Teaching "Lawfulness" With Kodu

David S. Touretzky
Computer Science Dept.
Carnegie Mellon University
Pittsburgh, PA 15213
dst@cs.cmu.edu

Christina Gardner-McCune
Dept. of Computer & Info.
Science & Engineering
University of Florida
Gainesville, FL 32611
gmccune@ufl.edu

Ashish Aggarwal
Dept. of Computer & Info.
Science & Engineering
University of Florida
Gainesville, FL 32611
ashishjuit@ufl.edu

ABSTRACT

This paper introduces reasoning about lawful behavior as an important computational thinking skill and provides examples from a novel introductory programming curriculum using Microsoft's Kodu Game Lab. We present an analysis of assessment data showing that rising 5th and 6th graders can understand the lawfulness of Kodu programs. We also discuss some misconceptions students may develop about Kodu, their causes, and potential remedies.

CCS Concepts

•**Social and professional topics** → **Model curricula; K-12 education;** *Computational thinking;*

Keywords

Kodu Game Lab; formal reasoning; programming idioms

1. INTRODUCTION

We believe that mastery of *lawfulness* is an important but often overlooked computational thinking skill. In computing, formal laws (or rules) govern how expressions are evaluated, actions are sequenced, and state is updated. Often these rules deal with low level processes: languages such as Python or Java are only a bit more abstract than assembly language. Children's languages such as Scratch and Alice remove some sources of difficulty by providing a drag-and-drop GUI editor to prevent syntax errors, and a queryable graphics canvas that makes some program state readily apparent. But these environments provide few tools for helping children think about state. Likewise, tutorials for beginning Scratch and Alice programmers don't try to teach students to reason formally about the state changes they observe.

While a graphics canvas is somewhat useful for reasoning about state, with no physics and no intrinsic interactions among objects, nothing happens on Scratch's 2D or Alice's 3D canvas unless the programmer makes it happen. Interesting programs therefore require tedious detail that makes reasoning about them more difficult. Thus, graphics primitives alone are not as abstract or powerful as we would like for beginning programmers.

For children to demonstrate mastery of "lawfulness" in a computational framework they should be able to (1) state the laws, (2) explain program behavior by referencing the laws, and (3) apply the laws to predict future behavior from current state. Fortunately there is wide lattitude in what these laws can be. They don't have to describe low level primitives as in Python or Java, or – notwithstanding the graphics canvas – Scratch or Alice. For example, a set of objects can be referenced by specifying a pattern that they match rather than collecting and maintaining them in a subscripted data structure. To see what higher level laws look like, we will examine Microsoft's Kodu Game Lab [1] and its underlying computational principles.

Kodu worlds are three-dimensional environments with fluidly articulated characters. These characters are not merely graphics objects: they are semi-autonomous robots that perceive and act on the contents of a dynamically changing world. There is real physics, including gravity, inertia, friction, elastic collisions, complex lighting, wind, water, and waves. Actions in Kodu have built-in sound effects, collisions are always noisy, and when a character isn't doing anything else, it mumbles and fidgets amusingly. A handful of lines of Kodu code can produce highly engaging behaviors. Research presented in this paper will show that given the right tools, children are able to reason about these behaviors and demonstrate an appreciation of their lawfulness.

In the following sections we discuss various aspects of Kodu and four conceptual tools we employed to help children recognize "lawfulness". We then analyze assessment data from 23 rising 5th and 6th graders who attended a five session Kodu Camp. Finally, we identify two important misconceptions about Kodu rules, and their likely causes.

2. KODU SYNTAX

Kodu syntax is trivially simple. Each character has its own program, which is a set of up to 12 numbered pages. A page is an ordered list of rules. A rule consists of a WHEN phrase followed by a DO phrase, either of which can be empty. A non-empty phrase begins with a predicate or action tile and continues with zero or more argument/modifier tiles. These can appear in any order with a few exceptions.[1]

[1]The "random" tile takes distinct left and right arguments. And in operations involving scores, the first tile after the verb is required to be a score color and is treated specially. This is a recent change in Kodu syntax made in June 2015.

SIGCSE '16, March 2–5, 2016, Memphis, TN, USA.

 ISBN 978-1-4503-3685-7/16/03…$15.00
DOI: http://dx.doi.org/10.1145/2839509.2844652

Finally, a rule's indentation level must be between 0 and one more than the indentation of the immediately preceding rule. That's it!

Kodu's GUI editor uses context-senstive menu selection rather than the drag-and-drop interface with fixed menus common in other visual languages. The menus only present options that are both syntactically and semantically legal at that point in the rule.

Although Kodu users can't make syntactic errors, it's still useful for them to think about syntax because it's an illustration of lawfulness. But a formal syntactic description like the one above is not easily digested by children. Touretzky created a set of plastic tile manipulatives whose colors and shapes reflect Kodu's syntactic constraints [6], as in Figure 1. The WHEN tiles are green, DO tiles are blue, and indentation levels are reified as yellow tiles.

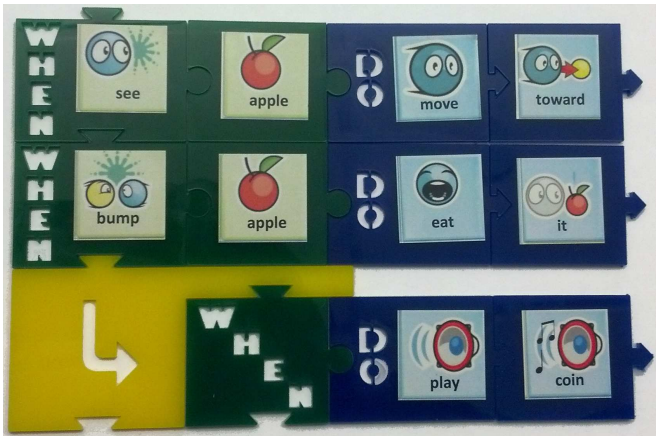

Figure 1: Kodu tile manipulatives.

During a Kodu session with second graders, a child tried to use the manipulatives to make the improper rule "WHEN see apple DO eat apple", but the final tile wouldn't fit because the connector didn't match. Another child spontaneously corrected him by suggesting the "it" tile, which does fit, yielding the syntactically correct rule "WHEN see apple DO eat it" [6].

3. PRINCIPLES OF KODU COMPUTATION

Kodu's power comes in part from providing higher level primitives such as pattern matching — the mechanism of Kodu perception — and rich actions, such as giving a held object to another character with automatically generated motion and sound effects. Control structure is more subtle than in conventional languages because there is only one statement type: the conditional rule. Additional control comes from other sources: rule dependency relationships expressed through indentation, an arbitration scheme that mediates conflicting actions, an implicit variable binding mechanism with "it" and "me" tiles, and a "perform action once" modifier. There is no explicit iteration. Instead, all the rules on the currently active page run repeatedly, 50-100 times per second, until the character switches to a different page.

To teach students to reason about lawful behavior in Kodu, Touretzky formulated a set of "principles of Kodu computation" [5]. These principles (or laws) can either be taught explicitly or worked out by the children themselves through guided discovery. To illustrate the latter, consider the first problem posed in many Kodu tutorials: eating a bunch of apples scattered over a field. The solution comprises a single two-rule idiom, Pursue and Consume, discussed in section 4. Once they've learned the idiom and run the solution, children are asked how the kodu character chooses which apple to pursue first. They are encouraged to move the kodu and/or the apples and rerun the program. They quickly discover that the kodu always goes to the closest apple. This is the first principle. It is reinforced by having students restate it in written exercises, and by requiring them to apply it to predict the paths characters will take to consume various collections of objects. Formally, the first principle is "a rule always selects the closest matching object."

The second principle is that perceptual predicates like "see" or "bump" work by pattern matching, with additional tiles further constraining the match. So "see" matches any object other than the character itself; "see apple" matches apples; "see red" matches red things; and "see red apple" matches red apples, as does "see apple red".

The third principle is that an indented rule is only eligible to run if the parent rule is eligible and the parent's WHEN part is true. This principle, which produces block structure, is crucial for many Kodu idioms such as Do Two Things.

The fourth principle students learn is that when two rules specify conflicting actions, the earlier (lower numbered) rule wins. This principle makes the Default Value idiom possible.

These four are the most important principles, but there are many more, reflecting the richness of the language. The current list has more than 30. Together they capture many of the fine points of Kodu semantics. An example of a more subtle principle is: a character cannot "see" the object it is holding, but it can see objects held by others. (The "got" tile can be used in place of "see" to examine a held object.) Another more advanced principle is: "timers don't nest".

4. A KODU IDIOM CATALOG

Lawfulness is one interesting aspect of computation; another is the patterns in which primitives frequently combine. These patterns, or *idioms*, are a means of achieving higher level goals. Reasoning in terms of idioms instead of the component primitives is an example of abstraction, another important ingredient in computational thinking [8].

Touretzky created a catalog of Kodu idioms that students can draw upon [5]. They are depicted on a ring of flash cards (inspired by the Scratch Cards of Natalie Rusk [2]), and were given to each student at the start of every session. Each idiom has a name, a schematic illustration, sample code, and usage notes. Students were taught these idioms explicitly, and drilled on them in various ways. For example, they were asked to apply the idiom in a new situation. Or they were shown novel code fragments and asked to explain why each one does or does not qualify as an example of the idiom. Table 1 lists a few of these idioms.

The most basic idiom, which every Kodu student encounters when confronting a field of apples, is Pursue and Consume. Figure 2 shows the flash card. A pursue rule involves motion toward an object. A consume rule makes an object non-pursuable, either by destroying it as "eat" does, or by changing it to no longer match the perceptual pattern of the pursue rule. Kodu's repetitive execution of all the rules on a page takes care of the rest.

Students develop a more abstract understanding of Pursue

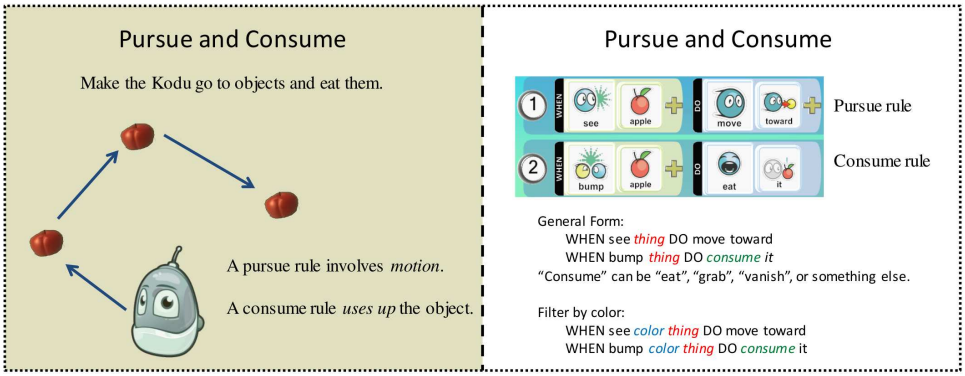

Figure 2: The Pursue and Consume idiom on a flash card.

Table 1: Some common Kodu idioms.

Name	:	Example
Pursue and Consume	:	Eat all the apples
Do Two Things	:	Eat an apple + play coin sound
Count Actions	:	Count the apples while eating
Default Value	:	Turn red if see octopus, else blue
Follow the Yellow Brick Road	:	Follow a path and react upon reaching the end
Let Me Drive	:	Steer character with joystick
Show Page As Color	:	Make a character's color indicate its current page
Parting Shot	:	Final action when killed or eaten

and Consume by exploring variants of the apple eating task. In one world they are asked to eat stars instead of apples. In another they are asked to grab (poisonous) blue apples instead of eating them; grabbing them makes them shrink and disappear. In yet another they pursue white or black huts and paint them blue, demonstrating that "consume" doesn't necessarily mean "remove".

An idiom is not just a code template; it's a rich semantic context in which to explore how primitives interact to achieve a result. We deepen students' understanding by asking them to give English definitions for "pursue" (to chase, or to proceed along a path) and "consume" (to eat or use up), and by having them predict the effects of altering parts of the idiom. For example, replacing "eat" with "boom" (explode) is fine, but replacing it with a non-consuming action such as playing a sound leaves the kodu stuck at the first object it pursued. A later exercise asks students for the opposite of pursue ("flee"), and then has them search for a movement action that will let the kodu flee an adversary. (Answer: "move away" instead of "move toward").

5. STATE MACHINES

State machines are among the most fundamental and ubiquitous concepts in computer science. They appear in digital logic design, formal languages, networking protocols, game programming, robot programming, and more. Learning to reason about state machines profoundly changes the way a person thinks about technology, from vending machines and traffic lights to smartphones and the Internet.

Kodu was inspired by behavior-based robotics [1] and is a natural state machine language. Each page functions as a state, with "switch to page" actions providing the transitions [3]. Some actual robot programming frameworks directly support state machines, but in conventional languages it falls to the programmer to implement them using a combination of conditionals and variables.

Kodu states are more complex than in most state machine languages because pages contain multiple conditional rules, all of which are embedded in an implicit while loop [7]. So it takes some effort to construct simple problems where multiple states are the most natural solution. One example is a world with red apples to be eaten and blue ones to be grabbed. With a single page and one "pursue apples" rule, the kodu will go to the closest apple no matter the color. With separate red and blue pursue rules, the rule ordering principle will cause the kodu to eat all the red apples first, before grabbing any blue ones, or vice versa. But if we want the kodu to alternate between red and blue apples, we must put the two pursue rules and their associated consume rules on separate pages and switch back and forth between them.[2]

To scaffold students' reasoning about state machines, they are taught the standard graphical notation using labeled nodes and arcs, and they practice mapping between Kodu code and state machine graphs. They are also shown how to use the graphs to reason about program behavior. For example, given the state machine in Figure 3, will the kodu ever eat another apple after grabbing a soccer ball?

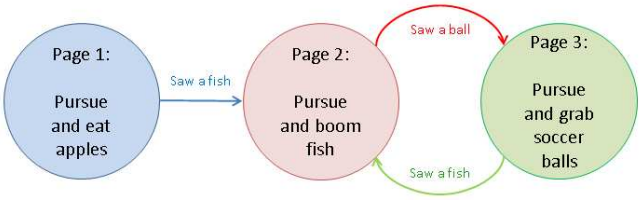

Figure 3: State machine reasoning problem.

Students use the Show Page As Color idiom to trace execution by having the kodu change its body color to indicate the current page. They also hand simulate a state machine, with one student playing the role of the kodu, to reinforce their understanding of page switching.

[2]We could instead use a variable to keep track of what color to pursue next, but that would just be implementing a state machine by hand, with more complicated rules.

6. METHODS

6.1 Participants

Twenty-three students participated in one of two week-long summer camps. Participants were recruited via emails to parents or teachers from the university's outreach office. There were 6 females and 17 males. The racial/ethnic breakdown was 14 White, 4 Asian/Indian, 1 Latino, 1 Multiracial, and 1 Native American. All were rising 5th or 6th graders. 21 students completed the entire camp; 1 boy in each camp missed the last day.

4 out of 23 students indicated that they had not programmed prior to the Kodu Camp. 12 out of 23 students had participated in 2 or more computing programs, with 5 having participated in 5+ computing programs. Students who had programming experience reported a wide variety of experience types, but only one had worked with Kodu before. Over half of the students had used Scratch (n=12); none had used Alice; 9 had used Minecraft; 9 had participated in Hour of Code; and 5 had done robotics. Students also reported some experience in Python (n=7) and HTML and Javascript (n=4).

6.2 Structure of the Curriculum

The curriculum consisted of the first five modules and half of the 6th module described in Table 2. On most days students completed roughly the second half of one module and the first half of the next one. Each module focused on one or two idioms or computational principles, illustrated with a collection of pre-made worlds that presented some sort of game or challenge. With the first world the instructor introduced the topic of the module, using the flash cards and/or tiles as appropriate, and took the students through the solution. In the second world students were asked to apply the idea in a slightly different context on their own. A third world might be used to introduce supplementary material. Students then completed a paper assessment, and after handing in their pages, the instructor presented the correct answers. Students were then given a challenge world where they were asked to implement a variant of the idiom or key concept, or apply it to a slightly more complex case. Finally, each day ended with 30-45 minutes of free exploration where students could make their own worlds or play in the worlds they'd previously written.

The assessments contained a combination of fill in the blank, multiple choice, and open ended items. Questions were of several types: review of terms ("What new idiom did you learn today?"); recognition of rule types ("Is this a pursue rule or a consume rule?"); synthesis of rules ("Write the rules to pursue and consume green fish."); recognition of idioms ("Are the rules below an example of Default Value?"); and mental simulation ("Given the rules below, in what order will the kodu eat these apples?", or "What will happen if this rule's indentation is removed?").

6.3 Interactivity and Gamification

All the worlds require students to program the kodu to solve a problem autonomously. Initially students would run the world and passively observe the result. However, several taught themselves the Let Me Drive idiom and chose to guide the kodu with the game controller. This was fun but didn't demonstrate mastery of the concepts they were supposed to be learning. For the second camp, we compromised by introducing an additional character in some worlds that students could control while the main character solved the problem. In the Apple1X world, for example, while the kodu pursued and ate the apples, there was a flying fish that students were taught to make drivable. The flying fish could help the kodu by pushing apples toward it, or harrass it by bumping it or knocking apples away. It could even maneuver the apples to fall into a pit, making them unreachable by the kodu. The kodu "won" if it managed to eat 4 of the 5 apples; the flying fish "won" if it got 2 apples into the pit before time ran out. Gamification increased student engagement, seemingly without compromising learning goals.

Table 2: Kodu curriculum modules.

Module	Content (• *idioms in italics*)
1	Characters, objects, and rules • *Pursue and Consume* • *Let Me Drive*
2	Pattern matching: color filters
3	Indentation (rule dependency); Scores • *Do Two Things* • *Count Actions*
4	Conflict and rule ordering • *Default Value*
5	State machines • *Show Page As Color*
6	Paths • *Follow the Yellow Brick Road*

6.4 Data Analysis Procedure

To investigate students' understanding of the material, we analyzed their responses on the paper assessments. In Phase 1 we marked each response as correct or incorrect, and then used a grounded theory approach [4] to find trends and themes underlying the incorrect responses. Incorrect answers were coded and grouped by similarity, and each coded group was given a descriptive label.

In Phase 2 we further categorized incorrect responses based on possible causes of error: (1) confusing wording or structure of the question, (2) lack of understanding of a well-formulated question, (3) a misconception about Kodu computation, or (4) negative transfer from a previous activity or question. Questions that we believe had confusing wording or structure were excluded from further analysis. Responses that were left blank, or that did not properly address the question, were categorized as "lack of understanding". Responses were categorized as "misconceptions" when we found multiple instances of the incorrect answer across both camps, and when a plausible reason for the incorrect response could be identified.

A legitimate misconception should be recognizable by a consistent pattern of wrong answers across questions. In Phase 3 we assembled the evidence in support of each hypothesized misconception. Due to space limitations, we will focus here mainly on misconceptions about rule ordering.

7. FINDINGS

7.1 Evidence of Student Learning

Students demonstrated their understanding of lawfulness

by predicting the behavior of Kodu characters, applying idioms, and reasoning about rules. For example, in questions measuring students' knowledge of Kodu's matching behavior (Modules 1 and 3), 82% ($n = 19$) and 86% ($n = 20$) of students correctly indicated that the kodu always went to the closest apple. Similarly, in Modules 1 and 2, 86% ($n = 20$)and 91% ($n = 21$) of students correctly applied the Pursue and Consume idiom to new situations. In Module 3, 73% ($n = 17$) of students correctly identified Pursue and Consume and Count Actions in a novel code snippet. Prior programming experience was not a predictor of correct answers in assessments in any of Modules 1–4.

Predicting future behavior requires mentally simulating state changes, which all the students were able to do. In Module 1 they were shown a diagram of a kodu and five red apples (similar in format to Figure 4, top) and asked to label the apples with numbers to indicate the order in which the kodu would eat them. 91% ($n = 21$) got it right.

7.2 Misconceptions About Rule Ordering

Many students had difficulty mastering rule ordering in Kodu. The rules on a page run repeatedly; WHEN parts are evaluated in parallel and DO parts are performed sequentially if the WHEN part was true. If there are no conflicting actions, rule ordering is irrelevant. We tell students this explicitly. On the other hand, if rules do have conflicting actions, the earlier (lower numbered) rule prevails. So order does matter, but the rules do not constitute a sequential procedure. Several factors work against students assimilating this knowledge, discussed below.

```
Q2:  [1] WHEN see apple DO move toward
     [2] WHEN bump apple DO eat it

Q3:  [1] WHEN see red apple DO move toward
     [2] WHEN see blue apple DO move toward
     [3] WHEN bump apple DO eat it

Q4:  [1] WHEN bump apple DO eat it
     [2] WHEN see blue apple DO move toward
     [3] WHEN see red apple DO move toward
```

Figure 4: Module 4 rule ordering questions (presented to the students as tile sequences, not text).

Two pursue rules. Students did better on questions that required them to apply the rule ordering principle to an example program than on questions that discussed rule ordering in the abstract. Given the example programs in Figure 4, students were asked to label each apple with a number 1-4 to show the order in which it would be eaten. With the color-agnostic pursue rule (Q2), 78% ($n = 18$) correctly applied the "closest match" principle. With a slight a decrease in performance, this response rate is consistent with prior demonstration of ability to mentally simulate the Pursue and Consume rule. Correct responses resulted in a 1-2-3-4 (red, blue, blue, red) labeling pattern, reading left to right. Of the five wrong answers, two completely misunderstood the question, and three responded 1-2-4-3, which produced an alternating red/blue pattern even though the rule didn't mention color. The alternating red/blue pattern might be explained by misjudging the distances between the second apple and the third and fourth, which is consistent with a simple elliptical path from the kodu's initial position through all the apples.

With separate red and blue pursue rules in Q3 and Q4, 70% ($n = 16$) correctly predicted a 1-4-3-2 pattern (red, red, blue, blue) in Q3. When the rule order was reversed in Q4, the same 16 correctly predicted 4-1-2-3 (blue, blue, red, red), and were not confused by the consume rule preceding the pursue rules. Of the 7 students who answered incorrectly, the same two students who misunderstood Q2 misunderstood Q3 and Q4. Of the remaining 5 students who answered incorrectly, two students thought the pursue rules would alternate in both questions based on the rule order (Q3: 1-2-4-3 or red, blue, red, blue, and Q4: blue, red, blue, red). When there are two pursue rules, thinking of a page of rules as a sequential procedure is consistent with alternating between red and blue apples. These students had not alternated on Q2; they were among the majority who correctly applied "closest match".

Two students followed a "closest object" principle and ignored the color filters in Q3 and Q4, resulting in the same answer as Q2. One student apparently thought the rules would interfere with each other, resulting in the kodu either eating only the first red apple and stopping (Q3) or eating no apples (Q4).

None of the three students whose answers alternated with the single pursue rule in Q2 alternated on Q3 or Q4. Two of them actually got both Q3 and Q4 correct. The third used "closest object" for both. This suggests that their answers on Q2 may not have been deliberate alternation.

Abstract reasoning about rule ordering. When asked which of the two idioms Pursue and Consume and Default Value relied on rule ordering, only 34% ($n = 8$) answered correctly that only Default Value did, 12 thought that both did, 1 said neither did, and 2 said only Pursue and Consume did. When asked why rule ordering matters for some idioms and not others, only 5 wrote answers that had some flavor of correctness; 3 wrote answers that weren't obviously wrong but didn't mention rule priority or conflicting actions; 12 gave answers that were either irrelevant or incoherent; and 3 indicated they thought rules executed sequentially, e.g., "you need to pursue for rule 1 and consume for rule 2".

Pages as sequential procedures. Even after students had been taught about the rule cycle in module 3 (and had run a program that repeatedly increments a score, watching the value run up at a blistering rate), many had trouble overcoming the tendency to view a page of rules as a sequential procedure. There are several reasons for this.

First, the Kodu rule editor numbers the rules, giving them the appearance of sequential steps. These numbers serve no purpose in the interpreter: they are there solely as an aid to discussion. And it's indeed convenient to refer to rules by number. But sequential numbering wrongly suggests sequential execution.

A second, compounding factor is that our example pro-

grams order the rules in a logical way. Since you cannot consume something until you've pursued it, "pursue" comes first in the idiom name, on the flash card, and in sample code. We find that demonstrating that the behavior is the same when the rules are reversed may not suffice to overcome students' bias toward sequentiality.

Third, movement is continuous. When executing the idiom a character will perform a great many pursue steps (50-100 per second) before the consume step, but to a student who does not yet understand the Kodu rule cycle, it looks as if the character performed a single pursue step several seconds in duration, followed by a shorter consume step: a sequential procedure. This false sequentiality becomes apparent when there are multiple pursue rules, because they don't take turns as some students assumed. The second rule can only take effect when the first rule no longer finds something to pursue.

7.3 Indentation and Rule Dependency

In module 3 students used the Do Two Things idiom to make the kodu play a sound when it bumped and ate an apple (rules shown in Figure 1). In a second world they used the related Count Actions idiom to count how many red hearts the kodu ate. In both cases, students were prompted to remove the indentation of the play or count rule and observe that the kodu played the sound continuously, or the score ran up continuously at a high rate.

What we wanted students to understand was that indenting a rule makes it dependent on the parent rule having a true WHEN part. But what we saw from interactions with some students is that they may view indentation simply as a way to block unwanted behavior, without fully comprehending the mechanism.

Given the rules below, students were asked when the kodu would play the coin sound:

```
Q5:    [1] WHEN see ball DO move toward
        → [2] WHEN DO play coin
       [3] WHEN bump ball DO eat it
```

18 out of 23 correctly answered "when it sees a ball" or "when it moves toward the ball". Three gave incoherent or vague responses. And two said "when it bumps the ball", which suggested they may have attached the indented rule to the rule below instead of the rule above, as in [6]. An alternative explanation is that they weren't reasoning about rule dependency at all, and this was an example of negative transfer.

7.4 Negative Transfer From Past Experience

One type of evidence for mastery is a student successfully applying what they've learned to novel, atypical situations. This requires strict application of computational lawsrather than reasoning by analogy to earlier situations. In several instances where students gave incorrect answers, they appear to have made this type of mistake.

In the indentation example Q5 cited above, the coin sound plays for as long as the kodu sees a ball. But in all the examples students had seen of Do Two Things or its special case Count Actions, the additional action had been indented under a "WHEN bump ... DO eat it" rule whose action negates the condtion. This may have led two students looking at Q5 to expect that the coin sound would play once, when the ball was bumped and eaten, with indentation preventing unwanted repetition.

Errors resulting from stereotyped reasoning might be prevented by more exposure to atypical cases. For example, we could have a world where Do Two Things implemented a persistent action as in Q5 rather than a transitory one, illustrating that indentation does not prevent repetition.

A similar negative transfer situation arose with the state machine problem in Figure 3. In all the examples students had seen, each state had some sort of goal, such as to pursue and consume an object, and a transition occurred when the goal was satisfied. But in Figure 3, states pursue their respective goals repeatedly, and a transition is triggered by the appearance of a goal object for another state. When asked to turn this diagram into rules, some students mistakenly attached the transition rule to the consume rule rather than having it look for the other type of object.

8. CONCLUSIONS

The power and simplicity of Kodu facilitate teaching students to reason formally about program behavior. We described a novel curriculum that fosters an appreciation of lawfulness through a combination of tile manipulatives, explicit computational principles, named idioms, and state machine formalism. Students demonstrated an understanding of lawfulness in concrete situations, but did less well on more abstract questions.

Two potential misconceptions were identified that are unique to Kodu's style of computation, one concerning rule ordering, and the other rule dependency (indentation). Future designers of Kodu-based curricula will want to check for these misconceptions when gauging student understanding.

Acknowledgments

Funded by a gift from Microsoft Research. Thanks to Brooke Ley, Swati Priyam, and Shweta Venkateswaran, who served as TAs for the Kodu camps.

9. REFERENCES

[1] M. B. MacLaurin. The design of Kodu: A tiny visual programming language for children on the Xbox 360. In *Proceedings of POPL '11)*, pages 241–246, 2011.

[2] N. Rusk. Scratch cards. Available at <https://scratch.mit.edu/help/cards>, 2009.

[3] K. Stolee and T. Fristoe. Expressing computer science concepts through Kodu Game Lab. In *Proceedings of SIGCSE'11*, pages 99–104, 2011.

[4] A. Strauss and N. Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage, Thousand Oaks, CA, 1998. 2nd ed.

[5] D. S. Touretzky. Kodu resources page, 2014. Available at <http://www.cs.cmu.edu/~dst/Kodu>.

[6] D. S. Touretzky. Teaching Kodu with physical manipulatives. *ACM Inroads*, 5(4):44–51, 2014.

[7] D. S. Touretzky, D. Marghitu, S. Ludi, D. Bernstein, and L. Ni. Accelerating K-12 computational thinking using scaffolding, staging, and abstraction. In *Proceedings of SIGCSE'13*, pages 609–614, 2013.

[8] J. M. Wing. Computational thinking and thinking about computing. *Philos Trans A Math Phys Eng Sci*, 366(1881):3717–3725, 2008.