Teaching Programming with Gamified Semantics

Ian Arawjo\textsuperscript{1}, Cheng-Yao Wang\textsuperscript{2}, Andrew C. Myers\textsuperscript{2}, Erik Andersen\textsuperscript{2}, and François Guimbretière\textsuperscript{1}

\textsuperscript{1}Department of Information Science, \textsuperscript{2}Department of Computer Science, Cornell University
\{iaa32, cw776, acm22, ela63, fvg3\}@cornell.edu

ABSTRACT
Dominant approaches to programming education emphasize program construction over language comprehension. We present \textit{Reduct}, an educational game embodying a new, comprehension-first approach to teaching novices core programming concepts which include functions, Booleans, equality, conditionals, and mapping functions over sets. In this novel teaching strategy, the player executes code using reduction-based operational semantics. During gameplay, code representations fade from concrete, block-based graphics to the actual syntax of JavaScript ES2015. We describe our design rationale and report on the results of a study evaluating the efficacy of our approach on young adults (18+) without prior coding experience. In a short timeframe, novices demonstrated promising learning of core concepts expressed in actual JavaScript. We also present results from an online deployment. Finally, we discuss ramifications for the design of future computational thinking games.

ACM Classification Keywords
K.3.2 Computer and Information Science Education: Computer Science Education; K.8.0 Personal Computing: Games

Author Keywords
Educational games; block-based programming; concreteness fading; novice programming

INTRODUCTION
There is a long tradition of constructionist approaches to teaching programming languages. Perhaps the most famous example is the language LOGO, introduced by Seymour Papert [61]. While specifics vary, these approaches focus on providing an engaging, interactive environment fostering self-discovery of basic programming concepts by trial and error. Some recent educational systems that take this approach are Alice [24] and Scratch [70]. Constructionist approaches have achieved widespread adoption in introductory programming classes [2] and online [23, 21], while visual feedback mechanisms inspired by Papert [61] continue to be integrated into widely-accessed beginner environments [44].

However, without mentorship or other external structure, construction-first approaches tend to expose learners only to a subset of the concepts that are important for general-purpose programming. Purely constructionist approaches are not well suited to teaching more challenging concepts, such as function definitions, variables, and higher-order functions [63, 54, 59]. Analyses of online Scratch programs show that many remain simplistic, often use no conditional statements, and overuse concurrency (among other issues) [58, 3, 4]. Part of the challenge of teaching programming is that more advanced programming constructs have complex semantics and are therefore not easily discoverable. Absent explicit instruction, beginning programmers will learn them—if at all—only through patient, tedious experimentation [15]. Understanding execution semantics is key to successful programming; a mismatch between what programs do and what novices think they do has long been recognized as a primary hurdle for novices [13, 30, 20, 67, 40].

In this paper, we explore the feasibility of a new approach: to embed comprehension of language semantics into a game in which students manually execute code. Building on the formal framework of structural operational semantics [65], we introduce computation as a succession of basic reduction steps. We present an example implementation of this approach: \textit{Reduct}, a puzzle game that builds student understanding of programming concepts. Our design utilizes theories of progression design and skill acquisition to scaffold concepts while incentivizing players to build correct mental models of code. The current end goal of the game is to teach up to the level of JavaScript’s \texttt{Map}, a sophisticated functional programming construct.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure1.png}
\caption{A level of the \textit{Reduct} programming game, faded to the syntax of Javascript ES2015.}
\end{figure}
We evaluated our prototype’s educational impact. For a population of young adults with no prior Computer Science knowledge, we found that after playing the game for a median time of about 34 minutes, participants correctly solved programming comprehension, recall, and JavaScript near-transfer post-test problems with an average accuracy of 75%. Moreover, quantitative and qualitative feedback, coupled with qualitative results of an online deployment, shows that overall, learners found the game engaging and wanted to play more. However, we found our application of concreteness fading [32], where graphics fade from concrete to abstract representations over time, showed little impact on the perception of the game or on learning outcomes. We discuss beneficial aspects of our design and implications from our study results.

RELATED WORK

Our research is informed by three primary threads of prior work: construction-first approaches, comprehension-first approaches, and methods for reducing barriers to entry. In this section we review prior research in each of these three threads.

Construction-first approaches

A construction-first approach focuses on the feedback loop of program construction: a programmer writes code, runs it, observes the output, and revises their program in an iterative process [62]. Seymour Papert first introduced this method of programming-by-discovery to children with his theory of constructionism [15]. A key feature of his work is the “turtle,” an embodied feedback mechanism representing the current draw position on the computer screen [56, 69]. The turtle provided children a body to reason about the logical world in a way analogous to their own bodies [61].

The constructionist method of teaching programming has been applied to educational technology in the proceeding decades. It is perhaps best exemplified by LOGO’s successor Scratch [70]. Scratch builds on LOGO’s initial ideas, such as allowing users to customize the appearance of their feedback cursor, direct multiple cursors, and snap together blocks of code. Scratch’s ideas and approach have achieved widespread adoption for entry-level programmers, particularly at the K–12 level [47]. While we cannot survey all construction-first approaches here, some representative examples are [24, 42, 75, 35, 79, 22, 43, 29, 76] and the most prominent tutorials featured on Code.org for the 2016 Hour of Code event [23].

One drawback of the constructionist approach is that language semantics can only be inferred indirectly through program output: i.e., by trial and error. Historically, construction-first environments were meant to be used in close consultation with a skilled mentor [62]. Without outside structure or mentorship, students are unlikely to discover the semantics and utility of higher-level concepts [54]. The danger of this is that novices can end up fiddling with their program until it does what they want [15], often without understanding why it works [66]. Empirical analyses show that constructionist learning environments struggle to teach higher-level concepts, such as boolean comparison, variables, and functions, and can over-encourage use of concurrency [54, 58, 59, 3, 4]. Moreover, students can be distracted by entertaining but extraneous features such as choosing sprites and drawing [15, 3, 27].

Comprehension-first approaches

In contrast, a comprehension-first approach focuses first on building an accurate mental model of language semantics, with the goal of building intuition that students can later use to effectively write programs with less semantic misconceptions [30]. In other words, it first teaches novices how to read code before letting novices write code from scratch. Through comprehension-first approaches, programmers may be exposed to common patterns that they can later use to solve problems, rather than having to come up with those patterns themselves [60, 39].

Instances of this approach include example-based learning, in which novices modify lines of existing code to achieve a goal rather than writing code from scratch [77, 66, 49]. Existing tools such as Online Python Tutor [36] improve comprehension by visualizing execution traces, although these tools remain either supplementary or for more advanced programmers. A recent exception is Gidget [49], a debugging-first puzzle game for teaching programming concepts. Though it indirectly teaches the semantics of a custom rather than a general-purpose language, Gidget showed promising results for teaching novices to use loops and functions in a short amount of time.

Reducing barriers to entry

Educational technology can provide structured learning environments in the absence of direct mentorship. Approaches include puzzle games, tutorials, and cognitive tutors (e.g., [35, 21, 9], respectively). Prior research suggests that puzzle-based learning of code concepts can be faster and more effective than tutorial-based learning [37], and that puzzle-based, comprehension-first approaches can potentially teach novices faster than a tutorial-based, construction-first approach [50]. Tutorials can also be detrimental in games where behaviors can be discovered through play alone [6]. Furthermore, puzzle-based approaches can introduce concepts with minimal natural language explanations, potentially reducing barriers to learning for non-English users and underrepresented populations [57]. We note that non-English speaking populations are often ignored in the US-based computer science education literature. Partly as a result of this prior research, we chose to design a puzzle game.

Block-based programming

Syntax is a known barrier to entry for novices [64]. To combat this, a long trend of research has developed graphical, block-based manipulatives whose visual properties signal how they snap together [56]. These manipulatives make syntactic errors impossible. Block-based code is well-represented in novice programming environments today, for instance by the puzzle-piece metaphors in Scratch [70], Blockly [34], and Tern [38]. Research suggests that block-based environments also improve engagement and comprehension over text-based environments, even outside of programming [46, 80]. Our design also adopts block-based manipulatives, in line with prior work.

Concreteness fading

When learning mathematics, novices have been known to “self-handicap,” or be discouraged by the appearance of mathematical notation [10, 71]. Partially to overcome this barrier,
concreteness fading (CF), first conceptualized by Bruner [17], teaches abstractions by gradually “fading” concrete representations to symbolic abstractions [55]. Recently, the game DragonBox [78] applied concreteness fading to algebra, becoming the #1 bestselling game in the Apple App Store in its home country of Norway [51]. DragonBox gamifies algebra by teaching linear equation solving via equations that are initially presented in a pictorial form and gradually faded to mathematical notation. In a large study, 96% of K-12 students who played for more than 90 minutes were able to solve three linear equations in a row with no errors in the game interface [52].

However, while some studies provide support for CF [31], other studies have not shown positive results, and so evidence remains divided on concreteness fading’s effectiveness [32]. A study of DragonBox that examined whether students improve on pen-and-paper tests did not find a statistically significant effect [53]. Fyfe et al. [32] speculate that effectiveness may vary based on demographics and the number of fade stages.

DESIGN
We set out to design a comprehension-first approach to teaching programming that departed from previous work. Rather than using example-based learning [49], our key design idea was to gamify the evaluation steps a computer takes to execute code. During gameplay, the player performs the steps of computation; in effect, the player takes the role of the computer executing code. Players internalize the semantics of each rule by solving a series of puzzles using these rules in combination.

Choice of computational model
Programming language theory offers several models of computation that break down execution into a recursive series of deterministic rules. After reviewing several formalisms such as lambda calculus [19] and linear logic [1], we settled on using the rules of small-step operational semantics [65] to provide the basic units of gameplay. This semantics defines a set of reduction rules that provide an algorithm for the evaluation of program expressions (see the Appendix for a brief overview). Our game progressively introduces reduction rules for a subset of JavaScript ES2015, allowing the player to construct increasingly complex computations. We settled on JavaScript ES2015 for its notational brevity, functional flavor, versatility, and widespread adoption. In 2015, JavaScript was the most active programming language on GitHub.com [48].

Performing reductions in practice
We now illustrate reduction rules in practice with the example shown in Figure 2. Here we see the player perform two reduction steps using our block-based manipulatives, which were inspired by Scratch [70]. First, the player binds a star value to the input of an anonymous function:

![Figure 2](image)

**Figure 2:** From left to right: A star constant is dropped over an anonymous function as input, binding x and producing star == star. Next, the player clicks to reduce the equality statement, producing the terminal value of true.

The player enacts this rule by dropping star over the function parameter to perform substitution in the function body. The player can click the resulting expression to reduce it to the boolean value true:

\[
((x) \Rightarrow x == x)(\text{star}) \rightarrow (\text{star} == \text{star})
\]

Unwittingly, the player is simultaneously building an execution that follows the operational semantics of the JavaScript language.

Gameplay overview
In each level of Reduct, the player sees three areas of gameplay: a board (a), a goal (b), and a toolbox (c) (Figure 3). The player is presented with a set of expressions in all three areas. For every level, the player’s goal is to successively reduce expressions on the main board to create an exact match between what is on the board and the expressions in the goal box. This match can be achieved by dropping an expression into another expression (for example to bind an expression to x), by deconstructing an expression, or by clicking on an expression to trigger a reduction. The toolbox is used to encourage forward planning and will be described in the next section.

Example levels
Figure 3 shows a typical level that players are presented with almost halfway through the Conditionals section. On the game board are incomplete equality and ternary (conditional) statements, concepts introduced in prior levels. With the goal of getting a single star value on the board, the player must select the appropriate primitive from their toolbox to complete the equality statement. For the solution, the player executes the following reductions:

\[
(\text{tri} == \text{tri}) \rightarrow \text{true} \\
(\text{true} ? \text{star} : \text{null}) \rightarrow \text{star}
\]
According to theories of skill acquisition, if the search space becomes too large for players to brute-force, the solution, we introduced the toolbox as a means to foster forward planning [73]. Expressions in the toolbox do not count toward or against the goal. The player can move expressions out of the toolbox and onto the board as they choose; however, once moved, an expression cannot be placed back into the toolbox. The decision is irreversible—the placed expression now counts toward the goal—and the player must use Reset (top right corner) if they have made a mistake. Because of this, players must plan the solution in their mind before executing.

**Reward structure**
When the player succeeds, the pieces on both the board and goal box glow, a short victory chime plays, and “You Win!” text fills the screen. This is the only reward structure in the game. We adopted this minimal reward structure in response to criticisms of gamification, which urge caution when prematurely adopting secondary reward structures [5, 3, 11, 18]. We also hoped to avoid factors which might complicate analysis of our evaluation.

**Concreteness Fading**
To address the potential problem of self-handicapping [10, 71], a unique feature of our approach is the application of concreteness fading (CF) to programming education. While prior tools like Toontalk [41] have explored accessible concretizations of abstractions, these abstractions have not been faded to abstract forms over time.

In implementing CF, we drew inspiration from *DragonBox* (DB) [78], a game to teach algebra. DB fades representations from cartoon monsters, to dice, to numbers and variables. We applied this approach to constructs in Reduct (e.g., Figure 4). In Table 1 we show the different representations of concepts introduced in Reduct, including the metaphor behind each concrete variant. We show in Figure 5 side-by-side the different stages of fading for functions, equality, and conditionals. Functions are first introduced with holes for input and a pipe for output, while equality is introduced as a mirror, and conditionals are introduced as a lock opened by a key representing a Boolean value. The function and conditional concepts have three or more stages of fading, while all other concepts have two. Some concrete variants can be seen in context in Figure 4.

To reinforce the correspondence between variations, Reduct includes an explicit transition between one level of concreteness to the next. Animations between concrete and abstract variants have shown higher transfer results than fading without such animations [72]. When a new level of abstraction is introduced, the element sparkles with green stars, as shown in Figure 6. Note that the transition between levels of concreteness is not perfectly aligned with each concept’s introduction, as a concept must first be instilled before it can be appropriately faded.
<table>
<thead>
<tr>
<th>Concept</th>
<th>Concrete Variant</th>
<th>Metaphor</th>
<th>Abstract Variant</th>
<th>JavaScript ES2015</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives</td>
<td>★ ★ ★</td>
<td>Shapes as types</td>
<td>★ ★ ★</td>
<td>Strings e.g. “star,” “rect”</td>
</tr>
<tr>
<td>Booleans</td>
<td>⚪ ⚬ ⚪</td>
<td>Key / Broken key</td>
<td>true, false</td>
<td>true and false</td>
</tr>
<tr>
<td>Equality</td>
<td>🌅 🌅 🌅</td>
<td>Reflecting glass</td>
<td>🌅 =</td>
<td>Equal to ==</td>
</tr>
<tr>
<td>Conditional</td>
<td>🗝️ ⚠️</td>
<td>Lock and Key</td>
<td>🟢 ? ★ : null</td>
<td>Ternary if ?:</td>
</tr>
<tr>
<td>Function</td>
<td>⚒ ⚒ ⚒</td>
<td>Hole and Pipe</td>
<td>(x) =&gt; x</td>
<td>One-parameter arrow function (x) =&gt;</td>
</tr>
<tr>
<td>Collection</td>
<td>🙁</td>
<td>Bag of items</td>
<td>[★,★,★]</td>
<td>Bracket syntax e.g. [1, 2, 3]</td>
</tr>
<tr>
<td>Map over a set</td>
<td>🛡️</td>
<td>Factory</td>
<td>★.map(★)</td>
<td>Array.prototype.map()</td>
</tr>
<tr>
<td>Null</td>
<td>(does not appear)</td>
<td></td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

Table 1: Concepts in the game and their JavaScript equivalents.

**Special considerations**

We should note that throughout the game, we sometimes deviate from a strict interpretation of a given construct in deference to simplicity and improving player engagement. For example, rather than returning an array, our `Map` function “spills” the contents of the array onto the board. As instances of `null` automatically disappear in a ‘poof’ animation in the game, the player can effectively construct a filter function by mapping over a collection and returning `null` for those elements that don’t fulfill the specified condition.

**Progression Design**

In total, `Reduct` contains 72 levels stretching across multiple programming concepts. Our ultimate goal was to introduce the player to the basic notion of `Map` to filter elements in a bag. More specifically, we set forth to have participants understand the following JavaScript statement:

```
[★,★,★].map((x) => (x == ★) ? ★ : null)
```

We felt that the `Map` concept represented a good balance between expressiveness and complexity (`Map` is typically introduced at around the sixth lecture in an advanced programming class). The different concepts present in the filter statement produce a dependency graph (e.g., conditionals require Booleans). While there are many different ways to traverse this graph, after several iterations in which members of the team played the game repetitively, we settled on introducing concepts in the following order: `Functions and Primitives, Booleans and Equality, Conditionals, Collections, Map`.

In our design, each concept shown in Table 1 is introduced over roughly ten levels, often starting with a single-step reduction and progressively exploring increasingly complex aspects of the concept. We show in Figure 7 where each concept is faded in the CF version of the game. Unlike textual introductions which must present many concepts “at once”—the complexity of which may unnecessarily hamper or confuse novices [8, 2]—our approach manages cognitive complexity by introducing concepts in relative isolation, then gradually mixing previously learned concepts to strengthen understanding of the whole, in a repeating pattern informed by theories of elaboration and flow [68, 25] and lessons from cognitive tutors [9].

For example, the first level contains only the identity function `(x) => x` and a `★` primitive. Dropping the star into the function is the only move. By the last level in this sequence, players are still restricted to functions and primitives, but they...
must now also understand replication and first-class function application (functions as input to other functions). The solution path is not as straightforward; instead of solving the level in one move, at least three are required. Similarly, equality is introduced in a level where the only move is to click the expression, while in the final level of that sequence, participants in our online evaluation reached over 500 unique game states in their search for the solution. The bottom of Figure 7 shows the expected difficulty as the relative number of optimal moves per level.

Implementation
Reduct was implemented in HTML5 and JavaScript ES2015, the language it teaches. It has been tested on desktop Chrome and Safari web browsers, and the version of the game as presented in this paper is publicly available online as of this writing.

IN-LAB EVALUATION
To evaluate how well Reduct’s design addresses our objective of teaching code comprehension, we conducted an in-lab evaluation focusing on how players react to the game’s basic principles, the game dynamics, and the ability of players to transfer what they learn in the game to an out-of-game problem. We were also interested in the potential impact of concreteness fading on learning, engagement, and play behavior.

Experimental Design and Procedure
Each participant was asked to play the game from the beginning until they reached the final level (72 levels in total), taking as long as they wanted to perform the task. For each game level, participants could press a reset button to reset the current level if they made an error. They could also use a “Skip” button in case they got stuck on a particularly difficult level. After completing the game, there was a 5-minute break before completing a performance survey testing recognition, recall, and near-transfer of concepts, described in the post-test design section below.

To avoid any skill transfer, we used a between-subject design to evaluate the impact of concreteness fading: half of our participants ran a game using concreteness fading (referred as CF henceforth), while the other half played the same game but with all visuals fully faded (referred to as FF henceforth) (see Table 1). Allocation was randomized. Participants were given the same procedure in both conditions, although participants in the CF group took an extra post-test on their knowledge of the concrete representations, which was later removed from the analysis. This test was administered after the abstract post test to reduce priming bias for the abstract post-test; thus, the two groups’ abstract post-tests could be directly compared. Finally, after the performance assessment we interviewed participants about their engagement using Likert scales as starting point.

Post-test design
Our abstract post-test contains 24 total questions, grouped into 3 sections testing recognition, recall, and near-transfer of programming concepts found in the game (Figure 8). There were an additional 2 difficult recall questions that served to probe players’ understanding, which we did not count towards our analysis. Our questionnaire used 9 multiple-choice questions to test recognition (Figure 8a), 9 fill-in-the blank questions to test recall (Figure 8b), 4 near-transfer comprehension questions, asking the participant to evaluate pure JavaScript code snippets without graphical scaffolding and type their response in the blank (Figure 8c), and 2 questions asking the participant to write a small JavaScript function to isolate an item in a collection (Figure 8d). Participants in the CF condition also
took a concrete post-test after the abstract post-test designed to identify where concreteness fading was the most useful. This had the same questions, but in concrete form.

**Qualitative measures**
At the very end of the evaluation, we asked participants whether the game was fun, easy-to-understand, whether they would play more, how difficult they found the game and whether the game was visually appealing. Participants in the CF condition were also asked whether they found the visualized metaphors helpful to their understanding. These questions were answered using a Likert scale from 1 to 5.

**Participants**
We recruited 24 mostly young undergraduates (18–25 years old, 16 female) at a large US university. Most participants signed up via an online recruitment system, while some signed up via flyers posted around campus. Participants were screened to have no recent experience in programming. Our exclusion criteria was the question: *In the last two or three years, have you written a program longer than 20 lines in any of the following languages or similar?* The programming languages explicitly listed were Python, JavaScript, Ruby, Java, C, C++, Lisp, Scheme, Racket, Matlab, Mathematica, and R. 22 of our participants came from non-mathematical majors such as Public Policy, Psychology, Biology, and Human Development; one came from Economics; one did not wish to mention their major.

**Results**
When computing averages and playtimes, we removed the first level from consideration due to possible confounding effects (e.g., the researcher may still be explaining the interface). Any level played for more than 5 min was considered an outlier and its time replaced with the maximum time for that cell.

**Playing the game**
Using a paired t-test with Bonferroni correction, there was no significant difference \( (p = 0.443) \) between play time for CF and FF: median playtimes were 33m37s and 34m33s, respectively. To understand how the players performed during the game, we analyzed our logs as a two-way Concept X Concreteness mixed design. We first consider the completion rate of each level and for this dependent variable, a mixed design two-way ANOVA shows no main effect of Concept \( (F(6,154) = 1.168, \ p = 0.326, \eta^2_F = 0.044) \) or Concreteness \( (F(1,154) = 0.551, \ p = 0.459, \eta^2_F = 0.004) \). There were no interactions between the two variables \( (F(6,154) = 0.551, \ p = 0.769, \eta^2_F = 0.021) \). We also looked at effort, which we define as the relative number of reduction steps players took for a given level over the minimum number of steps required to win \((\text{actual_moves - optimum}) / \text{optimum}\). The raw data is presented in Figure 9 and closely followed the expected difficulty of each level.

We also show the same data aggregated per concepts in Figure 10. A two-way Concept X Concreteness mixed design ANOVA on this dependent variable shows no main effect \( (F(20,551) = 0.878, \ p < 0.01, \eta^2_F = 0.449) \) with Map and Map2 being significantly lower than the other level \((p < 0.001)\). These interactions between the two variables did not reach significance \( (F(6,154) = 1.953, \ p = 0.08, \eta^2_F = 0.071) \). We further remark that the \( \eta^2_F \) effect size for these analyses was less than or equal to .006, which was very low effect sizes. To reach significance for playtime between CF and FF, we would need 205 participants per condition for a power of 0.9. To explore this possibility, we performed an online study with much larger group sizes (described in the next section).

**Learning Outcomes**
Since the game did not test the recall of syntax (for instance by asking them to type), on the post-test some participants entered minor syntax errors. We cleaned data in the following manner: For recall and near-transfer tests, we ignored capitalization errors (e.g., TRUE versus true). For one near transfer question whose answer was a String, we ignored whether or not quotes were around the solution (since our game did not teach the near transfer being the weakest, as can be expected, but no main effect of Concreteness \( (F(1,66) = 0.404, \ p = 0.527, \eta^2_P = 0.006) \) and no interaction between Concreteness and Learning Outcome. Overall, participants answered about
82% of the recognition questions, 75% of the recall questions and 58% of the Near-transfer function.

This expected trend was confirmed by direct participant feedback. 20 of 24 participants mentioned that the recall questions were more difficult for them in the interview. One participant stated, “The game was fun and addictive. I was focusing on figuring out the solutions by dragging those objects. I didn’t try to memorize them.” Another participant reported, “The second type of post-test questions were difficult. If I knew there would be questions like that, I would have paid more attention to those symbols in the game while I played it.”

For the two optional hard recall questions, only a few participants (one CF participant and three FF participants) were able to come up with an answer similar in syntax to the correct answer. This was consistent with our expectation. A consistent feature of their responses is the absence of the “(x) =>” arrow syntax. For those that remembered the syntax of equality, they jumped directly to equality, suggesting that they considered the function binding implicitly a part of Map.

Overall these results lead us to believe that our game was a success in introducing participants to basic concepts of programming. Probably the most surprising result was related to the low impact of concreteness fading on the outcome, a result consistent with our expectations. Mathematics interference

During our study, participants in both conditions exhibited a tendency to attempt to directly substitute for x by dragging a primitive over top of it. This is despite repeated levels which require players to apply an expression to the leftmost part of the lambda function (its input) to perform substitution indirectly. We further observed that 17 out of 24 participants reported the correct answer to the final near-transfer question, mapping the add function \((\chi) \mapsto \chi + 5\) over an array of numbers, despite both numbers and addition not being a part of the game. Moreover, the equality operator \(==\) was consistently confused for \(=\) in responses to the optional questions. Given the prominence of direct substitution for \(x\) in mathematics and the \(=\) sign for equality, we believe that this behavior may be the result of interference from mathematics knowledge.

ONLINE DEPLOYMENT

In order to evaluate Reduct on a larger scale, we posted a link to it through Reddit. 2942 players played Reduct as a result. The post received more than 100 comments. The two highest-rated comments started with “I love it!” and “This is cool.” The rating score increased to approximately 480 and the post hit the top of the front page. The median player played for 3 minutes and completed 8 levels. About 10% finished the game. In online deployments, when players are choosing to play, a sharp falloff with a long tail is common [5].

The online deployment contained an A/B test to further examine the impact of CF. 1502 players played the version with CF, and 1440 played the version without. We analyzed whether there was an effect for CF across concepts. In total, 355 players finished every level in the game (190 FF/165 CF). Looking at time to completion among these players (Figure 13), a
We also examined the impact of CF on player engagement, whether an extension of our approach can eventually transfer results are in-line with a similar trend from the in-lab study. We also examined the impact of CF on player engagement, measured as time played and levels completed. Using a Wilcoxon-Kruskal-Wallis two-tailed test, we did not find an effect for number of levels completed (Z = 0.32, p = 0.75) or time played (Z = -0.20, p = 0.84).

**DISCUSSION AND LIMITATIONS**

The results of our two evaluations seem to confirm that gamifying the operational semantics of a programming language such as JavaScript can be an engaging method of teaching programming concepts. In a larger context, our results provide evidence for the viability of comprehension-first approaches in programming education. This builds on prior work [77, 37, 50] suggesting comprehension-first approaches can be just as effective as construction-only models for teaching language semantics.

**Distinguishing features**

As mentioned in Related Work, many previous approaches rely on an embodied feedback mechanism [56, 47] (with some exceptions e.g. [74, 76]). While embodied feedback is a great introductory method for novices, especially children [56, 69], one drawback is that it can only teach language semantics indirectly. By gamifying reduction rules, our approach was able to teach semantics directly.

**Bridging block-based and general-purpose languages**

Our work represents an important stepping stone towards bridging the gap between block-based and textual environments [16]. Many previous tools (e.g. [24, 70, 49]) teach custom languages to lower barriers to entry [64], with the trade-off that novices cannot directly apply their coding knowledge outside of the learning environment [45]. To help mitigate this trade-off, some block-based environments now offer a correspondence between visual and general-purpose languages [34, 12]. Building on the benefits of block-based approaches, our design was able to introduce basic semantics of a general-purpose language directly by introducing and elaborating on concepts one-by-one [68]. In future work we hope to test whether an extension of our approach can eventually transfer novices to an actual JavaScript coding environment.

**Minimal reward structure**

A common concern [18] with educational games is whether they can introduce concepts without relying on secondary reward structures—such as badges, points, or narrative—to improve engagement, as these structures can distract from learning goals [5, 3, 11]. We believe this work illustrates that one can gamify programming without a secondary reward structure and still remain engaging. Our approach was able to introduce language concepts through construct behavior alone, and without tutorials that hinder applicability to non-English contexts [57]. Key enabling features are our progression design and our code replication and partial writing mechanics. Taken together, these allowed us to scaffold complexity to keep players engaged [9, 26] while fostering development of accurate mental models of language semantics [28].

**Ineffectiveness of concreteness fading**

Although participants found our approach engaging and our post-test results seem promising, our comparative evaluations showed no effect for concreteness fading. This is especially significant in the online case, where we had over 150 participants in each condition. Below, we discuss three possibilities for why we did not find an effect for CF.

**Implementation of CF**

There has been debate over how CF should be implemented [32, 31]. When implementing CF for Reduct, we tried to adhere closely to DragonBox’s (DB) implementation. However, our CF design differs from DB in two respects: concepts remain faded once faded (a feature of almost all previous studies [32]), and several concepts have only two graphic variants. Regarding the latter point, two previous studies with undergraduates and two variants found conflicting results [33, 14]. Fyfe et al. [32] speculate that three fade stages are necessary to achieve consistent success. Our fading was constrained by the length of our progression, where oftentimes there was only room for two variants. Future work could explore whether sporadic fading or the amount of fade stages alters CF’s effectiveness.

**Study demographics**

Fyfe et al. [32] speculate that the effectiveness of CF may depend on student background and readiness for abstract representations. Even though many of our participants were not in technical majors, their acceptance into university, predicated on high performance in standardized tests (e.g. SAT), required prior achievement in mathematics. The study may also have a selection effect where those afraid of programming most likely did not show up. As such, our sample may be-and-large not experience self-handicapping. Similarly, our online participants came from a novice programming forum, likely causing the same selection effect and demographic skew. A new study with children would need to be conducted to help determine whether CF has an effect for a population that self-handicaps [31]. However, we note that a few prior studies with undergraduates did show an effect for CF [33, 55].

**Evaluation of near-transfer**

Our post-test recognition and recall questions included Reduct’s graphical scaffolding around code snippets. While our near-transfer questions tested transfer to code without scaffolding, in retrospect, the number of questions and sample size was not large enough to detect an effect. We note that some prior studies found that CF’s effectiveness was limited to transfer of concepts to new representations [55, 32], while Fyfe et al. [31] found that participants with high prior knowledge only showed benefits for CF when given a more challenging problem. Future work in this space might explore the effect of game-based CF on near-transfer.

**Design limitations**

Our results for recall and optional question types suggest that we should extend our design to incentivize recall ability, for...
instance by requiring players to type out the syntax of an expression in order to attain or activate it. Though our design focused solely on recognition, in order to become programmers novices must also learn how to construct programs to solve problems [15]. With our approach, it is currently unclear what the optimal sequence and balance between construction and comprehension would be. Future research in this space might investigate how an oscillating mix of construction- and comprehension-based approaches compares to prior models.

As well, while Reduct fades to JavaScript text, the graphical scaffolding (e.g., colors, the shape for boolean type, etc.) is not completely faded. A longer version of the game could fade even the colors and shapes to approximate the look of real code (which might include a syntax coloring scheme common to plain text editors). The appearance of brackets <> in one participant’s (P8) optional question response suggests that some of the graphics interfere with learning syntax and should become standardized in the late game. As an aside, we also neglected quotes when representing literal string constants, which is a distinction that could appear later.

As presented here, Reduct did not cover core concepts of sequential execution, variables, function naming, and assignment. A future progression would need to incorporate these concepts in order to move from program comprehension to programming in full.

CONCLUSION AND FUTURE WORK

We presented Reduct, a comprehension-first approach to teaching programming by gamifying operational semantics. We presented results from a lab study demonstrating that novices can learn programming concepts by playing the game. We also presented results from an online deployment showing that the game was well received. However, we found inconclusive evidence for the impact of concreteness fading.

We believe that we have only scratched the surface of gamifying operational semantics. In future work, we plan to further investigate the potential benefits of concreteness fading for programming by testing it on younger audiences who may be more initially averse to programming formalism. We also intend to expand the content of the game. Ideally, players will eventually be able to learn the core set of constructs for commonly-used programming languages through continued play. Furthermore, to improve recall, we also intend to add fading of not just the visual elements but also the input mechanisms, so that the player learns new concepts through block-based manipulatives but eventually types code directly.

APPENDIX

Operational Semantics

An operational semantics for a programming language is a precise description of how to carry out the execution of a program written in that language. It differs from other kinds of language semantics such as axiomatic or denotational semantics in that it describes the mechanical steps involved in running a program.

A popular form of operational semantics is structural operational semantics, introduced by Plotkin [65]. While not all operational semantics needs to be mathematical, this style of operational semantics defines program execution as a set of rules for rewriting the syntax of a program. In particular, reduction rules specify how to rewrite certain program expressions into equivalent, but simpler expressions. By applying these rules repeatedly, the execution of the program makes forward progress, eventually arriving (if the program terminates) in an expression that represents the final state or result of the program.

For example, a structural operational semantics for arithmetic might allow reductions such as “2 + 2 → 4” and “3 + 4 → 7.” Both of these allowed reductions would be expressed as instances of a more general rule:

\[ \frac{n_3 = n_1 + n_2}{n_1 + n_2 \rightarrow n_3} \]

This rule states that the expression \( n_1 + n_2 \), where \( n_1 \) and \( n_2 \) are any numbers, may be replaced with the expression \( n_3 \), where \( n_3 \) is the number that is the sum of \( n_1 \) and \( n_2 \). Note that the symbol + in the conclusion (bottom) of the rule is just a symbol, whereas the + in the premise of the rule represents actual addition.

The Reduct language is based on the lambda calculus [19], which has only three syntactic forms (variables, function definitions, and function applications), and just one reduction rule, known as \( \beta \) reduction. Using the syntax of Reduct, this rule appears as follows:

\[ (\lambda x \Rightarrow e \Rightarrow e' \rightarrow e' / x) \]

Note that the expression to the right of the arrow is not syntax in the language; rather, it represents the result of substituting expression \( e' \) for all free occurrences of \( x \) in the function body \( e \). Remarkably, this single reduction rule is powerful enough to express all possible computations.

REFERENCES


78. We Want To Know. 2013. DragonBox. PC Game. (2013).
