Paper ID #22105

Misconceptions and the Notional Machine in Very Young Programming Learners (RTP)

Prof. Tony Andrew Lowe, Purdue University, West Lafayette (College of Engineering)

Tony Lowe is a PhD student in Engineering Education at Purdue University. He has a BSEE from Rose-Hulman Institute of Technology and a MSIT from Capella. To pass the time between classes he works for Anthem as a software architect and teaches as an adjunct at CTU Online.

Misconceptions and the Notional Machine in Very Young Programming Learners (RTP)

Abstract

This study looks at very young learners make mistakes and possibly form misunderstanding when learning to programming. A variety of national efforts are extending programming education to younger learners who are materials many adults struggle to learn. For decades literature has captured common misconceptions in using programming constructs (e.g. conditionals, loops, and recursion) in older learners, but early learners may wait years before they tackle these complex concepts. Many model misconceptions as a missing or inaccurate notional machine. The notional machine is an individual's mental model, representing how a programming language executes on a real device. The notional machine aligns with traditional learning models from several educational theorists, particularly Bruner's three stages of representations and Kahneman's neuroscience-based modeling of the mind. To better understand the early thought process of and learning theory for teaching novices, this study looks at videos of early elementary students working to create basic navigational programs for simple robots. We observed students in K-2 and categorized the mistakes made and strategies used to achieve their goals. Our findings align with prior misconception literature in very young learners around the 'problem' being the source of more misconceptions than the language. We also find promising cases which support learning theory around the notional machine, Bruner's representations and Kahneman's two mind model. Using this theory suggests possible approaches to consider in teaching young learners to program.

Introduction

Background and Motivation

Computers Science (CS) educators and researchers have long worked to capture the struggles novices face in learning to programming. Most efforts focus on older students, but from the early days researchers have looked at young children learning to program as well. Pea (1983) describes programming and problem solving in children as young as 8 years old extending work from Papert et al. (1978) who worked with 6th graders learning the LOGO programming language. Our wider research program looks at the early development of Computational Thinking (CT) in even younger learners, starting in Kindergarten and extending into the second grade. While Papert and Pea's learners spent time directly programming computers, we have chosen to introduce programming concepts through both "unplugged" activities (not requiring devices) and formal instruction and "play" activities using simple toy robots. Limiting high-tech devices makes the curriculum easier to access in more classrooms and allows focus on specific concepts rather than managing technology. Ironically, in 1986 it was noted that students in primary school could "use a machine for only one or two hours a week… [teachers, were] new to the enterprise… [and] programming as a recent subject area has a relatively undeveloped pedagogy" (Perkins, Hancock, Hobbs, Martin, & Simmons, 1986, p. 38). In 2018 circumstances have not dramatically changed. While computers may be more available in some classroom, time is often limited by the pressures of high stakes testing, teachers are still generally untrained in programming and managing/maintaining devices, and formal curriculum on programming is limited/emerging. An unplugged and 'low tech' approach can still build precursor skills while reducing the demand on teachers and resources while meeting state requirements for programming curriculum. Most importantly, it allows introduction of basic programming and CT competencies in a controlled and hopefully more successful platform.

The most important development for a novice programmer may be fostering a sense of self-efficacy, if not a passion for programming. Identity is often discussed as a limiting factor in the inclusion of underrepresented groups in STEM and CS education (Guzdial, 2015; Margolis, Estrella, Goode, Holme, & Nao, 2010; Mercier, Barron, & O'Connor, 2006). Teachers have long stereotyped students into two group when it comes to programming: "Johnny can do anything, but Ralph just can't seem to get the hang of it" (Perkins et al., 1986, p. 37). Some students seem to immediately pick up the concepts while other persistently struggle. Achievement may be impacted by many socio-economic factors yet Perkins et al. found "[a]nother factor clearly affecting many students is their attitude toward making mistakes" (1986, p. 43). Novice programmers must repeatedly make mistakes as a natural part of learning, yet how they perceive making mistakes may be critical to their success. In many subjects mistakes are a sign of failure. Students who fail to memorize facts and procedures (e.g. spelling, grammar, multiplication tables) are less apt, yet in being a programmer may require an alternative measure of success than counting correct and incorrect attempts. Programming does require the memorization of details but equally requires improvement through trial and error. Papert et al. in the 70's observed students learned through experimentation and play, noting "[p]resumably kids need this experience, no matter what the medium in which they are working" (Papert & others, 1978, p. 71). Yet the 6th graders with which Papert et al. were working were not able to successfully learn from experimentation alone. The way a programmer learns may require a difference sense of progress and success than other traditional subject matter and assessment approaches.

Novices and the Notional Machine

The notional machine is a theory for how we mentally model the workings of a computer language. A notional machine is "an idealized, conceptual computer whose properties are implied by the constructs in the programming language employed" (Du Boulay, O'Shea, & Monk, 1981, p. 237). Each learner develops their own mental model describing how a computer language executes on a device. The notional machine embodies the mental transformation of a programming language's syntax into the resulting execution on a computational device. Some theorize a novice must develop a notional machine in order to write programs (Khalife, 2006). Learning to program may be particularly difficult thing for many people. While humans naturally learn language, "instructing a computer is an 'unnatural' activity and not at all like instructing a person" (Du Boulay, O'Shea, & Monk, 1999, p. 239). Often novices are capable of replicating examples, but struggle to fix problems or create new projects. Even if they understand syntax rules and can type code, they do not seem to learn from trial and error. Papert et al. blame the inability of students, who otherwise are successful in procedural coding tasks, to learn from experimentation is "not surprising from Piagetian work" (Papert & others, 1978, p. 70), though do not explain the implications further. Perhaps novices struggle to construct experiments because they do not understand all a computer can do, rather than lacking Piagetian developmental maturity? Experiments are generally devised from a theory of how the observed system works, yet a computer is often a magical 'black box' that seems limitless in its potential actions. How can novices experiment if they lack a "gut feel" for what they should/could be doing with a programming language?

Bruner (1966a) describes a “gut feel” for a subject as an enactive representation. An enactive representation provides a description how the world works from which inferences or experiments can be conceived. Bruner uses the example how infants (and primates) initially learn using their senses (primarily, but not limited to vision), to form impressions of the world. Young children first understand objects by seeing and touching them and will “reach out” to an object to perceive it. The infant sees the toy and reaches out to hold the toy, but without seeing the toy it does not exist. It is not until later in the first year do they understand an object exists even without holding or seeing it. Papert et al.’s students are unable to experiment because they cannot “see and reach out” to constructs of the LOGO programming language because their only artifact is static and unmoving code disconnected from the actions on a screen. A programming language uses complex symbols to eventually tie back to behavior, thus Bruner might say they lack an enactive representation of how LOGO works until they connect the language to the actions induced. The symbolic representation is the highest form of the mental construct, so students are ‘forced’ to work backwards, starting with concept to develop their ‘gut’. Yet novices likely require an enactive representation to reach the developmental maturity needed to transfer coding procedural skills into solving new problems.

A rudimentary notional machine may be a prerequisite for the self-exploration Papert sees as critical to learning to program. Without an intuitive understanding of a programming language’s syntax and capabilities a novice may be unable to experiment. To form a programming experiment, the novice must understand how the construct could vary and how to create the code to test their variation. To experiment is to predict an outcome, thus novices who lack a notional machine seems unlikely to create much less benefit from a predefined experiment. Without a prediction, how can they be surprised with the results as all outcomes are equally plausible? McCauley et al. belive, “students learn from their mistakes only when the causes of the faulty mental models causing the errors are understood” (McCauley et al., 2008, p. 68). While experimentation is critical, it seems teachers must first focus on instilling and maturing a notional machine in students to enable leaning through experimentation.

The lack of a notional machine may describe other issues of learning and identity in learning to program. The presence of a strong notional machine may be the dividing line between those who easily pick up programming and those who struggle. Perkins (1986) describes two types categories: movers and stoppers. When “stoppers” cannot quickly move past an issue they encounter, they freeze and stop learning. Movers seem to embrace a challenge, almost reveling in creating methods to unravel how the language works. The development of a sufficient notional machine could be the dividing line between movers and stoppers. A basic notional machine gives movers confidence to experiment. “Tinkerers are by definition ‘movers’” (Perkins et al., 1986, p. 48). Movers use their notional machine to compare their expected results with those of their program and explore options to correct mistakes. Stoppers, if lacking a notional machine, are likely unable to see faults within the steps of the execution. Pea notes, when novices do not sufficiently understand a program they have written, they prefer to “rewrite [the] program from scratch rather than to suffer through the attention to detail required in figuring out where a program was going awry” (Pea, 1983, p. 7). Stoppers find it is easier to start from scratch rather than to ‘sort out’ errors. The lack of a notional machine, means stoppers do not experiment at the “line level” of the code, but rather as a whole approach to problem solving. The notional machine seems to provide the tools to enable other fundamental skills in learning to program.

Methods for Incubating a Notional Machine

The concept of a notional machine is well documented, yet so are the pitfalls in developing and maturing it. Du Boulay et al.'s suggests instruction, yet admits "[one] of the difficulties of teaching a novice how to program is to describe, at the right level of detail, the machine he [sic] is learning to control" (Du Boulay et al., 1981, p. 237). Developing a notional machine is a daunting task, as programming languages provide a staggering combination of instructions and ways of employing them. Du Boulay et al propose the best path to developing a notional machine is through careful instruction, which is the traditional approach to programming pedagogy. Du Boulay et al. continue by noting that teaching a simpler notional machine is not enough to assist novices, as there are always layers of abstraction hidden from the learner. Du Boulay et al.'s plan for instilling a notional machine likely fails because it assumes fact can be assembled into a working mental model. Many novices fail as "[m]ental models are often not the product of deliberate reasoning; they can be formed intuitively and quite unconsciously" (Sorva, 2013, p. 8:9). Developing a notional machine seems more like other procedural tasks, such as riding a bike. "If you have tried to … teach a child to ride a bike, you will have been struck by the wordlessness and the diagrammatic impotence of the teaching process" (Bruner, 1966b, p. 10). Bruner points out how useless instructions can be for some tasks which are easier learned through practice and repetition.

Bruner's theory gives an educational psycology model for learning which seems to be supported by biological models from neuroscience. Kahneman (2011) describes the brain's capability to quickly and effectively execute regular procedural tasks as "System One". System One allows mental tasks to be completed quick and effortlessly yet requires significant investment of time to properly train and mature. An experienced programmer effortlessly simulates chunks of code, not needing to recall the specific rules for each construct, despite clearly demanding the understanding of logic. System One provides automatic recall for simple tasks like 12 times 12, where Kaheman describes "System Two" as our rational, rule-based mind, which would be employed to procedurally calculate 1734 time 934. The notional machine, like basic arithmetic or spelling, is most effective when it is quick and automatic, not demanding focused thought and planning. The instruction Du Boulay et al suggest is would only suffice to build a System Two understanding which demands attention and focus. Novices can successfully use System Two, meticulously working through each statement, at the cost of demanding significant cognitive load (Plass, Moreno, & Brünken, 2010). Movers, like experts, seem to be unencumbered by mental stress in detangling line-by-line execution, which seems to imply a mature notional machine is a System One function. If Bruner and Kahneman's theories hold true, developing the notional machine requires altering common pedagogical approaches.

One popular approach to developing the notional machine beyond instruction is tracing existing code to predict the expected results.

> "Another fruitful line of attack is by implementing a language in such a way that either pictorial or written traces can be displayed that comment on the actions taken by the notional machine during a program run. " (Du Boulay et al., 1999, p. 242)

Tracing allows the novice to be guided through a symbolic representation (e.g. flow chart or source code) with tips on what would be happening in the real device. Yet Sorva observes "[it] seems likely that the novice programmer who merely studies visualizations of the computer running programs is less likely to become a good "notional machinist" than the another who actively engages with the program runtime" (2013, p. 8:22). Walking through code and watching, or even predicting behavior only goes so far to mature the notional machine. Kahneman (2011) describes learning, particularly in System One, as happening through repeated and varied practice. Tracing simulates execution of the code, but with little consequences (other than academic) if the prediction fails. Students will often use a 'gut feel' to form a bad guess over working through a program line by line, because System One guesses are easy and System Two requires focused attention that must deemed important and fruitful. A student lacking a mature notional machine and overwhelmed by the logic required to process code in System Two will likely stop rather than dive in deeper, as we see in the tendency to rewrite rather than debug programs. Research shows code tracing is not 'natural' to students (Perkins et al., 1986), they don't like it (Sorva, 2010), and it does not inevitably help them to later build their own code (Cunningham, Street, Blanchard, Ericson, & Guzdial, 2017). Instruction and code tracing may help in maturing the notional machine but are alone is not enough.

One assumption of working with extremely young learners perhaps is reducing the influences of prior experiences in using computers. Novice programmers are often considered a "blank slate". Pea notes "It is not that students don't know anything that is relevant to programming-they have an intuitive understanding of much of what we say about programming" (1986, p. 26). The concepts of programming have many analogs, but those analogies lack the precision and absolute obedience of a computer executing a program. Novices often believe generalized instructions are good enough for the computer to fill in the gaps, yet "[while] people are intelligent interpreters of conversations, computer programming languages are not" (Pea, 1986). Pea describes this misconception as the "superbug". Novices unconsciously believe that a 'right answer' is hidden in the code that the programmer can 'reveal'. The describe the computer an actor helping in solving the problem rather than obediently executing the exact instructions in the source code, for good or ill. Past studies have documented this and other misconceptions in novices (Mühling, 2014; Pea, 1986; Sorva, 2010).

Research Goals

This study is looking at some of the youngest plausible learners performing some of the simplest possible programming tasks to strip away other complexities and see what remains that is difficult. By using a simple language, the novices have less to remember and manage. By using a tangible task, we ask a minimal amount of visualization compared to most programming tasks. What is left is the essence of what it is to create an automated sequence in with highly abstract and easily understood commands, theoretical exposing basic misconceptions of what it is to program. The mistakes the participants make will either be easily corrected, or if repeated show foundational misconceptions which can categorized. Additionally, we can attempt to explain struggles as they relate to the concepts of the notional machine, Bruner's learning theory, and Kahneman's model of the brain to see if they provide a more accurate description of what it is to learn to program.

Methods

This paper performs a qualitative analysis in the spirit of Papert (1978) and Pea (Pea, 1986) on data collected from another study to be published. The participants include children in Kindergarten, First and Second grades attending an after-school daycare program and participating in CT focused exercises using the "Robot Mouse" toy from Learning Resources ("About Learning Resources," n.d.). Students were video recorded completing different programming exercises to direct a robot mouse through a maze to a piece of cheese, possibly completing other tasks along the way (e.g. traveling under a hoop, or to 'squeak' on a specific space). The original study tests how students use multiple representations to create a computer program (e.g. drawing a map, using direction cards to plan a route, and programming sequences in the mouse device). The other study defined the protocol for student activities, data collection, and used an alternative analysis.

For this paper, the videos were coded based on the success of the student outcomes, ignoring the medium. I compared their plan with the actual route(s) needed to successfully complete the task. Each unsuccessful attempt is further coded for the type of failure and/or the likely root cause of the failure using an open coding scheme seeking the range of possible misconceptions. While plenty of coding schemes on misconceptions exist in CS literature (Mühling, 2014; Pea, 1986; Sorva, 2010), this study attempts to, as much as possible, ignore existing categories of misconceptions to see if the traditional categories reappear in this simplified coding environment with such young and generally inexperienced learners.

Analysis of the findings occur both as a qualitative report of the types of mistakes and successful strategies learners use to program, and as a comparison with traditional findings of novice programmers. The aim is to compare misconceptions across age/experience groups and to evaluate theoretical frameworks around notional machine and how Bruner's learning model of enactive representations and the concept of the notional machine relate to children learning to program.

Results and Discussion

Activity and Exercise Overview

Students started the study by participating in a training session on the basics of using the robot mouse. The training sequence involved a pair of participants working with a researcher to learn and practice the basics of the mouse device, how to program the device and the types of problems to be solved. The device is a basic two-wheeled robot in the shape of a colorful mouse that has command buttons on its 'back' to program a forward or backward fixed distance move, rotate left or right 90 degrees, and make a 'squeak noise'. Two other buttons clear or execute the saved program. The programmer enters a sequence of movement/rotation/sounds and then can execute the sequence or clear it and start over. The researcher demonstrates the mouse on green plastic squares connected to form a map which can be customized into different configurations by adding purple walls, orange arches (under which the mouse travels) and a plastic wedge of cheese indicating the goal. Some tasks demand the mouse to travel to a specific square and squeak before continuing. The seven buttons make up the full 'syntax' of the 'programming language' (and thus notional machine) the participants are being asked to learn. The study also uses variations where the participants use cards which can be sequenced to plan the mouse's program. In some exercise a predefined route is provide which participants must predict the end location of the mouse.

Observed Misconceptions and 'Failures'

Participant failures can be categorized into three general causes, misconceptions about the domain, the language, or the participant's logic breaking down due to cognitive load, as broken down in Table 1. The most common mistakes related the domain related to spatial reasoning. Students often fail to compensate for prior moves by the mouse, instead programming turns from the original orientation thus a 'left' turn becomes 'right' relative to the mouse's new orientation. They also would move the mouse too few or occasionally too many forward steps as they planned their path. These are considered 'domain' failures, as they relate to the proper analysis of the environment and ordering of the path. The final domain failure relates to 'initializing the environment', or to ensure the map and mouse are properly setup. This does not directly relate to planning a program but did interfere as participants sometimes mistook a physical mistake (e.g. falling off the track) for a logical one in programming and would abandon a good program instead of trying again after properly starting the mouse.

The language rules failures are related to how the buttons translate into actions of the mouse, and thus are directly related to the notional machine. Users must understand how a program is saved, and when to clear (or not clear) the program. The programmer must understand how each button translates to motion on the table. As shown in Table 1, the most common of these is turning, where some students think the turn rotates and moves the mouse, where in fact it only rotates the mouse. It could be argued that some of the 'wrong number of step' domain issues could be misinterpreting the language. Some programming games will move forward to the end of a path on a command instead of 'one square' as the mouse does. There are no instances however of the students mentioning they thought it would go forward to the end of the track, and rarely did they make the domain mistake including a single forward move when multiple moves were required.

	Code	Description	Frequency
Domain Rules	Not physically set	Runs the program with the mouse askew or otherwise not set so it is not normal to the paths	Moderate
	Turn direction	Failed spatial reasoning has the plan turn right instead of left or vice versa	High
	Wrong number of steps	The plan includes too many or too few steps, going the right direction but the wrong distance	High
Language Rules	Does not clear	Program is re-entered, but the clear button is not pressed; the new plan is appended to the prior one	Low/Mod
	Excessive clear	Clears when a final step can be added to the plan, or clears and reprograms when nothing is wrong with the plan	Low/Mod
	Turn abstraction	Considers a turn to be not just a rotate, but a rotate and move to the next square	Low/Mod
	Cognitive overload	The number of steps required for the sequence, exceeds the programmer's ability to keep track of their intended design	Low

Table 1 Codes related to mistakes in the notional machine

The final category of failure occurred occasionally when students advanced to particularly longer sequences. Buoyed by early success, some participants jumped at advanced challenges abandoning the strategies that helped them to succeed and relying on memory alone. In these cases, their failure is being attributed to "cognitive overload" as they start strong, but the longer the sequence required the more likely they are to fail. For instance, one participant was highly successful in planning a route, yet after successful completing several maps, confused the meaning of 'turn' (rotate, not rotate and move) on one particularly long route with several intermediate goals. He had not made that mistake on any of the several prior turns but did at the very end after a dozen or more moves. It would seem the long route caused the mistake, rather than a misunderstanding. It is important to note the frequencies indicated in Table 1 are relative and not meant to convey quantitative meaning past some were more frequently found and repeated than others in the observations.

Interpreting 'Failures' through Theory

Young programmers more frequently make mistakes in domain rather than language, which aligns to the findings of Pea (1986). Fewer language failures seems obvious as the syntax consists of 7 buttons! The student's notional machine seems jumpstarted by the initial training, but only for tasks practiced in that training. For instance, students with a correct path, but missing the last step(s) would erase and reprogram the entire sequence, rather than simply adding the last step. The initial instruction did not demonstrate that the mouse's program could be extended, thus without specific practice, describing the function of each button did not create an understanding of the underlying mechanics. On the flip side, students get plenty of practice in clearing and reprogramming after a failure. Most of the failures would indeed require the old program to be cleared, but on multiple occasions students would clear the mouse and reprogram when the only known issue is the mouse fell off the track because it was placed poorly. All that was required was to let the program complete, place the mouse properly and re-execute, but habit is so deeply ingrained they participant would clear and reprogram, a slower and more error prone task, but less cognitively demanding than to stop and consider that the reprogram is not required. This is an example of Kahneman's 'lazy' System Two taking a back seat to the procedural habit developed in System One, and evidence the notional machine quickly is formed in System One. Whether we like it or not, how students practice will be more likely to form their understanding than the careful instructions provided.

The same example (falling off the track), in some cases provides evidence of a mover versus stopper. While some students reprogrammed out of habit, others would not simply reprogram, but also revisit the entire design. While Pea (1983) long ago observed this behavior, it is notable here that some of our participants could not even interpret a failure of hardware versus their designed software. It would seem obvious a mouse driving at an angle off of the track was unrelated to the planned program, but for some of our novices they did not distinguish the nature of failure, instead always resorting to starting from scratch. This may be better explained, as with the "habitual re-programmers", as a failure of instructional methods. As will be described later it is easy for the researcher/instructor to focus on the next problem-solving step, rather than to create a habit of mind to understand the failure before providing a fix. If we do not repeated practice analysis, the habit will be to start over rather than fix what we have, even when the 'code' is not particularly complex.

Instances of cognitive overload provide an interesting case in theory. If the notional machine resides as an enactive representation in System One then cognitive overload as we witnessed in the complex problem describe earlier should not be much of a factor once mastered. A System One response to 'turn' should not confused by a long sequence of events, as an expert program does not forget how loops work because of a particularly complex algorithm. In our case our participant has at most an hour of experience in using the device however and may indeed be splitting their plan across a maturing notional machine and double checking the plan in their rational System Two. Without asking, it is impossible to say how the mistake is made. It is important to note however as this could either a limitation of the model, or a supporting case that repetition is vital in learning.

Problem-Solving Strategies

Commented [AAL1]: Need to make sure this properly sets up this section

Participants adopted (from researchers) or created several problem-solving strategies to use in completing their programming tasks. It is important to differentiate the strategy for composing their design or algorithm, compared to the medium in which they capture their design. Since this study 'borrowed' the data, the original study had researchers introduce several ways learners could capture their design. Participants were asked to create plans by drawing maps or ordering cards with directional commands to plan their solution. This led to, or in some cases participants directly, entered the commands into the robot using buttons. The problem-solving strategy describes how participants decide the steps for their algorithm, not how they document them. It also describes strategies used to fix incorrect algorithms. This study is not comparing documentation styles, but simply is determining if the participant's solution is or would be successful and to categorize the mistake/misconception when not. The value of the problem-solving strategies, as captured in Table 2, is in describing the demonstrated process by which the student conceives of their work, which provides insight to their mental modeling of the problem and maturing of their process.

Code	Description	Frequency
One step at a time	Programs one step, and executes, then the next step and executes, typically not considering a plan	High
Walkthrough	Use the mouse or analog to trace the route on the map and create the plan	High
Cards on track	Place cards on the map to plan out the route	Low
Mental tracking	Trace the entire route in the mind and produce from memory	Low
Start from scratch	After a failed attempt, restart rather than debug	Moderate
Debug	Attempt to find the problem in the sequence and fix it	Low
Skip to Code	Don't' make a plan, but program the device directly	Moderate
Jump to solution	Skip the root cause of what went wrong, but jump to a fix and try it out	Moderate
Play/experiment	Try out plans/buttons/sequences unexpectedly to see how it works or what it can do	Low

Table 2 Problem solving and debugging strategies employed

Learners often solved the map by moving the mouse one step, or a few steps at a time. This could be *go forward*, then *execute*, then *clear, turn right*, then *execute*, then *clear, forward*, *forward*, and *execute*, for example, in a simple case, inching the mouse towards the solution but not capturing an actual sequence. This stepwise approach seems to build confidence and build an enactive understanding connecting the buttons (action) to how the mouse moves (results). Stepwise problem solving did not create deeper understanding of programming as participants did not develop longer sequences unless redirected. In fact, they tended to fall back into an advanced stepwise approach when the number of steps in a sequence seemed to exceed their ability to track and remember the path. The 'step by step' pattern returns in advanced students as they abstract a problem into constituent steps. For instance, one participant first directed his mouse to move to the spot he needed to squeak before then turning to the problem of going to the cheese. He took the goals of the problem as a level of abstraction to achieve one step at a time, much as other participants literally moving the mouse one step at a time. One step at a time seems to be a strong strategy for managing complexity but requires intervention to help the learner advance to the next stages.

The researchers pushed participants to program full sequences by requiring a plan before programming and execution. At this point most students still required a physical representation to design their path. Many students stepped the mouse or an analog object through the map and 'recorded' the next step using directional cards. Navigating on the board helped reduce the number of spatial reasoning errors, presumably as it demands less mental modeling of turns. In some cases the researcher did not allow the programmer to plan 'on the map' requiring the programmer to plan entirely in their mental model or attempt to use object in hand (or their hand) to simulate the orientation. These strategies were less successful. Being forced to remember position and orientation away from the track, even with the use of an object, often was too much to juggle and master in a single setting. These two approaches (on the track and off) seem to support the finding that domain issues (in this case spatial reasoning) were more of a challenge to the success of the programmers than understanding the 'language' over time.

Relating Problem Solving Approaches to Learning

The participants did not naturally take to 'designing' as a separate activity form coding. Unless directed otherwise, most would skip to directly programming the device. Initially first this could be a desire to 'play with the robot' but skipping planning persists through the hour session, each day, even after planning proved a successful strategy. The act of planning does not seemingly a natural trait in novices, as earlier literate documented as well. Given the nature of the mouse, the cards are the only way of capturing the algorithm outside the device, yet it is only used by participants to enter the program. As noted earlier, the preference, if a problem occurs, is to try again from scratch, so the participants generally used the cards only after being reminded. Without revisiting the past plan, it seems less likely the novice would update the notional machine based on traditional learning theory, as they spend no time reflecting on the mistake. Yet, perhaps slower than if they were reflective, they do gradually come to understand the controls and programming strategies for using the mouse. At times, adding a layer of design seems to initially hinder rather than support maturing the notional machine.

The strength of Bruner's and Kahneman's theories of learning applied to novice programmers is it describes how learning may be happening even with 'active resistance' to metacognitive methods. Often the focus of the participants, and sometimes the researcher, is achieving the goal. Little time, if any, is spent on meta-cognitive evaluation of which strategies are helpful to be reused later, or on reflecting ways the type of problems was solved in prior attempts. The lack of metacognition not surprising in learners so young, yet over time and repetition they do show growth! On each attempt they strengthen their enactive representation of the mouse and are better able to predict outcomes and thus invent new routes and with less dependency on the cards. If the notional machine is a logical function, the cards would become more important as the task becomes more complex, not less so. The enactive mental model, the notional machine, executed on System One strengthens and accommodates complexity by reducing the required cognition so long as the problem is similar. Using a cognitive model of programming, an external tool which reduces cognitive strain should be preferable to one that causes more work, yet one example shows this not to be the case. One researcher worked with participants to place the cards physically on the map to show the step by step execution, and then program the mouse. Using this approach, participants were easily able plan the route. The new strategy was dramatically successful and easily accessible for students, but novel, not their initial approach. Neither participant adopted this successful approach, but preferred the method they traditionally used, despite inferior results. Logical thinking says to abandon the lesser way and use the easier more productive methods. Yet System One does not often change based on a single observation, and in this case the students followed the researchers lead, rather than actively using the strategy. Since the participants were following the researcher's process only contributing small ideas, they did not adopt the full approach! Perhaps if they had been taught the full approach they may have chosen it, but in this case, it seems their enactive representation of how to program was the default over the cognitively superior model.

Bruner model says as students mature, they move into higher order representations, such as our cards. While most students eschewed the cards, one adept young lady found a way to make good use of them but with a conflicting result. This participant arranged her 'forward' cards to show the direction the mouse would be traveling on the map rather than the convention, to align the forward arrow 'up' (away from the participant) as the mouse is moving 'forward'. She used this technique to create a perfect route, but the researcher saw the 'wrong' card orientation and turned the forward arrows 'up' instead. The participant returned the cards to her orientation, which again the researcher 'fixes'. The researcher seems to be correcting the syntax error for the participant, but instead disrupts the participants mental model which conflicted with the "correct card representation" but aligned with the notional machine. Each 'correction' confuses the participant until she inevitable constructs a flawed route. Bruner's (1966a) full model has three levels of mental representations, enactive, iconic and symbolic. Our participant seems to hold a strong enactive representation (notional machine) and understands the mouse in action. She understands the simple symbolic representation (the buttons on the mouse) and how to use them to plot a route. Our clever participant created an iconic representation to include more information than is 'typical' for a design using the cards, also including an aid to spatial reasoning. Her design combined the symbol (going forward) with context in the domain (the orientation of the mouse on the map) yet did not change "what going forward means" in her notional machine. Inadvertently the researcher seems to have disrupted this link by forcing a specific design format. This participant was excelling at the task, yet this minor glitch in translating the notional machine to the physical space led to a misconception and bug in her program.

Open Play and Independent Experimentation

The research protocol included limited activities allowing the participants free time to 'play' with the mouse. Participants were encouraged to experiment with the buttons and corresponding motions. The paths generally were either very simple to test the commands or wildly random paths just to see what the mouse does, but not building to the idea of programming to meet a goal. The simple experiments are consistent with Papert's (1978) finding that student experimentation was generally unproductive to learning. Novices can discover a basic (enactive) understanding of how the button commands move the mouse but generally do not extrapolate the capability to create complex sequences to move mouse and achieve a goal without prompting. The step-by-step approach forms a basic understanding of the device but does little to spark deeper understanding. The random trials are so wild there is little observable control to allow a prediction to test if the route failed or why. Early play may be valuable in building a basic enactive understanding of the device but does little to mature the application of the notional machine. In this case, developmental maturity may be a limitation, as older learners may be more apt to find more complex tasks, yet literature seems to say otherwise as Papert did not observe value in random experimentation either.

On "Pair Programming"

While but was not specifically part of the research plan, there are interesting observations that can be made on how participants interacted while working on tasks. Each activity paired two participants to work with the researcher. "Pair programming" is a coding methodology often used in industry and education (Cockburn & Williams, 2001; Van Toll, Lee, & Ahlswede, 2007) to improve quality of work, and perhaps learning. For this study the pairs did not tend to spontaneously collaborate on the programming tasks. Some of the researchers instructed the pairs to divide the work where one participant would design and plan the route while the other would program the route into the mouse, but even this produced little interaction. The participants did little to discuss or collaborate on the route, instead just waiting for the design to complete and then program without critical review of the plan. For teams without identifying a given role, the 'second' tended to disengage, staring off, daydreaming, or passively observing the mouse activity. This age group did not seem to naturally work together in these activities.

Having a partner provided some benefits, but often was a drawback to learning. In several cases, two individuals are asked to complete the same task, but the second individual simply emulates the first's plan and quickly or even immediately produces a successful program. Alternatively, the second is entirely disinterested after seeing the goal met, seeing no value repeating. The task becomes one of memory, rather than practicing the intended skill, or is skipped altogether. The solution-focused attitude derails the much-needed practice and creates a misconception on how one learns to program. In one notable sequence, a solution-focused attitude disrupts both partner's learning. The first participant completes a design, close but one step short from correct. The second participant, programs the design but places mouse at a slight angle and the execution fails. Without any review of the plan, the second participant emotes "I told you that wouldn't work" and insists he "could not do it", becoming animated and loud. The researcher attempts to correct the plan, but the outburst overloads the designer participant. She alters the route but at the wrong place as she is further confused when the researcher started using the word "up" instead of "forward". The designer is the same young lady who wanted to use the alternative facing cards but was told to shift them, so when she hears up she is seemingly is unable to process both the change in verbiage and the outburst and abandons her plan. This case is an outlier but does raise questions about how working in pairs impacts novice learners in programming. If, as theory suggests, repetition is vital, and how early work is conducted impacts later habits, the value of pairs should be weighed against the need of individuals to work through problems. The correct solution is less instructional than the struggle of working through individual misconceptions, so it may be that working in pairs is better delayed until foundational skills are mastered.

Conclusion

Theoretical Findings

The theories of Bruner and Kahneman are useful in describing the learning of our young learners on programming tasks. Our participants worked with a simple device with a simple notional machine, yet still resemble early programming. The participants formed an understanding of the commands and actions as would be predicted by an enactive representation. We can see cases where repletion is more readily used than cognitively better approaches, suggesting the notional machine likely resides in System One rather than System two. It is practice, rather than reasoned logic and instruction that seems to mature the notional machine. We see some evidence of the Bruner's progressive representation as participants tend to reject advanced iconic representations until they are more comfortable with the basics. Until their notional machine matures, they do not, and possibly cannot, embrace the planning cards unless instructed to do so. This would suggest that the novices, and I would argue experts, find it easier to fall back on 'gut feel' and 'trial and error' rather than a reasoned and reviewed plan. Kahneman's (2011) would agree as System One requires less effort than System Two, and we tend. If the combination of neuroscience model for brain function and Bruner's model for learning hold true for larger pools of learners, it could suggest alterations to the practical approaches to pedagogy.

Practical Findings

Our young participants seem to mirror those both of young learners from 40 years ago as well as those of their older contemporaries. The basic and limited programming tasks in this study are achievable for the learners ranging as young as K-2. The mistakes they make come from misconceptions related to understanding how to program, much more than the language. Until they develop an intuitive feel for the problem space and how the device works, rather than logical reasoning of how programming works, they make inconsistent progress. Generally,

- Learners require a strong understanding of the connection between 'commands' and 'actions'. This is the basic idea of the notional machine.
- Learners also need a strong mental model of the domain. In this study they failed more often due to problems in spatial reasoning and setting up the physical environment properly than they did with misconceptions of the notional machine.
- It is important to promote habits of learning, rather than solving the problem. Learners are so often goal oriented they miss the fact there are multiple possible solutions, and the act of making their own solution is more valuable than, in our case, the mouse getting its cheese.

In general these ideas align with those of the older learners studied by Pea et al. (1986), but by stripping away most of the complexity in the language and environment and may get at some of the other aspects of learning to program. Our participants were dealing with physical devices and very tangible solution, instead of a virtual 'turtle' of the LOGO language. Even in our small study with a short time frame, we could see evidence of Perkin's (1986) movers and stoppers, as we saw with our young man's frustration and outburst. We can see how simple strategies and actions by researcher supported or hurt identity and self-efficacy. Learners of all ages could possibly benefit from formal and informal activities which generate and mature a 'gut feel' for how a language works and then how it connects to the commands and syntax. For young learners developing confidence in using the technology and an identity as someone capable of doing so is a valuable precursor skill towards later learning in programming.

Limitations and Next Steps

The main limitations in this research are the small number of trials and the 'second hand' nature of the data. In total just over five hours of video were analyzed with about 60 failed attempts, which means there are many more ways students could show misconceptions. The observations were cultivated from valuable coding activities but planned for a different research question. There is no way to gather a better sense of how often or likely a misconception is to occur, or more than a theoretical suggestion of what pedagogy would discourage or promote 'proper' notional machine development. These limitations provide a nice starting point for next steps. A study designed specifically to examine misconceptions found here and others not visible in our activity set could be constructed with additional interaction to capture the thought process of the learners as appropriate. Additionally, work could be done to see which methods of pedagogy best improve the notional machine in young leaners. Allowing participants to work individually could reduce the 'noise' helping or hurting them in solving problems as well as seeing which approaches they prefer and how they mature over time.

References

About Learning Resources. (n.d.). Retrieved May 2, 2018, from https://www.learningresources.com/category/company+information/about+us.do?code=CAT-HSH11

Bruner, J. S. (1966a). On cognitive growth. In *Studies in cognitive growth: A collaboration at the center for cognitive studies* (pp. 1–29). Wiley and Sons.

Bruner, J. S. (1966b). *Toward a theory of instruction* (Vol. 59). Harvard University Press.

Cockburn, A., & Williams, L. (2001). The Costs and Benefits of Pair Programming. *Extreme Programming Examined*, 223–243. https://doi.org/10.1108/00012530210448235

Cunningham, K., Street, C., Blanchard, S., Ericson, B., & Guzdial, M. (2017). Using Tracing and Sketching to Solve Programming Problems, 164–172.

Du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, *14*(3), 237–249. https://doi.org/10.1016/S0020-7373(81)80056-9

Du Boulay, B., O'Shea, T., & Monk, J. (1999). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Human-Computer Studies*. https://doi.org/10.1006/ijhc.1981.0309

Guzdial, M. (2015). Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics*, *8*(6), 1–165. https://doi.org/10.2200/S00684ED1V01Y201511HCI033

Kahneman, D. (2011). *Thinking, fast and slow*. Macmillan.

Khalife, J. T. (2006). Threshold for the introduction of programming: providing learners with a simple computer model. *28th International Conference on Information Technology Interfaces*, (September 2006), 71–76. https://doi.org/10.1109/ITI.2006.1708454

Margolis, J., Estrella, R., Goode, J., Holme, J. J., & Nao, K. (2010). *Stuck in the shallow end: Education, race, and computing*. MIT Press.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, *18*(2), 67–92. https://doi.org/10.1080/08993400802114581

Mercier, E. M., Barron, B., & O'Connor, K. M. (2006). Images of self and others as computer users: The role of gender and experience. *Journal of Computer Assisted Learning*, *22*(5), 335–348. https://doi.org/10.1111/j.1365-2729.2006.00182.x

Mühling, A. M. (2014). Investigating Knowledge Structures in Computer Science Education.

Papert, S., & others. (1978). Interim report of the LOGO project in the Brookline public schools: An assessment and documentation of a children's computer laboratory. *Artificial Intelligence Memo*.

Pea, R. D. (1983). Logo Programming and Problem Solving. *Conference Paper*, *150*(ir 014 383), 1–10.

Pea, R. D. (1986). Language-Independent Conceptual "Bugs" in Novice Programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689T-1R2A-X4W4-29J2

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of Learning in Novice Programmers. *Journal of Educational Computing Research*, *2*(1), 37–55. https://doi.org/10.2190/GUJT-JCBJ-Q6QU-Q9PL

Plass, J. L., Moreno, R., & Brünken, R. (2010). Cognitive Load Theory. *Americas*, *32*, 10013–2473. Retrieved from www.cambridge.org%5Cnwww.cambridge.org/9780521677585

Sorva, J. (2010). Reflections on threshold concepts in computer programming and beyond. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research - Koli Calling '10*, 21–30. https://doi.org/10.1145/1930464.1930467

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. https://doi.org/10.1145/2483710.2483713

Van Toll, T., Lee, R., & Ahlswede, T. (2007). Evaluating the usefulness of pair programming in a classroom setting. *Proceedings - 6th IEEE/ACIS International Conference on Computer and Information Science, ICIS 2007; 1st IEEE/ACIS International Workshop on E-Activity, IWEA 2007*, (Icis), 302–307. https://doi.org/10.1109/ICIS.2007.96