How Block-based Languages Support Novices

A Framework for Categorizing Block-based Affordances

David Weintrop
UChicago STEM Education
University of Chicago
Chicago, Illinois, USA
dweintrop@uchicago.edu

Uri Wilensky
Center for Connected Learning and
Computer-based Modeling
Learning Sciences and Computer Science
Northwestern University
Evanston, Illinois, USA
uri@northwestern.edu

Abstract *The ability to express ideas in a computationally meaningful way is becoming increasingly important in our technological world. In response to the growing importance of computational literacy skills, new intuitive and accessible programming environments are being designed. This paper presents a framework for classifying the ways that block-based introductory programming environments support novices. We identify four distinct roles that these graphical languages play in the activity of programming: (1) serving as a means for expressing ideas to the computer, (2) providing a record of previously articulated intentions, (3) acting as a source of ideas for construction, and (4) mediating the meaning-making process. Using data from a study of novices programming with a custom designed block-based language, we provide examples of each role along with a discussion of the design implications of these findings. In doing so, we contribute to our understanding of the relationship between the design of programming representations and their ability to support computational literacy. The paper concludes with a discussion of the potential for this framework beyond block-based environments to programming languages more broadly.*

Keywords *Block-based Programming, Cognition, Design, Learning*

1. Introduction

The skills and practices associated with computational thinking are critically important for learners in order to be full participants in our increasingly technological world [1–7]. Central to computational thinking is the ability to encode ideas into representations that can be executed by a computational device. Through mastering these skills, computational thinking can be infrastructural to learning across diverse domains and open pathways to new forms of expression. In this way, we align computational thinking with diSessa's [1] notion of computational literacy, which envisions a citizenry that are both consumers and producers of computational artifacts.

A key component for both comprehension and generation of computational artifacts is the representational infrastructure that mediates these processes. This historically has taken the form of text-based programming languages, but can also include visual programming languages or graphical interfaces that support the assembly of instructions [8], or applications designed to interpret drawings or glyphs created by the user [9]. Block-based programming languages in particular are becoming increasingly common in introductory programming contexts [10].

Each of these representational systems achieves the same ends (defining instructions for a computer to follow), but does so through very different means that directly influence the process. The characteristics of a representational system, including the visual presentation, syntax, relation to other representational systems, and expressive power, have a direct influence on how one goes about accomplishing a task and the resulting understanding that develops from that experience [1, 11, 12]. With the emergence of new forms of end-user programming languages and human-centered interfaces, providing a framework for categorizing the ways that representational tools facilitate these ends is important as it provides structure to understand the various roles that features of introductory programming languages play. Further, it can be used to improve the current generation of programming tools and inform the design of the next generation of expressive computational media.

In this paper, we present a framework for categorizing the various ways novices use block-based programming languages to express their ideas in a computational medium. Through analyzing novices playing a program-to-play constructionist video game, we identify four distinct usages of the programming language: (1) serving as a means for expressing ideas to the computer, (2) providing a record of previously articulated intentions, (3) acting as a source of ideas for construction, and (4) mediating the meaning-making process. This paper situates these roles in a larger framework and presents vignettes from a study to demonstrate what each use looks like when enacted. The contribution of this work is the development an empirically grounded framework that can be used to structure the study of block-based programing languages, advance our understanding of the learning that takes place through their use, and inform the design of future programming tools and expressive computational technologies. In the conclusion of the paper, we expand our focus to include non-block-based programming languages and discuss the potential broader applicability of the presented framework.

DOI reference number: 10.18293/VLSS2017-006

2. Orienting Framework

The constitutive role of language and tools on cognition has long been a topic of research. A central theme of Vygotsky's sociocultural theory of mind was the claim that mental functioning is mediated by tools and signs. "The sign acts as an instrument of psychological activity in a manner analogous to the role of tool in labor" [13, p. 52]. diSessa [1] calls this *Material Intelligence*, saying "we can instill some aspects of our thinking in stable, reproducible, manipulable, and transportable physical form" (p. 6). Work looking at the relationship between signs (or more broadly representations) and cognition has delineated the particularities of how representations are bound up with knowledge, learning, tasks and uses [1, 11, 12, 14-16]. Similar work focusing on the design of programming languages has shown how various features of the representation, be they visual [17, 18], semantic [19], or syntactic [20], all influence the ease of use of the resulting language.

In their work on the development of mathematical meaning in computational settings, Noss & Hoyles [21] developed the theoretical construct of *webbing* to capture the nature of the learning process in rich computational settings. Webbing describes a "structure that learners can draw upon *and reconstruct* for support – in ways that they choose as appropriate for their struggle to construct meaning" [21, p. 108]. The construct is intended to capture the full, interconnected set of resources available to the learner as they progress through their meaning-making endeavor and respects the fact that each learner is unique and will leverage different features of the language in different ways. Webbing was proposed as a way to describe how understanding emerges that is consistent with the situated nature of the learning task and acknowledges the central role of the tools used in the process. This construct is particularly valuable when analyzing the role of block-based programming languages in introductory learning environments as it provides a way to makes sense of the full set of features of the language (semantics of keywords, visual display, syntax constraints, etc.) and identify the differing roles they play during the learning process and across learners [22]. Likewise, it does not demand that each component of an environment be considered in isolation, a challenge often encountered when trying to study block-based programming environments [23]. Bringing this analytic lens to the study of block-based programming environments reveals that the language primitives and their presentation play a variety of roles in helping novices achieve their goals.

In this work, we bring a representation-as-mediational-means lens to block-based programming languages. As such, the unit of analysis for this work is not the individual blocks, nor the full library of blocks provided by a block-based environment, but instead, the unit of analysis is the block-based environment in conjunction with the user interacting with it. This is consistent with the theoretical construct of webbing and recognizes the central role of the learner in the learning experience. Thus, the framework and the examples provided, treat user and tool as co-constituents in the ongoing learning process. This lens brings specific features of the language (sematic and syntactic) into focus alongside the environment in which it is situated (programming activity and interface) and the unique experiences and prior knowledge of the learner.

Block-based programming environments leverage a programming-primitive-as-puzzle-piece metaphor that provides visual cues to the user about how and where commands can be used as their means of constraining program composition. Programming in these environments takes the form of dragging blocks into a composition area and snapping them together to form scripts. If two blocks cannot be joined to form a valid syntactic statement, the environment prevents them from snapping together, thus helping to alleviate difficulties with syntax while retaining the practice of assembling programs instruction-by-instruction. Block-based languages, unlike more conventional text-based languages make the atomic unit of composition a node in the abstract syntax tree of the program, as opposed to a smaller element (i.e. a character) or a larger element (like a fully formed functional unit). In making the abstract syntax tree node the constructible unit, the building block of the representation shifts, giving the user a different set of objects-to-think-with [4], and thus providing a different set of supports and enabling different types of uses relative to text-based alternatives. Understanding and giving structure to the new roles and affordances of the block-based modality is the central objective of this paper.

In formulating our framework for categorizing the ways that novices use block-based languages, we looked to the literature and found two distinct dimensions along which mediational roles differ that could lead to a productive classification that fit our emerging findings. Kaput [24], in his work on the roles of symbols in mathematics, identifies two complementary uses for the material form of mathematical expressions: "the support of internal cognitive processing and communication between persons" (p. 160). We categorize this difference as internal (cognitive) vs. external (communicative); these categories provide the first dimension of our framework. The second dimension along which programming representations can differ comes from the computer science education literature, where a distinction is made between the act of generating a program and that of comprehending one [25]. This difference in purpose (generative vs. interpretive) forms the second dimension of our framework, producing a 2x2 matrix (Table 1).

Table 1. The 2x2 matrix situating the four roles Block-based programming language primitives play in supporting novices.

	Generative	**Interpretive**
External (Communicative)	Means for expression	Record of previously expressed intentions
Internal (Cognitive)	Source of Ideas	Resource used in meaning-making

The four quadrants of this framework delineate the four roles we identify in our analysis. The External-Generative role is the one most closely aligned with the conventional view of the

purpose of programming languages: that of an expressive medium with which to encode ideas in a computationally executable form. In this role, the user conceives of a general idea or specific intention, and then uses the programming language to mediate the expression of that idea into a form the computer can carry out. The second identified use of the block-based representation is serving in an External-Interpretive role. In this capacity, the modality acts as an external record that preserves previous intentions, serving as the *memory* in the distributed cognitive system of the programming environment [26]. Unlike the first role, which defines the human-to-computer interaction, this role captures asynchronous human-to-human communication in the form of one user reading the instructions previous written by others. A third role that language primitives can play is acting as a source of ideas for constructions, which defines the Internal-Generative quadrant of our classification. In this role, the representational system is not mediating the expression of an idea, but instead, the language itself acts as a resource the user can leverage to form new ideas. Block-based languages are particularly well suited for this role given the way they are presented, as will be shown later in the paper. The final role of this orienting framework is Internal-Interpretive, which manifests itself as novices using the language as a cognitive resource to make sense of observed behaviors. In this role, the author uses the programming commands as a mechanism to help decipher and interpret observed behaviors of the program, serving as objects-to-think-with [4] in facilitating the meaning-making process.

While we see these four roles as distinct, in practice, they are often used in conjunction or quick succession as part of a single effort. We see this ontology as productive in that each dimension suggests a pattern of use for novices and provides a lens for studying the ways the representational system is being appropriated by the learner. Further, the application of this framework can be used to inform the evaluation and design of programming languages. This framework is not meant to be definitive, but instead is one possible way to categorize novice interactions with programming environments.

Finally, the framework was derived with block-based programming environments in mind, but may provide insights beyond block-based contexts. This aspect of the framework will be revisited at the conclusion of the paper.

3. Methods

To develop and validate this framework, we conducted a study asking programming novices to play RoboBuilder [27], a constructionist, program-to-play game [28] in which writing programs is the main mechanism of gameplay (Fig. 1). The central challenge of RoboBuilder is to design and implement strategies to make an on-screen robot defeat a series of progressively more challenging opponents. A player's on-screen robot takes the form of a small tank, which competes in one-on-one battles against opponent robots equipped with the same set of capabilities. Unlike a conventional video game where players control their avatars in real time, in RoboBuilder, players must program their robots before the battle begins. To facilitate this interaction, RoboBuilder has two distinct components: a graphical programming environment where players define their robots' strategy, and an animated battleground where their robots compete (Fig. 1). To implement their strategy, players use a domain specific, block-based programming language. The language includes movement blocks (ex: `forward, turn gun right, fire`) to control the robot's motion, event blocks (ex: `When I See a Robot, When I Get Hit`) to control when instructions will execute, and control blocks (ex: `Repeat, If/Then`) that can be used to introduce logic into the robot's strategy. RoboBuilder uses an event-based programming model where in-game events are linked to the language's event blocks, so that when a certain action occurs (like the robot hitting the wall), flow of control of the program is passed to the associated event (`When I hit a wall`).

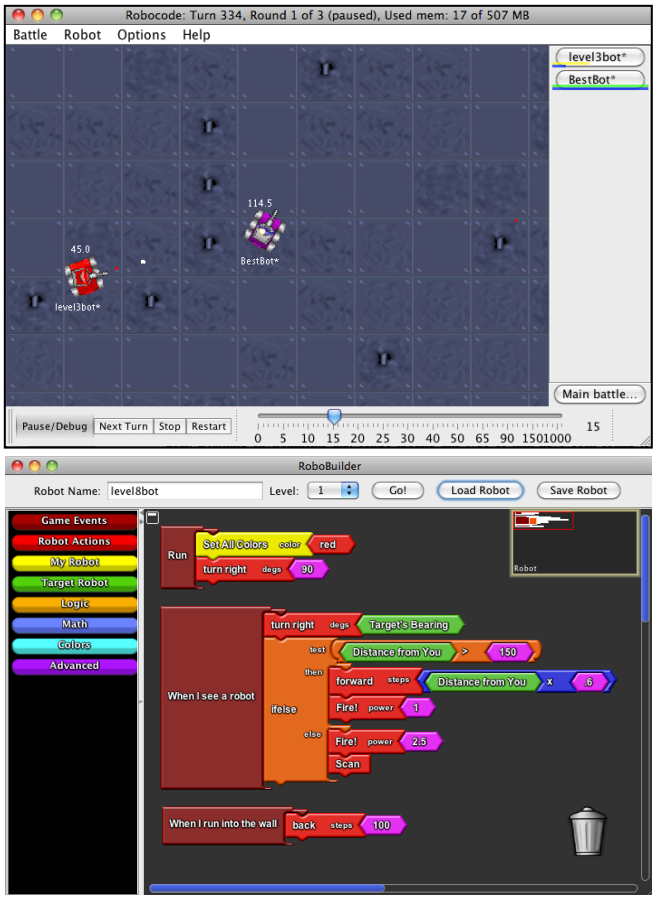

Figure 1. RoboBuilder's two screens. The battle screen (top) where players watch their robots compete and the construction space (bottom) where players implement their robot strategies.

The data presented in this paper are from 16 RoboBuilder play sessions conducted with programming novices ranging from middle school to graduate school. The university-aged participants were students at a Midwestern American university. Two of the younger participants were recruited through university connections, while the remainder of the participants were recruited through a community center in a Midwestern city that serves a predominantly African-

American, low SES population. Each participant played RoboBuilder for at least 40 minutes, resulting in a total of roughly 18 hours of interview and gameplay footage and over 200 robot strategies created.

The data were collected through one-on-one interviews in which a researcher sat alongside the participant as he or she played the game. At the outset of a session, the interviewer introduced the participant to RoboBuilder, explaining the game objective and the components of the game environment. The participant was then given a chance to ask questions before the actual game play procedure began. The gameplay portion of the session proceeded in an iterative, three-phase protocol. First, players are asked to verbally explain their intended strategies to the interviewer in conversation. Next, they are given the chance to implement their proposed strategy using the block-based language. Finally, participants click the 'Go' button, and then watch their robot compete, with the interviewer asking them to describe what they observe and whether or not it matches their expectations. At the conclusion of the battle, the next iteration of the protocol would begin with the interviewer asking participants what alterations they plan on making to their strategy to progress in the game. This three-phase cycle was repeated for the duration of the session. Throughout the session, the researcher's role was mainly to move the iterations forward by using various prompts to get participants to verbalize their thought process. The researcher also answered clarifying and technical questions when they arose. Each RoboBuilder session was recorded using both screen-capture and video-capture software.

4. Four Roles of Block-based Primitives

In this section, we provide vignettes and a discussion for each of the four roles of the framework. These vignettes are intended to demonstrate interactions for each quadrant of the framework and act as illuminating examples that can be drawn on to inform our thinking about how block-based languages support novices.

4.1. External-Generative: Primitives as an Means for Expression

In RoboBuilder, language primitives serving as a means for expression can be seen when a participant uses the language to implement an idea that he or she has conceived of, but not yet expressed in code. In other words, they are using the language to encode their intention so that the computer can execute them. An example of the block-based programming language playing this role involves Morris[1], a university student with no prior programming experience. At the outset of his interview, when asked what his strategy would be, Morris responded:

> *So, my master plan is to, like, be continuously moving, so it's harder to hit. If I get hit, kind of change the path so it's different than what you might be expecting however the sequence is running, and then, during that path, adjust to what the opponent is doing to hit them.*

He then proceeded with the construction of his robot strategy. After six minutes of working, he had produced his first program; the first three events of which are displayed in Fig. 2. Comparing the strategies Morris articulated in his initial remarks to the program he constructed, we can see the blocks taking on an expressive role, mediating and enabling the computational implementation of his ideas. His "master plan" included three distinct ideas, each of which can be seen in his resulting program. His first strategy: *"be continuously moving, so it's harder to hit"* is achieved with the `Run` method of his program (left side of Fig. 2). This series of instructions will result in his robot remaining in constant motion. Morris' second verbalized tactic: *"if I get hit, kind of change the path so it's different"*, can be found encoded in his `When I get Hit` event block (top right of Fig. 2). These two instructions will execute when his robot gets hit and will cause it to change its heading and move forward out of the current line of fire. His final idea: *"adjust to what the opponent is doing to hit them"* is captured by his `When I See a Robot` command (bottom right of Fig. 2), which makes his robot adjust its gun towards the location of his enemy and fire at it.

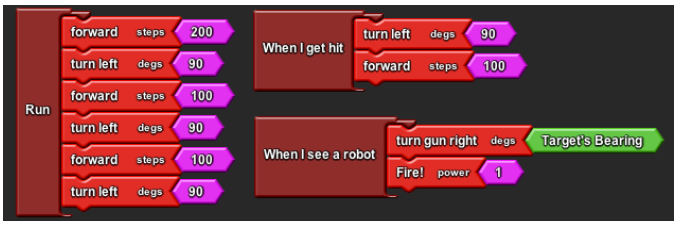

Figure 2. The first three events of Morris' initial RoboBuilder program.

From the first five minutes of Morris' RoboBuilder session we can see how the language primitives can serve as a means for expression. A second demonstration of the language serving in this capacity occurs roughly twenty minutes into Daniel's RoboBuilder session. Daniel is a tenth-grade student with no prior programming experience. After seeing his first two robot strategies struggle against the level-one opponent, Daniel decided he needed a new approach. He realized he was having difficulty locating his opponent; this prompted him to propose the following strategy: *"since* they *change the position of the robot every time, I won't know where it's at. So, I just want to make* [my robot]*, like, spin in a circle and shoot."* Having verbalized this new idea, Daniel proceeded to construct the strategy shown in Fig. 3.

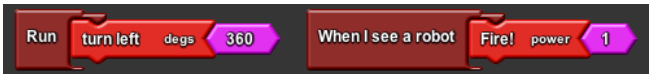

Figure 3. Daniel's implementation of his "*spin in a circle and shoot"* strategy.

The result of these commands is that his robot continuously rotates in a circle, shooting whenever the opponent comes into view. After trying out his new strategy, the interviewer asked

[1] All names are pseudonyms.

Daniel to describe what his robot was doing, Daniel responded: *"it's spinning in a full circle, and when he sees the robot he's shooting."* In other words, the robot is carrying out the strategy that Daniel had just vocalized. Here again we see the language primitives serving as a means of expression enabling the computer to carry out the intentions of the user.

These two vignettes were chosen because they provide clear demonstrations of the language primitives being used in the expressive capacity and serve as examples of the first identified role that language primitives can play in a programming activity: that of a mediating role between an idea generated by a user and a computationally executable reification of that same idea. This is a demonstration of language primitives being used in an External-Generative role, where the end result is a computationally executable form of the idea. It is important to mention that this idea-to-implementation process was not always so direct or easy. Often, over the course of our interviews, players either struggled to encode their stated intentions, or composed strategies that did not match their expressed intentions, at times relying on other features of block-based programming languages that will be discussed later in this analysis.

External-Generative: Discussion

The ability for a programming language to enable users to express ideas in such a way as to be executable by a computer is an essential feature of the representation, as, by definition, if it is not possible to write a program using the representational system, it can hardly be considered a programming language. That being said, it is certainly not the sole feature, and, arguably, not even the most important, as [29] famously says, "programs must be written for people to read and only incidentally for machines to execute". Programs, and programming languages, serving as a means of expression has long been argued as a pedagogical strength of the form [29]. This role is akin to the ability for the alphabet to be used to express ideas in the written form, the difference being in the case of programming languages, the audience is not solely another human, but also a computer.

In this way, programming languages serve as a bridge across what Hutchins et al. [30] call the gulf of execution, which describes the distance between a user's goals and the expression of those goals using the representations understood (and often defined by) the system. The design of the representational system can facilitate this bridging role "by making the commands and mechanisms of the system match the thoughts and goals of the user" (p. 318). In the case of RoboBuilder, to support programming novices in expressing their ideas with the provided representational system, the language primitives were designed to carry semantic meaning within the context of the game in such a way as to enable players to understand how they could be used. This can be seen in the close mapping between the verbal language of the player and the labels on the blocks, for example, Morris said: "*If I get hit"* and then used the `When I get hit` block.

In the first example, Morris relied on the natural language label on each block to select appropriate commands, the closeness of mapping to his intentions, and the shape of the blocks to facilitate his assembling them into a script. Daniel, along with these features, also used feedback from the environment in the form of seeing his opponent reposition itself, to inform the strategy he devised. All of these aspects have been identified as useful features of the block-based modality for learners [31]. These different supports designed into the language and environment contribute to the webbing upon which learners draw in order to support this first use of block-based languages. The two examples shown above highlight how not all users draw upon the resources available in a learning environment in the same way. In this way, block-based tools and their suite of scaffolds support an epistemological pluralism [32].

4.2. External-Interpretive: Primitives as a Record of Previously Expressed Intentions

The second role block-based languages can play is that of a record of previously expressed intentions, serving in an External-Interpretive capacity. After a user writes a program (i.e. uses the language in the previously discussed External-Expressive capacity), the language remains a visible, legible artifact that can later be referred back to and read either by the original author or other interested parties. Used in this capacity, the language serves as a record of previously expressed instructions, or as a resource to refer to for mapping outcomes onto expressed instructions. An example of this usage can be seen toward the end of Anne's RoboBuilder interview. Anne, a third-year undergraduate student, had just finished implementing the seventh iteration of her robot, during which she introduced the `When I get Hit` event to her strategy in hopes of addressing a weakness she had identified: if her robot got hit, it did not move; instead it stayed in place, making it easy for her opponent to hit her again. To address this issue, Anne decided to have her robot move to a new location if it got hit. Fig. 4 shows the two events from Anne's program that are relevant for this episode.

Figure 4. The two events of interest from Anne's robot strategy.

After starting a battle with this new behavior in place, her robot was behaving as expected until it was hit a few times in succession and backed into a wall. Her robot then remained pinned to the wall, motionless, getting hit until the match ended. Upon seeing this, Anne got a confused look on her face and said aloud: "*Wait, what happened?*" Not being able to make sense of what she was seeing based on what she remembered programming, Anne, speaking to herself, asked: "*Wait, but when I run into a wall, what'd I put?"* She then brought the programming window to the forefront and read through her instructions, quickly realizing the bug she had introduced. When her robot backed into a wall, her `When I Run into the Wall` logic would instruct her robot to back up an additional 300 steps; in doing so it hit the wall again,

thus producing an endless loop. To debug her strategy, Anne used the programming language in an External-Interpretive capacity; she read through the instructions using them as a record of her previously articulated strategy to identify the bug in her program.

External-Interpretive: Discussion

This vignette provides an example of the second role that programming language primitives can play during a programming task — that of a preserved record of the instructions followed by the computer that can later be referred to and analyzed. This use falls in the external dimension of our ontology as it relies on the communicative aspect of the blocks, but unlike the previous vignette, where the language was used in a generative capacity, here, Anne used the language to accomplish an interpretive goal. With computational representational systems, the primary audience for a constructed artifact is usually the computer on which it is going to be run, but there is also a secondary audience: any human tasked with interpreting, modifying, or extended the program. Because programs exist as sets of instructions that produce dynamic outcomes, it is essential for the language to support being read at a later time, either by the initial author or by others. Here again it is appropriate to cite [29] and their claim that "programs must be written for people to read and only incidentally for machines to execute". While it is being run, the written program serves as a blueprint, containing an explanation for the resulting behavior.

In this vignette, without referring back to her program, Anne was unable to make sense of what her robot was doing. To help her interpret its behavior, she re-read the program she had authored; using the language in a mediating role to provide guidance on what was happening. In this case, it was the original author who was reading her own code, but it is very common for programs written by one person to be read by others so they can understand, and ultimately use, or extend the program. In this way, programming languages serve as a means to mediate the expression of ideas as well as serve as a record of the ideas already expressed. Through the lens of webbing, the permanence of the constructed artifact, the previously mentioned closeness-of-mapping of the commands, and the visual execution of the program were all designed aspects of the environment that helped Anne debug her program. One goal for this framework is that it be useful for evaluating and improving programming environments.

In evaluating block-based programming's ability to be used in an External-Interpretive capacity, we see one potential direction for future improvement. Prior work has found that the block-based representation poorly supports longer programs [31]; as program length and complexity grow, the block-based modality can make the program more difficult to follow. In other words, block-based languages may struggle to support the External-Interpretative aspect of programming languages. In response to this drawback, new block-based tools are being designed to address this shortcoming by blending features of block-based and text-based modalities [33, 34] or by allowing users to move back-and-forth between modalities [35, 36].

4.3. Internal-Generative: Primitives as a Source of Ideas

When trying to develop an approach for accomplishing a desired computational goal, the language itself can be used as a resource. By internalizing the possibilities provided by the language, the author can use the language itself to bootstrap idea generation for potential solutions. This is one possible use of a programming language that falls in the Internal-Generative dimension of our framework. Block-based languages are especially well suited for this use as the visual arrangement and pre-defined categorization of the blocks make browsing and finding blocks easy. Our example of this usage comes from the start of the RoboBuilder interview conducted with Beth, an undergraduate student studying vocal performance. This was Beth's response to the initial question of how she was going to defeat her opponent:

> *Well, I...I don't know, it seems to make sense to have, to determine what would happen in every case, so I think I'll use these dark red buttons and try and figure out what I want to have happen.*

Beth then proceeded to go through each of the Game Events blocks (the "*dark red buttons*" she refers to in the quote), using them as a roadmap to develop her strategy. Fig. 5 shows Beth's first completed robot strategy alongside the Robot Events drawer that lists the available Game Event blocks.

Figure 5. On the left, is the Robot Events drawer; on the right is Beth's first implemented Robot.

What is especially interesting about Beth's first robot is that not only did she implement every event, but the order of the events in her program perfectly matches the presentation in the Robot Events drawer. The video from her interview shows Beth starting at the top of the events drawer and systematically working her way through the set of available blocks. This suggests that she did not have a clear, unified strategy when she began to program her robot. Instead, Beth built her program event-by-event, using the commands provided by the language to bootstrap the generation of a valid robot strategy. In this way, the language primitives supported Beth in conceptualizing possible actions that her robot could carry out.

Internal-Generative: Discussion

In this vignette, we see RoboBuilder's language primitives playing a distinctly different generative role than we saw in the vignettes in the External-Generative section. Whereas with Morris and Anne, the emphasis was on the language serving in an external and expressive capacity, with Beth, the primitives facilitate an internal, cognitive outcome; serving as a source of inspiration for generating ideas for her robot strategy. She even states her intention to use the language commands in this capacity, saying: "*I think I'll use these dark red buttons…and try and figure out what I want to have happen.*" Consistent with diSessa's [1] idea of "materially-mediated-thinking", in this episode we see Beth having ideas *with* the medium, as opposed encoding her preconceived ideas into the language. The language primitives are mediating her thinking about the challenge, seeding the ideation process for how to accomplish the in-game programming challenge. This use is further facilitated by the ease of testing and visualizing the behaviors of the blocks. The use of the language in this capacity also relates to Wilensky and Papert's [11] structuration theory linking representation and cognition, as the representation itself is making certain ideas more accessible. You can imagine that if instead of the descriptive blocks the game provides, the language was an abstract set of operations with labels like `operation1` and `state2`, then Beth would not have been able to use it in the way shown above, even if the language had the same computational capabilities. Here, the language serves in an Internal-Generative role, facilitating the generation of a new idea. When designing programming languages for novices, recognizing that primitives serve this role is important, as this use can help a novice achieve early programming successes. To the growing list of features that block-based languages include that support learners, we now add the organization and visual arrangement of the full set of blocks as another element of the webbing learners can draw on.

4.4. Internal-Interpretive: Primitives as a Resource Used in Meaning Making

The final quadrant of the framework describes programming languages serving in Internal-Interpretive roles. In this capacity, the language is used as a cognitive tool with which to interpret and make sense of the computational task at hand. Used in this way, the language need not be visible or even present, but instead is employed as a cognitive resource through which observed behavior can be understood. This vignette, also taken from Beth's RoboBuilder session, occurred during her second battle against the level-one opponent. The level-one robot's strategy is to remain motionless until its energy drops below 50, at which point it begins to move. At the start of the second battle, as Beth was watching the battle, she asked the interviewer when the opponent was going to start moving. The interviewer responded "*It happens at 50*", which prompted Beth to say:

> *It happens when it reaches 50? OK, so that robot must have something built into it when it reaches 50. OH! There we go, so that's what the, that's what the other boxes are for, so like if you reach a certain health level you can change the actions, oh, ok.*

This brief excerpt shows Beth using the language as a tool to mediate her understanding of the opponent's behavior without ever seeing the instructions externally represented. Her exclamation "*OH! There we go,*" suggests a moment of revelation, when some piece of the puzzle of how her opponent was behaving fell into place. She then explains that the "*other boxes*" (referring specifically to the conditional and robot state blocks, a fact that became clear later in the interview) can be used to create the behavior her opponent is carrying out. The key piece of this excerpt is her stating: "*if you reach a certain health level you can change the actions.*" This description maps perfectly onto the program that is controlling her opponent (shown in Fig. 6), but, importantly, these blocks are not visible to Beth, so she was unable to read the instructions, like we saw Anne do in the External-Interpretive vignette. Instead, she used the blocks as cognitive tools with which to interpret the opponent's behavior and devise a possible explanation for how its stationary-then-active strategy was achieved.

Figure 6. The hidden conditional logic inside the level-one opponent.

Internal-Interpretive: Discussion

In this fourth role, we see again how the language primitives can be used as objects-to-think-with [4] to support the meaning making process. This use fits with the diSessa's [1] Material Intelligence, where symbols serve as cognitive tools with which to make sense of the world. Likewise, it matches Kaput's [24] discussion of mathematical symbols and their role in delineating and providing structure for the mathematical activity at hand. What makes computational representational systems, and in particular block-based languages, especially capable for being used in the Internal-Interpretive capacity is their ability to offer a suite of resources, i.e. the webbing of the environment, to facilitate meaning making. This includes the ability to incorporate visual cues like color and shape that can make it easier to categorize how specific primitives can be used, and the embedding of existing, familiar symbol systems and representational conventions into the language's design, including natural language labels and mathematical symbols. This enables the set of primitives to include semantic hints in the form of meaning-carrying labels (such as `move forward` and `When I hit a wall`) that can bootstrap the cognitive process of interpreting observed behavior through the language itself.

5. The Challenge of Designing for All Four Roles

Recognizing the various roles programming primitives play has implications for designers of novice programming

environments and introductory programming languages. Attempting to design for all four quadrants of the Internal/External, Generative/Interpretive framework presents a challenge to the designer, as some design decisions made to support one usage may be at the expense of another. Each role suggests a different set of priorities and considerations for how the language should be designed and presented. An example from RoboBuilder's language makes this tension more concrete. The set of game events provided in RoboBuilder (`When I See a Robot, When I get hit,` etc.) were designed to provide conceptual hooks for players to introduce behavioral logic and enable them to use the blocks to guide the creation of strategies, as we saw in Beth's first vignette. However, by providing a fixed set of events, the language constrains how and when logic can be introduced in the game, limiting its expressive capabilities in the External-Generative capacity. This type of design decision comes down to a question of finding the right grain size for the language primitives. This challenge was encountered in the design of low-threshold computational modeling tools: "It is critical to design primitives not so large-scale and inflexible that they can only be put together in a few possible ways…On the other hand, we must design our primitives so that they are not so 'small' that they are perceived by learners as far removed from the objects they want to model" [37, p. 168]. Finding the right size primitives is one of the central challenges for designers when creating languages for novice programmers. Our decision to provide a standard set of events, as opposed to a customizable set, is an example of the design trade-offs one encounters when designing a representational system that can support all of the roles specified by this framework.

While the analytic framework we put forth in this paper was introduced and discussed as a means of understanding block-based languages, it need not be tied to that modality, as text-based or other graphical representations share these four distinct uses. While we expect the manifestations of the four quadrants would differ with other representational systems, we expect the framework would still be illuminating and fruitful.

6. Conclusion

When creating a new computational language for novices, a diverse set of uses should be considered. By providing a classification system for the roles block-based programming languages take in for novices, and providing examples of each, we seek to provide a set of aspects designers should consider when creating new computational tools. We also see this framework as a useful lens with which to analyze existing computational representational systems. Understanding how they are used is an important first step in refining existing and designing new tools.

In our use of webbing as a theoretical construct to ground the analysis, the findings were necessarily coupled with the block-based language under investigation, but it is easy to draw connections from this work to conventional text-based languages. Text-based programming languages provide the same fundamental capabilities as block-based tools, although at times the specifics may differ. As such, we believe this framework can be useful when applied to conventional text-based programming languages, but for now, this remains future work.

The creation of accessible, yet powerful, languages is a critical challenge we face in laying the infrastructure for the computationally literate society championed at the outset of this paper. By recognizing the various roles primitives can play in supporting novices in computationally expressing ideas, we as designers and educators can begin to develop new languages and environments that support these different usages to scaffold learners. In doing so, we can make progress toward this vision of a computationally literate 21st century.

References

[1] A. A. diSessa, *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA: MIT Press, 2000.

[2] M. Guzdial and E. Soloway, "Computer science is more important than calculus: The challenge of living up to our potential," *SIGCSE Bulletin*, vol. 35, no. 2, pp. 5–8, 2003.

[3] National Research Council, *Report of a Workshop on The Scope and Nature of Computational Thinking*. Washington, D.C.: The National Academies Press, 2010.

[4] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic books, 1980.

[5] D. Weintrop, E. Beheshti, M. Horn, K. Orton, K. Jona, L. Trouille, and U. Wilensky, "Defining Computational Thinking for Mathematics and Science Classrooms," *Journal of Science Education and Technology*, vol. 25, no. 1, pp. 127–147, 2016.

[6] U. Wilensky, "Modeling nature's emergent patterns with multi-agent languages," in *Proceedings of EuroLogo*, Linz, Austria, 2001, pp. 1–6.

[7] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, pp. 33–35, 2006.

[8] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[9] K. D. Forbus, R. W. Ferguson, and J. M. Usher, "Towards a computational model of sketching," in *Proceedings of the 6th International Conference on Intelligent User Interfaces*, 2001, pp. 77–83.

[10] C. Duncan, T. Bell, and S. Tanimoto, "Should Your 8-year-old Learn Coding?," in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, New York, NY, USA, 2014, pp. 60–69.

[11] U. Wilensky and S. Papert, "Restructurations: Reformulating knowledge disciplines through new representational forms," in *Proceedings of the Constructionism 2010 conference*, Paris, France, 2010.

[12] J. Kaput, R. Noss, and C. Hoyles, "Developing new notations for a learnable mathematics in the computational era," in *Handbook of International Research in Mathematics Education*, 2002, pp. 51–75.

[13] L. Vygotsky, *Mind in Society: The Development of Higher Psychological Processes*. Cambridge, MA: Harvard University Press, 1978.

[14] B. L. Sherin, "A comparison of programming languages and algebraic notation as expressive languages for physics," *International Journal of Computers for Mathematical Learning*, vol. 6, no. 1, pp. 1–61, 2001.

[15] U. Wilensky and S. Papert, "Restructurations: Reformulations of knowledge disciplines through new representational forms," Manuscript in Preparation.

[16] J. Zhang and D. A. Norman, "Representations in distributed cognitive tasks," *Cognitive Science*, vol. 18, no. 1, pp. 87–122, 1994.

[17] D. Weintrop and U. Wilensky, "Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs," in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15)*, New York, NY, USA, 2015, pp. 101–110.

[18] C. D. Hundhausen, S. F. Farley, and J. L. Brown, "Can direct manipulation lower the barriers to computer programming and promote

transfer of training?," *ACM Transactions on Computer-Human Interaction*, vol. 16, no. 3, pp. 1–40, Sep. 2009.

[19] J. F. Pane, B. A. Myers, and L. B. Miller, "Using HCI techniques to design a more usable programming system," in *Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments*, Los Alamitos, 2002, pp. 198–206.

[20] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 1–40, Nov. 2013.

[21] R. Noss and C. Hoyles, *Windows on Mathematical Meanings: Learning Cultures and Computers*. Dordrecht: Kluwer, 1996.

[22] D. Weintrop and U. Wilensky, "Situating programming abstractions in a constructionist video game," *Informatics in Education*, vol. 13, no. 2, pp. 307–321, 2014.

[23] D. Weintrop and U. Wilensky, "The challenges of studying blocks-based programming environments," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 5–7.

[24] J. J. Kaput, "Towards a theory of symbol," in *Problems of Representation in the Teaching and Learning of Mathematics*, C. Janvier, Ed. Hillsdale, NJ: Lawrence Erlbaum Associates, 1987, p. 159.

[25] A. Robins, J. Rountree, and N. Rountree, "Learning and teaching programming: A review and discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[26] J. Hollan, E. Hutchins, and D. Kirsh, "Distributed cognition: toward a new foundation for human-computer interaction research," *ACM Transactions on Computer-Human Interaction*, vol. 7, no. 2, pp. 174–196, 2000.

[27] D. Weintrop and U. Wilensky, "RoboBuilder: A program-to-play constructionist video game," in *Proceedings of the Constructionism 2012 Conference*, Athens, Greece, 2012.

[28] D. Weintrop and U. Wilensky, "Program-to-play videogames: Developing computational literacy through gameplay," in *Proceedings of the 10th Games, Learning, & Society Conference*, Madison, WI, 2014, pp. 264–271.

[29] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press 2nd ed, 1996.

[30] E. L. Hutchins, J. D. Hollan, and D. A. Norman, "Direct manipulation interfaces," *Human-Computer Interaction*, vol. 1, no. 4, pp. 311–338, Dec. 1985.

[31] D. Weintrop and U. Wilensky, "To block or not to block, that is the question: Students' perceptions of blocks-based programming," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*, New York, NY, USA, 2015, pp. 199–208.

[32] S. Turkle and S. Papert, "Epistemological pluralism: Styles and voices within the computer culture," *SIGNS: Journal of Women in Culture and Society*, vol. 16, no. 1, pp. 128–157, 1990.

[33] M. Kölling, N. C. C. Brown, and A. Altadmri, "Frame-based editing," *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, Jul. 2017.

[34] J. Mönig, Y. Ohshima, and J. Maloney, "Blocks at your fingertips: Blurring the line between blocks and text in GP," in *IEEE Blocks and Beyond Workshop*, 2015, pp. 51–53.

[35] D. Bau, D. A. Bau, M. Dawson, & C. S. Pickens, "Pencil Code: Block Code for a Text World," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*, New York, NY, USA, 2015, pp. 445–448.

[36] M. Homer and J. Noble, "Lessons in combining block-based and textual Programming", *Journal of Visual Languages and Sentient Systems*, vol. 3, no. 1, Jul. 2017.

[37] U. Wilensky, "GasLab: An extensible modeling toolkit for connecting micro-and macro-properties of gases," in *Modeling and Simulation in Science and Mathematics Education*, N. Roberts, W. Feurzeig, and B. Hunter, Eds. Berlin: Springer-Verlag, 1999, pp. 151–178.