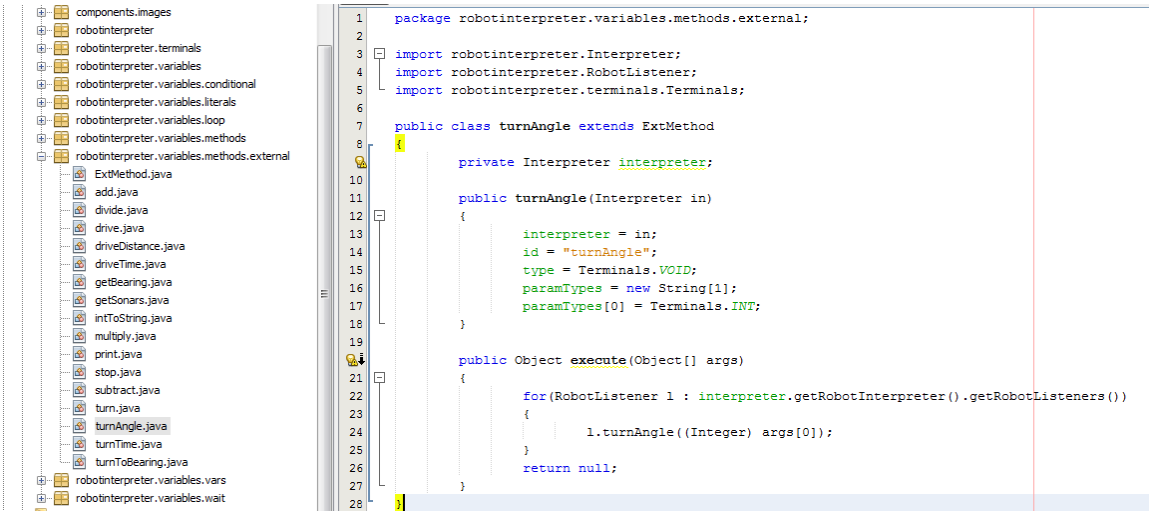


Figure 21. The external methods implemented in Genost, and the code of one of these external methods

3.4.4.4. Communication Between the Systems

One of the challenges in creating this system was creating a way for the GUI and simulator to communicate when both of these software are running in the browser. Not only were we required to find a way to "push" the algorithm from GUI to simulator, but we also must allow an arbitrary number of users to do this at the same time.

The classic HTTP model is a "pull" model – user agents request must explicitly information from the web to be downloaded and displayed in their browser. Remote systems are not able to, on their own initiative, push data to a browser, except in systems that are specifically designed to allow this. For this reason, we could not simply have the GUI software send the algorithm to the simulator software when both were running in the browser. The simulator had to, in some way, request it – a difficult thing to do when the simulator has no way of knowing that an algorithm even exists, much less is ready to be transferred.

In order to transfer the algorithm from the GUI to the simulator, the following procedure was defined. We programmed the GUI to, when sending an algorithm, first create a globally unique identifier (GUID), associate it with the algorithm code, and send both the algorithm and GUID to the Genost web server storage using a RESTful service. The GUI then opens up the simulator and, in doing so, includes the GUID as a URL parameter in the simulator URL. Once the simulator loads, it takes the GUID from its URL and calls another RESTful service, sending the GUID as input and receiving the associated algorithm in return. This solution both allows the algorithm to be transferred from GUI to server, and allows multiple users to run this at once, since the GUID is, by definition, globally unique. This solves the problem of communication between the two systems.

3.4.4.5. Technical Challenges

Many of the innovations discussed above were developed in response to technical challenges faced during the Genost development. For example, the method of sending the algorithm from GUI to simulator was built in response to the limitations of HTTP communications.

The GUI / simulator communication difficulty is a specific instance of the general challenge of having three different systems, all running on the web, communicating with each other. Not only are these three systems all running in separate environments, they are written in different languages. The challenge of having these three systems

synchronize and communicate would have been insurmountable without the use of service-oriented principles, and we believe that our intelligent use of SOA is a strength and innovation of our system.

The generality and customizability of our systems was also a challenge to implement, but was ultimately worth it due to our ability to easily reuse and extend the software. Once again this customizability is heavily powered by SOA principles (in the case of the GUI and simulator loading in XML files to configure themselves) but also by object-oriented principles (in the case of the high generality of the Interpreter).

A final difficulty worth mentioning is the complicated security issues that the GUI and simulator had to navigate. As these items run in a web browser, an inherently unsafe medium, our software had to contend with security provisions implemented by the browser and, in the case of the Java software, the language itself. We ultimately were required to purchase a Java security certificate to allow us to bypass the security protections in the browser.

3.5. THE ROBOT

In addition to allowing students to run their algorithm on the simulated robot in the Simulator, our system also features a physical robot prototype which students may send their algorithms to.

The robot may be seen below in Figure 22. It has all the same features that the simulated robot does – namely, the ability to drive and turn, sonar sensors on all four sides, and a compass sensor.

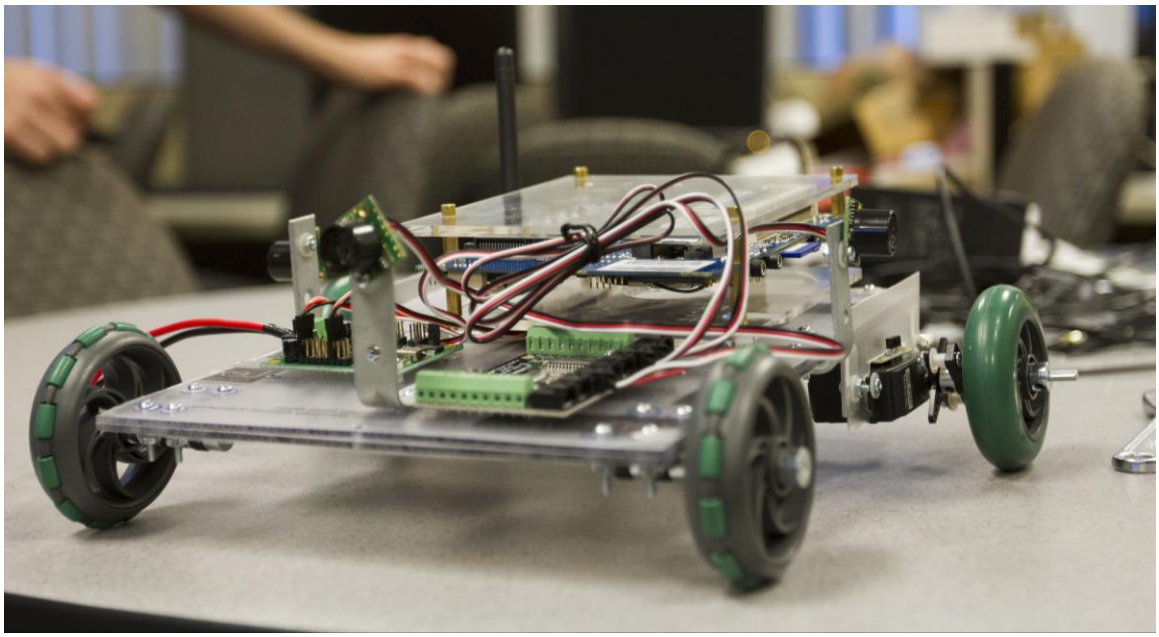


Figure 22. The Robot. Photo credit Jessica Hochreiter / ASU.

Currently the robot is powered by a full onboard computer in the form of an Intel Bay Trail SOC[31]. It connects to the internet wirelessly. When the robot is turned on, it immediately begins waiting for a program to be sent to it from the GUI. When the program is sent, the robot receives it, parses it and sets itself up to run the program. After receiving an "execute" signal from the GUI, the robot runs the program until it receives a Stop signal by the GUI. The communication described here is powered entirely by RESTful web services.

[31] http://ark.intel.com/products/codename/55844/Bay-Trail

The robot was built to be as low cost as possible, and costs less than $500. However, it should also be noted that the robot does not currently implement every feature that we planned for.

The robot may be used as the needs of the curriculum dictate – however, in our conception of the classroom use, the robot would be shared between students, who would use it to execute finalized algorithms which were developed using the virtual world. In this way a single robot could be shared between an entire classroom, and could be used as a "reward" for developing a successful algorithm.

3.5.1. Robot Design

The primary reasons for adding the robot to the Genost system was for both the fun that students have when working with robots (which therefore brings increased motivation, attention, and learning success), and for the execution model that a robot provides, which is more concrete than a virtual world. These benefits are discussed more in-depth in IG3.

In this section we will discuss our goals for the robot's design. We were not able to implement all of these goals in the final product – we will discuss the reason for this at the end of this section.

3.5.1.1. Goal 1: Focus on Computational Thinking (Not Engineering)

Our primary focus for Genost is teaching computational thinking (T4). As discussed in IG3, many systems that use robots tend to involve the engineering or building of the robots, to the detriment of the computer science education. For this reason, we want to set up our system such that computational thinking education is primary, and any other educational focus, such as engineering, is secondary.

We implemented this goal by creating a single prebuilt robot to give to the students instead of attempting to design a "kit" that the students will build. In educational use, students would use the prebuilt robot from Day 1, and would not build their own. The robot, then, serves solely as a tool to execute algorithms (and thereby teach computational thinking skills), instead of a focus in itself.

3.5.1.2. Goal 2: Inexpensive

We want our robot to be available to as many classrooms as possible – a goal discussed explicitly in IG3 and implied in T3 (adaptability). Just as our software should be adaptable, so should the system as a whole, and that means we should be able to make it work in schools of different types and means. One of the most important ways to do this is by making the robot inexpensive.

When designing and building our robot we attempted to make it as inexpensive to build as we could. Low cost parts were chosen, and we built our own robot chassis from basic materials instead of purchasing an expensive designer chassis. Ultimately, we were able

to build a fully functional robot that implements all the functions that the simulated robot does for less than $500USD. This is roughly one and a half times more expensive than a single Lego Mindstorms robot[32], and may serve an entire classroom.

Notably, our robot may actually be more expensive than it necessarily needs to be – we included a full SOC on the robot, which receives and executes the algorithm. A smaller embedded system which receives commands from a remote computer executing the algorithm would be less expensive, and is more in line with IG3. The reasons for not implementing this will be described further in Sections 3.5.1.5 and 3.5.2.

3.5.1.3. Goal 3: Robustness

Genost is an introductory educational system, meant to be used in a classroom with students, potentially young students. For this reason, the robot needs to be tough enough to survive "in the field". This is specifically discussed in IG3, and is somewhat in line with T3.

Robustness was an important, though not primary, goal for the robot we developed – we attempted to implement it by giving it a flexible body to absorb small shocks, as opposed to the more brittle body that the previous version of the robot had. This said, it is unclear to what degree our robot could actually survive a classroom environment, as we will discuss more in Section 3.5.2. In future robot development, robustness would likely be emphasized to a greater degree.

[32] At time of writing, the Lego Mindstorms EV3 Model 31313 retails for $350USD.

3.5.1.4. Goal 4: Same as Simulated Robot

We want our actual robot to respond exactly the same to the commands of an algorithm as our simulated robot. They should move in the same way, and use the same units in their sensor values, as far as possible. The physical robot and the simulated robot are supposed to be equivalent and interchangeable, so that a classroom using one will not be disadvantaged to a classroom using another.

To some degree this is also motivated by T1 – we do not want students writing an algorithm in the simulator to have to rewrite their code to make it work on the robot.

There is little to say in regards to our implementation of this. Currently the as robot has all the same capabilities as the simulated robot does, and will respond in the same way to an algorithm. However, it does not currently use the same units as the simulated robot does in its sensors.

3.5.1.5. Goal 5: Remote Code Execution

In IG3 we detailed the benefits of remote code execution - that is, the feature such that an algorithm executes on a desktop computer, and each command (drive, turn, etc.) is transmitted to the robot when necessary. These benefits include simpler (and therefore less expensive) robots, a benefit mentioned in Section 3.5.1.2; and more importantly, this allows students to debug their algorithm or step through it as it executes on the robot, assisting with teaching computational thinking, particularly CG1 and CG4.

There are considerable technical difficulties with this feature, primarily with ensuring that the commands reliably transmit to the robot in time. Additionally, because our robot uses sensors, the technology must support bidirectional transmitting. Possible ways of doing this include HTTP-based web services, or local wireless in the form of Bluetooth or others options. We initially attempted to implement this using the web services approach but found it far too slow to be useful. For this reason, we chose not to implement this goal in our current testing robot. Future development will attempt to implement this goal in an efficient manner.

3.5.2. Departures from Robot Design

The goals listed above are designed to produce an effective, fun, inexpensive and adaptable robot. Unfortunately, we were not able to implement all of these goals, and our robot currently serves as more of an effective testbed for executing Genost code than as a production robot that could be used in an actual classroom.

As described above, the robot is not terribly robust, does not run exactly the same way as the simulated robot, and does not implement remote code execution capabilities. It also is not consistent in its driving, turning, and sensor readings. The communication between the robot and the GUI is functional but not always reliable, and various ease-of-use features have not been implemented.

As mentioned above, we wish to implement a robot due to its benefits in motivation and in providing students with a concrete model of execution. Our robot currently proves that it can interface with the GUI, receive code, and execute it. The features that are not implemented are benefits ancillary to the main goal of the robot. Because these ancillary benefits are not present, we did not choose to use our robot in our tests or curriculum, as will be described in the section below. However, by adding the features we have described, we believe that an effective educational robot could be created and immediately integrated into the curriculum.

3.5.3. Robot Technology

In the previous subsection we described our intended robot design, as well as the ways that we departed from this. In this section we will describe the technology that went into building the robot. We will not discuss the proposed technology for features that were not actually included in our robot. This section will be divided into three subsections: the robot hardware, the robot software, and the technical challenges we faced in building the robot.

3.5.3.1. Robot Hardware

In this section we will describe the physical hardware that was used to build the robot.

The main computer on the robot is an embedded Intel architecture (IA) platform powered by an Intel Bay Trail SOC. The actual project board is a custom board created by Intel

and used for their ESDC 2014 competition[33]. Because we participated in this competition during the development of this thesis, we were able to utilize the project board. Aside from the Bay Trail processor, the board contains various input / output ports, GPIO pins and an SD reader for long-term storage. The board itself, and its standoff mounts and protective plastic covers, measures roughly 5.5" x 5" x 1.5".

An Anker laptop battery was used to power the Bay Trail board. The battery was chosen for its ability to self-regulate its output, adjusting the voltage between three separate settings and its current as needed. The advanced laptop battery also allowed us to charge the battery even while it is providing power to the computer without risk of damage to either the battery or the board.

The robot's drive system consisted of two Parallax high-speed bidirectional 360 degree motors, used to power the rear two wheels of the robot. The rear wheels themselves were made of rubber, chosen for grip. The front two wheels were special Vex omni-wheels[34], which contain small rollers along the wheel circumference which turn perpendicular to the main wheel axle. This allows the omni-wheels to "strafe", or in this situation, allowed the two rear wheels to turn the robot without resistance from the front wheels. A Parallax "Propeller" servo controller was used to control the motors. A basic 7.2V hobby battery was used to power the motors.

[33] https://software.intel.com/en-us/forums/2014-intel-cup-embedded-system-design-contest
[34] http://www.vexrobotics.com/omni-wheels.html

The robot's sensor system consisted of two types of sensors: sonar sensors and a compass sensor. Three Phidgets sonar sensors were utilized, one pointing to the robot's front, and one on each side. A Phidgets sensor interface board was used to interface with and control the sonar sensors. This sensor board connects to the main board via USB. The compass sensor was also made by Phidgets, and connects directly to the main board via USB without the need for an intermediary interface board. All of these sensors received power through the USB connection and did not require their own external battery connection.

The chassis of the robot consists of two polycarbonate sheets measuring 10" x 8" x 0.093". The two sheets were stacked and bolted together, and the various robot parts were attached. The polycarbonate sheet was chosen as a basic material that was strong, flexible and easy to work with, while also quite inexpensive.

The above hardware was identified and chosen over a semester-long prototyping process, and was preceded by an earlier prototype build which was built mostly to test out the motors. These parts were found to be the least expensive items that were still usable, reliable and robust enough for our design.

3.5.3.2. Robot Software

The operating system that is running on the robot's Bay Trail mainboard is Ubuntu, chosen to assist us in easy development while still providing a relatively small memory footprint. The actual software that we developed to run the robot is divided up into

different “modules”, all of which are written in Java. The modules are: the bootstrapper module, the networking module, the robot core, and the hardware drivers. Each of these modules are invoked at certain points in the operating cycle of the robot. We will discuss the technology involved with each of these modules in the order that they are invoked. The bootstrapper module is responsible for configuring the robot and loading up all other modules when the robot first boots up. It is automatically run at computer boot time and loads up the networking module and robot core for further processing. As the name indicates, its main purpose is bootstrapping the robot.

The networking module is started by the bootstrapper and, once it is started, connects to the internet wirelessly. It then begins polling a RESTful service defined on the management website to see if it an algorithm is ready for the robot to execute. Once the algorithm is ready, the networking module downloads it and sends it to the robot core for processing. It then begins polling for the command to execute. Once it receives this command the module instructs the robot core to begin executing, and starts polling once more for the “stop” command. Once it receives this stop command, the module stops the robot, and starts the polling process over, polling for a new algorithm.

The robot core is made up of the interpreter, and the drivers for the motors, sonar sensors and compass sensor. The interpreter that is used here is exact same one that is used in the simulator (described in Section 3.4.4.3) – just as the interpreter’s event handlers are implemented in the simulator to allow the interpreter to control the simulated robot, so the physical robot’s code implements event handlers to control the real robot. When the

networking module sends the algorithm to the robot core and commands it to execute, the interpreter is started up with the algorithm as input. The execution proceeds just as it does with the simulator.

In order to send commands to the motors and retrieve data from the sensors, we required drivers to interface with the hardware. The Phidgets sonar sensors and compass sensor have prepackaged drivers that allow easy communication, but the motors did not. Communication with the "Propeller" servo controller requires the use of a special language called "Spin" (Scanlan, 2007) – for this reason we wrote a translator driver to translate Java commands into Spin commands.

3.5.3.3. Technical Challenges

Many challenges were encountered in developing the robot, a handful of which are worthy of mention. The main challenge that we will discuss has to do with the way in which commands are sent to the robot. As described in Section 3.5.1.5, a major goal of our system was to run the interpreter in the browser or on a local computer, instead of on the robot itself; this would require us to relay robot action commands like "drive" or "turn" to the robot in some manner.

In order to implement this we initially pursued the Robot as a Service (RaaS) SOA model, a paradigm described by Chen et al in (Chen Y. Z.-A., 2010) and (Chen Y. H., 2013). The RaaS methodology allows a robot to be controlled through services hosted on the robot itself – for example, the robot's motors could be controlled via a motor service,

and the sonars could be accessed through a sonar service. This is an admirable paradigm and would fit our needs very well, allowing us to simply send HTTP packets to the robot with our commands as they were issued by the interpreter. Using RaaS would also allow us to avoid requiring additional hardware (like IR sensors or Bluetooth) to communicate with the robot.

Past experiments have verified the validity of the RaaS paradigm. In 2010 Chen, Du and García-Acosta implemented some aspects of the RaaS paradigm in Microsoft VPL, utilizing its DSS service framework (Chen Y. Z.-A., 2010); in 2013 Chen and Hu created a full prototype of the RaaS paradigm, in which a robot hosted multiple services (Chen Y. H., 2013). Each service represented an instruction to move or turn the robot in a certain direction; as a test, a website was created containing buttons that called the services when pushed. These experiments prove the technical possibility of the RaaS paradigm.

We attempted to use the RaaS paradigm to control the Genost robot. Setting up the robot services themselves was a relatively simple task, but connecting to them was a difficult challenge. The robot, because it utilizes dynamically assigned IPs when connected to the internet, does not have a URL or other static identifier. If the IP address of the robot is known, then we can send messages to it; the problem is knowing the IP address, which is not known until the robot is turned on in the classroom. A possible solution is to have the teacher using the robot look up and enter the robot's current IP address on the management website, from which it could be utilized by the robot services; but we cannot

rely on teachers being technically savvy enough to accomplish this step, and anyways it is quite pedantic. It would be far better if this could be accomplished automatically.

A step was made towards solving this problem by having the networking module look up its own IP address and send it to the Genost server via a RESTful service. Once sent, the IP could then be utilized by the interpreter to connect to the robot. However, if the robot was inside of a NAT, then this method would only work so long as the interpreter itself was also running inside of that NAT – which, if the interpreter is running in the browser (i.e. off a remote Genost server), will never be the case. This difficulty was never entirely resolved, due to the result of the challenge described next.

It was found that, after setting up the SOA robot such that it could run within a NAT, communication between the interpreter and the robot was excruciatingly slow. HTTP is, by its design, a "best effort" policy, so it is not surprising that the packets had a round trip time nearly on the order of seconds. This was far too slow for robot operations. Consider the scenario in which the robot is told to drive forward until a sonar sensor sees a wall five inches away – by the time the interpreter has queried the sonar sensor, and the sonar sensor has responded with the sensor value, the robot had already had at least a second or two to continue driving, and may have already slammed into the wall. Due to the slow response time, and the difficulty in connecting to the robot from the Genost server, the RaaS paradigm was mostly abandoned, and the current method of running the interpreter on the robot itself was adopted.

Abandoning RaaS does not mean abandoning the use of HTTP RESTful services, as can be seen with our method of polling the RESTful services to retrieve the algorithm and execute command from the server. While this does work, it is far from an ideal solution since it involves polling, a non-ideal solution in any circumstance. We wish to maintain our use of HTTP due to the aforementioned benefits of universality and generality that it brings, but since we cannot "push" to the robot polling was judged an acceptable compromise. All this said, the RaaS paradigm is a valuable one and we believe with additional time and funding the identified problems could be overcome.

Another challenge that was not ever fully overcome involves creating a way to connect the robot to a wireless network. If the robot is to be run under arbitrary wireless networks in a classroom, the teacher would need a way to enter the network credentials into the robot. Our goal was to create a program that would run when the robot was connected to a desktop computer via USB; this program would allow the teacher to enter the wireless credentials, which would be saved on the robot. We were never able to create this program due to time constraints and technical difficulties. Currently the wireless credentials for ASU's network are hardcoded onto the robot, so the robot will function anywhere on ASU campus. Configuring the robot to work outside of campus on a different network would require connecting the Bay Trail board to a monitor and mouse, and changing the hardcoded credentials, a method which is not at all acceptable or sustainable. This is a challenge that we hope to overcome in the future.

A final challenge worth mentioning is the simple difficulty in designing an inexpensive robot that can drive and turn accurately. Due to our goal of low cost, certain sacrifices had to be made in quality, and as a result our final robot does not drive perfectly straight, nor does it turn perfectly accurately. Early in the project we planned to implement external sensors to help track the rotation of the wheels and use this information as feedback to adjust the robot's movement on the fly. While these sensors were bought and installed, we did not have the time to implement them into the code. We believe this challenge can be overcome with, as usual, more time for development.

3.6. THE MANAGEMENT WEBSITE

The Management Website or "Teacher Website" is a tool to manage the curricula, mazes, and student access for Genost. Genost is a highly customizable tool – the Management Website partially powers this customizability by hosting the content which Genost uses to configure itself.

There are a handful of different data objects which are hosted on the Management Website. These are: Classes, Students, Curricula, Lessons, Mazes and Toolboxes.

- **Maze:**

  A **maze** hosted on the Management Website is an XML file that, when loaded into the Simulator, defines the position of all the obstacles and coins in the maze, what graphics they will use, the initial position of the robot, and the goal of the maze.

- **Toolbox:**

  A **toolbox** hosted on the Management Website is an XML file that, when loaded into the GUI, defines what code blocks are available to the user and what those code blocks do.

- **Lesson:**

  A **Lesson** is a structure hosted on the Management Website that references a particular Maze and a particular Toolbox. In other words, a Lesson links together a maze and the code blocks that the student will have access to in order to solve the maze. A **Lesson** is loaded into the GUI when the GUI is first started, and when students change the Lesson manually.

- **Curriculum:**

  In context of the management website, a **Curriculum** is a linked series of Lessons, hosted as a data object on the Management website. Teachers or other managing users can define a Curriculum and set up which lessons they wish to be included in it, as well as specify the order that these lessons should proceed in. When a user on the GUI loads the “next” or “previous” lesson, it is the

Curriculum that is referenced to determine which lesson is in fact “next” or “previous”.

- **Students:**

  All users who utilize Genost will have a **Student** account on the Management website, though only teachers and other managerial users will have access to the management content. Creating user accounts for each student allows us to track student activity – when interacting with the GUI, all students must first log into their Student account, and from then on their activity can be recorded, tracked and analyzed.

- **Class:**

  A **Class** is a logical grouping of **Student** objects. A teacher may set up their **Class** to contain all their students, and then associate one or more **Curricula** with the **Class**. It is through the **Class** object that students access the **Curricula**, and therefore how they load in their lessons. The **Class** object also is where the various metrics, such as student completeness, are centered and reported.

Figure 23 shows a screenshot of the management website’s Class report, which shows a list of lessons for the class, a list of students for the class, and a report showing which students have completed which mazes. In this way, the Management Website may serve

as both a customizing tool, allowing teachers to set up mazes and toolboxes, an organization tool, allowing teachers to set up classes and curricula, and as a reporting tool or gradebook, allowing teachers to see how their students are doing.

3.6.1. Management Website Design

The primary design goals of the management website are to be easy to use (T1) and to allow users to quickly and simply set up powerful customizations (T3). Furthermore, the feedback functionality supports T4 in a broader sense, as it will allow teachers and researchers to get feedback on their educational techniques, improve these techniques, and ultimately teach computational thinking education more effectively.

The website itself is built using the Drupal[35] content management system, and uses RESTful web services to communicate with the GUI, simulator and robot. Because this website is fairly simple, no more needs to be said about it.

3.6.2. Management Website Technology

As mentioned above, the management website was built from the Drupal CMS, which is written in PHP. Drupal is an extraordinarily flexible tool and this is why it was chosen for our management website. It also has a very active community which has developed many "modules" to extend the base CMS. Some of these modules, like Views[36], have allowed

[35] https://www.drupal.org/

[36] https://www.drupal.org/project/views

us to easily generate useful displays of student data, such as that seen in Figure 23. Other modules, like the Taxonomy module[37], allow us to easily define, create and categorize large amounts of data; the Taxonomy module was used to store all the data collected on the student interaction with the software.

[37] https://www.drupal.org/documentation/modules/taxonomy

Lessons for FSE100 Fall 2014 Extracurricular Activity

1-1-1_driving_example_1
1-1-2_driving_example_2
1-2-1_procedural_example_1
1-2-2_procedural_example_2
1-2-3_procedural_takehome_1

1 2 3 4 5 6 7 8 9 … next › last »

Students in FSE100 Fall 2014 Extracurricular Activity

abra
bellsprout
bulbasaur
caterpie
charmander

1 2 3 4 5 6 next › last »

Student	1-1-1_driving_example_1	1-1-2_driving_example_2	1-2-1_procedural_example_1	1-2-2_procedural_example_2	1-2-3_procedural_takehome_1	1-2-4_procedural_takehome_2
abra	Completed	Completed	Completed	Completed	Completed	Completed
bellsprout	Completed	Completed	Completed	Completed	Completed	Completed
bulbasaur	Completed	Completed	Completed	Completed	Completed	Completed
caterpie	Completed	Completed	Completed	Completed	Completed	Completed
charmander	Completed	Completed	Completed	Completed	Completed	Completed
cubone	Completed	Completed	Completed	Completed	Completed	Completed
doduo	Completed	Completed	Completed	Completed	Completed	Completed
geodude	Completed	Completed	Completed	Completed	Completed	Completed
machop	Completed	Completed	Completed	Completed	Completed	Completed
mankey	Completed	Completed	Completed	Completed	Completed	Completed
meowth	Completed	Completed	Completed	Completed	Completed	Completed
nidoran	Completed	Completed	Completed	Completed	Completed	Completed
paras	Completed	Completed	Completed	Completed	Completed	Completed
pidgey	Completed	Completed	Completed	Completed	Completed	Completed
poliwag	Completed	Not yet completed	Completed	Not yet completed	Not yet completed	Not yet completed
sandshrew	Completed	Completed	Completed	Completed	Completed	Completed
seel	Completed	Completed	Completed	Completed	Completed	Completed
spearow	Completed	Completed	Completed	Completed	Completed	Completed
squirtle	Completed	Completed	Completed	Completed	Completed	Completed
staryu	Completed	Completed	Completed	Completed	Completed	Completed
tangela	Completed	Completed	Completed	Completed	Completed	Completed
tentacool	Completed	Completed	Completed	Completed	Completed	Completed
venonat	Completed	Completed	Completed	Completed	Completed	Completed
voltorb	Completed	Completed	Completed	Completed	Completed	Completed
vulpix	Completed	Completed	Completed	Completed	Completed	Completed
weedle	Completed	Completed	Completed	Completed	Completed	Completed

Figure 23. Screenshot of the Management Website

One of Drupal's very useful functions is the "menu hook" functionality[38], which allows one to define an arbitrary URL and associate it with a custom function. The function may

[38] https://api.drupal.org/api/drupal/modules%21system%21system.api.php/function/hook_menu/7

be written in PHP and can therefore perform arbitrary activity, and can return any data to the system that requests the URL. In this way we could use easily define as many RESTful services as we needed and program them to do whatever we needed them to do.

3.6.2.1. Technical Challenges

The "menu hook" solution for defining the RESTful services was chosen only after trying many other solutions. One solution which is representative of the ones we tried is the Services module[39], which allows the definition of RESTful services through the website front end. While this module is useful for creating certain types of services (mostly those that allow the creation, updating, and selection of Drupal content items or "nodes") it does not allow us to define the more arbitrary, functional services that we needed.

The "menu hook" solution was one of the last solutions tried due to its lack of support for creating the RESTful services – all relevant items had to be created by scratch when using the "menu hook". However, after trying it we found that this did not require nearly as much effort as expected, and so this solution was the one we settled on.

3.7. THE CURRICULUM

As has been mentioned many times, the Genost system consists of two major parts: the software, and the curriculum. The previous sections have all dealt with various aspects of the software (or the hardware, in regards to the robot.) This section will focus entirely on

[39] https://www.drupal.org/project/services

the other half of the Genost system, the curriculum. The software is a tool which has been designed to be maximally effective in teaching computational thinking skills – the curriculum, however, is where these skills are actually taught.

As part of this thesis, we have designed a curriculum for both theoretical reasons (we wish to design not just the tool but also to explore its use) and practical reasons (we need a curriculum with which to test our system.) Another motivating factor in developing the curriculum is the benefits that pairing a curriculum with software bring, as discussed in IG4.

3.7.1. Curriculum Overview

The subject of our Genost curriculum is computational thinking, and we teach in two major ways: first, by introducing the fundamental programming structures (loops, if statements, variables, etc.) as general concepts (this covers CG1 and CG2) and second, by teaching the skills involved with analyzing a problem, breaking it down, and turning it into an algorithm.

The curriculum is divided logically into four sections. These sections are each dedicated to introducing a different fundamental programming structure; the theme of algorithm design is dispersed throughout.

The four sections are:

1. **Actions:** introduces the basic robot actions and procedural programming
2. **Loops:** introduces the two Loops available in Genost, as well as basic algorithm design
3. **Waits:** introduces the Wait statements, as well as generalized algorithm design
4. **Ifs:** introduces the If statements and their various uses, and ties the concepts together.

It should be noted that this curriculum does not include Variables or Function, even though the Objective G language does contain this functionality. We did not include these items because our curriculum, as it stands, is already very large, and we would not be able to effectively teach these large concepts with the little time we had to test. Future iterations of the curriculum will include sections for these concepts.

Each of the four sections described above is divided up into multiple subsections. Each is focused on a subtopic of the overall section topic – for example, a subsection of the Loop section might focus on Loop For specifically. The subsections are themselves divided up into individual lessons – there are almost always four lessons per subsection, though occasionally there are more. Each lesson comprises a maze and a worksheet. The maze is designed to require use of the subsection topic to (easily) solve, and the worksheet is used to help the student solve the maze.

The worksheets usually contain questions that the students are asked to answer. The questions ask the student to think about the topics at hand, and usually involve breaking down a problem into subproblems, selecting solutions for the subproblems, and putting them back together to create a final algorithm. Example worksheets may be seen in Appendix C.

The four lessons in a subsection follow a "fading" progression, an approach described by Atkinson that involves first teaching a topic through a complete example, then slowly working the student through examples of increasing incompleteness, until finally the students are asked to solve a full problem. Research on this fading technique has found it to be an effective method of knowledge transfer (Renkl, 2002). We "fade" through the use of four sequential lessons per topic, each of which are described below.

The four lesson model is as follows:

1. **Initial Lecture**: the first lesson is usually provided in the form of a lecture, which introduces the topic of the subsection. The general idea behind the topic is covered, as well as the need for the structure, and its uses. Each lecture also includes a worked example code exercise in which the instructor walks students through the development of an algorithm. The algorithm walkthrough focuses on the structure being taught; the instructor explains the development every step of the way, and the students follow along on their own computer. The focus of the initial lecture is *introducing* students to the concept. A review recently conducted

by Atkinson et al has concluded that this worked example method of teaching does provide real, flexible knowledge transfer (Atkinson, 2000); we are therefore confident that this method of introducing a topic will provide a solid foundation for students to work through the subsequent lessons.

2. **Guided Practice:** the second lesson is provided through a handout worksheet. The worksheet serves as another worked example, as it walks students through the development of a second algorithm, which again features the programming structure being taught. The worksheet will involve basic questions that will walk the student through breaking down the problem and developing the algorithm. The focus of the guided practice is to build *confidence* in the new concept.

3. **Simple Exercise:** in the third lesson the students are given a simple maze to solve that requires use of the topic structure. Usually, the simple exercise will involve some questions that will help the student break down the problem and develop the algorithm. The goal of the simple exercise is to *test* the student's *familiarly* of the concept.

4. **Challenging Exercise:** in the final lesson the student is given a more challenging maze to solve, and is given very little or no guidance in solving it. The student can use the skills he learned from the previous worksheets and his own creativity to solve the maze. The goal of the challenging exercise is to *challenge* the student's *mastery* of the concept.

For most lessons, the subsection topic is taught through direct instruction. In some lessons we intentionally have the student to make errors, in order to illustrate a principle or show a common mistake, as well as how to recover from these errors. In other lessons we have the student solve a maze in an inefficient or difficult way, in order to illustrate the need for a new structure. Often a certain maze will be used multiple times throughout the curriculum – the first time it is used it will be solved inefficiently using early techniques, and later on it will be solved in a more efficient way using better techniques.

It should be noted that this curriculum is designed to be used with the GUI and the virtual worlds in the simulator, and does not utilize the robot. We discussed our reasons for not using the robot in this curriculum in Section 3.5.2.

3.7.2. Curriculum Topics

This section will cover the different topics covered in the curriculum. We will cover each of the four sections and briefly describe the subsections within them.

3.7.2.1. Section 1: Actions

Section 1 is intended to introduce the basic operation of the Genost software, the definition of the Genost language, the Action blocks, Sockets and Parameters. A basic introduction to breaking down a problem and building an algorithm is provided.

Section 1 has two subsections:

- **Section 1.1:** this subsection focuses on introducing the Drive Distance action, Sockets, and the Integer data block. It only has two lessons, both of which are lectures.

- **Section 1.2:** this subsection focuses on introducing the concept of procedural code flow and the Turn Degrees action. Section 1.2 contains the standard four lessons, plus an additional challenging lecture to make five lessons total.

3.7.2.2. Section 2: Loops

Section 2 is focused on two major subjects: introducing the concept of loops, and fully introducing problem breakdown and algorithm design. In this section, the algorithm design steps are taught very mechanically – students are told to use the visual patterns in the maze to determine how an algorithm should be broken down, and the visual patterns are introduced as directly indicative of what processes and code blocks should be used to solve them.

Section 2 has five subsections:

- **Section 2.1:** this subsection introduces loops, and teaches the Loop For block. It also introduces the algorithm design process more fully than was covered in Section 1. It has the standard four lessons.

- **Section 2.2:** this subsection introduces the use of sequential Loop For blocks, and continues teaching the algorithm design processes. It also introduces the concept of setting up the system to run a loop – that is, positioning the robot in a certain way solely for the convenience of the loop. It has the standard four lessons.

- **Section 2.3:** this subsection introduces the use of nesting with Loop Fors. "Looping loops" is introduced and integrated into the algorithm design process. Again, this has the standard four lessons.

- **Section 2.4:** this subsection introduces the use of the Loop Until, which necessitates the introduction of Conditions. Using Loop Untils allows us to write general code that can solve multiple mazes, and techniques to do this are introduced. The four standard lessons are used.

- **Section 2.5:** the final subsection combines Loop Until and Loop For and uses both sequential and nested loops. A technique is introduced that allows the robot to drive forward forward until a condition is met. The four standard lessons are used.

3.7.2.3. Section 3: Wait Statements

Section 3 introduces the Wait statements, which enhances the generality of code by allowing the robot to continuously perform an action until a certain condition is met. The

Drive and Turn blocks are introduced as separate concepts from Drive Distance and Turn Degrees. The difference between the algorithm's execution and the robot's movement is explored. This section has three subsections.

- **Section 3.1:** the first section in Section 3 introduces the Wait Until block, the Drive block, and the difference between code execution and robot movement. Algorithm design is explored in greater detail here, with an eye towards generality. This section has the standard four lessons.

- **Section 3.2:** this section combines Wait Untils and Loops, and a more general form of algorithm analysis is introduced. This newer form of algorithm design moves away from the mechanical form used in Section 2 to a more creative form. This section has the standard four lessons.

- **Section 3.3:** this section teaches Wait For, and continues teaching the difference between algorithm execution and robot movement. Wait Fors and Wait Untils are combined to allow greater and safer generality. This section has the standard four lessons.

3.7.2.4. Section 4: If Statements

Section 4, the final section, introduces If statements and the concept of the algorithm executing differently depending on the circumstances. Ifs, Else Ifs and Elses are all introduced, and at the very end of Section 4, Logical AND and OR are introduced.

Algorithm design is taught in its fullest and most creative method. This section has four subsections.

- **Section 4.1:** this section introduces the If block, which is introduced in the context of executing "additional actions" in certain circumstances. Algorithm design is once again tweaked to accommodate this new and more general possibility. The standard four lessons are used here.

- **Section 4.2:** this section introduces the Else block, and an If paired with an Else is described as allowing the program to make a decision or choice with two possibilities. This allows for the creation of very general algorithms. The standard four lessons are used here.

- **Section 4.3:** this section introduces the Else If block, and this block is described as allowing the program to make choices with 3+ outcomes. The method of reducing a single choice between *n* options to a series of choices between two options is described. The standard four lessons are used here.

- **Section 4.4:** the AND / OR logical blocks are introduced along with complex logical conditions. Students are shown how to convert complex If-Else If-Else chains into smaller chains using these logical blocks. Four lessons are used here.

This concludes the review of the Genost curriculum topics. It is important to note that this curriculum is not the *only* curriculum that can be used with Genost, though we believe that it is a fundamental or "core" curriculum that other curricula can be based on.

3.7.3. Curriculum Design

We have described the layout of our curriculum above, and to some degree justified our general design. In this section we will describe the design goals, justification and implementation of some of the major elements of the curriculum, including both aspects of the content and aspects of its presentation.

3.7.3.1. Goal 1: Teach Fundamental Programming Structures

In Section 3.7.1 we described the two major focuses of our curriculum. The first of these focuses, teaching the fundamental programming structures, will be described in this subsection. The other major focus will be discussed in the next subsection.

"Fundamental programming structures" refers to the structures in programming that are common to most or all programming languages and paradigms. Variables, Functions, Loops, and Ifs are all examples of these fundamental structures. They are fundamental because, ultimately, all algorithms are made up of various combinations of these concepts.

In order to program in any language, a student must have some understanding of the general concepts behind the code they are writing. As we have shown in Sections 2.3.1

and 2.3.2, however, these concepts are not taught *generally* in US introductory computer science education – they are instead taught *specifically,* in the context of a particular programming language.

We believe that a student with a grasp of these fundamental concepts as general ideas will easily be able to read algorithms that implement them regardless of the syntax, fulfilling CG1, and will also be able to effectively break down problems and build algorithmic solutions, fulfilling CG3. It is for these reasons we wish to teach these concepts directly as general concepts, instead of indirectly within the context of a specific formal language. Furthermore, teaching these items as general ideas transcending languages instead of specific items already encoded in a formal language is directly in line with CG2, abstraction.

Our curriculum focuses on teaching four of the major fundamental programming structures: Actions (the concept of a structure that results in discernible output), Loops (the concept of a specific section of code repeating), Waits (the concept of pausing execution of an algorithm for a certain amount of real-world time) and Ifs (the concept of choosing between alternatives, i.e. branching). Each major section focuses on one of these, and they are explicitly explained as general concepts to the students.

The usefulness and "purpose" of these concepts are taught in a few different ways. As mentioned above, in each section we directly explain to students what the section topic is good for. We also introduce the need for these structures, and their common uses. In

some cases we introduce these items in an indirect manner. For example, some lessons ask students to solve a maze that would best be solved using one concept, but we do not allow the students to use this concept in the solution. In these cases the student solution is inevitably messy, inefficient and generally poor, and when comparing this poor solution to a solution that utilizes the proper concept, students can easily see why the concept is useful, as well as how the concept can be used.

Our approach to teaching these concepts closely mirrors the ITEST group's description of a Use – Modify – Create learning cycle. In the ITEST cycle, the first step, Use, asks the student to simply use the concept, to see it in action. The second step, Modify, asks the student to modify an existing algorithm using the concept. Finally, the third step, Create, asks the student to create an algorithm from scratch with this concept (Allen).

Our four-lesson cycle is very similar to this. The first lesson, the lecture, asks students to Use the concept. The second lesson, a guided worksheet, has students both Use the concept and Modify its use in some ways. The third lesson, a simple challenge, asks students to either Modify an existing algorithm or Create a new one, depending on the lesson. The fourth lesson always has the students Create a new algorithm using the concept.

In this way, our curriculum attempts to teach the fundamental programming structures.

3.7.3.2. Goal 2: Teach Problem Breakdown and Algorithm Design

The second of our two major focuses in the curriculum is the ability for students to break down a problem and create an algorithm that solves it. We have argued above for the importance of this skill for computational thinking. One of the four components of computational thinking, CG3, is directly dedicated to it. As always, CG2, abstraction, is also an important part of this skill.

We teach this ability explicitly all throughout the four sections of our curriculum. A common mantra that students are taught is the "four steps of algorithm analysis", which are:

1. Fully understanding a problem
2. Breaking the problem down into subproblems
3. Solving the subproblems
4. Combining the subproblem solutions to form an algorithm

In most lectures (the first lesson of the four-lesson model) students are explicitly walked through these four steps to solve a problem. In the guided worksheet (the second lesson of the four-lesson model) students are usually directed to proceed through each of these steps by solving worksheet problems corresponding to each step. These worksheet problems are also sometimes present in the third lesson.

Students are usually asked to work through the first and second steps on their worksheets by literally drawing on a picture of the maze to indicate their division of the maze into subproblems. Students are encouraged to look for similar shapes in the maze to help them break the maze down. See Figure 10 for an example of a maze with drawings to highlight how it can be broken down into subproblems: the large maze is broken down into four separate identical crosses. These function as subproblems whose solution can be looped to create a full algorithm to solve the maze.

Students work through the third and fourth steps by writing pseudocode in special text areas on their worksheets. Students are asked to write out code for the individual subproblems at first, and then to write pseudocode that combines the individual solutions, along with "glue code" that is inserted in between the subproblem solutions to allow them to work together. Most guided worksheets end with having the student write out the entire algorithm in pseudocode.

Early worksheets in Section 2 tend to teach the algorithm design skill in a very mechanical, procedural way: students are told to rely on the visual appearance of the maze to break down the algorithm, and to find guidance in repeating physical shapes (like the crosses in Figure 10). In Section 2 the combining step, Step 4 in the list above, is always a loop, and students are told to select the number of iterations by looking at the number of repeating physical shapes there are. In this way the skill is first introduced mechanically.

In Section 3 and Section 4, as students are taught to solve multiple mazes with the same algorithm, and are introduced to heuristics to help them in doing so. Students are encouraged to walk through an algorithm in their heads, to abstract away certain elements while paying attention to others. By Section 4, students are given little mechanical guidance, and the guided algorithm analysis is mostly performed with heuristics, though students are still asked to keep track of their algorithm solutions and subsolutions by writing down their pseudocode.

In this way, our curriculum attempts to teach the ability to break down problems and design algorithms.

3.7.3.3. Goal 3: Teach Habits of Good Program Design

Once again, this goal corresponds directly to an element of computational thinking, in this case CG4 (algorithm quality). As we have argued before, we do not only want to teach students to design algorithms, we want to teach them to design *good* algorithms. In our curriculum we tell the student that high quality algorithms are not just better technologically but also better ethically, and that high should be considered just as important as functionality.

The importance of high quality in programming is discussed heavily and reinforced throughout the curriculum. Just like with problem breakdown / algorithm design, the curriculum contains a mantra about programming quality that is repeated many times

throughout the lessons, and is specifically tested on the pretest and posttest. The mantra is describes the “three goals of programming”, which are:

1. Reduce the size and complexity of your algorithm.
2. Create algorithms that are more general and can be used in multiple mazes.
3. Have fun programming!

Goals 1 and 2 focus on quality – Goal 3 is a reminder that high quality programming is fun.

Algorithm size reduction is consistently taught and practiced. Students are reminded whenever they write pseudocode that “less is more”. Many worksheets include a specific number of blank lines for pseudocode that match up with the most minimal (and therefore least complex) algorithm possible. The practice of cutting out unnecessary code is consistently encouraged.

Beginning with late Section 2, and continuing with earnest in Section 3 and 4, students are not only encouraged but also required to write algorithms that are general and reusable. Most lessons in Sections 3 and 4 involve creating a single algorithm for multiple mazes, which requires the introduction of reusability and generality into the algorithm design process.

In this way, our curriculum attempts to teach the production of high quality algorithms.

3.7.3.4. Goal 4: Design Curriculum to Scaffold Students

In IG4, we noted the applicability of Ausubel's "anchoring ideas" or "advance organizers" to programming education (Ausubel, 1968). We argued in that section that in order for students to be able to explore programming, they must first be instructed in the fundamentals (this is why we reject a heavy play emphasis for our introductory curriculum). A similar argument may be made that in order to understand advanced applications of the fundamentals, students must first understand basic applications.

We have put considerable effort into designing the curriculum such that students are never asked to utilize a skill or apply knowledge that has not first been deeply taught. The four major sections are arranged in such a way that each section can be taught using only the skills used up to that point. This applies not only to the major fundamental programming structures (i.e. we teach Loops without utilizing Ifs) but also to the problem breakdown techniques, algorithm design techniques, minimizing techniques, and all other items that have been discussed so far. Each subsection contains (almost always) four lessons presented in a specific order to deeply teach the concept. Within the four-lesson model, a concept is always first introduced, explained and specifically applied before the student is asked to utilize it on their own.

Even combinations of fundamental structures and techniques are held off from until both structures involved in the combination have been taught individually. For example, in Section 3, we do not utilize Loops with Waits until Section 3.2, after we have introduced the concept of Waits on its own in Section 3.1.

In addition to the curriculum design, we also scaffold through clever use of the customizability of the Genost GUI (described in Section 3.4). In that section we noted that each lesson has its own "toolbox", that is, has its own XML file defining which blocks are available for use. Instead of making all blocks be available in every lesson, we limit the blocks to only those that the student has learned so far. This helps us scaffold the student – the student is not overwhelmed by a large number of blocks before he has learned them all, and the GUI "learns" along with him as it makes more blocks available.

In this way, our curriculum attempts to scaffold the students as they learn.

3.7.3.5. Goal 5: Strike Balance between Instruction and Creativity

In T5, we discuss the need for the curriculum to be focused on explicitly teaching computational thinking skills first and foremost, as opposed to other newer systems which focus more on play, storytelling or competition. The reasons for this are discussed in the previous subsection (Section 3.7.3.4), in IG4 (Section 2.4.9) and throughout Section 2.4. However, we also do not want to exclude creative action entirely. A balance must be struck, and this is what we have attempted to do.

As described above, we initially teach algorithm analysis mechanically, but later move on to allowing students to use their own creative heuristics once we are confident they have been properly scaffolded. Similarly, the four-lesson model features explicit instruction in

the first two lessons, and allows for creative problem solving in the remaining two. These are some of the ways that we attempt to strike the balance between instruction and creativity.

Genost's customizability and maze building functions may also be leveraged to allow some student creativity, once the explicit concepts have been taught. For example, a possible future section of the curriculum may involve allowing students to make their own mazes that are centered on a certain concept, or are intended to teach a certain fundamental structure. After building the maze, students may then share (Chamillard, 2000) them with their friends. This would allow for the social creativity that Resnick and others often extol, while still providing a direction for the student to work in. Although this feature is not currently part of our curriculum, we have considered it and may implement it in future versions, as we will describe more in Section 6.4.

3.7.3.6. Goal 6: Individual Effort

The final goal for the curriculum has much more to do with the way the curriculum is taught than the curriculum itself. We have attempted to teach our curriculum such that all lessons are an individual effort, and that students work on their own as much as possible.

There are many benefits to individual effort. For example, Chamillard has noted that it reduces plagiarism and allows for teachers to better evaluate student learning (Chamillard, 2000). Perhaps more importantly, however, individual effort ensures that each student is responsible for his or her own learning.

Many arguments for group work in programming education stem from the idea that it helps teach the teamwork skills necessary for working in industry (see (Williams, 2002) for an example of this). This may be true. However, our curriculum is centered on computational thinking education – something that precludes most other programming skills, including working on a team with other engineers. For this reason we do not believe that anything is lost by having students work individually, and much is gained.

As our curriculum is designed, there is technically nothing stopping it from being taught using student groups. However, some aspects of the system do encourage individual effort – each GUI allows only one student to log on at a time, and the Management Website's tracking is set up under the assumption that each Genost account represents a student, and not a team. In our own tests we have had each student work individually, though discussion with other students was allowed.

These are some of the ways we have attempted to implement an individual education basis in Genost.

3.8. COMPARISON OF GENOST TO NEWER SYSTEMS

In Section 2.4 we performed a review of eight different "newer" systems that attempt to teach computational thinking ideas, and it was from this review that we took lessons and takeaways for what an "ideal" system would look like. These lessons and takeaways are specifically described in Section 2.4.9. Using those lessons, we designed and built the Genost system, described in great depth above. We will now perform a direct comparison

between Genost and the other eight systems, utilizing the lessons in Section 2.4.9 as our items of comparison. To see a visual representation of this comparison, please see the charts in Appendix G.

3.8.1. Drag and Drop Language

Judging by its universal adoption in almost all recently created educational systems, a drag and drop language is virtually a requirement for an introductory educational system. Alice, Scratch, Lego Mindstorms and Microsoft Robotics Developer Studio all contain this feature, while older systems such as Logo, Myro, IBM Robocode and FIRST Robotics Competition do not. Genost does feature a drag and drop language, as described in Section 3.2.

A drag and drop language brings many benefits with it. By its very nature, a drag and drop language makes it impossible for students to make syntax errors – this is a benefit present in all the reviewed systems containing the drag and drop language, including Genost. These five systems also implement their drag and drop languages such that actions – the commands being sent to the item being programmed – are abstracted to a high level, a feature argued to lighten the cognitive load for younger programmers (Caitlin Kelleher, 2007).

There are certain beneficial features of drag and drop languages that are not implemented by all new systems containing such languages, however. For example, due to a language's graphical design, the language "blocks" can be shaped or colored in such a

way that their appearance indicates their use. Most systems like Alice, Scrath or Mindstorms feature this ability, but Microsoft Robotics Developer Studio does not. Genost does contain this feature: we currently color the blocks differently depending on their use, and in the future we plan to alter their shapes as well.

Just as there are many benefits to the drag and drop languages, there are also certain items that we wish to avoid when implementing them. A major item we wish to avoid is *oversimplifying* the language, something that we have argued that Lego Mindstorm's NXT-G language does by, for example, limiting the depth of nesting. Genost does not feature any artificial limitations of this nature, and any algorithm that can be created in a formal procedural language can ultimately be created in Genost.

Finally, we note again that the "look" of a language can help or hinder student transition from the introductory system to a formal language. We have argued above that languages like VPL in MRDS or the Mindstorms NXT-G language, which are visually dissimilar to formal languages and look more like flowcharts, can hinder the transition to an actual formal language (we have also argued that the visual structure of formal languages, in some ways, orients students towards certain computational thinking ideas). Genost's language is designed such that Genost algorithms are visually similar to algorithms written in a formal language, reading from top to bottom and containing indentation where appropriate.

3.8.2. Virtual Worlds

We argued for the enormous benefits of virtual worlds in programming education in Section 2.4.1 and elsewhere. It is apparent that this feature is widely recognized as beneficial, as all but three systems (Myro, FIRST Robotics Competition and Lego Mindstorms) feature virtual worlds as part of their regular educational system. Genost does feature a virtual world, as described in Section 3.3.

Genost's virtual world is somewhat different in nature to the virtual worlds featured in, for example, Alice or Scratch. The Alice and Scratch virtual worlds are very open in nature, allowing for students to create wide-ranging stories with many characters (in the case of Alice) or manipulate 2D graphics and media in just about any way, in the case of Scratch. We have argued that this openness, while often touted as beneficial, may actually be a hindrance when teaching introductory computational thinking skills. Sources such as (Paul Mullins, 2009), (Orni Meerbaum-Salant, Habits of Programming in Scratch, 2011), (Orni Meerbaum-Salant, Learning Computer Science Concepts with Scratch, 2013) or (Maloney, 2008) have noted that some students may be distracted by the open-world playfulness of these virtual worlds to the detriment of their education. Genost has attempted to alleviate this by providing a less open, more directed virtual world in which students are given an explicit goal to solve. The Genost world is not fully locked down, as students have complete freedom to choose how to pursue this goal; we believe that this design is a good compromise that will better direct students towards learning computational thinking, while still allowing them to have fun.

3.8.3. Robots

Just as virtual worlds bring educational benefit and motivation to students, so we have argued in Section 2.4.1 and elsewhere that robots also bring educational benefits and motivation, though in a different way and to a different degree than virtual worlds. If paired correctly we believe that virtual worlds a nd robots may complement each other, and for this reason we want our system to include a robot as well as a virtual world. Four of the eight "newer" systems feature a robot – these are Myro, FIRST Robotics Competition, Lego Mindstorms and Microsoft Robotics Developer Studio. As described in Section 3.5, Genost also features a robot. Note that only Microsoft Robotics Developer Studio and Genost feature both virtual worlds and robots.

While the four mentioned systems do feature robots, these robots are often implemented in ways that we find problematic. For example, the FIRST Robotics Competition system and the Lego Mindstorms system both focus heavily on *building* the actual robot, which certain studies have noted takes time and emphasis away from learning computer science (Delden, 2008) (Buckhaults, 2009). Genost attempts to avoid this problem by not asking students to build the robot itself, and not involving any mechanical engineering items in the Genost curriculum. This decision has an added benefit of preventing any teacher or student alienation due to unduly technical content, which has been noted to negatively affect adoption of more complicated systems (Tucker Balch, 2008) (Long, 2007).

Genost took inspiration from the Myro system and added to our robot design the goal of having the algorithm controlling the robot be executed on a local computer and

transmitted to the robot, instead of executing on the robot itself. This design choice was made for the benefits described in Section 3.5.1.5 and elsewhere in this thesis: namely, the ability to debug algorithms or step through the code as it executes. Most other robotic systems, such as FIRST, Mindstorms or Microsoft Robotics Developer Studio, do not contain this feature. Myro implemented this feature using local wireless communication to transmit commands to the robot. For reasons described in Section 3.5.3 we attempted to implement this using HTTP. While we were not able to implement this feature for this release, we plan to implement it in future releases in order to attain the benefits it brings. Aside from Genost, only Myro contains this feature.

Finally, the Genost system, due to its general, customizable design is usable with multiple robots. FIRST and MRDS also contain this feature; Myro and Mindstorms do not. We believe that the ability to build and work with multiple robots makes a system more adaptable and allows it to be useful in more varied environments and classrooms – for this reason we are proud to have this feature in Genost.

3.8.4. Curriculum

From the beginning of development, Genost was developed with the needs of the curriculum in mind, and our finished product has an official curriculum to go along with the software. All things being equal, it would seem that a system that has a curriculum to go with it is better than one that does not (since software without a curriculum is simply a tool). Furthermore, we have explicitly argued above that developing these software and

the curriculum alongside one another allows for better "fit" between the two, as opposed to a curriculum developed after the fact (perhaps by a third party).

Very few systems actually feature an official "curriculum". Myro and Alice both have curricula developed and released by the same entities that produced the software, but Scratch, Mindstorms, and Microsoft Robotics Developer Studio do not. A curriculum is not applicable for FIRST Robotics Competition and Robocode (since these are more of a competition and a game, respectively), and it is unclear whether Logo has an official curriculum or not due to its considerable age. Ultimately, then, the fact that Genost has an official curriculum that was developed alongside the software differentiates it significantly from the other systems.

The content and structure of Genost's curriculum also differentiates it from the other systems. Genost teaches computational thinking explicitly through the use of procedural programming – we put the computational thinking instruction first and foremost. Contrast this with systems like FIRST, Alice or Scratch which have a heavy competition, storytelling or "tinkering" focus, respectively. We have noted above in Section 2.4.9 that these features are not necessarily bad, and in fact may be quite beneficial, but that they should not be the major focus for introductory computing education. Genost attempts to attain the proper balance between explicit instruction and allowance for creativity through its four-lesson "fading" structure, in which the first two lessons for a particular topic are guided, and the final two lessons allow the student to solve the problem in their own way.

We have noted above that Microsoft Robotics Developer Studio, in contrast to all of the other systems (Genost included) attempts to serve multiple roles – it attempts to be both an educational system and a useful IDE for industrial robot prototyping and development, and in doing so plays neither role well. Genost consciously avoids this mistake and attempts to focus only on being an educational system.

Finally, we note that the Genost curriculum is designed to be simple and fundamental, and to avoid inappropriately advanced items, such as the computer vision or AI found in the Myro system. We intend Genost to be used only for introductory computer science, and therefore have limited our curriculum accordingly.

3.8.5. Other

In this last section of the comparison between Genost and the eight "newer" systems reviewed in Section 2.4 we will discuss some of the differences that do not fit anywhere else.

A major difference between Genost and these other systems, and one that we are most proud of, is Genost's ability to be extended and customized. Genost's mazes, code blocks, lessons, and even curriculum can all be customized by end users. All of these items can be created, uploaded and used within the Genost system. Almost no other system that we have considered features customization to this degree. The only system aside from Genost that features this customizability is Microsoft Robotics Developer Studio, which allows end users to develop their own DSS services. We consider Genost's

customizability a large benefit in making the system adaptable to different classrooms, easy to develop for, and in general, robust.

We argue above that allowing students to visually follow along with their program code as their algorithm executes (thereby "pairing" the code and the execution) provides educational benefits. Genost features this "pairing" in the sense that the GUI and the simulator live in separate windows, and students may arrange these windows such that both the code and the simulator are simultaneously visible. In the future we plan to make this pairing more explicit, and perhaps even enforced. This feature is not present in many other systems – none of the four code-based systems feature it. Only Alice and Scratch contain this feature. This feature is not applicable to Lego Mindstorms and Microsoft Robotics Developer Studio, because these systems use a real robot instead of a simulated one.

The last item that we will discuss is the integration of a system with social media, which is the only feature that we identified as valuable that Genost does not implement. Scratch is the only prevalent example of this – as described above, all Scratch projects may be uploaded and shared using Scratch's own social media site. This social feature has been described as a primary benefit of the Scratch system. None of the other seven systems we reviewed contain this feature. Genost also does not contain this feature, and we do not have any explicit plans to add it at this time. We chose not to include this due to the difficulty of implementation, and the relative small benefit that it would bring to the Genost system as currently designed. While social media integration does have the

potential to increase both the motivational aspect and the educational efficacy of a system, we are not confident that the increase such an integration would provide would be worth the considerable time and expense that the integration would bring.

3.8.6. Comparison Conclusion

The above five sections feature comparisons between Genost and the eight systems we reviewed. The following table summarizes these comparisons by showing the number of positive and negative features in each system:

Table 3

Number of positive and negative features displayed by each system

System	# Positive Features	# Negative Features
Logo	3	1
IBM Robocode	1	2
Myro	3	2
FIRST Robotics Competition	2	5
Alice	7	1
Scratch	7	1
Lego Mindstorms	5	5
Microsoft Robotics Developer Studio	6	2
Genost	**11**	**0**

This table shows that Genost contains the largest number of positive features (11), and the smallest number of negative features (zero), out of all of the systems. Genost does not implement a single negative feature, and implements all but one positive feature.

The fact that Genost is, according to this analysis, the "best" system is perhaps not surprising, considering that Genost was designed specifically with the lessons and takeaways of these systems in mind. Ultimately, it would appear that the careful planning that went into Genost's design therefore has paid off, as our analysis shows that Genost, at time of writing, is the superior system for teaching introductory computer science.

3.9. GENOST DESCRIPTION CONCLUSION

In the above section, we have described and justified the Genost software and curriculum. Section 3.8 shows that Genost implements almost all the positive features, and none of the negative features of the ideal educational system, as described in Section 2.4.9. By this analysis, Genost is superior to the eight other prominent introductory computer science education systems on the market.

Section 3 argued for Genost's effectiveness in teaching computational thinking by describing its implementation of features that have been proven to work in other systems, and its avoidance of features that have proven to hinder student education. This argument alone is not sufficient to establish Genost's efficacy in teaching computational thinking. For this reason we have run two tests to evaluate whether Genost is in fact effective at teaching computational thinking. The next section, Section 4, will describe the two tests that we have run.

4. TEST DESCRIPTION

In order to determine whether our system was effective at teaching computational thinking we performed two different tests. This section will describe the tests that we held. For each test we will describe the design of the test, our recruitment efforts, the number of students that participated, the allotted time and environment of each test, and the data that was collected. We will discuss the actual results of the test in Section 5.

We held two tests: one test was held at the Arizona School for the Arts[40], and this test will be referred to as the "ASA test." The other was held at Arizona State University using students from ASU's FSE100 (Introduction to Engineering) classes, and this test will be referred to as the "FSE100 test". In the rest of Section 4 we will first describe the elements common to both tests. After this, we will discuss the ASA test first, and the FSE100 test after it.

4.1. COMMON DESIGN OF THE TWO TESTS

In this section we will discuss the general design of the two tests, and the commonalities between them in some depth. Both tests were roughly similar, though each had certain unique elements that will be discussed in the following two sections.

Each test began by giving the participating students a pretest intended to measure computational thinking. Students were then taught the Genost curriculum over a period of weeks. At the end of the class, a posttest was given to measure computational thinking

[40] http://www.goasa.org/

ability again. The pretest and the posttest both used the same testing tool, and the same testing tool was used for both the ASA and the FSE100 tests.

The two tests were carried out over different lengths of time, with different numbers of students, and with different student grade levels. The same lecture slides, worksheets, and mazes were used between the two tests, but the presentation of the material was adapted to the differences between the tests. For example, the FSE100 test had less teaching time overall, so less time was spent on individual lectures and topics as compared to the ASA test.

Despite the differences, each teaching day in either test followed roughly the same pattern. A certain number of curriculum subsections (usually two or three) would be covered each day. Each subsection would begin with a brief lecture to the students. I was the sole lecturer for these two tests. During the lecture we would often do examples, which would involve the students following along with me as I developed an algorithm illustrating the curriculum section topic. During these lecture exercises the students were asked to build the algorithm on their own computers as I walked them through it.

After each lecture, the students were given a certain amount of time to complete the worksheets and mazes for the subsection. Students were asked to work individually and not in groups, though they were allowed to speak with each other and give each other advice. Students were asked to finish worksheets and mazes that were not completed in class outside of class, though this was not stated to be a requirement.

Both test designs were approved by Arizona State University's Independent Review Board. Copies of the approval are available upon request.

4.1.1. The Testing Tool

We created a single testing tool to be used as both the pretest and the posttest. The test is intended to be measure of computational thinking ability in relation to the topics covered during the curriculum, which are basic actions, loops, wait statements, if statements, problem breakdown, and algorithm analysis. The questions are mostly stated in terms of Objective G code, but due to Objective G's clear design (described in Section 3.2.1) we believe that anyone with computational thinking ability and critical thinking skills would be able to understand the questions on the test.

The test has 13 questions. The questions on the test cover the following topics:

- Basic identification of fundamental programming structures (Question 1)
- Reading and evaluation of algorithms (Question 11, Question 12)
- Understanding algorithm design (Question 2, Question 3)
- Understanding algorithm execution (Question 4, Question 5)

- Understanding and applying fundamental programming structures
    - Loops (Question 6, Question 7)
    - Wait Statements (Question 8)
    - If Statements (Question 9)
    - Logical Operators (Question 10)

- Debugging (Question 13)

The test questions have many different formats. The formats of the questions are as follows:

- Multiple choice (Question 7, Question 8, Question 9, Question 10, Question 13)

- True / False (Question 4, Question 11)

- Fill-in-the-blank (Question 2, Question 3, Question 6)

- Matching (Question 1)

- Ordering (Question 5)

- Free response questions (Question 12, Question 13 Extra Credit)

Each question takes up one full page on the test. At the bottom of each page is a sentence stating “I don’t know how to solve Question #X” – students were instructed both verbally and in the test directions to circle this sentence if they do not know the answer to the question. This was included in an attempt to deter guessing.

Each question was worth one point, save Question 13, which was worth one normal point plus one extra credit point. Questions with multiple parts were given a value of one point overall, with the point distributed evenly over the different parts of the question. So, for a question with three parts, each part was worth 1/3 of a point.

The full test may be viewed in Appendix D.

We argue that our testing metric is both reliable and valid. The reliability of the testing metric will be evaluated through the use of Cronbach’s Alpha in Section 5.1. The validity of the testing metric can be assessed by examining the questions and comparing their content to the goals of computational thinking education listed in section 2.1. For reference, those goals are:

1. Ability to Read and Understand Algorithms
2. Ability to Engage in Abstraction
3. Ability to Decompose a Problem into Solvable Processes
4. Ability to Identify the Quality of a Solution

Goal 1 skills are required to answer almost all of the questions on the test, since most of the questions require students to read an algorithm and complete it. Specific questions that focus on Goal 1 skills include Questions 5, 11 and 12. Question 11 asks students to read an algorithm and evaluate whether it will work or not; Question 12 asks students to read an algorithm and trace the path of the robot as it executes the algorithm.

Goal 2 skills are also required to answer virtually all of the questions, since abstraction is required at all levels of computational thinking. Specific questions that require abstraction include Question 1, which asks students to think of fundamental programming structures such as loops or if statements as abstract items, and Question 4, which asks students to move between two different levels of abstraction (the execution of the code vs. the movement of the robot).

Goal 3 skills are required to answer Questions 2, 6, 7, 8, 9, and 10. Questions 6 – 10 all ask the student to complete an algorithm, which requires the student to examine the algorithm and the maze, break them down, and figure out what the best answer is from the available choices to complete the algorithm. Question 2 asks students to explicitly state the four steps of breaking down an algorithm.

Goal 4 skills are required to answer Questions 3 and 13. Question 3 asks students to explicitly state some of the criteria by which a solution is judged for quality. Question 13 asks students to examine and debug a solution, to figure out why it doesn't work and how it can be fixed.

Skills in all four of the listed goals are required to complete the test. Therefore, we argue that our test is a valid measure of computational thinking ability.

4.1.2. The Feedback Forms

In order to collect feedback on the Genost system, we created and distributed two different forms to both groups of students that went through the Genost intervention. Each form collected feedback on a different subject: the first form, which we will call the Likert form, collected feedback on the Genost system as a whole, including the software, the curriculum, and our presentation through the use of Likert scale questions. The second form, which we will call the Free Response form, collected feedback specifically on the curriculum and my presentation, through the use of free response questions. We will describe both forms below.

The Likert form contained ten separate Likert scale questions that ask students to rate their feelings on various metrics related to the Genost system as a whole. Three major metrics were measured on the Likert form, which were:

- Ease of use of the Genost system (Question 1, Question 5, Question 8)
- Educational value of the Genost system (Question 2, Question 4, Question 7)
- Student enjoyment from using the Genost system (Question 3, Question 6, Question 9)

Question 10 asked students to rate their overall satisfaction with their Genost experience and therefore contains elements of all three metrics.

The Likert form may be viewed in Appendix E. Note that the Likert form had a field at the top for students to enter their pseudonym. This field was inadvertently added and was not caught until the forms had been printed. Students were told not to fill out this name field.

The Free Response form contained four separate free response questions asking students to provide feedback on the Genost curriculum and my presentation of it. The students were asked to write down three things they liked, three things they disliked, three things they would change, and any additional comments they had. Students were instructed not to include specific software bugs or improvements in their feedback on the Free Response form. The Free Response form had no field for students to fill in their pseudonym, and students were instructed not to write this pseudonym anywhere on the form.

The Free Response form may be viewed in Appendix E.

For both the FSE100 test and the ASA test, the feedback forms described above were administered after participating students took the posttest.

4.2. ASA TEST

The ASA test was performed at the Arizona School for the Arts, a charter school located in Phoenix, Arizona. The school serves students from grades 5 through 12, though the students participating in our test were all in grades 7 through 12. Tracy Ryan, a teacher at ASA, assisted us with organizing and carrying out the test.

The ASA test was a two-group design, with one group receiving the Genost treatment and the other group receiving no treatment. We will call the group that received the Genost treatment the independent group, and the group receiving no treatment the control group. The two groups were not related and no student was a member of both groups simultaneously.

Both the independent and control group were given the aforementioned pretest at the beginning of the testing period. After the Genost treatment had been given to the independent group, the posttest was given to both groups. This is the basic design of the ASA test.

4.2.1. Student Numbers and Recruitment

The recruitment methods and number of students were different between the independent and control groups. We will review this data for both of the groups in this subsection.

4.2.1.1. Independent Group

Our recruitment target for the independent group was 30 students between 7th and 12th grade. Participation in the group was voluntary, and students were recruited through an online form, advertised through email and in person at the school. Both parents and students were informed of the opportunity. No incentive was provided for participation.

The online recruitment form was opened on August 19th, 2014, and an email advertisement was sent out to ASA newsletter recipients on this same day. The text of the advertisement may be seen in Appendix F. The online form was limited to 30 students – students who filled out the form after it had received 30 submissions were added to a wait list. Throughout the recruitment period, some students who had been part of the initial 30 signups withdrew their signup, and were replaced with students on the wait list.

29 students were present on the first day of the class. Due to one student not being present, another student on the wait list was selected to participate on the second day, bring the class up to 30 participants. Students were allowed to withdraw from the class at any time, and 10 students did withdraw over the course of the Genost class. Ultimately, 20 students from the class completed the pretest, the Genost class, and the posttest, and therefore our final independent group dataset consists of 20 students. No other selection methods were applied to populate the independent group.

The final independent group contained 6 students in 7th grade, 10 students in 8th grade, 1 student in 9th grade, 1 student in 10th grade, and 2 students in 11th grade.

4.2.1.2. Control Group

The control group was recruited from a group of 7th to 12th grade students in a study hall class. The class was asked to take the pretest at the beginning of the Genost intervention period, and was asked again to take the posttest at the end of the Genost intervention period. The control group therefore consists of the students from this class that took both tests. Students were incentivized to participate by offering candy to participants.

Some small selection criteria was applied. Two students who were known to have high programming ability, and were therefore unsuitable for introductory programming education, were excluded from the control group. No other selection criteria was applied.

The final control group consisted of 17 students. The students' grade level were self-reported on the tests, and because not every student filled this out, we do not have accurate grade level information for the control group.

4.2.2. Time Allotted

The Genost intervention that the independent group took part in was administered over two weeks. Class was intended to be held for two hours after school on Monday through Friday from 3PM to 5PM, making 10 days of class total. Due to unexpected events, class was actually only held for 8 days – one class was cancelled due to unexpected weather, and another was cancelled due to early release at ASA. Therefore, the total amount of time students spent in the Genost intervention was 16 hours.

On most class days, two curriculum sections were presented. Students spent the first 30 minutes listening to and participating in a lecture on the first curriculum section, and the second 30 minutes doing worksheets and self-directed algorithm creation for that section. The second hour was spent similarly, 30 minutes in lecture and 30 minutes doing worksheets. Students who did not finish their worksheets in class were asked to complete them outside of class.

4.2.3. Test Environment

The eight classes that were part of the Genost intervention were held in a classroom on the ASA campus in Phoenix, Arizona. Each student was given a Macbook laptop with an internet connection – students were able to use the web browser on these laptops to interact with the Genost software.

Students were seated at long tables instead of individual desks, due to the way the room was organized. The students were allowed to sit wherever and with whomever they wanted, which in some cases led to excessive socialization and off topic play during the class. In some cases we reseated students in an attempt to prevent this.

The classroom had a projector at the front of the room, which was used to project the lecture onto a screen.

4.2.4. Data Collected

The primary data that were collected in the ASA test were the scores on the pretest and posttest. We also collected feedback data from the student through two forms. These two sets of collected data will be described below, and the actual data collected, and our analysis of that data, will be presented in Section 5.

In addition to the items mentioned above, we also collected the following information from the students:

- Attendance records for the Genost classes

- Records regarding the worksheets and mazes completed by each student

- Various data related to the interaction of the student with the Genost software, such as the number of times students simulated an algorithm in a specific maze, or the date and time that a student switched from one lesson to the next. Each datum that was collected was tagged with a date and time stamp, along with other contextual information.

These three items will not be presented or analyzed in this thesis.

4.2.4.1. Pretest / Posttest Data

The same test that was used for both the pretest and the posttest. This test is the one that was discussed in Section 4.1.1 and shown in Appendix D. All students in the independent and control groups took both a pretest and a posttest.

We collected the score for each individual question on this test, and the overall score, for each student and on both administrations of the test. Each test was hand-graded. No manipulation or adjustment was done to any of the test data.

4.2.4.2. Feedback

The Likert and Free Response forms described above in Section 4.1.2 and shown in Appendix E were given to the ASA independent group on the last day of class. All students in the independent group filled out both forms.

The data collected on the Likert form was collected and averaged. Students were instructed to only circle one number on the Likert scales; when multiple numbers were circled, we chose the lowest number circled.

The data collected on the Free Response form was analyzed, and responses were noted, classified and counted.

4.3. FSE100 TEST

The FSE100 test was held at Arizona State University using students from ASU's Fall 2014 FSE100 Introduction to Engineering class. This class is a required introductory course for many majors offered through the Fulton Engineering School, including computer science, computer systems engineering, mechanical engineering, electrical engineering, and industrial engineering.

The FSE100 test was a three group design. The three groups are as follows:

- The Genost group, which participated in an extracurricular class designed to teach the Genost curriculum.

- The Python group, which participated in CodeAcademy.com's online Python course[41].

- The Control group, which did not undergo any treatment.

The Genost class was given at the beginning of the semester. The Python course, because it is an online course, was available throughout the semester for any student who desired to take it. At the end of the semester the FSE100 grade data from all participants in the three groups was collected – these data served as our main data set for the FSE100 test.

[41] http://www.codecademy.com/en/tracks/python

The main goal of the FSE100 test was to compare the grade data of the three groups and determine whether there was any significant difference between them.

In addition to the grade data, we also collected pretest and posttest data from the Genost group. The Genost group was given the pretest at the beginning of the Genost class, and the posttest at the end of this class. However, we were not able to administer the pretest and posttest to the other two groups.

4.3.1. Student Numbers and Recruitment

ASU's FSE100 course is offered in different "flavors" – for example, one FSE100 course may be taught with a computer science flavor, intended for the CS and CSE majors, while another may be taught with a mechanical engineering flavor, intended for the ME and Civil Engineering majors. We limited our recruitment to the computer-flavored FSE100 classes, of which there were 13 in Fall 2014. Each of the 13 FSE100 classes had 43 students in it, for a total of 559 students[42]. It was from these 559 students that we recruited our participants in the FSE100 test.

Participation in all three groups was entirely voluntary. At the beginning of the semester, I visited each of the 13 FSE100 sections and gave a short speech explaining our research and asking students for two things: first, to consent to releasing their FSE100 grade data to us, and second, to sign up for either the Genost group or the Python group if they were interested in participating. Students consenting to releasing their grade data signed a

[42] Note that this is the number of seats filled at the beginning of the semester; the number of students that completed the FSE100 course is fewer than 559.

release form during my visit. The speech that was given during these initial classroom visits may be viewed in Appendix F.

Students were offered an incentive to participate in either the Genost or the Python group. All students who completed participation in either of these groups (defined as either completing the Python course or taking both the Genost pretest and posttest) received a 10% extra credit bonus to their FSE100 grade. Students were not offered an incentive to release their grade data to us.

At the end of these classroom visits, we had received grade release forms from many students, and contact information from students interested in either the Genost or Python courses. The next three sections will detail the student numbers and any further recruitment efforts for the three groups.

4.3.1.1. Genost Group

After the classroom visits at the beginning of the semester, we found that the number of students that had signed up for the Genost extracurricular activity was very large. We were not able to accept all signups due to the nature of the Genost extracurricular activity as an in-person class with limited space. We therefore were forced to narrow the number of students from the pool.

We first created a questionnaire form and emailed it to all students who had signed up.

The form asked for basic contact information along with the following questions:

1. Are you interested in participating in the ASU Genost-based extracurricular activity?

2. Will you be able to come to all six sessions?

3. Can you commit to completing the activity from beginning to end?

4. Depending on the way the meetings go, we may ask you to do some extra practice work outside of class. Is this something you would be willing and able to do?

5. What is your current grade level?

6. Please rate what you consider your programming ability, from 1 (no ability) to 10 (expert)

By the time we closed the form for submissions roughly one week after sending it out, we had received exactly 100 unique responses. Our goal was to narrow the pool down to 30 students.

We began by eliminating all students that did not answer “yes” to questions 1, 2, 3 and 4. This eliminated 29 students.

We next examined the remaining rows and their answers to questions 5 and 6. We removed all students who were not freshmen and who rated themselves as having a programming ability greater than or equal to a 5. This eliminated 15 students. Our justification for this narrowing measure is that we are attempting to teach introductory programming, and are less interested in teaching students who already have considerable programming ability.

After taking the measures above, 56 rows remained. We narrowed the pool from 56 to 30 through directed random selection, done in the following way:

1. We arranged the 56 rows into 13 groups, grouping them by their FSE100 class.

2. We selected 2 students randomly from each of the 13 groups, resulting in 25 students selected. One group only had one student, which is why we finished this step with 25 instead of 26 students.

3. Removing the selected students from the 13 groups, we then chose the 5 most populous groups, which had populations of 7, 6, and 3 groups of 5. Note that there were actually 5 groups with population 5 – the 3 groups chosen from those 5 were chosen randomly.

4. We randomly selected one student from each of these 5 most populous groups. This resulted in 30 randomly selected students.

We therefore began the Genost test with 30 students. On the first day of class, 4 students did not show up, resulting in a starting group size of 26. 9 students withdrew from the study or otherwise did not complete it, resulting in a final Genost group size of 17 students.

These 17 students all signed an additional consent form releasing the data collected with Genost to us for study. This consent form may be viewed in Appendix F.

4.3.1.2. Python Group

The Python group consists of all students who signed up for the Python course during the classroom visits (or contacted us about their desire to participate sometime during the semester), completed the Python course, and released their grade data to us. Each student that completed the Python course was counted by sending us link to or screenshot of their CodeAcademy account, which shows the completion of the course.

No further selection or narrowing of the Python group was performed. There are 38 students in the Python group.

4.3.1.3. Control Group

The control group was made up of all students who released their grade data to us but did not participate in either the Genost or the Python group. There are 317 students total in the control group.

4.3.2. Time Allotted

The Genost class was administered over the course of three weeks. Students met twice a week on Mondays and Wednesdays for two hour sessions, 7PM to 9PM. On most class days, two curriculum sections were presented. Students spent the first 30 minutes listening to and participating in a lecture on the first curriculum section, and the second 30 minutes working on worksheets for that section. The second hour was spent similarly, 30 minutes in lecture and 30 minutes doing worksheets. Additionally, all students were asked to complete outside of class all worksheets not completed in class.

Students who did not attend one class were asked to come in thirty minutes early to the subsequent one for makeup. Not counting this makeup time, students spent 12 hours as part of the class.

The Python class on CodeAcademy.com is a set of step-by-step tutorials and does not feature any human instructor, so students were able to take this course at any time during the Fall 2014 semester. We required Python group students to complete the online course by the final day of classes for the Fall 2014 semester, which was December 5th. Aside from that requirement, students were allowed to start the course whenever they wanted and to take as long as they wanted to complete it.

4.3.3. Test Environment

The six classes that were held as part of the Genost intervention were held in a computer lab on ASU's Tempe campus. Each student used a desktop computer running Windows

7. Students used the computer's web browser to interact with the Genost software. Early in the test, some students chose to use their own laptop instead of the desktop lab computers.

Students were seated at long tables instead of individual desks, due to the way the room was organized. The students were allowed to sit wherever they wanted during the class.

The classroom had a projector at the front of the room, which was used to project the lecture onto a large screen.

4.3.4. Data Collected

The primary data that were collected in the FSE100 test were the student FSE100 final grade percentage. In addition to this grade data, we also collected the Genost group's scores on the pretest and posttest. Because these tests were not administered to the Python and Control groups, no score data were collected from them. Finally, we collected feedback data from the student through two forms. These three forms of data collection will be described below, and the actual data collected, and our analysis of that data, will be presented in Section 5.

In addition to the items mentioned above, we also collected the following information from the students:

- Attendance records for the Genost classes

- Records regarding the worksheets and mazes completed by each student

- Various data related to the interaction of the student with the Genost software, such as the number of times students simulated an algorithm in a specific maze, or the date and time that a student switched from one lesson to the next. Each datum that was collected was tagged with a date and time stamp, along with other contextual information.

These three items will not be presented or analyzed in this thesis.

4.3.4.1. Student Grades

The full gradebook for all students who released their data to us by signing a consent form was sent to us by the FSE100 professors at the end of the Fall 2014 semester. While the full gradebook was available, only the final FSE100 score was used in our analysis. This score is a grade percentage and therefore is on a scale from 0 to 100. The 10% extra credit incentive offered to participants in the Genost or Python groups is not included in this final percentage.

Our hypothesis that if Genost does succeed in teaching computational thinking skills, then this will be reflected in higher grades in FSE100 for students who underwent the Genost intervention than those who did not. We are assuming that students with higher

computational thinking skills will receive higher FSE100 grades. Note that we are not claiming that the FSE100 grade is a direct measure of computational thinking ability, only that computational thinking skills will result in higher grades in computer science courses. This is consistent with what we have argued above in regards to computational thinking as a "prerequisite" to computer science.

4.3.4.2. Pretest / Posttest Data

The same testing instrument was used for both the pretest and the posttest. This test is the same one that was discussed in Section 4.1.1 and shown in Appendix D. All students in Genost group took both a pretest and a posttest.

We collected the score for each individual question on this test, and the overall score, for each student and on both administrations of the test. Each test was hand-graded. No manipulation or adjustment was done to any of the test data.

4.3.4.3. Feedback

The Likert and Free Response forms described above in Section 4.1.2 and shown in Appendix E were given to the Genost group on the last day of class. All students in the Genost group filled out both forms.

The data collected on the Likert form was collected and averaged. Students were instructed to only circle one number on the Likert scales; when multiple numbers were circled, we chose the lowest number circled.

The data collected on the Free Response form was analyzed, and responses were noted, classified and counted.

5. DATA RESULTS AND ANALYSIS

In Section 4 we described the design of our tests and the data that was collected. In this section we will present the results of these tests and our analysis of the collected data.

We will begin by presenting the results from our reliability analysis of the testing metric that was used in both tests, in Section 5.1. We will then present the results and analysis of the ASA test in Section 5.2, and the results and analysis of the FSE100 test in Section 5.3. Finally, we will discuss possible weaknesses of our data in Section 5.4.

5.1. PRETEST / POSTTEST RELIABILITY ANALYSIS

In Section 4.1.1 we argued for the validity of the testing tool. In this section we will argue for the reliability of the testing tool, as measured by Cronbach's Alpha.

The test was given six times: four times in the ASA test and twice in the FSE100 test. We performed a Cronbach's Alpha test on each of these administrations. The results of this analysis may be viewed below:

Table 4

The results of the Cronbach's Alpha test on six administrations

Test	Cronbach's Alpha	# of Items Removed Due to 0 Variance	# of Items Which Would Increase Cronbach's Alpha if deleted
ASA – Control Pretest	.859	0	4 (Q1, Q3, Q11, Q12)
ASA – Control Posttest	.828	2 (Q9, Q10)	0
ASA – Independent Pretest	.758	0	2 (Q4, Q7)
ASA – Independent Posttest	.782	0	2 (Q4, Q11)
FSE100 – Pretest	.643	1 (Q7)	4 (Q2, Q3, Q6, Q9)
FSE100 – Posttest	.449	2 (Q8, Q10)	4 (Q4, Q5, Q7, Q13)

In 4 out of 6 administrations, Cronbach's Alpha was greater than .7. In one of the two administrations that did not have a Cronbach's Alpha greater than .7, the Cronbach's Alpha score was very close to .7 (.643). The remaining administration had a low score of .449. Because the majority of administrations had a high Cronbach's Alpha score, we are confident that our test has a high level of internal consistency and is valid.

As can be seen in
, some items were removed from some of the Cronbach's Alpha test due to zero variation. The ASA Control Group posttest had two items removed – this was because no student got these two questions right. The same is true for the one question removed from the FSE100 pretest – no student scored any points on this item for that administration of the test. Finally, in the FSE100 posttest, no student got Q8 or Q10 wrong, and so this question also had zero variance.

On all administrations of the test except the ASA Control Posttest, there were some questions which, if they were removed, would increase the Cronbach's Alpha score. These items are listed in the last column of
.
Table *5* shows the distribution of this data across the thirteen questions.

Table 5

Number of Times Each Question's Removal would Increase Cronbach's Alpha

Question #	**# of times removal would increase Cronbach's Alpha**
1	1
2	1
3	2
4	3
5	1
6	1
7	2
8	0
9	1
10	0
11	2
12	1
13	1

The data in
Table *5* is fairly well distributed across the thirteen questions – most questions were removed at least once, some were removed twice, and one was removed three times. Because this data is well distributed, and no one question has a removal number much larger than the others, we do not believe that any of the questions on our testing metric is inherently unreliable.

Section 4.1.1 establishes the validity of our testing metric. The current section also establishes its reliability. Because of this, we believe that we can proceed to analyze the scores on administrations of this test with the confidence that these scores do represent the degree to which students understand fundamental computational thinking ideas.

5.2. ASA TEST RESULTS AND ANALYSIS

Our ASA test, as described in Section 4.2, was a two-group test, one of which was a control group and the other of which was the independent group. We collected pretest / posttest data from both groups, and feedback data from the independent group. In this section we will present our analysis of this data.

5.2.1. Group Similarity Test

Because we are comparing two separate groups of students, we want to first establish that these students are drawn from similar populations and the groups do not significantly differ. In order to do this, we performed an independent means t-test on the pretest scores of these two groups.

As can be seen in Table 6 below, no significant difference was found between the two groups ($t(42) = -.797$, $p > .05$). The mean pretest score of the control group ($m = 20.59$, $sd = 22.176$) was not significantly different from the mean of the experimental group pretest scores ($m = 15.89$, $sd = 16.842$).

Table 6

Independent means t-test to establish similarity of populations between ASA test Control and Independent groups

		Levene's Test for Equality of Variances		t-test for Equality of Means				
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference
13.	Equal variances assumed	2.495	.122	-.797	42	.430	-4.699	5.898
	Equal variances not assumed			-.748	27.500	.461	-4.699	6.280

This indicates that the populations in the two groups were not significantly different from one another, and that we may compare the performance of the two groups.

5.2.2. Test Score Analysis

An ANCOVA test was performed on the ASA data to determine whether the posttest scores were significantly different between the independent group and control group, with the pretest score treated as a covariant.

Note that, in the tables below, Group 1 is the independent group, and Group 2 is the control group.

5.2.2.1. ANCOVA Assumptions

To draw conclusions from an ANCOVA test, a series of assumptions about the data must be made. We ran a series of tests on the data to validate these assumptions, which will be described below.

A test of between-subject effects was run in order to determine the homogeneity of regression slopes. The interaction term (group * pretest) was found not to be statistically significant, $F(1, 33) = .017$, $p = .897$. Table 7 below shows this data.

Table 7

Test of between subjects effects for ASA data to establish homogeneity of regression slopes.

Dependent Variable: Posttest

Source	Type III Sum of Squares	Df	Mean Square	F	Sig.
Group	13950.736	1	13950.736	43.039	.000
Pretest	2426.957	1	2426.957	7.487	.010
Group * Pretest	5.530	1	5.530	.017	.897
Error	10696.772	33	324.145		
Total	98806.692	37			

A Shapiro-Wilk's test was run to ensure that the standardized residuals for the interventions and for the overall model were normally distributed. This test showed that these were normally distributed ($p > .05$). These results can be seen below in Table 8.

Table 8

Shapiro-Wilk test of normality of residuals for ASA data

	Group	Shapiro-Wilk		
		Statistic	df	Sig.
Standardized Residual for Posttest	1	.961	20	.564
	2	.907	17	.088

Homoscedasticity was established by a visual inspection of a scatterplot of the residuals. There were also no outliers in the data, as assessed by no cases appearing with standardized residuals greater than $\pm$3 standard deviations.

Certain assumptions were not met. There was not a linear relationship between the pretest scores and posttest scores, and there was not homogeneity of variances, as assessed by Levine's test of homogeneity of variance ($p = .023$). These violations may partially be explained by a floor effect in our pretest data. Despite not meeting these assumptions, we proceeded with the ANCOVA test, as these violations are not large, and ANCOVA is known to be robust against violations of assumptions. Furthermore, the strong significance found after running the ANCOVA makes it unlikely that these assumption violations led to a Type I error.

5.2.2.2. ANCOVA Results and Analysis

After investigating the assumptions, we ran the actual ANCOVA test. The result of this test may be seen below in Table 9.

Table 9

Results of ANCOVA on ASA test scores

Dependent Variable: Posttest

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Partial Eta Squared
Pretest	2680.604	1	2680.604	8.516	.006	.200
Group	27811.964	1	27811.964	88.355	.000	.722
Error	10702.302	34	314.774			
Total	98806.692	37				

The result of the ANCOVA test show that, after adjustment for pretest score, there was a statistically significant difference in posttest scores between the control and independent groups, $F(1,34) = 88.355$, $p < .0005$, partial $\eta_2 = .722$.

Post hoc analysis was performed with a Bonferroni adjustment. This analysis shows that posttest scores were statistically significantly greater for the independent group ($p < .0005$). This data may be seen below in Table 10.

Table 10

Post-hoc analysis on ASA ANCOVA

Dependent Variable: Posttest

(I) Group	(J) Group	Mean Difference (I-J)	Std. Error	Sig.[b]
1	2	55.839*	5.940	.000
2	1	-55.839*	5.940	.000

Based on estimated marginal means

*. The mean difference is significant at the .05 level.

b. Adjustment for multiple comparisons: Bonferroni.

The Cohen's D value for this data is 9.4. This is an extremely large effect size, perhaps suspiciously so. Possible reasons for the large effect size come from the control group, whose scores actually decreased from pretest to posttest. Possible problems with the control group include:

- The control group knew that they were part of a test, and may have experienced an observation expectancy effect.

- The control group may have had low motivation to take the test seriously – this may explain the decrease in scores.

- Many people withdrew from the study in both groups.

These facts should be kept in mind when evaluating the ASA test results.

5.2.2.3. Mann-Whitney U Assumptions, Results and Analysis

Due to the fact that certain assumptions for the ANCOVA were not met, as discussed in Section 5.2.2.1, a Mann-Whitney U test was run to confirm our ANCOVA result. The Mann-Whitney U test was run to determine if there were differences in change scores (posttest minus pretest) between the independent group and the control group. Based on a visual assessment of a population pyramid, we determined that the distributions of change scores were similar enough to allow us to use the Mann-Whitney U test to compare medians (see Figure 24).

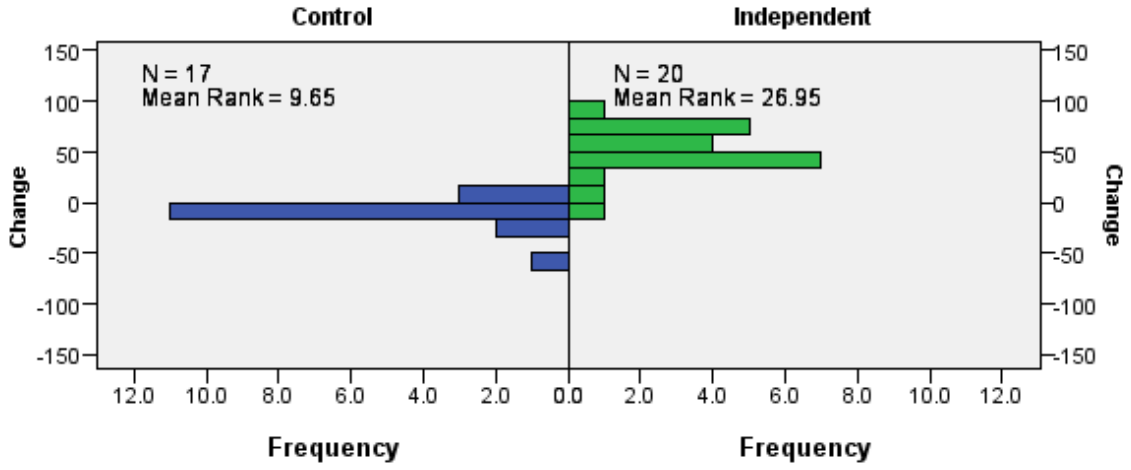


Figure 24. Mann-Whitney U population pyramid

The result of the Mann-Whitney U test was as follows: median change scores was statistically significantly higher in the independent group (48.065) than in the control group (-5.154), U = 11, z = -4.846, p < .001, using an exact sampling distribution for U.

This result confirms our ANCOVA analysis, and allows us to state with confidence that the Genost treatment did elicit an increase in computational thinking scores.

5.2.3. Feedback Analysis

Two sources of feedback were collected from the independent group during the ASA test, as described in Section 4.2.4.2.

5.2.3.1. Software Feedback (Likert Scales)

All students in the independent group filled out a feedback form containing ten Likert scales, with three questions for the categories of ease of use, educational efficacy and enjoyment of the Genost software. Additionally, one question was dedicated to rating the overall experience of using Genost. The Likert scales measured on a scale of 1 to 10, with 1 being the lowest score and 10 being the highest. The feedback collected was averaged and the results are as follows:

Table 11

The Likert scale averages from the Genost software feedback form, ASA test

Scale	Average Score
Ease of Use (Q1, Q5, Q8)	7.767
Educational Efficacy (Q2, Q4, Q7)	8.183
Enjoyment (Q3, Q6, Q9)	7.15
Overall Experience (Q10)	7.71

Because our form has not been administered for other educational systems, we cannot do numeric comparisons to determine whether this feedback is higher or lower than, say,

Scratch, Alice or NXT. However, we can at least see that the numbers themselves are on the high end of the scale.

5.2.3.2. Curriculum Feedback (Free Response)

The second sources of feedback collected from the independent group was a series of four free response questions focusing on the Genost curriculum and our presentation. Students were asked to list three things they liked, three things they disliked, three things they would change, and any other comments they had, and were specifically instructed to limit these things to the Genost curriculum and our presentation. We examined these responses and, for the first three free response questions (liked, disliked, and would change) we tabulated common themes in the responses. From this free response feedback, we can examine common themes for what the students liked, found easy, found educational, as well as what they did not like, found difficult, or did not understand.

We will examine this free response data over the next three subsections.

5.2.3.2.1. Free Response Data – What Students Liked

Table 12 shows common themes that the students identified in their free response feedback for the question "Please write three or more things that you liked about the Genost curriculum and my presentation".

Table 12

Free response tabulation for "Liked" question, ASA Test

# Responses / % of Total	Theme
10 / 16.9%	High quality / clarity of presentation by instructor
10 / 16.9%	Curriculum easy to learn / use / understand
5 / 8.5%	High quality / clarity of presentation on lecture slides
5 / 8.5%	Felt that curriculum was a good introduction to programming
4 / 6.8%	Had fun
3 / 5.1%	Written response was unclear
3 / 5.1%	Enjoyed the visual programming basis (i.e. liked drag/drop better than coding)
3 / 5.1%	Student felt like he or she learned something
2 / 3.4%	Good introduction to computational thinking
2 / 3.4%	Good difficulty progression (started out easy, became more challenging)
2 / 3.4%	Good conceptual progression (student felt scaffolding was appropriate)
2 / 3.4%	Open nature of the software ("multiple answers to problems and no strict rules")
1 / 1.7%	Related to software
1 / 1.7%	Lessons well ordered
1 / 1.7%	Related to class section (Small class size, flexible etc.)
1 / 1.7%	Liked mazes
1 / 1.7%	Liked video game theme
1 / 1.7%	Good presentation speed
1 / 1.7%	Liked "achievement based" style
1 / 1.7%	Felt that lectures related to exercise

We can see that the two most prominent themes were that the curriculum was easy to learn and understand, and that we presented it well. Other major themes that students liked include well-constructed lecture slides, and a belief that the Genost course was a good introduction to programming. It is notable and encouraging that these responses are exactly what we were aiming for in constructing the Genost system.

5.2.3.2.2. Free Response Data – What Students Disliked

Table 13 shows common themes that the students identified in their free response feedback for the question "Please write three or more things that you disliked about the Genost curriculum and my presentation".

Table 13

Free response tabulation for "Disliked" question, ASA Test

# Responses / % of Total	**Theme**
9 / 17.3%	Class was taught too fast
9 / 17.3%	Class featured too many worksheets
8 / 15.4%	Response related to software bugs
5 / 9.6%	Not enough time for exercises
4 / 7.7%	Lack of formal language education
3 / 5.8%	Not fun
2 / 3.7%	Response related to software design
2 / 3.7%	Student didn't understand something
2 / 3.7%	Worksheets were too long
1 / 1.9%	Class was taught too slow
1 / 1.9%	Curriculum featured repetitiveness
1 / 1.9%	Review questions were too easy / repetitive / unnecessary
1 / 1.9%	Simulated robot moved slowly
1 / 1.9%	Poor design of the lecture slides
1 / 1.9%	Class time too long
1 / 1.9%	Genost did not run on well on all computers
1 / 1.9%	Related to class section (time of day, classroom management, etc.)

Overwhelmingly, the response data for the disliked question were related to the class design, specifically its length of time. Three of the top four responses have to do with this, meaning that, had the class been taught over a longer period of time, these common complaints may have been alleviated. This is not surprising, as the Genost curriculum

contains a large amount of information and requires a considerable amount of work. It is difficult to fit into the relatively short 16 hour instruction time of the ASA course.

The only response in the top four not dealing with the class design has to do with software bugs, a common complaint despite students being instructed not to include these on their free response forms.

Another interesting response is the lack of real life language involvement – four students noted that they would have liked to see more information relating Genost to a real life language, such as Java or C. Part of the reason for this response may have been student expectations – some students entered the Genost class under the impression that they would be learning a formal language, and at least some of the students already had experience with formal languages.

Relatively few students reported that they found the curriculum unenjoyable, and only two students reported not understanding something. However, due to the overwhelming response related to the class design, it may be that responses of this nature were suppressed. We may at least conclude from this data that whatever lack of enjoyment or lack of understanding that students experience was overwhelmingly outweighed by their dislike of the short amount of instruction time.

5.2.3.2.3. Free Response Data – What Students Would Change

Table *14* shows common themes that the students identified in their free response feedback for the question “Please write three or more things that you would change about the Genost curriculum or my presentation”.

Table 14

Free response tabulation for "Would Change" question, ASA Test

# Responses / % of Total	**Theme**
13 / 30.2%	Response related to software bugs
6 / 14%	Would add more time to class (Not enough time to go over everything in class, taught too fast, etc.)
5 / 11.6%	Fewer worksheets
4 / 9.3%	Related to class section / study design (class time, pretest, posttest, number of students, etc.)
4 / 9.3%	Improve pace - faster on easy stuff, slower on hard stuff
2 / 4.7%	Make things more fun
1 / 2.3%	Link to formal language education
1 / 2.3%	Change presentations - make more concise, less redundant
1 / 2.3%	Put content online / make it accessible
1 / 2.3%	Improve maze design / theme
1 / 2.3%	Improve worksheet quality - Add tips, hints
1 / 2.3%	Do not show solution in first lecture
1 / 2.3%	Do not require filling out worksheets
1 / 2.3%	More independent work
1 / 2.3%	Make different ways to solve a maze

Once again, despite being instructed not to include this information, many students responded with requests to solve software bugs. Following this, the top three responses all were related to the class design, especially the length of time. The most common

theme that was not involved with class design was instead related to the pace of the curriculum. There were no commonly suggested changes that had to do with the curriculum content itself.

As with the "dislike" question, it may be that the class design problems stood out so much to the students that problems related to the curriculum were suppressed. However, it is also notable that the second most common request was for *more time* with the curriculum, which seems to indicate at least that students did not dislike their time spent working with Genost.

5.3. FSE100 TEST RESULTS AND ANALYSIS

Our FSE100 test, as described in Section 4.2, used a three-group test, with one control group and two in dependent groups (the Genost group and the Python group). We collected the final FSE100 grade from all participants in the three groups. Additionally, we collected both pretest / posttest data and feedback data from the Genost group. In this section we will present the analysis of this data.

5.3.1. FSE100 Grade Analysis

The primary test that was performed on the FSE100 grade data was a one-way ANOVA, to determine whether the final FSE100 grades were significantly different between the three groups.

5.3.1.1. One-Way ANOVA Assumptions

To draw conclusions from an ANOVA test, a series of assumptions about the data are made. We ran a series of tests on the data to validate these assumptions.

The first assumption is that there are no significant outliers in any of the groups. We examined the data for outliers with the use of a boxplot, and identified as an outlier any value greater than 1.5 box-lengths from the edge of the box. Upon inspection, we identified 9 outliers, students whose grades were 75% or lower. Because the vast majority of grades were much higher than this (usually in the range of 80% to 100%), and because many of these outliers were far below the 75% mark, we believe that these outliers represent students who did not attend class (but also did not withdraw), or students who were not serious about the course for other reasons. For this reason, we removed the outliers from the data.

The second assumption, normality of the data, was assessed using a Shapiro-Wilk test of normality. This was performed on the data. The results may be seen below in Table 15.

Table 15

Shapiro-Wilk test of normality for FSE100 grade data

	Group	Shapiro-Wilk		
		Statistic	df	Sig.
Grade	Control	.705	317	.000
	Python	.910	38	.005
	Genost	.979	17	.942

a. Lilliefors Significance Correction

The Genost group is seen to easily be normally distributed ($p > .05$). However, the Python and Control groups both fail to reject the null hypothesis, and therefore we cannot claim that this data is normally distributed. This is in line with our earlier observation about the heavy grade skew towards the high range for the FSE100 grade data. While the Genost group is small enough for this skew to not be significant ($n = 17$), the Python group ($n = 37$) and Control group ($n = 309$) are large enough that this skew becomes highly visible.

Despite this assumption violation, we chose to carry on with the one-way ANOVA. It has been argued that ANOVA is robust to deviations from normality in cases where sample sizes are large (Lix, 1996), and in cases where the skew is similar across groups (Sawilowsky, 1992). Because the groups in which the assumption is violated are both large, and because the skew is the same across all groups, we chose to proceed.

The final assumption tested was homogeneity of variances. Levine's test for equality of variances was run, and it was found that the variances for the FSE100 grade data were homogenous (p = .166). This can be seen in Table 16 below.

Table 16

Results of Levine's test for homogeneity of variances, FSE100 grade data

Grade

Levene Statistic	df1	df2	Sig.
1.805	2	360	.166

5.3.1.2. One-Way ANOVA Results and Analysis

Having run the assumptions tests above, we ran the ANOVA test on the data.

The descriptive final grade data is presented in Table 17 below.

Table 17

The descriptive statistics for the FSE100 grade data

Grade

	N	Mean	Std. Deviation	Std. Error	Minimum	Maximum
Control	309	91.585568	8.7981871	.5005114	57.7400	106.7848
Python	37	92.741530	6.8492320	1.1260068	74.3208	107.3308
Genost	17	96.028957	7.0763748	1.7162730	83.8517	111.3680
Total	363	91.911486	8.5846856	.4505791	57.7400	111.3680

Data are mean +- standard deviation. As can be seen, the final grade increased from the control group (n = 309, 91.59 $\pm$ 8.8), to the Python group (n = 37, 92.74 $\pm$ 6.85), to the Genost group (n = 17, 96.03 $\pm$ 7.08), in that order.

The actual ANOVA test results can be seen in Table 18 below.

Table 18

ANOVA test results for FSE100 grade data

Grade

	Sum of Squares	df	Mean Square	F	Sig.
Between Groups	346.525	2	173.263	2.369	.095
Within Groups	26331.726	360	73.144		
Total	26678.251	362			

The differences between the FSE100 grade data, though they were arranged in concordance with our expected results (control group had the lowest grades, followed by Python group, followed by Genost group with the highest grade), were not statistically significant, $F(2, 360) = 2.369$, $p = .095$. Because the group means were not statistically significant ($p > .05$), we cannot reject the null hypothesis and cannot accept the alternative hypothesis.

A univariate analysis of variance test was run in order to confirm this result, and to identify the observed power of the test. The results of this can be seen below in Table 19:

Table 19

Test of between-subject effects for FSE100 grade data

Dependent Variable: Grade

Source	Type III Sum of Squares	df	Mean Square	F	Sig.	Noncent. Parameter	Observed Power[b]
group	346.525	2	173.263	2.369	.095	4.738	.478
Error	26331.726	360	73.144				
Total	3093201.053	363					

b. Computed using alpha = .05

We can see from this analysis that the same result is found: the differences between groups are not statistically significant, $F(2, 360) = 2.369$, $p = .095$. We can also see that the observed power of the experiment was .478, a low observed power.

We postulate that the low observed power indicates that this experiment was underpowered. Possible reasons for this underpowered design including utilizing data which was heavily skewed towards the high range, and large differences between the group sizes. Because the experiment is underpowered, while we cannot conclude that Genost had an effect on FSE100 grade data, we also cannot conclude that the Genost intervention had no effect. A future experiment may attempt to rectify this underpowered design by testing Genost's effect on the final grades of a different class with a more normal grade distribution, and by equalizing the group sizes.

5.3.2. Test Score Analysis

As with the ASA test, we collected pretest / posttest scores from the Genost independent group. However, we did not collect this data from the other two groups, and so we are limited on what analyses we can perform on this data, and what we can conclude from it.

We chose to run a paired-sample t-test on this data in order to determine whether there was a statistically significant mean difference between the pretest and posttest scores.

5.3.2.1.1. Paired-Samples T-Test Assumptions

In order to draw conclusions from a paired-samples t-test, two assumptions must be met.

The first assumption is that there are no outliers in the data. A boxplot was created to test this assumption. Visual inspection of the boxplot showed that there were no outliers in the pretest / posttest data.

The second assumption is that the differences between the two groups are normally distributed. A Shapiro-Wilk test was performed to investigate this. The results of this test may be seen below in Table 20:

Table 20

Shapiro-Wilk test of normality for FSE100 pretest / posttest data

	Shapiro-Wilk		
	Statistic	df	Sig.
difference	.851	19	.007

This test reported that the differences were not normally distributed (p = 0.007). Despite the data not being normally distributed, we proceeded to perform the paired-samples t-test. We continued because the paired-samples t-test is robust against normality assumption deviations.

5.3.2.2. Paired Samples T-Test Results and Analysis

The results of the paired-samples t-test may be seen below in Table 21:

Table 21

Paired Samples T-Test for FSE100 pretest / posttest data

		Mean	Std. Deviation	Std. Error Mean			
Pair 1	Posttest - Pretest	56.84210526084	19.61696781889	4.50044106849	12.630	18	.000

As can be seen in Table 21, the posttest scores (88.28 +- 11.6) were significantly higher than the pretest scores (31.44 +- 17.44). The Genost treatment elicited a statistically

significant increase in computational thinking test scores of 56.84 points (95% CI, 47.39 to 66.3), $t(18) = 12.63$, $p < .0005$.

A Cohen's d value was calculated to determine the effect size of the Genost treatment. This value was $d = 2.9$. According to Cohen's effect size interpretation guidelines, this is a large effect ($d > 0.8$).

This analysis does not necessarily indicate that the Genost treatment was responsible for this increase in scores, since we do not have a control group to compare it to. However, this data supports the analysis of the ASA ANCOVA test in Section 5.2.2.2, which *did* have a control group, and found that the Genost treatment was responsible for the improvement in scores.

5.3.3. Feedback Analysis

Two sources of feedback were collected from the independent group during the FSE100 test, as described in Section 4.2.4.2.

5.3.3.1. Software Feedback (Likert Scales)

All students in the Genost group filled out a feedback form with ten Likert scales, with three questions for the categories of ease of use, educational efficacy and enjoyment of the Genost software. Additionally, one question was dedicated to rating the overall

experience of using Genost. The Likert scales measured on a scale of 1 to 10, with 1 being the lowest score and 10 being the highest. The feedback collected was averaged and the results are as follows:

Table 22

The Likert scale averages from the Genost software feedback form, FSE100 test

Scale	Average Score
Ease of Use (Q1, Q5, Q8)	7.474
Educational Efficacy (Q2, Q4, Q7)	7.596
Enjoyment (Q3, Q6, Q9)	6.123
Overall Experience (Q10)	7.037

Once again, we can see that these numbers are on the high end of scale. Interestingly, they are somewhat lower than the ASA test. Possible reasons for this may include less time spent with the software (12 hours for the FSE100 test vs. 16 hours for the ASA tests) or age differences between the two groups (high-school aged vs. college aged.)

5.3.3.2. Curriculum Feedback (Free Response)

The second sources of feedback collected from the Genost group was a series of four free response questions focusing on the Genost curriculum and our presentation. Students were asked to list three things they liked, three things they disliked, three things they would change, and any other comments they had, and were specifically instructed to limit their responses to the Genost curriculum and our presentation. We examined these responses and, for the first three free response questions (liked, disliked, and would change) we tabulated common themes in the responses. From this free response feedback,

we can examine common themes in what the students liked, found easy, found educational, as well as what they did not like, found difficult, or did not understand.

We will examine this free response data over the next three subsections.

5.3.3.2.1. Free Response Data – What Students Liked

Table 23 shows common themes that the students identified in their free response feedback for the question "Please write three or more things that you liked about the Genost curriculum and my presentation".

Similar to the ASA free response feedback, the two most liked items were related to the content of the curriculum, and its presentation. The most frequent item that students noted they liked was learning the fundamental concepts of programming in a general manner. Other highly rated items include the example-based nature of the curriculum, the frequent practice and repetition of concepts, and the general education of computational thinking. Once again, these are exactly the things that we aimed for when developing Genost.

Table 23

Free response tabulation for "Liked" question, FSE100 test

# Responses / % of Total	**Theme**
8 / 13.1%	Genost taught the fundamental concepts in a general – to – concrete way
8 / 13.1%	High quality / clarity of presentation on lecture slides
6 / 9.8%	Used examples to teach in a concrete – to – general manner.
6 / 9.8%	High quality / clarity of presentation by instructor
5 / 8.2%	Enjoyed the repetition and practice
4 / 6.6%	Related to the software design
3 / 4.9%	Taught computational thinking
3 / 4.9%	Easy to learn
3 / 4.9%	Related to Class Section (Small class size, flexible etc.)
3 / 4.9%	Had fun
3 / 4.9%	Liked quality of the curriculum
2 / 3.3%	Written response was unclear
2 / 3.3%	Felt lessons were well ordered
1 / 1.6%	Liked receiving extra credit for participating
1 / 1.6%	Liked similarity to formal programming languages
1 / 1.6%	Liked mazes
1 / 1.6%	Liked video game theme
1 / 1.6%	Liked drag-and-drop programming interface

5.3.3.2.2. Free Response Data – What Students Disliked

Table 24 shows common themes that the students identified in their free response feedback for the question “Please write three or more things that you disliked about the Genost curriculum and my presentation”.

Table 24

Free response tabulation for "Disliked" question, FSE100 Test

# Responses / % of Total	**Theme**
6 / 12.5%	Class was taught too fast
4 / 8.3%	Certain worksheets were too hard
3 / 6.3%	Curriculum was repetitive
3 / 6.3%	Review questions were too easy / repetitive / unnecessary
3 / 6.3%	Late lessons were too hard
2 / 4.2%	Lack of real life language involvement
2 / 4.2%	Early lessons were too easy
2 / 4.2%	Felt certain worksheets were unnecessary
2 / 4.2%	Felt certain concepts were missing concepts (variables, etc.)
2 / 4.2%	Simulated robot moved slowly
2 / 4.2%	Poor design of the lecture slides
2 / 4.2%	Worksheet goals were unclear
2 / 4.2%	Class featured too many worksheets
1 / 2.1%	Class was taught too slow
1 / 2.1%	Not enough time for exercises
1 / 2.1%	Student didn't understand something
1 / 2.1%	Disliked maze focus
1 / 2.1%	Found early exercises (i.e. exercises without the best tools available) too hard
1 / 2.1%	Focused too much on learning Genost, and not general ideas
1 / 2.1%	Software crashed often
1 / 2.1%	Did not have fun
1 / 2.1%	Poor lesson design led to lack of student participation
1 / 2.1%	Instructor did not review takehome assignments in class
1 / 2.1%	Class ran for too long
1 / 2.1%	Genost did not run well on all computers
1 / 2.1%	Related to class section (time of day, classroom management, etc.)

As with the ASA test, the most common complaint has to do with the speed at which the class was taught, due to the limited time available. However, many of the other common complaints have to do with the curriculum itself. Two common complaints state that parts

of the curriculum were too difficult. Another common dislike is the repetition of the review questions at the end of each lecture. Finally, as with the ASA test, some students did not like the lack of formal language integration into the curriculum.

This feedback indicates that the curriculum may be difficult for some students to learn. As suggested in Section 5.3.3.1, part of the reason for this difficulty may be the short amount of time involved with the FSE100 course, though we do not suggest that this accounts for all of the difficulty students experienced.

5.3.3.2.3. Free Response Data – What Students Would Change

Table 25 shows common themes that the students identified in their free response feedback for the question "Please write three or more things that you would change about the Genost curriculum or my presentation".

Table 25

Free response tabulation for "Would Change" question, FSE100 Test

# Responses / % of Total	**Theme**
9 / 18.4%	More Time (Not enough time to go over everything in class, taught too fast, etc.)
4 / 8.2%	Fewer Worksheets
4 / 8.2%	Related to class section / study design (class time, pretest, posttest, number of students, etc.)
2 / 4.1%	Change presentations - improve design, make easier to read
2 / 4.1%	Change presentations - improve review questions
2 / 4.1%	Put content online / make it accessible
2 / 4.1%	Engage class more
2 / 4.1%	Teach more concepts
2 / 4.1%	Improve pace - faster on easy stuff, slower on hard stuff
2 / 4.1%	Improve worksheet quality - Add tips, hints
2 / 4.1%	Make class more concise
1 / 2%	Unclear
1 / 2%	Related to software
1 / 2%	Link to Real Code
1 / 2%	More Explanation of a Subject
1 / 2%	Should synchronize Genost curriculum with another freshman class
1 / 2%	Change presentation – be more concise, less redundant
1 / 2%	Change presentation – give more examples
1 / 2%	Improve worksheet quality – focus more on key points
1 / 2%	Make class more difficult
1 / 2%	Focus less on Genost and more on concepts
1 / 2%	Rely less on Powerpoint slides when lecturing
1 / 2%	Improve the maze design and theme
1 / 2%	Go over takehome lessons in class
1 / 2%	Add practice tests to curriculum
1 / 2%	Shorten worksheet length
1 / 2%	Teach more on algorithm design

Once again we see that the overwhelming suggestion for change is to increase the length of the course. The second most common suggestion is a reduction in the number of

worksheets. Interestingly, the third most commonly suggested changes had to do with the particulars of the class design – usually this had to do with the class time, which was held relatively late in the day at 7:00PM.

To conclude our review of the free response feedback, we note that the most common complaints seem to be that not enough time was spent in the course; at the very least, this seems to indicate that students did not dislike the Genost course, and wanted to spend more time learning.

5.4. POSSIBLE DATA WEAKNESSES

Our analyses above conclude that Genost did increase the computational thinking skills of the participating students. We have argued that we can conclude from this that Genost does effectively teach the computational thinking skills we have designed it to teach. In this section we will note possible data weaknesses to keep in mind when drawing this conclusion.

5.4.1. Possible Data Weaknesses in the ASA Test

The ASA test had relatively small sample sizes, with 20 students in the independent group and 17 students in the con troll group. This small sample size limits our ability to extrapolate the results of the test to larger populations.

Another possible weakness of the ASA data has to do with the nature of student recruitment. Because we recruited by advertising the Genost class to interested students,

instead of selecting participants from a pool, some self-selection may have occurred in the independent group. Further selection bias may have occurred as students dropped out during the course, and were therefore removed from the independent group. The control group is not likely to have experienced similar self-selection since they were recruited from a single class, and did not join the experiment based on an interest in learning to program.

5.4.2. Possible Data Weaknesses in the FSE100 Test

The FSE100 test, as we have argued above, was a low power test, at least in regards to the grades. This severely limits our ability to draw any conclusions from it.

The FSE100 test is not likely to have experienced selection bias to the same extent as the ASA test may have. As discussed in Section 4.3.1.1, we received a large amount of volunteers for the Genost group, and narrowed our selection from 100 to 30. At least half of the students eliminated were done so randomly. For this reason, initial selection is not likely to suffer selection bias in student selection. However, as with the ASA test, some selection bias may have occurred due to students dropping the course over its run.

Finally, we note again that the FSE100 grade data was unexpectedly skewed highly towards the high range of the grade distribution. Almost every student that we collected data from received at least an 80%. This may be part of the reason for the experiment's unexpectedly low power.

6. CONCLUSION

We will conclude this thesis by summarizing the results of our research and our review, and by performing a final analysis on the information gathered. We will summarize the results of our own tests on the Genost system, and discuss limitations on these results. Finally, we will complete the thesis with a discussion of future improvements that may be made to the Genost system.

6.1. RESEARCH SUMMARY

We began this thesis by considering the oft-reported STEM crisis. While the nature and magnitude of this impending crisis is disputed depending on the source, we found that all sources agree on the importance of teaching computational thinking. Most sources believe that computational thinking is an important (perhaps even crucial) skill for the modern world, and that it should be taught not just to computer science students, but to all students, regardless of their major.

In response to this we performed an investigation into the definition of computational thinking and identified four major components of computational thinking that ought to be taught. These are: the ability to read and understand algorithms, the ability to engage in abstraction, the ability to decompose a problem into solvable processes, and the ability to evaluate the quality of a solution. In Section 2.1 we performed a deeper look at each of these subjects and identified several educational goals for teaching each one.

Further research led to the conclusion that computational thinking ought to be taught in a student's "introductory" computer science course. A review of the literature led us to conclude that computational thinking and programming are two separate skills, and that the former ought to be taught prior to, and independently of, the latter. In this way we established that introductory computer science education ought to consist solely of computational thinking instruction.

We performed an investigation into existing introductory computer science education in the United States. This investigation found that neither high school education nor college education adequately teaches students computational thinking. In both high school and college, existing introductory courses focus on teaching the syntax of a formal programming language, and while in some cases computational thinking skills are taught as well, they are rarely taught explicitly, and are almost always taught in the context of programming in whatever particular language has been taught. After performing this review, we noted that existing introductory computer science education produces very poor results: these classes have high failure rates, high attrition rates, and the students who pass these classes are often unable to effectively design and write programs. We argue that the lack of computational thinking education is, at least in part, responsible for these poor results.

We are not the only ones to come to this conclusion, and many "newer" educational systems have been created to attempt to teach computational thinking. We performed a review of eight of these newer educational systems. As part of this review, the good and

bad qualities of these systems were considered, and from each system we drew a series of lessons and takeaways. The goal of this review was to determine what qualities are good for an introductory computer science education system focused on computational thinking, and what qualities are problematic.

We found four major qualities that an "ideal" introductory computer science educational system should include, which are are: drag and drop style programming, virtual world integration, robot integration, and an official curriculum developed alongside the software. Other particular desirable features that were identified include a well-designed drag and drop language that abstracts actions to a high level, and is visually designed to indicate how the blocks can be combined; a robotic integration that allows the robot code to execute on a local computer instead of on the robot itself, and also allows multiple models of robots to be used; general customizability of the system, and integration with social media. Particular features that systems should avoid include shifting focus away from computational thinking education onto other items like syntax or mechanical engineering, unduly high technical complexity, oversimplification, high expense, and a curriculum that focuses more on play, competition or storytelling than explicit computational thinking instruction.

6.2. GENOST TEST SUMMARY

Using the research findings described above, we designed and built Genost, an educational system focused on teaching introductory computational thinking skills. Genost's design and technology is described extensively in Section 3.

In Section 3.8 we compared Genost to the other eight systems that were reviewed in Section 2.4 and found, based on our takeaways from that review, that Genost implements more desirable features than any other system, and avoids all problems that the reviewed systems have. A table showing this comparison can be found in Appendix G.

In order to test Genost's effectiveness in teaching computational thinking, we performed two major tests of the systems, which involved teaching the Genost curriculum to two separate groups of students and evaluating these students' abilities in computational thinking. The designs of these tests are described in Section 4, and the results are described and analyzed in Section 5.

As stated in Section 5, we found in both tests that students participating in a Genost class saw a significant increase in their computational thinking skills and abilities. It is notable that the students in these classes went to having almost no computational thinking skills, attaining averages on the computational thinking pretest in the range of 0 to 10%, and often not even attempting most questions, to scoring in the range of 50% to 60%, with most students attempting most of the questions. The ASA test, due to its design, allows us to conclude that Genost was responsible for this increase. The FSE100 test does not allow us to draw this same conclusion due to the lack of a control group for the pretest / posttest, but the extremely large size of the increase in computational thinking skills, and the results of the ASA test, makes us believe that Genost was indeed responsible for the increase. In both tests the effect size of Genost on computational thinking abilities was classified as "large".

In addition to measuring computational thinking ability, we also collected grade data from the FSE100 students in our FSE100 test. We found that students participating in the Genost exercise had the highest raw grade average, followed by the students participating in a Python exercise, and finally by the remainder of the class. However, the differences between these grade means were not large enough to be significant, and so we cannot conclude that Genost raised student grades in FSE100.

The final data set that we reviewed was student feedback, provided in two forms: Likert data for the Genost software, and free-response data for the Genost curriculum and presentation. All students participating in the Genost groups provided us with this feedback, and we believe it is an accurate representation of student thoughts and feeling on the system.

The Likert responses measure student thoughts on the ease of use, fun and educational efficacy of the software. These responses were on the high end of the scale, measuring roughly between 6 and 8 for all categories. Ease of use was rated between 7.5 and 7.8 on average; educational efficacy was rated 7.6 to 8.1 on average; and enjoyment was rated 6.1 to 7.1 on average. The overall Genost software experience was given a rating of 7 – 7.7. These scores are on the high end of the scale, and lead us to conclude that students found the Genost software easy to work with, that they enjoyed working with it, and that they believed that they learned something from their interaction with the software.

The free response feedback, which measures student thoughts about the Genost curriculum and our presentation of it, was also very positive. The positive feedback that students provided indicated that students found the system easy to understand and educationally rigorous. This feedback also indicates that the students found our presentation of the curriculum clear and engaging. Many of the educational benefits that we argued that Genost provides, such as a general presentation of the fundamental programming concepts, were identified by the students as something they liked. This feedback makes us somewhat confident that we "hit the mark" with Genost.

An overwhelmingly high percentage of the negative feedback and the feedback on what the students would change focuses on the relative short amount of time students spent with Genost, and the large amount of work that we attempted to fit into that time. 44.2% of the student complaints in the ASA test had specifically to do with the short amount of time in the Genost class; lack of time was also the highest percentage complaint for the FSE100 test. The only major complaint that did not have to do with the short amount of time, or with other aspects of the class section (such as the class size or time of day at which it was taught) was a complaint by four FSE100 students that the worksheets were too hard to understand. A similar pattern can be seen in the feedback on what students would like changed – the most common suggestion for the FSE100 test, and the second highest suggestion for the ASA test (second only to a request to fix bugs with the software) was to add more time to the class. These complaints and suggestions indicate to us that none of Genost's deficiencies are strong or glaring enough to seriously bother the students; and indeed, the fact that students wish to spend *more time* with the Genost

system further reinforces our conclusion that students like the system, and find it educational and enjoyable.

The results of our comparison between Genost and the eight "newer" systems, and the results of our test, lead us to conclude that the Genost system is effective in teaching computational thinking, is easy and fun for students to use, and is in many ways superior to the other systems currently on the market.

6.3. THESIS LIMITATIONS

In Section 5.4.2 we noted some of the possible weaknesses of our results analysis. These include violations of some assumptions for both the ANOVA test in the ASA test (used to analyze the computational thinking pretest / posttest results) and the paired-samples t-test in the FSE100 test (also used to analyze the computational thinking pretest / posttest results). Despite these assumption violations, we have proceeded with these tests and believe that the results we have retrieved from them are valid. We are confident that this is acceptable due to the known robustness of these tests to violations, but also to the fact that the significance and effect size of our results are very large, meaning that Type I errors are less likely.

We also discussed possible biases in our tests in Section 5.4.2. Students participating in the FSE100 test were chosen pseudo-randomly according to the process described in Section 4.3.1.1, and therefore selection bias is not likely here. The ASA students were chosen on a first-come first-serve basis, however, and so the group may have been biased

towards students interested in learning programming. Furthermore, both independent groups had students withdraw, and therefore some bias may have been introduced towards more hardworking or persistent students. These sources of bias are virtually unavoidable in tests of this nature since participation is and must be voluntary. Despite this, we again believe that the very strong significance and effect size indicates that, whether selection bias was in fact present or not, it is unlikely to be high enough to invalidate our conclusions.

In retrospect we are able to see that FSE100 was not the ideal class to use for our test – as previously described, the scores for all groups were skewed strongly towards the high end of the scale, instead of providing a strong normal distribution. This feature and others are likely what led to the FSE100 grade data analysis being underpowered. This is ultimately not a true limitation since we cannot draw any conclusions from this data.

Aside from the limitations on the data we collected and analyzed discussed above, more general limitations may be identified when considering our literary review. For example, our review of the introductory computer science education classes in the United States was conducted using the best information available to us, which were online curricula. These curricula, while informative, do not and cannot give a whole picture of the class; and often the true value of the class comes from the teacher teaching it. It is possible that amount of computational thinking education in existing introductory computer science is higher than we have identified (it could also be lower).

A similar limitation should be noted for our review of the eight "newer" systems. Many of these systems are quite new and literature on them is limited. Furthermore, literature on how well these systems teach introductory computational thinking skills is a small percentage of the total amount of research for these systems. We believe that the general conclusions we have drawn in regards to this system are valid, but could certainly be refined. Similarly, our comparison between these systems and Genost are rough comparisons, and further research and explicit tests to compare these systems are certainly warranted.

6.4. FUTURE IMPROVEMENTS

We plan to add many features to Genost in order to improve its design and ability to teach computational thinking. We will conclude this thesis by listing some of these planned improvements.

Our immediate goals for the current version of Genost is to fix the bugs in the GUI and simulator, improve the visual design of the blocks to better indicate how they fit together, and improve the general usability of the system. We also hope to complete development on the robot and integrate it into the curriculum. Finally, we hope to rewrite some of the curriculum lessons to aid in clarity and educational efficacy.

Due to the feedback received as part of this test, we plan to either split future Genost courses up into multiple classes, or increase the length of a single course, in order to give students more time for individual lessons.

Our long-term goals for Genost involve a fairly complete rebuild of the system. We plan to rebuild the Genost system in JavaScript in order to take advantage of the language's native functionality in the browser. This will allow Genost to run cleanly in the browser with absolutely no required software or plugins. We also plan to add many new features to the mazes, in order to increase the richness of the different lessons. We hope to create multiple models of robots to provide customers greater flexibility in selecting a robot that meets their needs, and we also hope to implement a Genost interpreter on existing robot systems, such as Lego Mindstorms. Along with these features and changes, we also hope to refactor the system to allow the robot algorithm to be executed on a local computer instead of the robot itself. We plan on adding a debugger to the GUI, and in general redesign the software to be far more usable.

In addition to a redesign and reimplementation of the Genost software, we also plan to redesign the curriculum, and move it away from its current lecture-based focus to a collaborative development style of learning. We plan to add a larger focus on debugging to the curriculum, as well as introducing constant testing and refinement of student knowledge through online quizzes. We plan to design our curriculum such that all resources and materials may be accessed online.

Future implementations of Genost may include an online course, automated or facilitated, and possibly a video game.

This thesis has described the need for a system to effectively teach computational thinking, and we believe that Genost fills this need. This belief is informed both by a description and justification of Genost's design, and the results of the tests we have run that show that Genost results in increased computational thinking ability. Informed by this success, we hope to utilize the Genost system in the future to perform the vitally important task of teaching students computational thinking.

REFERENCES

2015 FRC Control System. (2015). (FIRST Robotics Competition) Retrieved 2 25, 2015, from WPILib: https://wpilib.screenstepslive.com/s/4485

About EV3. (2015). (The LEGO Group) Retrieved 2 25, 2015, from Lego Mindstorms: http://www.lego.com/en-us/mindstorms/about-ev3

About Scratch. (n.d.). (MIT) Retrieved February 5, 2015, from Scratch: http://scratch.mit.edu/about/

About the Alice 3 Instructional Materials. (2015). (Carnegie Mellon) Retrieved February 25, 2015, from Alice: http://www.alice.org/3.1/index.html

All About EV3 - Curriculum & Tools. (2014). (The LEGO Group) Retrieved February 25, 2015, from Lego Education: https://education.lego.com/en-us/lesi/middle-school/mindstorms-education-ev3/all-about-ev3/curriculum

Allen, W. B.-S. (n.d.). *Computational thinking for Youth.* Retrieved March 4, 2015, from stelar: STEM Learning and Research Center: http://stelar.edc.org/sites/stelar.edc.org/files/Computational_Thinking_paper.pdf

An Introductory Computing Curriculum Using Scratch. (n.d.). (Harvard) Retrieved February 25, 2015, from Creative Computing: http://scratched.gse.harvard.edu/guide/

AP Computer Science Principles. (2015). (The College Board) Retrieved February 27, 2015, from Advances in AP: https://advancesinap.collegeboard.org/stem/computer-science-principles

Atkinson, R. K. (2000). Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research, 70*(2), pp. 181-214.

Ausubel, D. P. (1968). *Educational psychology: A cognitive view.* New York: Holt, Rinehart and Winston.

Barry Fagin, L. M. (2003). Measuring the Effectiveness of Robots in Teaching Computer Science. *Proceedings of the 34th SIGCSE technical symposium on Computer science education (SIGCSE '03)* (pp. 307-311). New York: ACM.

Beaubouef, T. J. (2005). Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations. *ACM SIGCSE Bulletin, 37*(2), pp. 103-106.

Bennedsen, J. M. (2007). Failure Rates in Introductory Programming. *ACM SIGCSE Bulletin, 39*(2), pp. 32-36.

Blank, D. J. (2012). Calico: A Multi-Programming-Language, Multi-Context Framework Designed for Computer Science Education. *Proceedings of the 43rd ACM technical symposium on Computer Science Education (SIGCSE '12)* (pp. 63-68). New York: ACM.

Blum, L. T. (2007). CS4HS An Outreach Program for High School CS Teachers. *Proceedings of the 38th SIGCSE technical symposium on Computer science education (SIGCSE '07)* (pp. 19-23). New York: ACM.

Bornat, R. (2014). *Camels and humps: a retraction.* London: School of Science and Technology, Middlesex University.

Buckhaults, C. (2009). Increasing Computer Science Participation in the FIRST Robotics Competition with Robot Simulation. *Proceedings of the 47th Annual Southeast Regional Conference (ACM-SE 47).* New York: ACM.

Build a Robot. (2015). (The LEGO Group) Retrieved February 25, 2015, from Lego Mindstorms: http://www.lego.com/en-us/mindstorms/build-a-robot

Bundy, A. (2007). Computational Thinking is Pervasive. *Journal of Scientific and Practical Computing, 1*(2), 67-69.

BurningGlass. (2014, February). *Real-Time Insight into the Market for Entry-Level STEM Jobs.* Retrieved February 27, 2015, from Burning Glass: Careers in Focus: http://www.burning-glass.com/media/3326/Real-Time%20Insight%20Into%20The%20Market%20For%20Entry-Level%20STEM%20Jobs.pdf

Buss, S. R. (2001). The Prospects for Mathematical Logic in the Twenty-First Century. *The Bulletin of Symbolic Logic, 7*(2).

Caitlin Kelleher, R. P. (2007). Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)* (pp. 1455-1464). New York: ACM.

Calico. (2014, August 18). (Institute for Personal Robotics in Education) Retrieved February 25, 2015, from IPRE Wiki: http://calicoproject.org/

Carey, K. (2010, November 7). *Decoding the Value of Computer Science*. Retrieved March 19, 2015, from The Chronicle of Higher Education: https://chronicle.com/article/Decoding-the-Value-of-Computer/125266/

Carnevale, A. P. (2011). *STEM: Science, Technology, Engineering, Math.* Washington DC: Georgetown University Center on Education and the Workforce.

Carnevale, A. P. (2011). *STEM: Science, Technology, Engineering, Mathematics.* Washington, DC: Georgetown University Center on Education and the Workforce.

Carter, L. (2006). Why Students with an Apparent Aptitude for Computer Science don't choose to Major in Computer Science. *Proceedings of the 37th SIGCSE technical symposium on Computer science education (SIGCSE '06)* (pp. 27-31). New York: ACM.

CBI. (2013, June 20). *Changing the Pace: CBI / Pearson Education and Skills Survey 2013.* Retrieved February 27, 2015, from CBI: http://www.cbi.org.uk/media/2119176/education_and_skills_survey_2013.pdf

Chamillard, A. K. (2000). Evaluating Programming Ability in an Introductory Computer Science Course. *ACM SIGCSE Bulletin, 32*(1), pp. 212-216.

Charette, R. N. (n.d.). *The STEM Crisis Is a Myth.* Retrieved from IEEE Spectrum: http://spectrum.ieee.org/at-work/education/the-stem-crisis-is-a-myth

Chen, Y. H. (2013). Internet of intelligent things and robot as a service. *Simulation Modelling Practice and Theory, 34*, 159-171.

Chen, Y. Z.-A. (2010). Robot as a Service in Cloud Computing. *2010 Fifth IEEE International Symposium on Service Oriented System Engineering (SOSE)* (pp. 151-158). Nanjing: IEEE.

Committee on the Engineer of 2020, Phase II, Committee on Engineering Education, National Academy of Engineering. (2005). *Educating the Engineer of 2020: Adapting Engineering Education to the New Century.* Retrieved February 27, 2015, from The National Academies Press: http://www.nap.edu/catalog/11338/educating-the-engineer-of-2020-adapting-engineering-education-to-the

Computer science courses get highest drop outs - study. (2010, October 28). Retrieved February 27, 2015, from Silicon Republic: http://www.siliconrepublic.com/innovation/item/18532-computer-science-courses-ge

Cowen, T. (2013). *Average is Over: Powering America Beyond the Age of the Great Stagnation.* New York: Dutton.

Creating DSS Service Projects. (2012). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb483009.aspx

CSTA Curriculum Improvement Task Force. (2006). *The New Educational Imperative - Improving High School Computer Science Education.* New York: ACM. Retrieved from http://csta.acm.org/Communications/sub/DocsPresentationFiles/White_Paper07_06.pdf

Cuny, J. (2011, June). *Transforming Computer Science Education in High Schools.* Retrieved February 27, 2015, from Exploring Computer Science: http://www.exploringcs.org/wp-content/uploads/2011/11/IEEE-Transforming-Computer-Science-Education-in-High-Schools.pdf

David Barr, J. H. (2011, March). Computational Thinking: A Digital Age Skill for Everyone. *Learning & Leading with Technology, 38*(6), 20-23.

David J. Malan, H. H. (2007). Scratch for Budding Computer Scientists. *Proceedings of the 38th SIGCSE technical symposium on Computer science education (SIGCSE '07)* (pp. 223-227). New York: ACM.

Deepak Kumar, D. B. (2008). Engaging Computing Students with AI and Robotics. *AAAI Spring Symposium: Using AI to Motivate Greater Participation in Computer Science*, (pp. 55-60).

Dehnadi, S. R. (2006, February 22). *The camel has two humps (working title).* Retrieved February 27, 2015, from Science and Technology WebServer: www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf

Delden, S. v. (2008). Effective Integration of Autonomous Robots Into an Introductory Computer Science Course: A Case Study. *Consortium for Computing Sciences in Colleges, 23*(4).

Diana Franklin, P. C.-T. (2013). Assessment of Computer Science Learning in a Scratch-Based Outreach Program. *Proceeding of the 44th ACM technical symposium on Computer science education (SIGCSE '13)* (pp. 371-376). New York: ACM.

Drew, C. (n.d.). *Why Science Majors Change their Minds (It's Just So Darn Hard).* Retrieved from The New York Times: http://www.nytimes.com/2011/11/06/education/edlife/why-science-majors-change-their-mind-its-just-so-darn-hard.html

FIRST. (2015). *FIRST LEGO League*. (FIRST) Retrieved February 27, 2015, from FIRST LEGO League: http://www.firstlegoleague.org/

For Parents. (n.d.). (MIT Media Lab) Retrieved February 27, 2015, from Scratch: http://scratch.mit.edu/parents/

Get Started (LEGO MINDSTORMS). (n.d.). (Carnegie Mellon) Retrieved February 27, 2015, from Carnegie Mellon Robotics Academy: http://www.education.rec.ri.cmu.edu/content/lego/start/

Gouws, L. (2013). Computational thinking in educational activities: an evaluation of the educational game light-bot. *Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITiCSE '13)* (pp. 10-15). New York: ACM.

Gross, S. M. (2014). Fostering Computational Thinking in Engineering Education. *Global Engineering Education Conference (EDUCON)* (pp. 450-459). Istanbul: IEEE.

Guzdial, M. (2004). Programming Environments for Novices. In S. Fincher, *Computer Science Education Research.* Taylor & Francis.

Guzdial, M. (2011). Education: From Science to Engineering. *Communications of the ACM, 54*(2), 37.

Harms, D. (2013). A Preliminary Analysis of the Effectiveness of Myro / Java in Computer Science 1. *CompSysTech'13 Local Proceedings* (pp. 42-46). Ruse: University of Ruse.

Hofstadter, D. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid.* New York: Basic Books.

House of Lords Select Committee on Science and Technology. (2012). *Higher Education in Science, technology, Engineering and Mathematics (STEM) subjects, 2nd report of session 2012-2013.* House of Lords. London: The Stationary Office Limited. Retrieved from http://www.publications.parliament.uk/pa/ld201213/ldselect/ldsctech/37/37.pdf

Hu, C. (2011). Computational Thinking - What it Might Mean and What we Might Do About It. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education (ITiCSE '11)* (pp. 223-227). New York: ACM.

Ian Utting, S. C. (2010). Alice, Greenfoot and Scratch - A Discussion. *ACM Transactions on Computing Education, 10*(4), 1-11.

Institute for Personal Robots in Education. (2008, February 27). *Institute for Personal Robots in Education*. (Institute for Personal Robots in Education) Retrieved

February 27, 2015, from Institute for Personal Robots in Education: http://www.roboteducation.org/

Jackie O'Kelly, J. P. (2006). RoboCode & problem-based learning: a non-prescriptive approach to teaching programming. *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE '06)* (pp. 217-221). New York: ACM.

James J. Lu, G. H. (2009). Thinking about Computational Thinking. *Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09)* (pp. 260-264). New York: ACM.

Jobs, S. (1995). https://www.youtube.com/watch?v=IY7EsTnUSxY.

K-12 Computer Science Education: Unlocking the Future of Students. (2012, August). ACM. Retrieved from http://www.acm.org/public-policy/2012_CS_Slides_Aug.pptx

Karin Johnsgard, J. M. (2008). Using Alice in Overview Courses to Improve Success Rates in Programming 1. *IEEE 21st Conference on Software Engineering Education and Training, 2008 (fCSEET '08)* (pp. 129-136). Charleston, SC: IEEE.

Kay, A. (2008, August 24). *Alan Kay on 'The Camel has Two Humps'*. Retrieved February 27, 2015, from SecretGeek.net: http://secretgeek.net/camel_kay

Kelleher, C. (2007). Retrieved February 27, 2015, from Storytelling Alice: http://www.alice.org/kelleher/storytelling/index.html

Knuth. (1997). *The Art of Computer Programming Second Edition, Volume 1 Fundamental Algorithms.* Reading, Massachusetts: Addison-Wesley.

Knuth, D. (1985, March). Algorithmic Thinking and Mathematical Thinking. *The American Mathematical Monthly, 92*(3), pp. 170-181.

Larsen, F. N. (2013, February 27). *ReadMe for Robocode.* Retrieved February 27, 2015, from Robocode: http://robocode.sourceforge.net/docs/ReadMe.html

Learn to Program. (2015). (The LEGO Group) Retrieved February 27, 2015, from Lego Mindstorms: http://www.lego.com/en-us/mindstorms/learn-to-program

Learning with Scratch. (n.d.). Retrieved February 27, 2015, from Scratch: https://llk.media.mit.edu/scratch/Learning-with-Scratch.pdf

Lego MINDSTORMS Education EV3. (2014). (The LEGO Group) Retrieved February 27, 2015, from Lego Education: https://education.lego.com/en-us/lesi/middle-school/mindstorms-education-ev3

Lego Mindstorms NXT. (2015). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb905443.aspx

Lix, L. M. (1996). Consequences of assumption violations revisited: A quantitative review of alternatives to the one-way analysis of variance F test. *Review of educational research, 66*(4), 579-619.

Long, J. (2007). Just for Fun: Using Programming Games in Software Programming, Training and Education - A Field Study of IBm Robocode Community. *Journal of Information Technology Education: Research, 6*(1), 279-290.

Maja J Mataric, N. K.-S. (2007). Materials for Enabling Hands-On Robotics and STEM Education. *AAAI Spring Symposium: Semantic Scientific Knowledge Integration* (pp. 99-102). AAAI.

Maloney, J. (2008). Programming by Choice: Urban Youth Learning Programming with Scratch. *ACM SIGCSE Bulletin, 40*(1), pp. 367-371.

McGill, M. M. (2012). Learning to Program with Personal Robots Influences on Student Motivation. *ACM Transactions on Computing Education, 12*(1), 1-32.

Michael McCracken, V. A.-D. (2001, December). A Multi-National Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *ACM SIGCSE Bulletin, 33*(4), pp. 125-180.

Microsoft. (n.d.). *A National Talent Strategy: Ideas for Securing US Competativeness and Economic Growth.* Retrieved February 27, 2015, from News Center: http://news.microsoft.com/download/presskits/citizenship/msnts.pdf

Microsoft Robotics - Overview. (2012). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb483024.aspx

Mitchel Resnick, J. M.-H. (2009, November). Scratch: Programming for All. *Communications of the ACM, 52*(11), pp. 60-67.

Mohtadi, C. M. (2013). Why Integrate Computational Thinking into a 21st Century Engineering Curriculum. *41st SEFI Conference.* Leuven, Belgium.

Moursund, D. (2013, October 12). *Computational Thinking.* Retrieved February 27, 2015, from IAE-Pedia: http://iae-pedia.org/Computational_Thinking

Myro Development. (2009, August 4). (Institute for Personal Robotics in Education) Retrieved February 27, 2015, from Calico Project: http://calicoproject.org/Myro_Development

NSF. (2014, December 8). *College Board launches new AP Computer Science Principles course*. (National Science Foundation) Retrieved February 27, 2015, from National Science Foundation: http://www.nsf.gov/news/news_summ.jsp?cntn_id=133571

NXT-G. (2014). (Tufts University Center for Engineering Education and Outreach) Retrieved February 27, 2015, from LEGO engineering: http://www.legoengineering.com/program/nxt-g/

Object-Oriented Programming. (2014, November 19). Retrieved February 27, 2015, from Scratch Wiki: http://wiki.scratch.mit.edu/wiki/Object-Oriented_Programming

Ormrod, J. E. (2012). *Human Learning* (6th ed.). Upper Saddle River, New Jersey: Pearson Education, Inc.

Orni Meerbaum-Salant, M. A.-A. (2011). Habits of Programming in Scratch. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education (ITiCSE '11)* (pp. 168-172). New York: ACM.

Orni Meerbaum-Salant, M. A.-A. (2013). Learning Computer Science Concepts with Scratch. *Computer Science Education, 23*(3), 239-264.

Papert, S. (1993). *Mindstorms: Children, Computers, And Powerful Ideas.* Basic Books.

Partovi, H. (2014, June 19). *What % of STEM Should Be Computer Science?* (Code.org) Retrieved February 27, 2015, from Anybody Can Learn: http://codeorg.tumblr.com/post/89267280803/stem

Paul Mullins, D. W. (2009). Using Alice 2.0 as a First Language. *Journal of Computing Sciences in Colleges, 24*(3), 136-143.

Paul, J. (2012, June). Living in a Computing World - A Step Towards Making Knowledge of Computing Accessible to Every Student. *ACM Inroads, 3*(2), pp. 78-81.

President's Council of Advisors on Science and Technology. (2012, February). *Report to the President: Engage to Excel: Producing One Million Additional College Graduates With Degrees in Science, Technology, Engineering and Mathematics.* Retrieved March 4, 2015, from Whitehouse.gov: http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-engage-to-excel-final_2-25-12.pdf

Renkl, A. R. (2002). From Example Study to Problem Solving: Smooth Transitions Help Learning. *The Journal of Experimental Education, 70*(4), 293-315.

Resnick, M. (2007). All I Really Need to Know (About Creative Thinking) I Learned (By Studying How Children Learn) in Kindergarten. *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition (C&C '07)* (pp. 1-6). New York: ACM.

Robins, A. (2010, April 7). Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education, 20*(1), 37-71.

Rothwell, J. (2014, July). *Still Searching: Job Vacancies and STEM Skills.* Retrieved March 4, 2015, from Brookings: http://www.brookings.edu/~/media/research/files/reports/2014/07/stem/job%20vacancies%20and%20stem%20skills.pdf

Ryan Garlick, E. C. (2010). Using Alice in CS1: A quantitative experiment. *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education (ITiCSE '10)* (pp. 165-168). New York: ACM.

Sawilowsky, S. S. (1992, March). A more realistic look at the robustness and type II error properties of the t test to departures from population normality. *Psychological Bulletin, 111*(2), 352-360.

Scanlan, D. A. (2007, October). Programming the Eight-Core Propeller Chip. *Journal of Computing Sciences in Colleges, 23*(1), 162-168.

Scratch Curriculum Guide. (2014, August 7). (Harvard) Retrieved February 27, 2015, from ScratchED: http://scratched.gse.harvard.edu/resources/scratch-curriculum-guide

Selby, C. C. (2012). Promoting Computational Thinking with Programming. *Proceedings of the 7th Workshop in Primary and Secondary Computing Education (WiPSCE '12)* (pp. 74-77). New York: ACM.

Stark, E. (2013, January 29). Commentary: US Kids Need Computer Science Education. *USA Today*.

Supported Robots. (2015). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb905441.aspx

Testimonials from Alice users. (2015). (Carnegie Mellon) Retrieved February 27, 2015, from Alice: http://www.alice.org/index.php?page=testimonials

The FIRST Robotics Competition: CAREERS. (2014, August). Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/uploadedFiles/Robotics_Programs/FRC/FRC_Communications_Resource_Center/Flyers/FRC_CareersFNL.pdf

The FIRST Robotics Competition: EVALUATION. (2013, August). Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/uploadedFiles/Robotics_Programs/FRC/FRC_Communications_Resource_Center/Flyers/FRC_EvaluationFNL.pdf

The FIRST Robotics Competition: HOW IT WORKS. (2014, August). Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/uploadedFiles/Robotics_Programs/FRC/FRC_Communications_Resource_Center/Flyers/FRC_HowitworksFNL.pdf

The FIRST Robotics Competition: OVERVIEW. (2014, August). Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/uploadedFiles/Robotics_Programs/FRC/FRC_Communications_Resource_Center/Flyers/FRC_OverviewFNL.pdf

The FIRST Robotics Competition: SUCCESS. (2013, August). Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/uploadedFiles/Robotics_Programs/FRC/FRC_Communications_Resource_Center/Flyers/FRC_AcademicSuccessFNL.pdf

Thomas R. Flowers, K. A. (2002, May). Teaching Problem Solving, Computing and Information Technology with Robots. *Journal of Computing Sciences in Colleges, 17*(6), 45-55.

Thompson, A. (2007, September 21). *New Robotics Curriculum from Microsoft Robotics Studio*. (Microsoft) Retrieved February 27, 2015, from Microsoft Education Blogger: Teaching Computer Science: http://blogs.msdn.com/b/alfredth/archive/2007/09/21/new-robotics-curriculum-from-microsoft-robotics-studio.aspx

Tom Lauwers, I. N. (2009). CSbots: Design and Deployment of a Robot Designed for the CS1 Classroom. *Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09)* (pp. 428-432). New York: ACM.

Tucker Balch, J. S. (2008, April). Designing Personal Robots for Education: Hardware, Software and Curriculum. *Pervasive Computing, 7*(2), pp. 5-9.

Ursula Wolz, H. H. (2009). Starting with Scratch in CS1. *Proceedings of the 40th ACM technical symposium on Computer science education (SIGCSE '09)* (pp. 2-3). New York: ACM.

Valerie Barr, C. S. (2011, March). Bringing Computational Thinking to K12 - What is Involved and What ist he Role of the Computer Science Education Community? *ACM Inroads, 2*(1), pp. 48 - 54.

Visual Programming Language - Using Services. (2012). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/dd146310.aspx

VPL Introduction. (2012). (Microsoft) Retrieved February 27, 2015, from Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb483088.aspx

W.T. Tsai, Y. C. (2008). An Introductory Course on Service-Oriented Computing for High Schools. *Journal of Information Technology Education: Research, 7*(1).

Wagstaff, K. (2012, July 16). *Can we Fix Computer Science Education in America*. (Time, Inc) Retrieved March 4, 2015, from Time: http://techland.time.com/2012/07/16/can-we-fix-computer-science-education-in-america/

Wanda Dann, S. C. (2009, August). Education: Alice 3: Concrete to Abstract. *Communications of the ACM, 52*(8), pp. 27-29.

Watson, C. F. (2014). Failure Rates in Introductory Programming Revisited. *Proceedings of the 2014 conference on Innovation & technology in computer science education (ITiCSE '14)* (pp. 39-44). New York: ACM.

Welcome to the FIRST Robotics Competition. (n.d.). (FIRST) Retrieved February 27, 2015, from FIRST Robotics Competition: http://www.usfirst.org/roboticsprograms/frc

What is Alice? (2015). (Carnegie Mellon) Retrieved February 27, 2015, from Alice: http://www.alice.org/index.php?page=what_is_alice/what_is_alice

What Is Logo? (2011). (The Logo Foundation) Retrieved February 27, 2015, from Logo Foundation: http://el.media.mit.edu/logo-foundation/logo/index.html

Williams, L. E. (2002). In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education, 12*(3), 197-212.

Wilson, C. L. (2010). *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age.* ACM. Retrieved from http://runningonempty.acm.org/fullreport2.pdf

Wing, J. (2006). Computational Thinking and CS@CMU.

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM, 49*(3).

Wing, J. M. (2008, October 28). Computational Thinking and Thinking about Computing. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences , 366*(1881), 3717-3725.

Wing, J. M. (2010, November 17). *Computational Thinking: What and Why?* Retrieved March 4, 2015, from Center for Computational Thinking: http://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf

Yinong Chen, W.-T. T. (2014). *Service-Oriented Computing and Web Software Integration.* Dubuque, IA: Kendall Hunt.

APPENDIX A:

COLLEGE CURRICULA REVIEW NOTES

The following tables contain the overall rating and language used notes for the classes reviewed. Detailed commentary is available upon request.

College Curricula Review Notes for Overall Top 25 US Colleges
(As determined by US News, accessed 11/29/2014)

College	Course	Rating	Language Used	Date Accessed
Princeton University	COS109	1	HTML, Javascript	11/30/2014
	COS126	2	Java	11/30/2014
Harvard University	CS50	2	C, HTML, PHP, Javascript, Scratch	11/30/2014
	COMPSCI1	No syllabus obtainable		
	COMPSCI51	No syllabus obtainable		
Yale University	CPSC112	No syllabus obtainable		
	CPSC201	2	Racket	11/30/2014
Columbia University	COMSW1004	No syllabus obtainable		
Stanford University	CS105	1	HTML, CSS, PHP	11/30/2014
	CS106A	3	Karel Robot, Java	11/30/2014
	CS106B	2	C++	11/30/2014
University of Chicago	CMSC15100	1	Racket	12/16/2014
	CMSC15200	1	C	12/16/2014
MIT	6.01	2	Python	12/16/2014
	6.02	1	Python	12/16/2014
Duke University	CompSci101	2	Python	12/16/2014
University of Pennsylvania	CIS110	1	Java	12/16/2014
	CIS120	2	Java, OCAML	12/16/2014
California Institute of Technology	CS1	1	Python	12/16/2014

	CS2	No syllabus obtainable		
Dartmouth College	COSC1	2	Python	12/18/2014
	COSC10	2	Java	12/17/2014
John Hopkins University	EN.600.107	1	Java	12/17/2014
Northwestern University	EECS101	0	N/A	12/17/2014
	EECS111	3	C, Python	12/17/2014
Washington University in St. Louis	CSE131	2	Java	12/17/2014
	CSE132	2	Java	12/17/2014
Cornell University	CS1110 / CS1133	1	Python	12/18/2014
	CS1112 / CS1132	2	MATLAB	12/18/2014
Brown University	CSCI0150	1	Java	12/18/2014
	CSCI0170	2	Racket, OCAML, Java, Scala	12/18/2014
University of Notre Dame	CSE20211	2	C	12/18/2014
	CSE20212	1	C++	12/18/2014
Vanderbilt University	CS101	No syllabus obtainable		
	CS201	No syllabus obtainable		
Rice University	COMP140	3	Python	12/19/2014
	COMP160	1	Python	12/19/2014
University of California - Berkeley	CS61A	3	Python	12/19/2014
	CS61B	3	Java	12/19/2014
	CS10	5	Snap (Scratch)	12/19/2014
Emory University	CS170	2	Java	12/19/2014
	CS171	1	Java	12/19/2014
Georgetown University	COSC051	2	C++	12/20/2014

	COSC052	1	C++	12/20/2014
University of California - Los Angeles	COMSCI31	2	C++	12/20/2014
	COMSCI32	2	C++	12/23/2014
University of Virginia	CS1110 / CS1133	3	Java	12/20/2014
Carnegie Mellon University	15-112	1	Python	12/21/2014
	15-122	1	Python	12/21/2014
University of Southern California	CSCI103	2	C++	12/21/2014

College Curricula Review Notes for Top 25 US Colleges for Computer Science (As determined by US News, accessed 11/29/2014)

College	Course	Rating	Language	Date Accessed
MIT	6.01	2	Python	12/16/2014
	6.02	1	Python	12/16/2014
Harvard University	CS50	2	C, HTML, PHP, Javascript, Scratch	11/30/2014
	COMPSCI 1	No syllabus obtainable		
	COMPSCI 51	No syllabus obtainable		
Stanford University	CS105	1	HTML, CSS, PHP	11/30/2014
	CS106A	3	Karel Robot, Java	11/30/2014
	CS106B	2	C++	11/30/2014
University of California - Berkeley	CS61A	3	Python	12/19/2014
	CS61B	3	Java	12/19/2014
	CS10	5	Snap (Scratch)	12/19/2014
Princeton University	COS109	1	HTML, Javascript	11/30/2014
	COS126	2	Java	11/30/2014
University of Texas - Austin	CS312	2	Java	12/23/2014
University of California - San Diego	CSE8A	3	Java	12/23/2014
	CSE11	1	Java	12/23/2014
University of Southern California	CSCI103	2	C++	12/21/2014
Georgia Institute of Technology	CS1301	3	Python	12/23/2014
	CS1331	2	Java	12/23/2014
University of California - Los Angeles	COMSCI31	2	C++	12/20/2014

	COMSCI32	2	C++	12/23/2014
Carnegie Mellon University	15-112	1	Python	12/21/2014
	15-122	1	Python	12/21/2014
University of California - Irvine	I&C SCI 31	3	Python	12/23/2015
University of Illinois - Urbana-Champaign	CS125	2	Java	12/23/2015
University of Maryland - College Park	CMSC131	1	Java	12/23/2015
California Institute of Technology	CS1	1	Python	12/16/2014
	CS2	No syllabus obtainable		
University of Michigan	EECS280	3	C++	12/23/2015
University of Washington	CSE142	1	Java	12/24/2015
	CSE143	1	Java	12/24/2015
University of California - Davis	ECS30	2	C	1/5/2015
	ECS40	2	C++	1/5/2015
Columbia University	COMSW1004	No syllabus obtainable		
Purdue University	CS18000	2	Java	1/6/2015
Ohio State University	CSE1223	2	Java	1/6/2015
	CSE2221	2	Java	1/6/2015
Cornell University	CS1110 / CS1133	1	Python	12/18/2014
	CS1112 / CS1132	2	MATLAB	12/18/2014
University of Minnesota - Twin Cities	CSCI1133	3	Python	1/6/2015
Pennsylvania State University	CMPSC121	2	C++	1/6/2015
Texas A&M University - College Station	CSCE121	2	C++	1/6/2015

APPENDIX B:

OBJECTIVE G LANGUAGE DEFINITION

1. **Action Blocks**
   The Action Blocks are all the blocks that command the robot to do something. Some blocks return a data value. The Action Blocks are:

---

   a. **Drive Distance**: takes a single Integer-type block as a parameter, which tells the robot how far to drive. One can tell the robot to drive forwards or backwards by making the Integer value positive or negative, respectively.

   b. **Drive:** tells the robot to drive in a certain direction. Takes a single parameter, a Direction data block with parameters “Forward” or “Backwards”.

   c. **Turn Degrees:** takes a single Integer-type block as a parameter, which tells the robot how far to turn. One can tell the robot to turn right or left by making the Integer value positive or negative, respectively.

   d. **Turn to Bearing:** takes a single parameter, which may be either an Integer-type block or a Bearing data block. The robot will turn to either the cardinal direction specified, if a Bearing block is given, or to the bearing represented by the integer value, if an Integer block is given.

   e. **Turn:** tells the robot to turn in a certain direction. Takes a single parameter, a Direction data block with parameters “Right” or “Left”.

   f. **Get Distance:** takes a single parameter, a Direction data block with values “Forwards”, “Backwards”, “Left” or “Right”. Will take a sonar sensor reading from the sonar sensor specified by the Direction block and return it as an Integer.

   g. **Get Bearing:** takes a reading on the compass sensor and returns the value as an Integer. Does not take a parameter.

   h. **Stop:** stops the robot. Does not take a parameter.

2. Data and Robot Data

The data blocks function as input for other blocks. These blocks cannot stand alone and must be placed in a socket. There are two types: primatives (the value of which students must specify) and constants (the value of which are fixed)

---

a. Integer: a primitive block which allows students to enter an integer value.

b. String: a primitive block which allows students to enter a String value.

c. Boolean: a primitive block which allows students to enter a Boolean value.

d. Direction: a constant block. Students may select six different constants: Forward, Backwards, Left, Right, Front and Rear. The above list of six constants is filtered depending on what socket the Direction block is placed in.

e. Bearing: a constant block. Students may select four different constants: North, South, East and West.

---

3. Loops

A Loop block has a Condition and a Body. The Condition determines how many times a loop iterates; the Body is the code that is iterated. There are two loop types.

---

a. Loop For: takes an Integer, Integer-type Variable, or Integer-type Method as a Condition. Executes the code body the exact number of times specified by the Condition. If the Condition Integer is negative or zero, the loop body does not execute.

b. **Loop Until:** takes a Comparison or Logic block as its Condition. When the Loop Until first executes, the Condition is evaluated, and if it is FALSE, the loop body executes. After executing the body, the Condition is evaluated again. The loop body will execute until the Condition evaluates to TRUE. For this reason, it is possible to use Loop Until to create infinite loops.

4. **Wait Statements**
   A Wait Statement pauses the interpreter for a certain amount of time. No code is executed during this time, although any ongoing action the robot is performing like Drive or Turn will continue during the pause. There are two wait statement types.

   a. **Wait For:** takes an Integer, Integer-type Variable or Integer-type Method, and pauses execution for the number of milliseconds equal to the Integer parameter. If this Integer is zero or negative, the Wait will not pause anything.

   b. **Wait Until:** takes a Comparison or Logic block as a parameter. Pauses execution until the parameter evaluates to TRUE. Checks this parameter as frequently as it can. It is possible to create an infinite Wait Until.

5. **If Statements**
   An If statement, and its companions Else If and Else, allow a certain segment of code to be executed only if a certain Condition is TRUE (or FALSE). All If statements must contain an If block, and may optionally contain an Else If and / or an Else. An If Statement may contain one to many individual If / Else If / Else blocks, only one of which will have its body executed.

   a. **If:** an If block is the basic form of an If statement. It accepts a Condition (either a Comparison or a Logic block) and has a body. When the If block is executed, the Condition is evaluated. If the Condition is TRUE, the body is executed, and after the body is executed the next line of code outside the If Statement is executed. If the Condition is FALSE, either the next item in the If Statement is executed (either an Else If or an Else if there is one), or the next line of code outside the If Statement is executed if there is no other items in the If Statement.

   b. **Else If:** the Else If block also contains a Condition and a body. Else If blocks may only follow If blocks or other Else If blocks. When an Else If block is executed, the Condition is evaluated. If the Condition is TRUE, the body is executed, and after the body is executed the next line of code outside the If Statement is executed. If the Condition is FALSE, either the next item in the If Statement is executed (either an Else If or an Else if there is one), or the next line of code outside the If Statement is executed if there is no other items in the If Statement.

   c. **Else:** the Else block contains only a body, and has no condition. Else blocks may only follow an If or Else If block. An Else block terminates an If

Statement – no other Elses or Else Ifs may go after it. When an Else block is executed, the body is automatically executed, and after this the next line of code outside the If Statement is executed.

---

6. **Variables and Assignments:** a Variable is a temporary container for storing a value. An Assignment is a block which assigns a value to a Variable.

---

   a. **Variables:** All Variables have a type which may be either Integer, String or Boolean. Variables may be plugged into sockets, but may not stand alone. Variables may either return or receive a value, depending on where they are used. Variables are created or deleted in the Variable panel.

   b. **Assignment:** an Assignment statement has two sockets. The left hand socket may only accept a Variable. The right hand socket accepts a block of the same type as the Variable in the left hand socket – for example, a Data block of the same type, another Variable of the same type, or a Function of the same type.

---

7. **Logic and Comparison:** logic and comparison blocks may be entered into sockets which accept Conditions. These items cannot stand on their own. When evaluated, Logic and Comparison blocks return a Boolean value.

---

   a. **Comparison:** a comparison is a test between two blocks of the same type. Possible Comparison blocks are == (equals), != (not equals), <= (less than or equal to), >= (greater than or equal to), > (greater than), and < (less than). Two sockets are present in each Comparison block to accept the values being compared. Integer values may be used in all six tests; String or Boolean values may only be compared using == or !=. When evaluated, if the test passes, TRUE is returned; else, FALSE is returned.

   b. **Logic:** a Logic block has two sockets, both of which accept either Comparison blocks or another Logic block. There are two Logic blocks: AND and OR. When evaluated, the contents of both sockets are evaluated. If the Logic block is an AND, then the block will only return TRUE if both sockets evaluate to TRUE, and will return FALSE otherwise. If the Logic block is an OR, then the block will only return FALSE if both sockets evaluate to FALSE, and will return TRUE otherwise.

8. Methods

A Method is a set of instructions which may be called through a Method block. Methods have Types and Parameters; the Type determines what value the Method returns, and the Parameters are used to pass data into the Method.

---

a. Method: a Method may have a type of Integer, Boolean, or String, which indicates that the Method returns that type of data. A Method may also have a type of Void, which indicates the Method returns nothing. A Method may have parameters, which will be detailed later. The Method has a body; when the Method is executed, the body code is executed. A Method is executed by placing a Method block in either the main canvas or the body of some other block (including another Method). Methods may be created and deleted in the Method panel.

b. Parameter: a Method may have zero or more parameters. Each parameter has a type of either Integer, String or Boolean. When a Method is called, all Parameters must be passed in by inserting blocks of the appropriate types into the sockets. Inside the Method body the Parameters are available to be used in a similar way to Variables.

APPENDIX C:

EXAMPLE GENOST CURRICULUM WORKSHEETS

The following worksheet is a guided practice worksheet from Section 1. This worksheet walks the student through developing the algorithm to solve the maze.

NICKNAME: _____
DATE: _____

1.2.2 – Procedural Code and Turning Part 2

GOAL: DRIVE TO THE FINISH AND STOP

Introduction

Here's our next **maze:**

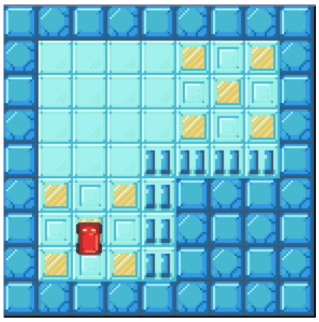


Figure 1: Maze 1.2.2

With your knowledge of **driving** and **turning** this shouldn't look too difficult. Let's begin by **planning out our algorithm.**

Pre-Exercise

We'll begin by writing out our **algorithm** in **general terms:**

1. **Drive forward some distance** _____
2. **Turn right** _____
3. **Drive forward some distance** _____

As before, we'll need to alter this **algorithm** to add in the **specific information**.
We've given you **space below** to write in the **exact distance / degree values**. Analyze the maze and then write in the values.

1. **Drive forward** _____ **pixels** _____
2. **Turn right** _____ **degrees** _____
3. **Drive forward** _____ **pixels** _____

Now that you've entered your values, you should be able to build the **code** from this **algorithm.**

Exercise

1. Once you have **changed your maze** to **1.2.2**, begin translating the **above algorithm** into **code** by placing the **three actions** into the **canvas,** in the **proper order**, as seen below.

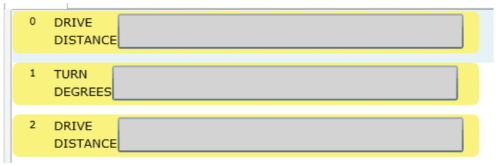


Figure 2: Three actions placed in the proper order, without ints

2. Now **add the Int blocks** into the **sockets** of the **three actions.**

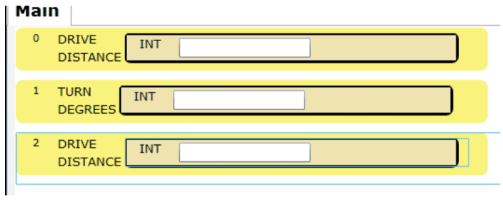


Figure 3: Three actions with Int parameters added

3. All that remains is to **enter the distance / degree values** you came up with in the **Pre-Exercise**. Go ahead and do so now, then **simulate** the maze.

4. If you entered the **right values**, your robot will drive to the end. If not, **close the simulator** and **try again!**

Post-Exercise

Discussion

This maze is **another** good example of **Procedural Code Flow**. The **robot executed** each **code block** only after the **previous block** was **executed.** Think about it this way:

1. The **Drive Distance** block on line **0** was executed first. The robot **drove** all the way to the end of the first **corridor**.

2. The **Turn Degrees block** executed **immediately after** the **first Drive Distance block**. It continued **executing** until the **robot** had fully made its turn.

3. Finally, the **second Drive Distance block** on **line 2** was executed. The **robot** drove **all the way to the end of the second corridor**.

4. The **program** then ended, and the **robot stopped.**

At no time did one block **execute** while another one was executing. Additionally, **each block** ran **immediately** after the previous one was done. The **robot** tries to execute its blocks **as fast as it can**.

The following worksheet is a simple exercise from Section 2. The worksheet does not walk the student through solving the maze, but does provide guidance on how to get there.

NICKNAME: _____
DATE: _____

2.3.3 – Nested Loop For Takehome 1

GOAL: COLLECT ALL THE COINS AND STOP

Introduction

We've walked you through **two examples** of **formal algorithm analysis** – now it's time for you to do it on your own. This example will give you a bit of guidance, but much of the process done in the last two mazes will be up to you.

Just remember how we did things previously, follow the formula, and pay close attention to the algorithm, and you'll do fine!

Here is the maze we are solving:

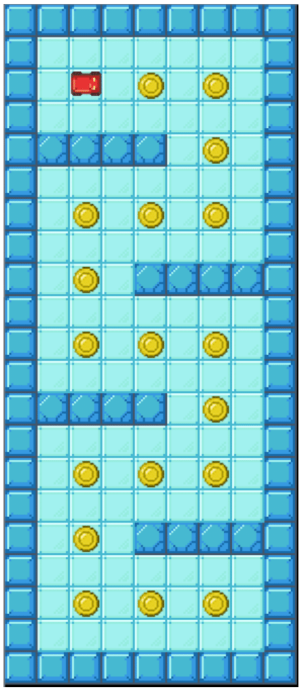


Figure 1: The maze for 2.3.3

Pre-Exercise

Step 1: Fully Understand the Problem

Does this maze look familiar to you? It should! This was the maze for **2.1.4**, where we solved it using a **single loop**. If you remember, the code was rather **ugly** – in this maze, we'll get another **opportunity** to write a **cleaner** algorithm using **nested loops**.

Try to remember the way you solved the maze **previously**. This may help you in solving the maze.

Step 2: Break the Problem Down into Subproblems

Now we must break the **maze** down into **subsections**. We are going to do so **twice** - we will first break the maze into **subsections**, and then break those **subsections** down further into **subsections**.

Divide the Maze (Level 1) into Three Subsections (Level 2)

To help you out this first time, we will **describe** the subsections that you need to break the maze down into.

- There are **two subsections:**
- The **two subsections** should have an **identical shape**.
- **Subsection 1** should contain **8 coins**.
- The **two subsections** will not cover the **entire maze**. **Two coins** will be not be covered.

Now, **draw the two subsections** on the **maze below, labeling them Subsections 1 and 2**.
Make sure to **label them clearly!**

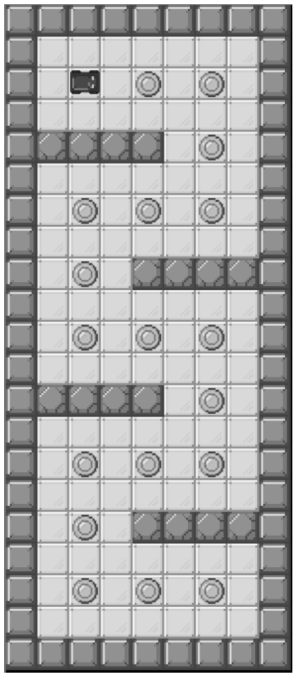


Figure 2: Draw the two Level 2 subsections on this maze.

Divide Subsections 1 and 2 (Level 2) into Two Subsections Each (Level 3)

Now you will need to further **subdivide Subsections 1 and 2**.

Again, here are some **hints** that will tell you how to **subdivide Subsections 1 and 2**

- **Subsections 1 and 2** should have **two subdivisions each,** making **4 Level 3 subdivisions total**.
- The **subdivisions** for **Subsections 1 and 2** should be **identical**
- The **subdivisions** should be **DT Loops**.

Draw the **Level 3 subsections** on the **maze below**, labeling them **Subsections A and B**.
Note that there should be two Subsection A's and two Subsection B's

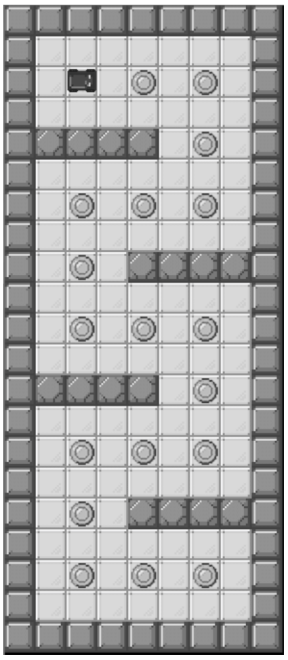


Figure 3: Draw the four Level 3 subsections on this maze.

Once you have completed this, you should have **three levels** fitting the **following descriptions:**

PROBLEM LEVELS FOR 2.3.3:

1. **Solution for 2.3.3**
2. **Solution for Subdivisions 1 and 2**
3. **Solution for Subdivisions A and B.**

This completes **Step 2.**

Step 3: Solve the Subproblems

Now that we have **identified** our **subsections**, we need to **write code** for them.
We will give you **templates** to write the code in, which should help you figure it out.

First, we will write code for **Subsection A:**

CODE FOR SUBSECTION A:

1. **Loop For _____ times_____**
    a. **_____**
    b. **_____**

Next, you must write code for **Subsection B:**

CODE FOR SUBSECTION B:

1. **Loop For _____ times_____**
    a. **_____**
    b. **_____**

This completes **Step 3.** We have now solved **Level 3**.

PROBLEM LEVELS FOR 2.3.3:

1. **Solution for 2.3.3**
2. **Solution for Subdivisions 1, 2 and 3**
3. **Solution for Subdivisions A and B. <SOLVED>**

Step 4: Combine the Subproblem Solutions into a Single Solution for the Whole Problem

Finally, we must **combine** our **code** on the various levels to produce **a single algorithm**.

Solving Level 2

We'll begin by **combining** the code for **Subsection A** and **Subsection B** to create code that will solve **Subsections 1 and 2.** What you need to do here is take **Subsection A** and **Subsection B** and combine them together in a way that the **single combination** will solve both **Subsection 1** ***and*** **Subsection 2**.

We will give you a **template** that will help you with this. Your code should fit on **six lines.**

CODE FOR SUBSECTION 1 / 2:

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____

Now we have solved **Level 2.**

PROBLEM LEVELS FOR 2.3.3:

1. **Solution for 2.3.3**
2. **Solution for Subdivisions 1, 2 and 3 <SOLVED>**
3. **Solution for Subdivisions A and B. <SOLVED>**

Solving Level 1

Finally, we must **combine the solutions for Subsections 1 / 2** to create a **single solution** for **2.3.3.**

Note that Subsections 1 and 2 do not cover the entire **maze**.
Therefore, after combining **Subsections 1 and 2**, you will have to write some **additional code** to collect the **last two coins.**

If you've done everything right up to this point, then this shouldn't be too hard for you.

Here are your **hints** for solving **Level 1:**

- Your code should occupy **8 lines**
- Your code should contain **one outer loop** with **two inner loops**.
- The **additional code** that will collect the **last two coins** should consist solely of **one line.**

CODE FOR 2.3.3:

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____

If you entered the above code right, then we have now solved **Level 1!**

PROBLEM LEVELS FOR 2.3.3:

1. **Solution for 2.3.3 <SOLVED>**
2. **Solution for Subdivisions 1, 2 and 3 <SOLVED>**
3. **Solution for Subdivisions A and B. <SOLVED>**

Exercise

1. Take the **code** you wrote out in the **above section "Code for 2.3.3"** and **transform it into Genost code**.
2. **Simulate** your code and make sure that it works.

The following worksheet is a challenging exercise from Section 3. This worksheet provides no guidance for the student to solve the problem. Also, note how this worksheet requires the student to solve multiple mazes.

NICKNAME: _____
DATE: _____

3.3.4a / 3.3.4b – Wait Fors Takehome 2a / 2b

GOAL: COLLECT ALL THE COINS AND STOP

Introduction

This is the **final maze** for **Section 3.** As usual, you will have to use **all the concepts you have learned so far** to **solve it.**

The **mazes** you will be solving are below:

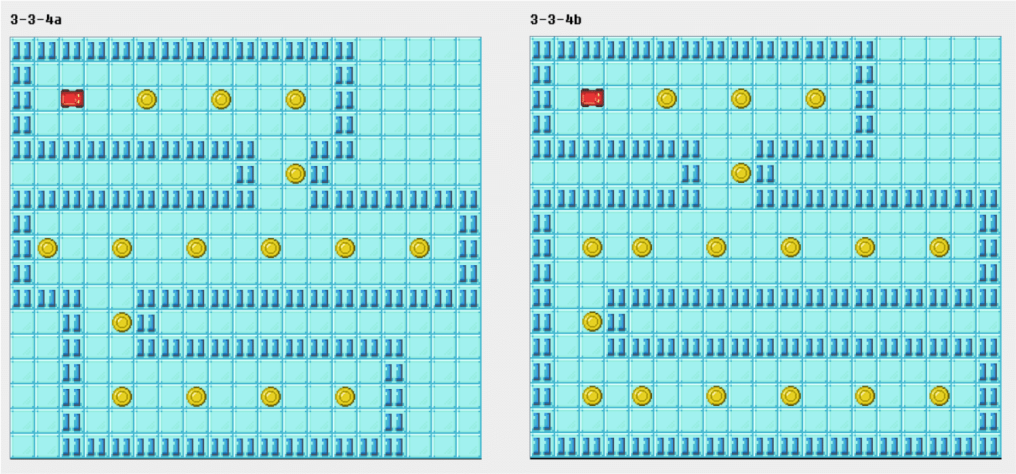


Figure 1: The two mazes for 3.3.4

Pre-Exercise

Please write **a single algorithm** that will solve **both mazes**. As usual, you will find it helpful to use **algorithm analysis**. **Good luck!**

Step 1: Fully Understand the Problem
Step 2: Break the Problem Down into Subproblems
Step 3: Solve the Subproblems
Step 4: Combine the Subproblem Solutions into a Single Solution for the Whole Problem

APPENDIX D:

COMPUTATIONAL THINKING TESTING INSTRUMENT

Nickname: _____

Genost Evaluation

Date: _____

Please answer the following questions as best you can.

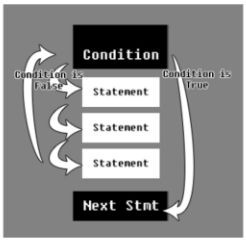
There will be no penalty for failing to answer a question or answering incorrectly. We just want to see how much you know and how much you have learned. ☺

If you don't know the answer to a question, please circle **I don't know how to answer Question #X**.

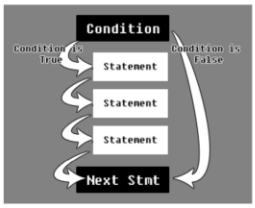
---

1. **Concept Matching**
   Each of the following **pictures** on the **right hand side** represent a **concept** on the **left hand side**.
   Please ***draw a line*** from each **concept** to the **picture** that represents it.

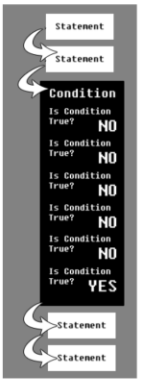
IF



WAIT UNTIL



LOOP UNTIL



I don't know how to answer Question #1.

2. **Algorithm Analysis**
   There are **four steps** involved in **Algorithm Analysis**. ***Fill in the blanks*** below to **complete** the **four steps.**

1. _____ the problem
2. _____ the problem down into subproblems
3. _____ the subproblems
4. _____ the subproblem solutions into a single solution for the whole problem.

I don't know how to answer Question #2.

3. **Three Goals of Programming**
   When developing **algorithms** we have **three goals.** Please ***fill in the blanks*** below to complete the **three goals**.

1. **Reduce** the _____ **and** _____ of your **algorithm.**
2. **Create** algorithms that are more _____ and can be used in _____ **mazes**.
3. Have _____ programming!

I don't know how to answer Question #3.

4. **Program Execution vs. Robot Action**.
   The three **statements** below are about **Program Execution vs. Robot Action**.
   Please indicate whether the **statement** is **correct or not** by ***circling TRUE or FALSE***

1. The **robot's movement** is **always in sync** with the **algorithm execution.** ---------- **TRUE FALSE**
2. It is **possible** for the **robot** to **stop moving** while the **algorithm continues executing**. ---------- **TRUE FALSE**
3. It is **possible** for the **algorithm** to **finish executing** while the **robot continues moving**. ---------- **TRUE FALSE**
4. It is **possible** for the **algorithm** to **pause execution** while the **robot continues moving.** ---------- **TRUE FALSE**

I don't know how to answer Question #4

5. **Data Flow**
The **image below** shows an **If block** containing a **Comparison** between two **sonar sensor values.**
These **blocks** will execute in a certain **order.** ***Number, from 1 to 4, 1 being first and 4 being last***, the **four blocks** to indicate which **order** they will execute in.

_____ - The **< (less than) block**

_____ - The **Front Sonar Sensor block**

_____ - The **If block**

_____ - The **Rear Sonar Sensor block**

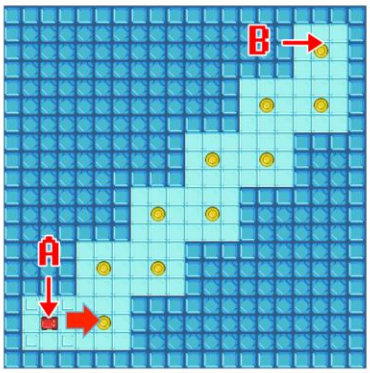
I don't know how to answer Question #5

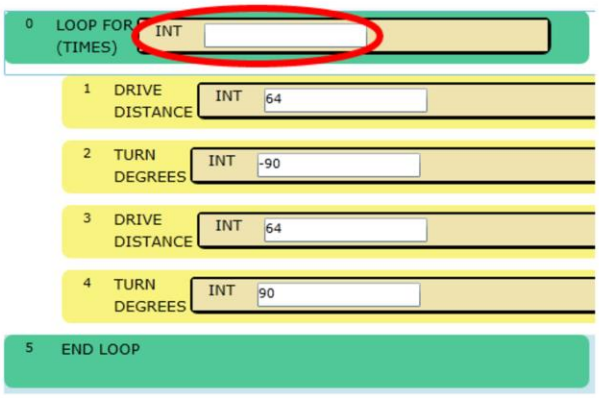
6. **Loop For**

Consider the following **code** and **maze**.

The code below will **drive** the robot to the **end** of the **maze**, but the **code** is **incomplete**. Note the **Int block** circled in **Red**.

Write an integer in the box below that will complete the code and solve the maze.





[ ] **ANSWER:**

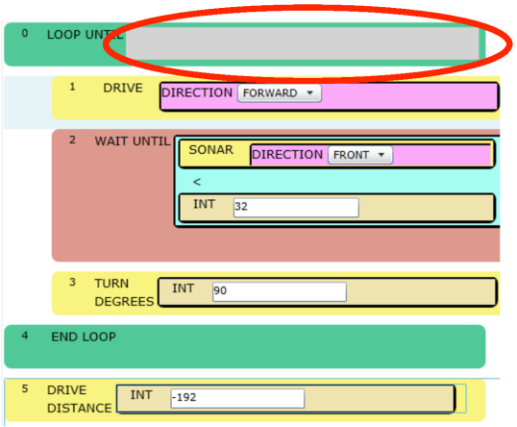
I don't know how to answer Question #6.

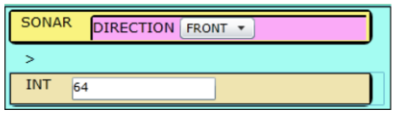
7. **Loop Until**

Consider the following **code** and **maze**.

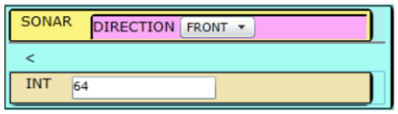
The code below will **drive** the robot to the **end** of the **maze**, but the **code** is **incomplete**. Note the **empty socket** in the **Loop Until block** circled in **Red**.

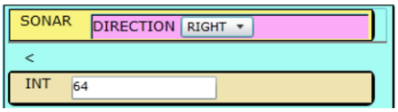
Choose, from the following four options, the Comparison block which will complete the code and solve the maze.



a. 

c. 

b. 

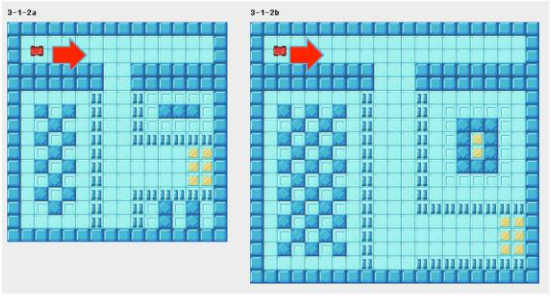
d. 

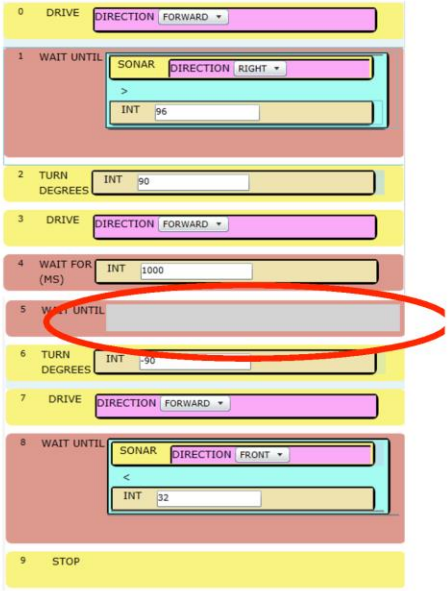
I don't know how to answer Question #7

8. **Wait Until**

Consider the following **code** and **mazes.** We need to create **one algorithm** that will solve **both mazes.**
Note that one **Wait Until comparison block** is not **filled in** – it's **circled in red.**

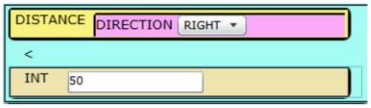
Choose, from the following four options, the Comparison block which will complete the code and solve the maze.





a. 

b. 

c. 

d. 

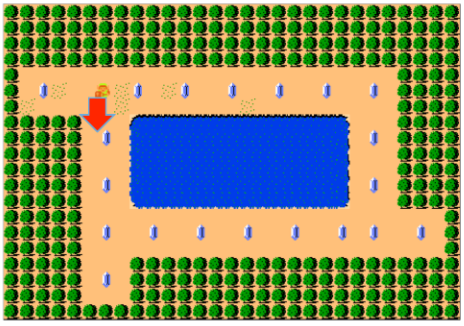
I don't know how to solve Question #8

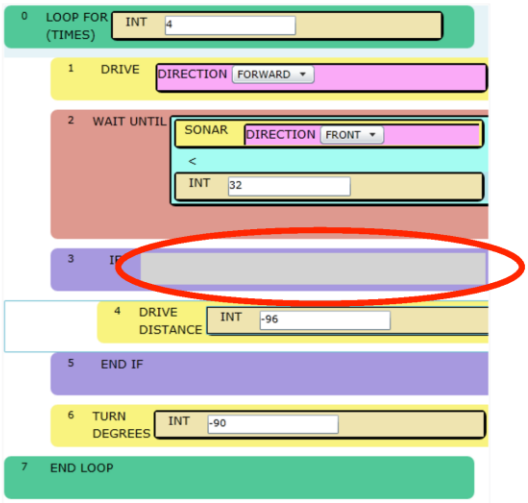
9. **Ifs**

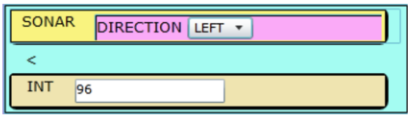
Consider the following **code** and **maze.**

The code below will **drive** the robot to the **end** of the **maze**, but the **code** is **incomplete**. Note the **empty socket** in the **If block** circled in **Red**.

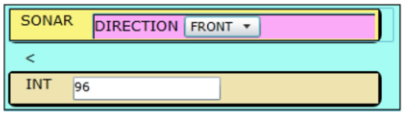
Choose, from the following four options, the Comparison block which will complete the code and solve the maze.





a. 

c. 

b. 

d. 

I don't know how to answer Question #9

10. Logical AND / OR

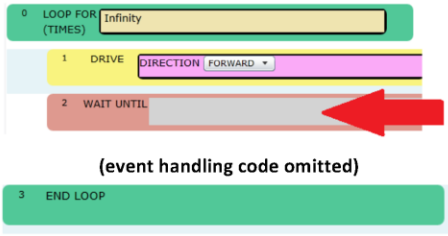
A **right wall following** algorithm tells us that the robot should drive **forward along a wall on the robot's right side** until one of the **following two events occur:**

- The **front sonar sensor** detects a wall in **front**.
- The **right sonar sensor** detects that the **right wall** is no longer there.

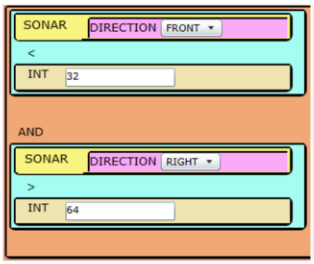
After **one of these events is triggered**, we **handle** the event accordingly.

We want to create **Genost code** that will **drive the robot forward** until **either of those events are triggered.** The **code below** shows **part** of our **Genost code**, but the **triggering condition** is missing. Note the **empty socket** in the **Wait Until block** marked by the **red arrow.**

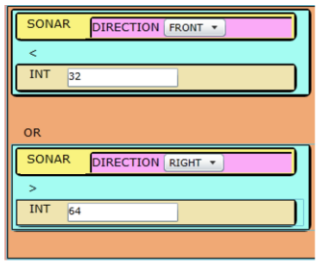
Choose, from the following four options, the Comparison block which will complete the right wall following code.



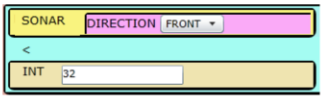
a.



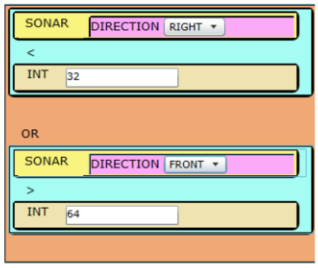
c.



b.



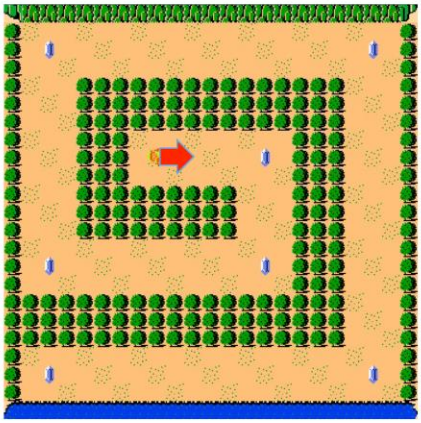
d.

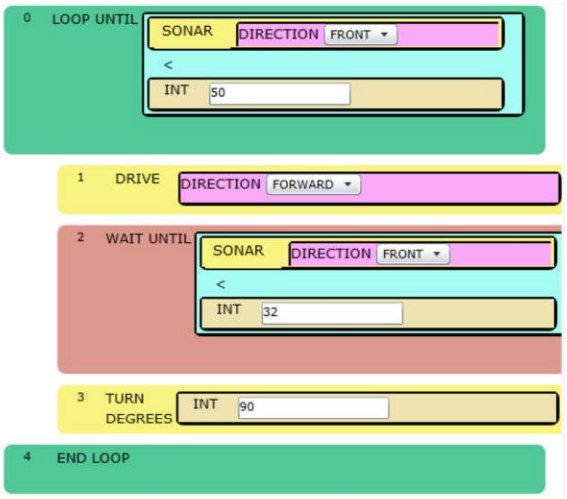


I don't know how to answer Question #10

11. Maze Solving

Consider the following **code** and **maze. Will the given code solve the maze?** ***Circle YES or NO.***





YES **NO**

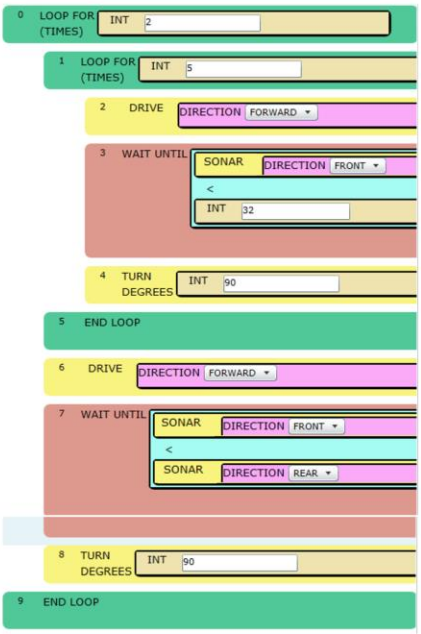
I don't know how to answer Question #11.

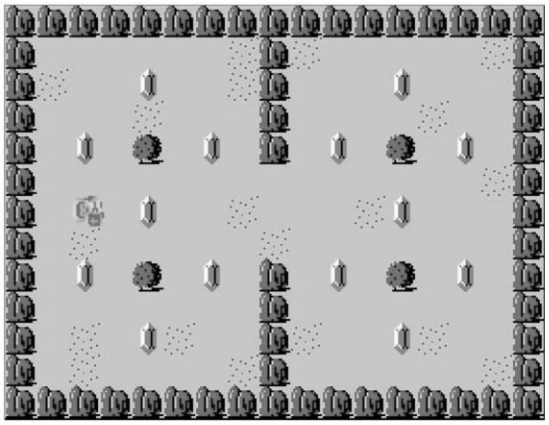
12. Tracing Code

Consider the following **code** and **maze**.

Please read the **code** and make sure you understand it.

Then, ***using arrows, draw the path in the maze that the robot will take as it executes the code.***





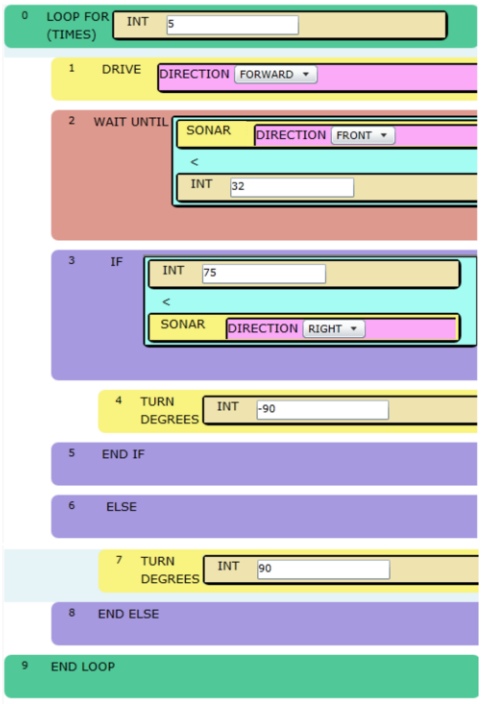
I don't know how to answer Question #12.

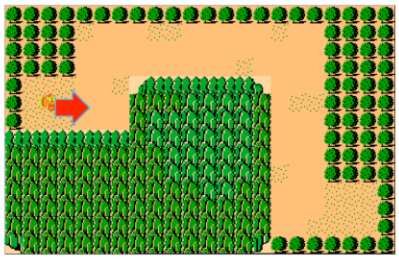
13. Debugging

Consider the **code** and the **maze below.** The **code** is set up to solve the maze, but there is a **problem** with it.

Select, from the FOUR options below, the line number that contains the problem.

For **extra credit**, write in your own words how to **solve** the problem.





a. Line 2 **b. Line 3**

c. Line 0 **d. Line 7**

EXTRA CREDIT: write, in your own words, how to solve the problem.

I don't know how to answer Question #13.

APPENDIX E:

FEEDBACK FORMS

The following form is the Likert scale form used to collect feedback on the Genost software.

<Test Name> - Genost Education and Usability Test
System Usability Test
(System)

Nickname: _____

Date: _____

Please CIRCLE the number that best describes your feeling about Genost!

1. It was easy to use Genost

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

2. I feel that I learned something by using Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

3. Using Genost was fun for me.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

4. I have a better understanding now of programming concepts thanks to Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

5. It was easy to learn how to use Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

6. I had a pleasant experience using Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

7. Genost effectively taught me programming concepts.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

8. I feel comfortable using Genost

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

9. I enjoyed using Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

10. Overall, I am satisfied with my experience using Genost.

(Not at all) (Somewhat) (Definitely)
1 . . . 2 . . . 3 . . . 4 . . . 5 . . . 6 . . . 7 . . . 8 . . . 9 . . . 10

The following is the free response form used to collect information on the Genost curriculum and presentation.

GENOST FEEDBACK FORM

Please fill this in with as much or as little detail as you want.
Please be honest and give me all the feedback you can, so I can improve my curriculum! *Thank you.*

Please write **three or more things** that you **liked** about the **Genost curriculum** and my **presentation:**

Please write **three or more things** that you **disliked** about the **Genost curriculum** and my **presentation:**

Please write **three or more things** that you would **change** about the **Genost curriculum** or my **presentation:**

Please write **any other comments** that you have related to the **Genost curriculum** or my **presentation:**

APPENDIX F:

RECRUITMENT MATERIALS AND CONSENT FORMS

The following text contains the speech made to the FSE100 students recruiting them to participate in the test.

Howdy everyone, I'm Garret Walliman. Welcome to your first day of FSE100.

I'm here to talk to you about an opportunity that may help you out a lot in FSE100, as well as your other programming classes throughout your time here at ASU.

Wanted to ask – how many in here are CS? CSE? Other degrees?
For what reason did you choose your degree?

All right. ASU is conducting research into a new educational program that we're developing. We're wondering whether a curriculum that focuses heavily on programming fundamentals and algorithm development helps students in future programming classes.

To this end we will be holding an extracurricular activity for FSE100 students during the first three weeks of September. We will meet six times and go through a small, custom set of lessons that we've developed using a new educational tool that we've built.

It is our hope that students participating in this course will gain a deeper insight into programming and that the ultimate result will be better understanding and higher grades. In addition, all students who complete this activity will be given 10% extra credit in FSE100 – that's a full letter grade.

So I'm going to ask you for two things. The first thing I want to ask is for anybody who wants to participate in this extracurricular activity to sign this signup sheet.

The second thing I would like to ask is for all students, whether you want to participate in the extracurricular activity or not, to sign a consent form to release your FSE100 grade data to our study. This data will allow us to compare the performance of those who participate in the exercise to those who do not.

I want to make it clear that you are not required to participate in the activity or to release your grade data to us. Your participation in these things is not required for you to attend FSE100 or ASU. You will not be penalized for choosing not to participate. You can also choose to stop participation at any time.

If you do choose to consent to releasing your grade data, it will be anonymized and will not be connected to you in any way. At ASU we comply with all ethical standards related to student data so you can trust that we'll keep it safe.

So – if you wish to allow us to collect your grade data for FSE100, please sign these consent forms. If you wish to participate in our extracurricular activity, please sign this signup sheet. For those who sign up, if you are chosen to participate, we will contact you closer to the activity.

Again, your participation is not required, but it will help us, and hopefully it will greatly benefit you as well.

I'd like to remind you again that participating in our activity and completing it will earn you 10% extra credit in your FSE100 class. If you are interested in earning the 10% extra credit but do not want to participate in our activity, there is an alternative activity available that I can describe to those who are interested. If you want to participate in the ASU activity, please sign the signup sheet. If you would like to participate in the alternative activity, please send me an email at the email address on the board.

Thanks guys!

The following text was sent to parents in an email, and displayed on the ASA website, to recruit ASA students to participate in the independent group.

Educational Programming Opportunity!

Programming and computational thinking are now recognized as a fundamental skill, as important as reading and writing, and a necessary ability for the modern world. We believe that a skill this fundamental should be given fundamental education.

An introductory programming class will be held at the Arizona School for the Arts from **September 8 – 12 and 15 - 19, 3PM to 5PM**. We are looking for **30 students, grades 7th to 12th**, to attend!

In this class, you will learn the fundamental skills involved in creating algorithms and programming. These skills include:

- Performing problem analysis
- Designing algorithms
- Recognizing and understanding programming concepts
- Applying and combining fundamental programming structures to implement an algorithm in code.
- General computational thinking principles

We will be using a new program being developed at Arizona State University, called **Genost**, to help teach these fundamental skills. This class is being held as part of a study conducted at Arizona State University to determine the educational effectiveness of **Genost**.

All participants, and their parents, will be required to sign consent forms to participate in the class. These forms may be downloaded below, and provide more information about the research being conducted.

We look forward to seeing you!

The following form is the grade release consent form used for the FSE100 students.

Arizona State University Fall 2014 FSE100 Extracurricular Activity:
Genost Educational and Usability Study
Student Consent Form for FSE100 Grade Release (All FSE100 Students)

My name is Garret Walliman, and I am a graduate student in the College of Computer Informatics, Systems and Design Engineering at Arizona State University under the direction of Dr. Robert Atkinson. I am conducting a research study to learn whether Genost, a new educational program being developed at Arizona State University, is effective at teaching the introductory programming skills you will be learning in FSE100.

I am inviting your participation in my study. I am conducting an extracurricular activity during this semester for some FSE100 students where fundamental programming and algorithmic development skills will be taught. In order to test the effectiveness of this activity, I would like to collect and analyze student grades in FSE100, both for students who participate in this extracurricular activity and those who do not.

This consent form is asking you to consent to releasing your FSE100 grade information to me for your Fall 2014 FSE100 class. If you consent to this, then your professor will provide me with all of your grade information for FSE100, both for individual assignments and overall.

Only FSE100 grade data will be collected if you consent. No grade data will be collected from your other classes. All grade data that is collected will be anonymized so that there is no connection to your name or any other personally identifiable information. All grade data that is collected will be stored securely during and after the study.

Your participation in this study is voluntary. You do not have to participate in the study to attend FSE100, nor will choosing not to participate affect your grade in FSE100 in any way. If you choose not to participate or to withdraw from the study at any time, there will be no penalty to you in any way. If you choose not to participate, we will not collect, store, analyze, or otherwise use any of your grade data.

By participating in this study, we hope that the data collected will allow us to improve the Genost software. There are no foreseeable risks or discomforts to your participation.

If you consent to allowing us to collect grade data, we will anonymize your grade data upon receiving it by replacing your name with a pseudonym. We will never store your grade data in any way that connects it to you. Any original documents which contain personally identifiable information will be destroyed immediately after anonymizing the data.

The results of this study may be used in reports, presentations, or publications but your name or any other personally identifiable information will not be used.

If you have any questions concerning the research study, please contact me at garret.walliman@asu.edu. If you have any questions about your rights as a subject/participant in this research, or if you feel you have been placed at risk, you can contact the Chair of the Human Subjects Institutional Review Board, through the ASU Office of Research Integrity and Assurance, at (480) 965-6788. Please let me know if you wish to be part of the study.

By signing below you are agreeing to be part of the study.

Your Name:

Your Professor's Name: Class Time:

Your Signature: Date:

The following form is the consent form for participants in the FSE100 Genost group.

Arizona State University Fall 2014 FSE100 Extracurricular Activity:
Genost Educational and Usability Study
Student Consent Form for FSE100 Extracurricular Activity (ExtracurricularStudents)

My name is Garret Walliman, and I am a graduate student in the College of Computer Informatics, Systems and Design Engineering at Arizona State University under the direction of Dr. Robert Atkinson. I am conducting a research study to learn whether Genost, a new educational program being developed at Arizona State University, is effective at teaching the introductory programming skills you will be learning in FSE100.

I am inviting your participation in my study. I am conducting an extracurricular activity during this semester for some FSE100 students where fundamental programming and algorithmic development skills will be taught. I would like you to participate in this extracurricular activity, and with your consent I would like to collect data about your experience with Genost.

The data I would like to collect is:

1) Data collected automatically during your interaction with the Genost program, such as successful solutions, lesson changes, and general activity.
2) Results from two short 10-question examinations that will be given to you in the course of the activity.
3) Results from a short 10-question survey about your experiences with Genost.

Your participation in this extracurricular activity is voluntary. You do not have to participate in the activity to attend FSE100, nor will choosing not to participate affect your grade in FSE100 in any way. However, you must consent to the above data gathering if you wish to participate in the extracurricular activity. If you choose not to participate or to withdraw from the study at any time, there will be no penalty to you in any way. If you choose not to participate, we will not store, analyze, or otherwise use any data related to your interaction with Genost.

By participating in this study, we hope that the data collected will allow us to improve the Genost software. There are no foreseeable risks or discomforts to your participation.

Your responses to the tests and survey, and any data recorded during your interaction with the teaching software, will be kept confidential. If you choose to participate, you will be given a nickname that you will use on pertinent tests and surveys. While we will keep a document linking your nickname to your real name for the length of the course, we will destroy this document afterwards, and once this document is destroyed the collected data will not be linked to you in any way.

The results of this study may be used in reports, presentations, or publications but your name will not be used.

If you have any questions concerning the research study, please contact me at garret.walliman@asu.edu. If you have any questions about your rights as a subject/participant in this research, or if you feel you have been placed at risk, you can contact the Chair of the Human Subjects Institutional Review Board, through the ASU Office of Research Integrity and Assurance, at (480) 965-6788. Please let me know if you wish to be part of the study.

By signing below you are agreeing to be part of the study.

Name:

Professor's Name: Class Date / Time:

Signature: Date:

The following form is the consent form for participants in the ASA independent group.

Arizona School for the Arts Fall 2014 - Genost Educational and Usability Study

My name is Garret Walliman, and I am a graduate student in the College of Computer Informatics, Systems and Design Engineering at Arizona State University under the direction of Dr. Robert Atkinson. I am conducting a research study to learn whether Genost, a new educational program being developed at Arizona State University, is good at teaching students like you how to program.

I am inviting your participation, which will involve going to an after-school activity for two hours a day over the course of a week, taking two short tests, and participating in a short multiple-choice survey. You have the right not to answer any question, and to stop participation at any time.

Your participation in this study is voluntary. If you choose not to participate or to withdraw from the study at any time, there will be no penalty to you in any way.

By participating in this study, we hope that you will learn fundamental programming skills. There are no foreseeable risks or discomforts to your participation.

Your responses to the tests and survey, and any data recorded during your interaction with the teaching software, will be kept confidential. If you choose to participate, you will be given a nickname that you will use on all tests and surveys. While we will keep a document linking your nickname to your real name for the length of the activity, we will destroy this document afterwards, and once this document is destroyed your responses will not be linked to you in any way.

The results of this study may be used in reports, presentations, or publications but your name will not be used.

If you have any questions concerning the research study, please contact me at garret.walliman@asu.edu. If you have any questions about your rights as a subject/participant in this research, or if you feel you have been placed at risk, you can contact the Chair of the Human Subjects Institutional Review Board, through the ASU Office of Research Integrity and Assurance, at (480) 965-6788. Please let me know if you wish to be part of the study.

By signing below you are agreeing to be part of the study.

Name:

Signature: Date:

APPENDIX G:

SYSTEM COMPARISONS

Positive Comparisons	Features Virtual Worlds	Features Robot	System Paired with Official Curriculum	Simplified Language	Impossible to Make Syntax Errors	Block Design Indicates Block Use	System Abstracts Actions to High Level	System Pairs Program Steps with On-Screen Execution	System is Extendable By End User	System Usable With Multiple Robots	Robot Code Runs on Computer	System Integrates with Social Media
Logo	YES	NO	?	YES	NO	N/A	YES	N/A	NO	N/A	N/A	NO
IBM Robocode	YES	NO	N/A	NO	NO	N/A	NO	NO	NO	N/A	N/A	NO
Myro	NO	YES	YES	NO	NO	N/A	NO	NO	NO	NO	YES	NO
FIRST Robotics Competition	NO	YES	N/A	NO	NO	N/A	NO	NO	NO	YES	NO	NO
Alice	YES	NO	YES	YES	YES	YES	YES	YES	NO	N/A	N/A	NO
Scratch	YES	NO	NO	YES	YES	YES	YES	YES	NO	N/A	N/A	YES
Lego Mindstorms	NO	YES	NO	YES	YES	YES	YES	N/A	NO	NO	NO	NO
Microsoft Robotics Developer Studio	YES	YES	NO	NO	YES	NO	YES	N/A	YES	YES	NO	NO
Genost	YES	YES	YES	YES	YES	YES*	YES	YES*	YES	YES	YES*	NO

* Feature is planned or partially implemented but not fully implemented in current version.

Negative Comparisons	Requires Learning Complex Syntax / Grammar	Focuses on Building / Engineering Robot	Focuses Heavily on Competition, Creative "Play" or Storytelling	Visual Language Looks Dissimilar to Formal Language	Language is Oversimplified	System Not Limited to Educational Focus	System is Expensive	Robot is Technically Complex	Curriculum Features Inappropriately Advanced Topics
Logo	YES	N/A	NO	N/A	NO	NO	NO	N/A	NO
IBM Robocode	YES	N/A	YES	N/A	NO	NO	NO	N/A	NO
Myro	YES	NO	NO	N/A	NO	NO	NO	NO	YES
FIRST Robotics Competition	YES	YES	YES	N/A	NO	NO	YES	YES	NO
Alice	NO	N/A	YES	NO	NO	NO	NO	N/A	NO
Scratch	NO	N/A	YES	NO	NO	NO	NO	N/A	NO
Lego Mindstorms	NO	YES	NO	YES	YES	NO	YES	YES	NO
Microsoft Robotics Developer Studio	NO	NO	NO	YES	NO	YES	NO	NO	NO
Genost	NO	NO	NO	NO	NO	NO	NO	NO	NO

In the previous chart, "N/A" was colored the same as "Yes" to indicate that the system should not be faulted for not including a positive for a feature it does not have. In this chart, "N/A" is colored differently from "Yes", to indicate that the system should not be credited for not suffering a failure for a feature it does not have.

APPENDIX H:

IRB APPROVAL DOCUMENTS

ASU Knowledge Enterprise Development

EXEMPTION GRANTED

Robert Atkinson
CIDSE: Computing, Informatics and Decision Systems Engineering, School of
480/727-7765
Robert.Atkinson@asu.edu

Dear Robert Atkinson:

On 8/13/2014 the ASU IRB reviewed the following protocol:

Type of Review:	Initial Study
Title:	Arizona School for the Arts Fall 2014 - Genost Educational and Usability Study
Investigator:	Robert Atkinson
IRB ID:	STUDY00001418
Funding:	None
Grant Title:	None
Grant ID:	None
Documents Reviewed:	• Child Assent Form, Category: Consent Form; • Parental Consent Form, Category: Consent Form; • gwalliman_hrp_503a.docx, Category: IRB Protocol; • Pretest / Posttest, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Genost Survey, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Website Recruitment Description, Category: Recruitment Materials;

The IRB determined that the protocol is considered exempt pursuant to Federal Regulations 45CFR46 (1) Educational settings on 8/13/2014.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (HRP-103).

Sincerely,

IRB Administrator

cc: Garret Walliman
Garret Walliman

ASU Knowledge Enterprise Development

EXEMPTION GRANTED

Robert Atkinson
CIDSE: Computing, Informatics and Decision Systems Engineering, School of
480/727-7765
Robert.Atkinson@asu.edu

Dear Robert Atkinson:

On 8/14/2014 the ASU IRB reviewed the following protocol:

Type of Review:	Modification
Title:	Arizona School for the Arts Fall 2014 - Genost Educational and Usability Study
Investigator:	Robert Atkinson
IRB ID:	STUDY00001418
Funding:	None
Grant Title:	None
Grant ID:	None
Documents Reviewed:	• Parental Consent Form, Category: Consent Form; • Child Assent Form, Category: Consent Form; • gwalliman_hrp_503a.docx, Category: IRB Protocol; • Pretest / Posttest, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Genost Survey, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Website Recruitment Description, Category: Recruitment Materials;

The IRB determined that the protocol is considered exempt pursuant to Federal Regulations 45CFR46 (1) Educational settings on 8/14/2014.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (HRP-103).

Sincerely,

IRB Administrator

cc:

EXEMPTION GRANTED

Robert Atkinson
CIDSE: Computing, Informatics and Decision Systems Engineering, School of
480/727-7765
Robert.Atkinson@asu.edu

Dear Robert Atkinson:

On 8/15/2014 the ASU IRB reviewed the following protocol:

Type of Review:	Initial Study
Title:	Arizona State University Fall 2014 FSE100 Introduction to Engineering Extracurricular Activituy - Genost Educational and Usability Study
Investigator:	Robert Atkinson
IRB ID:	STUDY00001417
Funding:	None
Grant Title:	None
Grant ID:	None
Documents Reviewed:	• Consent Form for All FSE100 Students, Category: Consent Form; • Consent Form for Extracurricular Activity Participants, Category: Consent Form; • gwalliman_hrp_503a.docx, Category: IRB Protocol; • genost_test_version_A.pdf, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Survey, Category: Measures (Survey questions/Interview questions /interview guides/focus group questions); • Verbal Recruitment Statement to Students, Category: Recruitment Materials;

The IRB determined that the protocol is considered exempt pursuant to Federal Regulations 45CFR46 (1) Educational settings on 8/15/2014.

In conducting this protocol you are required to follow the requirements listed in the INVESTIGATOR MANUAL (HRP-103).

Sincerely,

IRB Administrator

cc: Garret Walliman
Garret Walliman