Supporting Computational Expression: How Novices Use Programming Primitives in Achieving a Computational Goal

David Weintrop[1,2] and Uri Wilensky[1,2,3]
[1]Center for Connected Learning and Computer Based Modeling,
[2]Learning Sciences,
and [3]Electrical Engineering and Computer Science
Northwestern University

Supporting Computational Expression: How Novices Use Programming Primitives in Achieving a Computational Goal

Introduction

It has been argued that computational thinking and the skills associated with it are of critical importance and deserve a position alongside reading, writing, and arithmetic as part of the core knowledge one needs to be successful in the 21st century (National Research Council, 2010, 2011; Papert, 1980, 1996; Wing, 2006). Central to computational thinking is the ability to translate or encode ideas into representations that leverage computational power. In this way, computational thinking is closely aligned with Papert (1980), diSessa (2000) and colleagues' notion of computational literacy, in which people are seen not just consumers of computational artifacts, but producers as well, creating what diSessa calls a "two-way literacy".

To better understand this process of computationally encoding ideas, RoboBuilder (Weintrop & Wilensky, 2012), a program-to-play constructionist video game (Weintrop, Holbert, Wilensky, & Horn, 2012), was designed to challenge players to invent gameplay strategies and express them computationally. Early in our analysis of participants playing RoboBuilder, it became apparent that the game's language primitives were being used in a variety of interesting ways during gameplay. This led to an analysis of the question: What roles do language primitives play in novice programmers' generating computational expressions?

This paper presents findings from this investigation. The paper continues with a brief introduction to RoboBuilder and a discussion of the theoretical framework used to structure our analysis. Next, we present vignettes from a pilot study conducted illustrating the three distinct usages of the game primitives we identified: 1) serving as a means for computational expression, 2) providing a source of inspiration, and 3) acting as a resource for generating explanations of observed behavior. The paper concludes with a discussion of the implications of these findings.

Meet RoboBuilder

RoboBuilder (Figure 1) is a constructionist, blocks-based, programming game that challenges players to design and implement strategies to make their on-screen robot defeat a series of progressively more challenging opponents. The players' 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 as the players' robot has. The objective of the game is to defeat your opponent by locating and firing at it while avoiding incoming fire from your adversary. Unlike a conventional video game where players control their avatars live during battle, in RoboBuilder, players must program their robots before the battle begins. To facilitate this interaction, RoboBuilder has two distinct components: a programming environment where players define their robots' strategy, and an animated robot battleground where their robots compete. The game is played by having the player first use the programming interface to construct their robot's behaviors before launching the battleground screen where they can see their programmed strategies enacted. To implement their strategy, players are provided with a set of language primitives; the language includes movement blocks (ex: `forward, turn left, 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.

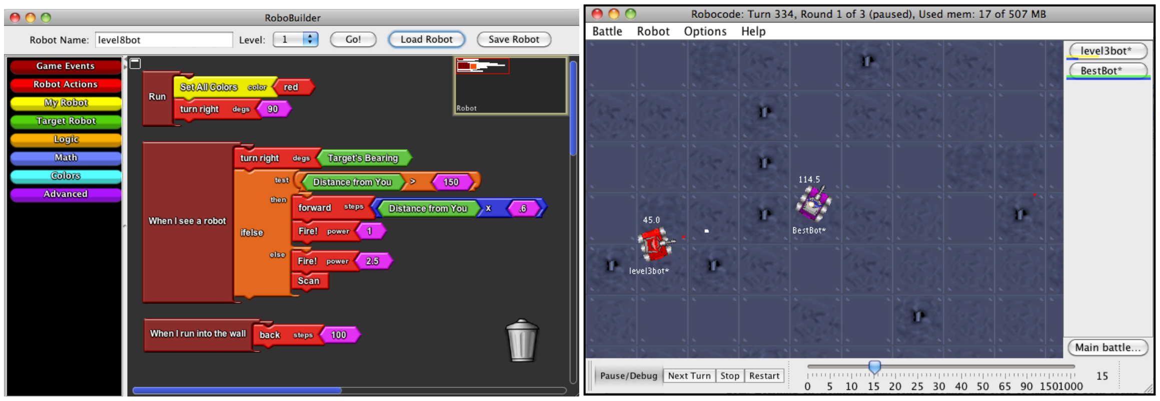


Figure 1. RoboBuilder's two screens. The battle screen, on the left, is where players watch their robot compete; the construction space, on the right, is where players implement their strategies.

Theoretical 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 (Wertsch, 1991). "The sign acts as an instrument of psychological activity in a manner analogous to the role of tool in labor" (Vygotsky, 1978, p. 52). 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 (diSessa, 2000; Parnafes & diSessa, 2004; Sherin, 2001; Wilensky & Papert, 2006, 2010; Zhang & Norman, 1994). diSessa (2000) called this idea "materially-mediated-thinking" arguing that "thinking in the presence of a medium that is manipulated to support your thought is simply different from unsupported thinking" (p. 115). He goes on to say "we don't always have ideas and then express them in the medium. We have ideas *with* the medium" (p. 116, emphasis in original). Wilensky & Papert (2006, 2010) have argued in their restructuration theory that the notions of task and content themselves cannot be separated from the representational infrastructure employed, content shapes and is shaped by the representational language.

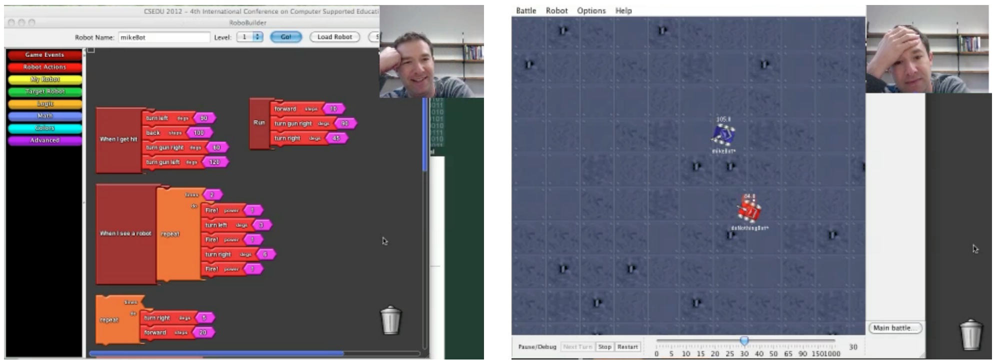
Given the foundational and constitutive role of the representations provided, to understand the cognitive task of programming, the analytic lens cannot focus purely on the individual, but instead must also include the signs (programming language) and tools (computer) provided to accomplish the task. In their cognitive account of direct manipulation interfaces, Hutchins et al. (1985) argue that the “distance” between the goals of the user and the tools and resources provided by the physical system can aide or hinder users in accomplishing the task set before them. Distance, they argue, is not a property of the interface alone, but instead “involves a relationship between the task the user has in mind and the way the task can be accomplished via the interface” (Hutchins et al., 1985, p. 318). Taking seriously the mediating role of the programming language in RoboBuilder, the analysis presented below focuses not on the individual alone, nor the language primitives themselves, but instead on the two acting in concert; what Wertsch (1991) calls an “individual-acting-with-mediational-means”. Through employing this analytic lens, an examination of RoboBuilder gameplay is revealing that the language primitives play a variety of roles in helping novices achieve their goals.

Methods

In the remainder of this paper, we present a series of vignettes from a pilot study we conducted intended to explore the varying roles that the programming language primitives played in accomplishing the in-game challenge. We have piloted RoboBuilder with a wide age-range of users, from middle school students through graduate students, all sharing the characteristic of having little or no prior programming experience. The university-aged participants were students at a large Midwestern 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 large Midwestern city that serves a predominantly African-American, low SES community. We will introduce the participants at the outset of each vignette. 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 robots being constructed.

The data presented below were collected through one-on-one interviews in which a researcher sat alongside the participant as 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, the player verbally explained her intended strategies to the interviewer in conversation. Next the participant was given the chance to implement her proposed strategy using the blocks-based language as the interviewer looked on. Finally, she would click the ‘Go’ button, and then watch her robots compete, with the interviewer asking her to describe what she observed and whether or not it matched her expectations. At the conclusion of the battle, the next iteration of the protocol would begin with the interviewer asking the participant what alterations she planned on making to her strategy to progress in the game. This three-phase cycle was repeated for the duration of the hour-long session.

Each RoboBuilder session was recorded using both screen capture and video-capture software. We also stored a digital copy of each robot strategy constructed during the RoboBuilder interview for further analysis. Two sample frames from the video produced during a RoboBuilder session can be seen in Figure 2.



```
When I get hit
    turn left degs 90
    back steps 100
    turn gun right degs 60
    turn gun left degs 120
Run
    forward steps 10
    turn gun right degs 90
    turn right degs 45
When I see a robot
    repeat times 2
        do
            Fire! power 1
            turn left degs 3
            Fire! power 1
            turn right degs 6
            Fire! power 1
repeat times
    do
        turn right degs 5
        forward steps 20
```

Figure 2. Two sample screenshots from a RoboBuilder session video. On the left, the participant is programming his robot; on the right he is watching his robot compete against an opponent

Differing Roles of Language Primitives in RoboBuilder

In our analysis of the data collected during our RoboBuilder sessions, we have identified a number of differing roles that language primitives have played in supporting the player in accomplishing the in-game objective. The three distinct mediating roles for the language primitives we have identified are: 1) serving as a means for computational expression, 2) providing a source of inspiration, and 3) acting as a resource for generating explanations of observed behavior. In this section, we provide vignettes for each of these mediational roles and provide a brief discussion of each.

Primitives as an Means for Expression

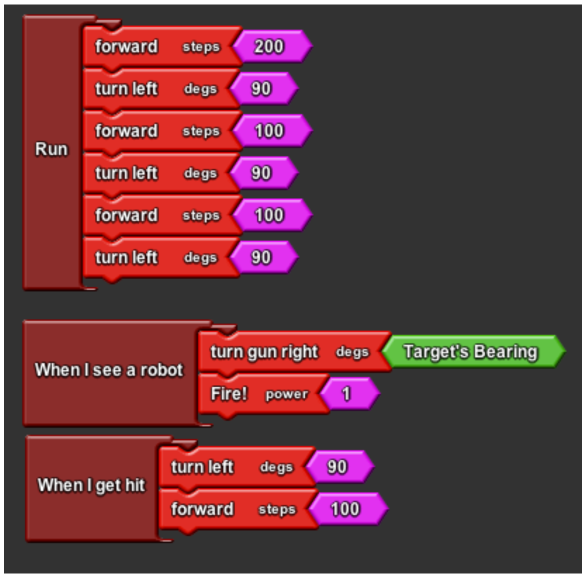
The first usage of the language primitives identified in our analysis is the one most closely aligned with the conventional view of the role programming languages play in a programming task; that of an expressive medium with which to encode ideas in a computationally executable form. In this role, the player conceives of a general idea or specific strategy for their robot, then uses the programming language to mediate the expression of that idea into a form the computer can interpret and execute. In this way, the language is used as a representational system with which players express their ideas.

Primitives as a Means for Expression: Two Vignettes

In RoboBuilder, language primitives used in this capacity take the form of a participant using the language to implement an idea that has been verbally expressed, but not yet implemented. An example of this can be seen at the beginning of Morris' RoboBuilder session. Morris is a university student in his junior year with no prior programming experience. When asked what his strategy was for defeating his first opponent, Morris responded:

Morris: So my master plan is to, like, be continuously moving, so it's harder to hit. If I get hit, kind of change that 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 implementation of his robot strategy. After six minutes of working, he had produced his first program; the first three events of which are displayed below in Figure 3.



```
Run
    forward steps 200
    turn left degs 90
    forward steps 100
    turn left degs 90
    forward steps 100
    turn left degs 90

When I see a robot
    turn gun right degs Target's Bearing
    Fire! power 1

When I get hit
    turn left degs 90
    forward steps 100
```

Figure 3. The first three events of Morris' first RoboBuilder program.

Comparing the strategies Morris articulated in his initial remarks to the program he produced, we can see the primitives taking on an expressive role, mediating and enabling the computational reification of his strategy. His "master plan" included three distinct ideas, each of which can be seen in his resulting program. Table 1 showing the correspondence between the verbalized component of his initial strategy and its computational instantiation, along with a brief description of the resulting behavior of the displayed code.

Verbalized Strategy	Computational Implementation	Comment

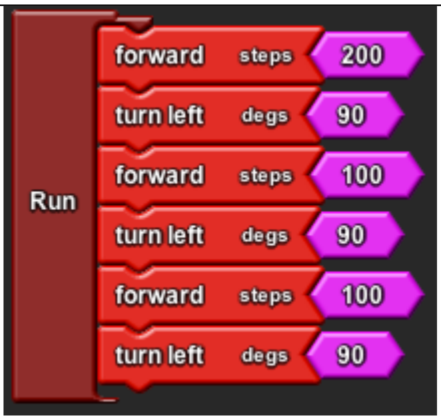
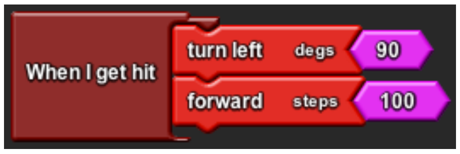
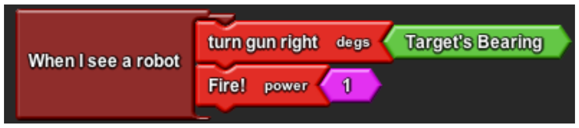
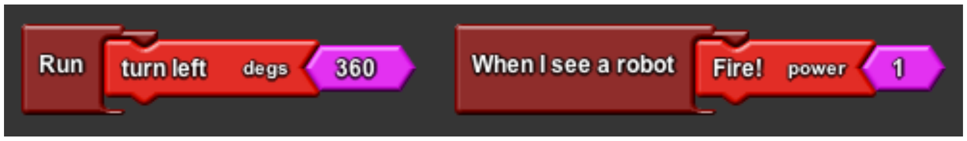
"Be continuously moving, so it's harder to hit"	 Run forward steps 200 turn left degs 90 forward steps 100 turn left degs 90 forward steps 100 turn left degs 90	This code will result in his robot remaining in constant motion, following a rectangle-like path.
"If I get hit, kind of change that path so it's different"	 When I get hit turn left degs 90 forward steps 100	These instructions will move his robot out of the current line of fire.
"Adjust to what the opponent is doing to hit them"	 When I see a robot turn gun right degs Target's Bearing Fire! power 1	This code will make the robot's gun turn towards it's opponent and fire at it.

Table 1. The three verbalized tactics and the corresponding computational implementation for each from Morris' initial RoboBuilder Robot.

From the first five minutes of Morris' RoboBuilder session we get a nice demonstration of how the language primitives can be used as a means for expression. Our second vignette occurs roughly twenty minutes into Daniel's RoboBuilder session. Daniel is a tenth grade student with no prior programming experience who was recruited to participate through an afterschool program at a neighborhood community center. After seeing his first two robots struggle against the level one opponent, Daniel decides he needs a new strategy. Daniel had initially thought his opponent would start the battle in the exact same position every battle, so was trying to instruct his robot to move toward that location. After watching his robot compete a few times, he realized the starting locations are random; this prompts 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 gets to work, quickly implementing this robot strategy, the entirety of which is shown in Figure 4.

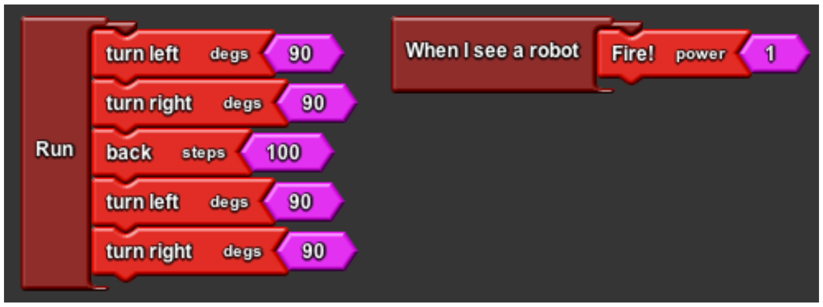


```
Run  turn left  degs  360
When I see a robot  Fire!  power  1
```

Figure 4. Daniels implementation of his 'spin in a circle and shoot' strategy[1].

The behavior of the `Run` event in RoboBuilder is to execute all the commands inside it from top-to-bottom, then, upon completion, return to the top, creating a repeating pattern. After try out his new strategy, the interview asks Daniel to describe what his robot is doing, Daniel, with a big smile responds: *"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, like with Morris' vignette, we see the language primitives serving as a means of expression.

From these two vignette's, we don't mean to give the impression that expressing ideas with the language primitives provided is always as easy or straightforward as these two examples might suggest. For example, if we continue to use Daniel's session as a case study, we can see some of the challenges associated with this goal. Having defeated the first (and second) robots with his spin-and-shoot strategy, Daniel encounters the level three robot, whose strategy of remaining stationary with its gun focused on the opponent is proving superior to his own. To combat this new tactic, Daniel proposes the following strategy: *"make him move more, and keep that, when you see the other robot shoot, because the other robot is standing in one place, and just make my* [robot] *move more."* Having verbalized his plan, he then went ahead, attempting to implement it, producing the strategy shown in Figure 5.



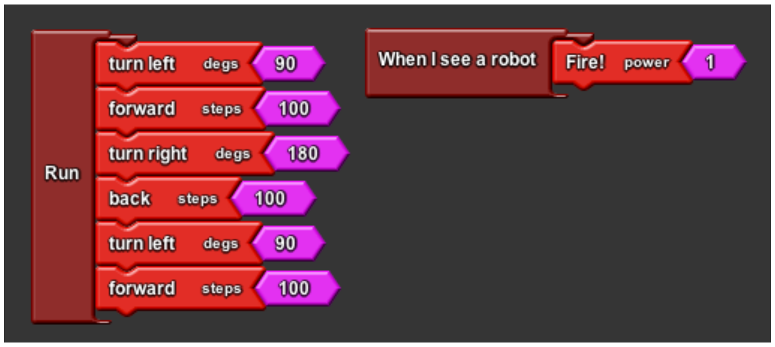
```
Run  turn left  degs  90
     turn right  degs  90
     back  steps  100
     turn left  degs  90
     turn right  degs  90
When I see a robot  Fire!  power  1
```

Figure 5. Daniel's new strategy.

With his new implementation in place, Daniel launches the battle screen and is surprised to see his robot not behaving as expected. *"What I thought was going to happen, isn't happening, I thought he was going to move from one side of the board to the other side, but he's just standing in one spot. Going left, right, left, right."* Because the `turn`

[1] Note: The spatial arrangement of the event blocks (in this case Run and When I see a robot) do not affect the order in which they are run, so a side-by-side arrangement is equivalent to having the events stacked on top of each other.

`right` and `turn left` blocks only rotate the robot, Daniel was not successful at implementing his latest strategy of making his robot move around. Figure 6 shows his next robot construction, where, having learned from his previous attempt, he successfully implements his idea of getting his robot to keep its "when I see a robot shoot" logic while also introducing movement, so it does not remain stationary.



```
Run
    turn left   degs   90
    forward     steps  100
    turn right  degs   180
    back        steps  100
    turn left   degs   90
    forward     steps  100

When I see a robot
    Fire!  power  1
```

Figure 6. *Daniels final robot in which he successful encodes his verbalized strategy using the language primitives provided.*

Primitives as a Means for Expression: Discussion

The ability for a programming language to enable users to express ideas in such a way that they can be interpreted and executed by a computer its primary role, as, by definition, if it is not possible to write a program using the representational system, it can hardly be considered a programming language. 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 another human, but instead a computer. In this way, programming languages serve as a bridge across what Hutchins et al. (1985) 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. Based on the designed affordances of the programming language, this task can be made more accessible or more difficult. Hutchins et al. (1985) argue that users can be aided in effectively bridging this gulf "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 to express an idea. This can be seen in the above vignettes 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. Similarly, his statement that he wanted his robot to "*be continuously moving*" resulted in him using the blocks `forward` and `turn right`, two commands that can be expected to produce movement in the robot. Likewise, with Daniel, comments such as *"spin in a circle and shoot"* closely map onto the language primitive provided.

The mediational role played by the primitives made it possible for Morris, Daniel, and other participants that played RoboBuilder to encode their ideas in a way that the computer could understand. The representational system acts as a bridge between the robot strategies he has imagined and the computational environment where the strategies are enacted. These vignettes are two of many from our research 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 player and computationally executable reification of that same idea.

Primitives as a Source of Inspiration

The second identified role that language primitives play is that of inspiration. When participants use the language primitives in this capacity, the primitives are not used to express ideas the participant has already come up with, instead, the language primitives act as a source of ideas, mediating the task of generating an idea for the challenge at hand.

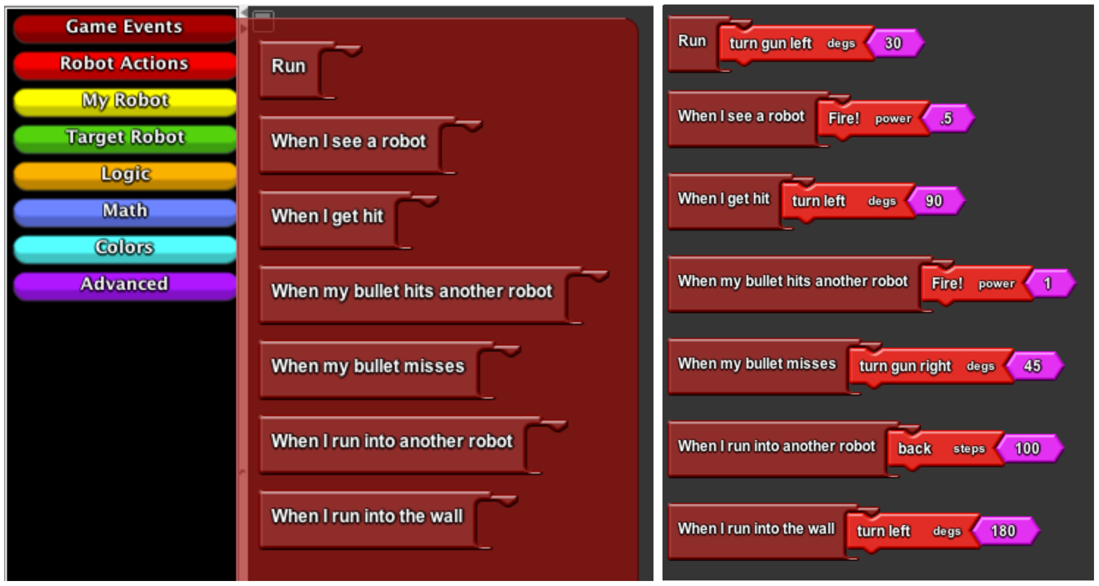
Primitives as a Source of Inspiration: Two Vignettes

Our two examples of this usage of language primitives come from a series of RoboBuilder interviews conducted with Beth. Beth is a vocal performance major at a large Midwestern university who had no prior programming experience and had admitted to not having taken a math course since high school. After being introduced to the game, Beth was asked how she was going to defeat her first opponent, she responded:

> *Beth: 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*[2]*, or whatever they are, and try and figure out what I want to have happen.*

Beth then proceeded to go through each block in the Game Events drawer, using them as a roadmap to develop a strategy for her robot. In Figure 7 on the right, we can see Beth's first completed robot strategy; on the left we see the Robot Events drawer that lists the available Game Event blocks.

[2] The dark red buttons Beth is referring to are the Game Event blocks. A full list of them can be seen in the left pane of Figure 7.



```
Run                               turn gun left   degs   30
When I see a robot                Fire!           power  .5
When I get hit                    turn left       degs   90
When my bullet hits another robot Fire!           power  1
When my bullet misses             turn gun right  degs   45
When I run into another robot     back            steps  100
When I run into the wall          turn left       degs   180
```

Figure 7. 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 possible event, but also 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 list of events and working her way down. This suggests that she did not have a clear, unified strategy when she began to program her robot, but instead, built her program event-by-event, following the road map set out by the order of the blocks within the Game Events drawer, using these primitives to bootstrap the generation of a valid (and successful) robot strategy.

By the time Beth had reached the level 5 robot, she had been playing the game for almost 80 minutes, and, over the course of gameplay, encountered almost all of the blocks provided by the game either by using them, or reading through them as she was trying to figure out what she wanted her robot to do (much in the same way she read through the Game Events blocks as described above). In constructing a strategy to defeat the level 4 opponent, Beth, for the first time, used a block from the Target Robot drawer, which contains primitives that provide information about the opponent robot. In her strategy she used the `Target's Bearing` block, which returns where the enemy is relative to the direction your robot is facing in terms of degrees, ranging from -180 and 180. Figure 8 shows the relevant events of Beth's latest robot as well as the contents of the Target Robot drawer.

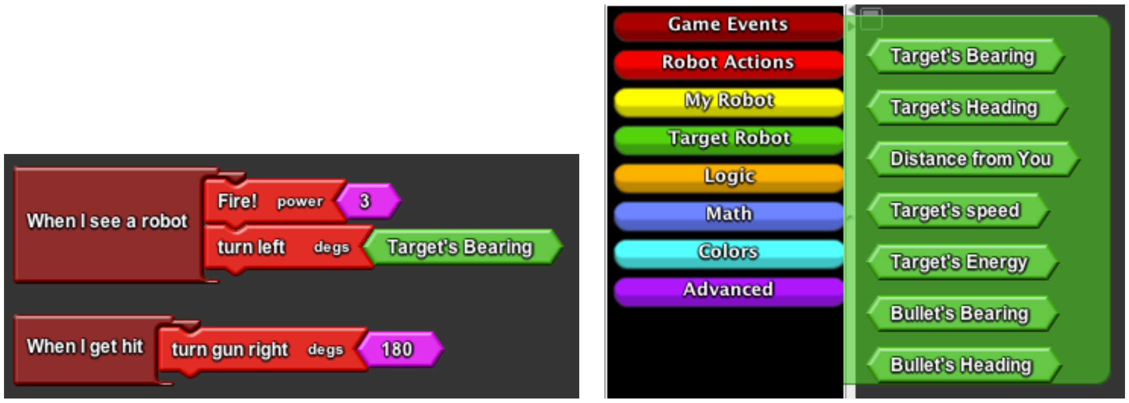


Figure 8. The left image shows the relevant events of Beth's 5th robot; on the right is the Target Robot drawer.

Beth was trying to add logic to her robot so it would turn at face her enemy robot but having difficulty[3]. This difficulty prompted the following exchange:

Beth: What else can I know about the target's robot? (Beth opens the target robot drawer[4]) *Alright, it's heading, I suppose that would be like, the information from the target's heading would like, tell me in which direction it was going?*

Interviewer: (Interviewer answers her question)

Beth: yeah - So that might be helpful, ok, (Beth continues reading from the drawer) *so distance from me, nah - that doesn't help, target's speed - target's energy - bullet's bearing, bullet's heading.*

Interviewer: Those last two, bullet's bearing and bullet's heading are for when I get hit, that's when you know information about the bullet, because it's kind of that same idea of like, when you get hit, you know information about the bullet when you see a robot, you know information about the robot.

Beth: Oh - ok, so I would use bullet's bearing, bullet's heading when I get hit, like that would tell me, that could tell me where it is - oh! (Her voice rises in excitement) *because, the bearing would be where it is coming from, right?* (Interviewer nods) *So...so... ok, I'm just going to get rid of this (*she drags the turn gun target's heading block into the garbage*) because that's not helping me, but when I get hit, oh, oh!* (Her excitement is growing) *so turn gun right 180, that's just arbitrary* (she drags `turn gun right 180` from `When I get`

[3] In this case, her confusion stemmed from the values returned by the `target's bearing` command, which return positive numbers if the target is to the player's right and negative if the target is to the player's left. Beth's usage of turn left resulted in unexpected and unintended behavior, as it caused her robot to turn in the wrong direction.

[4] Non-italicized text inside parenthesis within a quote provides further context for what was said or describes non-verbal actions that occurred during the utterance.

> `Hit` into the garbage*) whatever, sometimes that worked, sometimes it didn't but turn, when I get hit, turn gun right...ok, and now I'm going to try, so if I, here I am, here's the bullet, ah I just got hit, so I want to turn my gun right, how am I going to determine. Can I just do bullet's bearing? Or is that not going to work…?*

There are two interesting things to note from this excerpt. First, Beth is reading through the blocks in the Target Robot drawer, considering how she might use each. She entertained the idea of using `Target's Bearing` and `Target's Heading`, quickly dismissed the next three blocks as not being useful for the problem she is trying to solve, before landing on the two blocks related to the bullet direction. This reading of language primitives without a clear idea of how she going to accomplish the goal she has in mind is evidence that the language primitives are not being used in an expressive way, but instead are playing a generative role in helping her come up with ideas. Using diSessa's language, she is "thinking *with* the medium".

The second interesting thing to note is the change in her overall strategy that occurred after hearing an explanation of what information the two bullet related blocks provide. Immediately upon hearing the interviewer's explanation of the `Bullet's Heading` and `Bullet's Bearing` blocks, she adopted a new strategy that made these blocks central. Beth's new strategy was to turn her gun towards the source of the bullet whenever her robot got hit. If her opponent was not moving (which was the case for the robot she was currently competing against), this action would point the gun at her opponent. Figure 9 shows the same two events as Figure 8 after the above exchange transpired, all the other events in her program were untouched during this iteration.

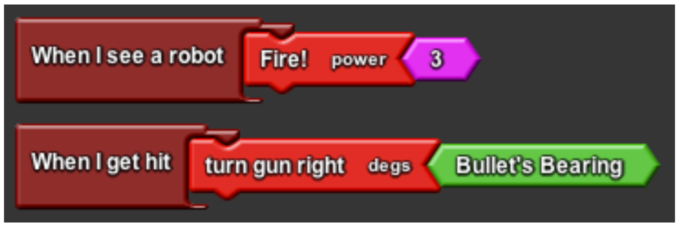

```
When I see a robot  Fire! power 3
When I get hit  turn gun right degs Bullet's Bearing
```

Figure 9. The two main methods of Beth's strategy after discovering the `Bullet's Bearing` *block.*

Primitives as a Source of Inspiration: Discussion

In these two vignettes from Beth's RoboBuilder interview, we see RoboBuilder's language primitives playing a distinctly different role from the way they were used in the vignette in the prior section. Whereas Morris used the language primitives in a mainly expressive capacity, here we see Beth utilize the language primitives as a source of inspiration for the strategies she eventually implements. . In the first vignette she even states her intention to use the language primitives in this capacity, saying: "*I think I'll use these dark red buttons…and try and figure out what I want to have happen.*" Later, in the second vignette, we saw how, once again, she used the language itself as a resource to help hear accomplish the larger goal of defeating her opponent. By reading through the set of primitives provided, and using them as objects-to-think-with (Papert, 1980) as she worked to develop her robot strategy, Beth was able to successfully program her robot despite not having any formal programming training. Consistent with diSessa's (2000) idea of "materially-mediated-thinking", in these episodes it appears that Beth is having ideas *with* the medium. The language primitives are mediating her thinking about the challenges, bootstrapping the process of come up with ideas for how to go about accomplishing the computational challenge set forth by the game.

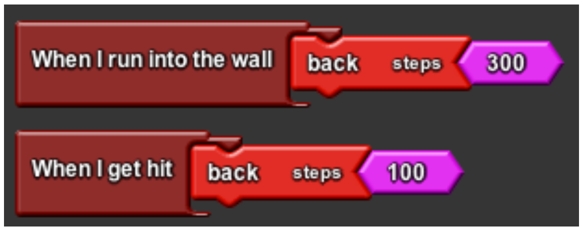
Along with diSessa's "materially-mediated-thinking" and Papert's "objects-to-think-with", we can see how the use of representational systems in this capacity relates to Wilensky and Papert's (2006, 2010) notion of a structuration, as the representation itself is enabling certain ideas to be expressed more easily. You can imagine that if instead of using the blocks provided, if the language was an abstract set of operations with labels like: `operation1`, `state2`, and `movementX`, that even if the two languages were equivalent, that Beth would have produced a very different robot strategy. In this way, the language itself supports and makes more accessible the expression of certain ideas. In the case of designing environments for novices, recognizing that the primitives are used in this capacity is critical for scaffolding the user in having early success as it can provide a roadmap to follow for getting started in expressing ideas with the tools provided.

Primitives as a Resource for Explanation

The third observed use of RoboBuilder's primitives was that of an explanatory tool with which to decipher and interpret observed robot behavior. In this role, the language primitives were used to mediate the process of developing an understanding of what was observed during battle. Observed behaviors were interpreted *through* the primitives; the primitives provided a means to explain the observed behavior.

Primitives as a Resource for Explanation: Vignette

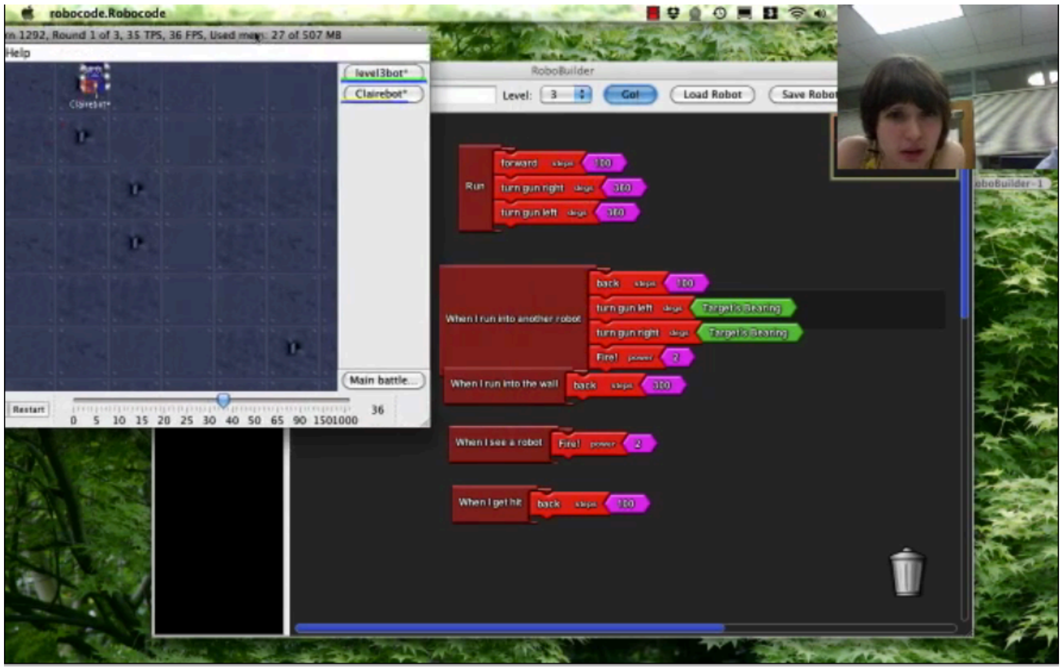
An example of this usage can be seen toward the end of Anne's RoboBuilder interview. Anne is a junior at a Midwestern University, who like all other participants in the study, had no prior programming experience. At the time of the interview, Anne was finishing up her undergraduate degree and had no prior programming experience. She had just finished implementing the seventh iteration of her robot, during which she introduced the `When I get Hit` event in hopes of addressing a weakness in her strategy she had identified during her last battle. Figure 10 shows the two events from Anne's program that are relevant for this episode.



```
When I run into the wall   back steps 300
When I get hit   back steps 100
```

Figure 10. The two events of interest from Anne's robot implementation.

Having made this addition, she launched the battle screen to test her changes. Things were looking promising until her robot was hit a few times in succession and backed into the wall. Her robot proceeded to remain pinned to the wall, motionless, getting hit until the match ended. When this happened, Anne got a confused look on her face and asked: "*Oh, what am I doing now? Wait, what happened?*" Not being able to make sense of what she was seeing based on what she remembered programming, Anne rhetorically asked: "*Wait, but when I run into a wall, what'd I put?*" She then dragged the battle screen to the left so that she could see both the battle screen and her code simultaneously (Figure 11).



```
Run                           forward steps 100
                              turn gun right degs 360
                              turn gun left degs 360

When I run into another robot back steps 100
                              turn gun left degs Target's Bearing
                              turn gun right degs Target's Bearing
                              Fire! power 2
When I run into the wall      back steps 300

When I see a robot            Fire! power 2

When I get hit                back steps 100
```

Figure 11. A screen shot of Anne reading her code while a battle is occurring to understand the unexpected behaviour she just observed.

Upon seeing her code, she realized the bug she had introduced to her robot strategy. 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, which had the result of keeping her pinned against the wall, essentially creating an endless loop until the end of the battle. By going back to her construction, and examining the instructions she had defined, she was able to understand why her robot would stop moving once it had backed into a wall. Having first seen an unexpected behavior, she was able to use the construction as a tool to mediate her developing an understanding of the unexpected behavior she had observed.

Primitives as a Resource for Explanation: Discussions

This vignette provides an example of the third role that programming language primitives can play during a programming task that is quite distinct from the first two. Whereas the first two uses are generative in nature, this third usage shows the central role the language primitives can play in mediating a comprehension task. Where our first identified role of supporting computational expression was analogous to an alphabet being able to support writing, this third role highlights the need for the representational system to be able to be read. This highlights the dual audiences of computational representations. With computational representational systems, the primary audience for a constructed artifact is 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 computational artifact. The fact that a program written with a computational representational system serves as a static set of instructions that produce a dynamic outcome, in our case the in-game robot behaviors, supports its being used in a meaning-making capacity. While the program is being run, the constructed artifact can serve as a blue print, containing an explanation for how and why the program is behaving as it is.

In the above vignette, without referring back to her program, Anne was unable to make sense of what she saw her robot doing. In this way, the language was playing a mediating role in helping her come to an understanding of how her program worked. 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, we see how programming languages can be used as a way not to mediate the expression of ideas, but also to mediate the understanding of ideas already expressed.

Implications of this Finding

Recognizing the various roles programming primitives play has implications for designers of novice programming environments and beginner programming languages. For a language designer, the differing usages of language primitives each suggest a different set of priorities and considerations for how the language should be designed and presented. If the goal is expressiveness, a language that supports as many implementations as possible would be ideal; this suggests a small grain-size and large set of options (these characteristics are valued by professional programming languages such as Java and c++). Alternatively, when designing for an audience who might use the language as a source of ideas, the language must fit the task it was designed for in such a way that the primitives themselves can bootstrap the generation of successful programs.

In this case, language primitive might be of a larger grain-size, carrying more meaning in the context of the programming challenge at hand. “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” (Wilensky, 1999, p. 168). Finding the right size primitives is one of the central challenges for designers where creating a language and environment for novice programmers.

Designing for all three of the usages presented in this paper presents a challenge to the designer as in some cases; design decisions made to support one usage may be at the expense of another. An example from RoboBuilder’s language will make this tension more concrete. The set of game events provided in RoboBuilder (like `When I See a Robot` and `When I get Hit`) were designed to provide conceptual hooks for players to introduce behavioral logic to their robot that would execute when their robot entered that state. By providing a fixed set of events, the language constrains how and when behavior logic can be introduced in the game. An alternative approach would have been to expose more basic primitives that would have allowed players to specify their own game states that could then be used to introduce specific behaviors to their robot. The decision to provide a standard set of events, as opposed to a customizable set has consequences with respect to all three of the identified primitive usages. First, with respect to the usage of primitives as sources of inspiration, by providing a fixed set of event blocks that a custom to the RoboBuilder game context, the event blocks can play a generative role in developing a robot strategy. We saw this in Beth’s vignette described above, where her strategy was strongly influenced by the set of provided game events.

Second, with respect to the role of primitives as an expressive medium, the consequence of providing a fixed set of events is that the provided event blocks constrain the set of battlefield states that a player can respond to. During the pilot study, participants did generate ideas that could not be expressed with the provided event blocks. For example, after Anne realized her `When I hit a Wall` logic would only work when she hit a walling going forward, she proposed the idea of having a different behavior if the robot hit the wall moving forward as opposed to backwards. Due to the provided event blocks, it was not possible to implement this solution as she described it. If the primitives were of a finer grain and thus more expressive, it might have been possible for her to have defined two different events, one for running into the wall while going forward, a second for running into the wall going backwards.

Finally, with respect to the primitives playing an interpretive role, the consequence of providing pre-defined event blocks was that sometimes it was difficult for players to understand exactly what their robot was doing because they themselves did not program the block. A number of players asked for clarification on what it meant for a robot to ‘see’ another robot (in reference to the `When I See a Robot` event block). Because this block black-boxed the logic to interpret the detection of an opponent robot, players had to ask the researcher for an explanation of the event, as opposed to deriving meaning from the code itself. An example of this came during Morris' interview, his `When I See a Robot` event was not behaving as expected, he went back and read his code, but was still not clear on what was happening, which prompted him to ask:

Morris: So when does [my robot] like, register as "seeing the [robot]", 'cause in my idea it's like, what I'm trying to say is: I see the tank, I should turn and aim at it, but it does that, but then it aims just off.

In response to this question, the researcher had to explain the radar system used by the game. Had the language provided finer-grained events, this explanation could have been avoided as the language itself could have supplied the explanation Morris was seeking.

Conclusion

At the outset of the paper, the idea of a computationally literate society championed by Wing and diSessa was introduced. While RoboBuilder on its own will most likely not be the technological break through that results in this vision being achieved, there is a chance that the findings from studying how novice programmers interact with it can act as a harbinger for other educational tools and environments that can start us down the path toward the computationally literate society envisioned by these scholars. By recognizing the various roles primitives can play in supporting novices in computationally reifying ideas, we as designers and educators can begin to develop new languages and environments to support these different usages to scaffold the progression of a learner from a novice to a computationally fluent individual. In doing so, we can make progress toward this vision of a computationally literate 21st century.

References

diSessa, A. A. (2000). *Changing minds: computers, learning, and literacy*. Cambridge, MA: MIT Press.

Hutchins, E. L., Hollan, J. D., & Norman, D. A. (1985). Direct manipulation interfaces. *Human-Computer Interaction*, *1*(4), 311–338.

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

National Research Council. (2011). *Report of a Workshop of Pedagogical Aspects of Computational Thinking*. Washington, D.C.: The National Academies Press.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic books.

Papert, S. (1996). An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning*, *1*(1). doi:10.1007/BF00191473

Parnafes, O., & diSessa, A. (2004). Relations between types of reasoning and computational representations. *International Journal of Computers for Mathematical Learning*, *9*(3), 251–280.

Sherin, B. L. (2001). A comparison of programming languages and algebraic notation as expressive languages for physics. *International Journal of Computers for Mathematical Learning*, *6*(1), 1–61.

Vygotsky, L. (1978). *Mind in society: The development of higher psychological processes*. Harvard Univ Press.

Weintrop, D., Holbert, N., Wilensky, U., & Horn, M. S. (2012). Redefining Constructionist Video Games: Marrying Constructionism and Video Game Design. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Weintrop, D., & Wilensky, U. (2012). RoboBuilder: A Program-to-Play Constructionist Video Game. In C. Kynigos, J. Clayson, & N. Yiannoutsou (Eds.), *Proceedings of the Constructionism 2012 Conference*. Athens, Greece.

Wertsch, J. V. (1991). *Voices of the mind: A sociocultural approach to mediated action*. Harvard Univ Press.

Wilensky, U. (1999). GasLab: An extensible modeling toolkit for connecting micro-and macro-properties of gases. *Modeling and simulation in science and mathematics education*, *1*, 151.

Wilensky, U., & Papert, S. (2006). *Restructurations: Reformulations of knowledge disciplines through new representational forms*. (Manuscript in preparation).

Wilensky, U., & Papert, S. (2010). Restructurations: Reformulating Knowledge Disciplines through New Representational Forms. In I. Kallas (Ed.), *Proceedings of the Constructionism 2010 conference*. Paris.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, *49*(3), 33–35.

Zhang, J., & Norman, D. A. (1994). Representations in distributed cognitive tasks. *Cognitive Science*, *18*(1), 87–122. doi:10.1016/0364-0213(94)90021-3