

Teaching Programming with Gamified Semantics

Ian Arawjo¹, Cheng-Yao Wang², Andrew C. Myers², Erik Andersen², and François Guimbretière¹

¹Department of Information Science, ²Department 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 *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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CHI 2017, May 06 - 11, 2017, Denver, CO, USA

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4655-9/17/05...\$15.00

DOI: <http://dx.doi.org/10.1145/3025453.3025711>

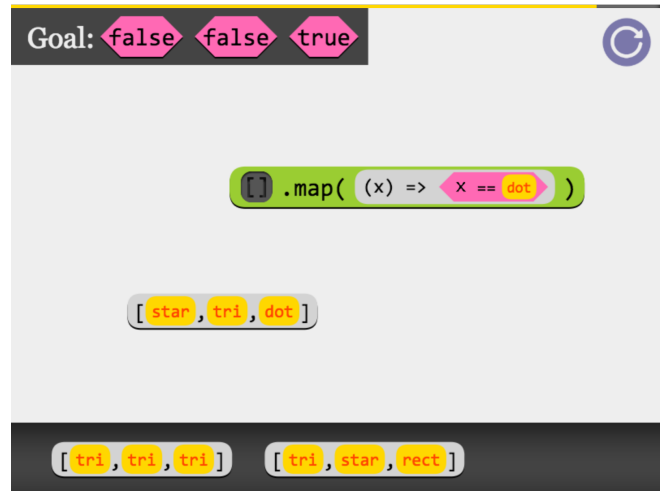


Figure 1: A level of the *Reduct* programming game, faded to the syntax of Javascript ES2015.

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: *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 `Map`, a sophisticated functional programming construct.

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:

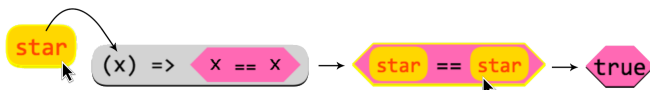


Figure 2: From left to right: A *star* constant is dropped over an anonymous function as input, binding *x* and producing *star == star*. Next, the player clicks to reduce the equality statement, producing the terminal value of *true*.

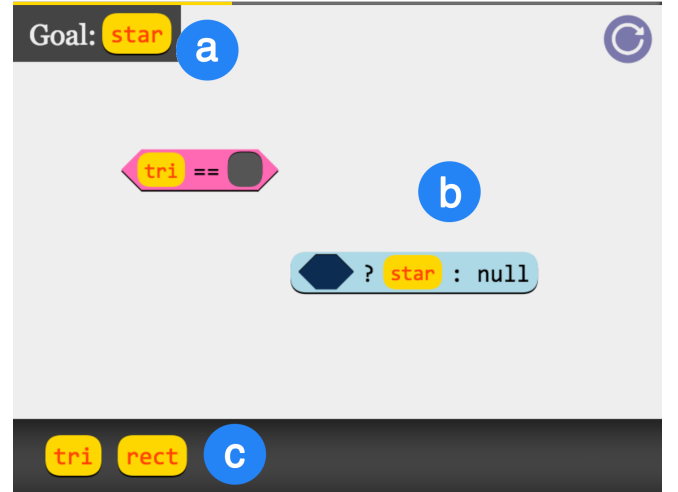


Figure 3: Level 25 with fully-faded representations. (a) The goal, a *star* value. (b) The game board, with equality and ternary expressions. (c) The toolbox, with two primitives. Players must construct a *true* value to unlock the *star*.

$$((x) \Rightarrow x == x)(\text{star}) \rightarrow (\text{star} == \text{star})$$

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*:

$$(\text{star} == \text{star}) \rightarrow \text{true}$$

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:

$$\begin{aligned} (\text{tri} == \text{tri}) &\rightarrow \text{true} \\ (\text{true} ? \text{star} : \text{null}) &\rightarrow \text{star} \end{aligned}$$

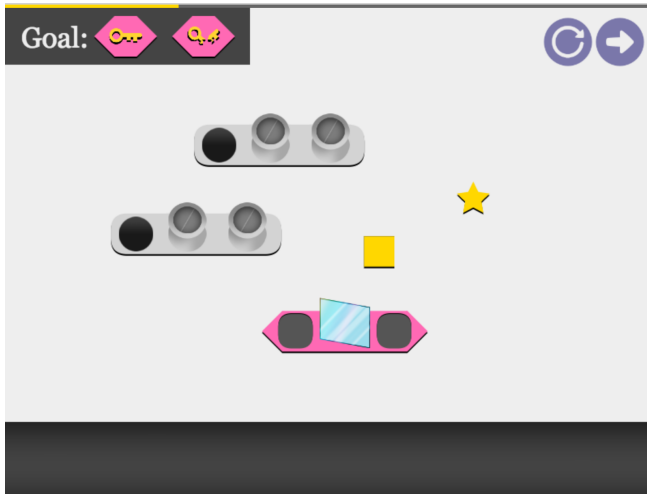


Figure 4: The hardest level in the Boolean section, with concrete representations. Reduction steps are not clear and require the player to recall how the elements on the board reduce to values.

To illustrate a more challenging puzzle, the final and hardest level in the Boolean section is depicted in Figure 4. In this level, it is not at first clear how to reach the goal. A forward-thinking strategy is to realize that in order to get one key and one broken key (true and false, respectively), one needs 3 of one primitive (e.g., 3 squares) and 1 of the other. Once the player recognizes this fact, the solution follows: duplicate one primitive, place one of the resulting primitives into the equality expression, and duplicate that expression. The player can then put the remaining two primitives into the two equalities and click them to reduce, producing one key and one broken key.

Fostering Learning

Our core motivation when designing *Reduct* was to develop the player’s mental model of language constructs over the course of gameplay. This design choice was motivated by substantial research which suggests that incorrect mental models of code are the main cause of novice misconceptions [13, 30, 20, 67, 40]. To accomplish this goal, we applied theories of skill acquisition and progression design, described below.

Skill Acquisition

According to theories of skill acquisition, if the search space of a problem becomes too large, novices must develop coping strategies to progress [7]. When mastering games, players gain expertise by developing “maxims” to reduce complexity and think several moves ahead, which in turn depends on competence in basic game rules [28]. Our design utilizes this principle to cement novices’ competence in language semantics over time. From a puzzle design standpoint, this approach requires presenting levels with search spaces too large for players to brute-force.

Unfortunately, programming constructs are heavily deterministic and lead to only a few possible combinations. Thus to increase the complexity of search space in many levels, our design introduces two features: “holes” or *partial writing* of expressions (e.g., `_==_`), as well as the ability to *replicate* expressions. Replication was inspired by lambda calculus

[19], where $\lambda x. (x\ x)$ is a function in whose body the parameter x appears twice. In *Reduct*, unlike lambda calculus, the expression $(x) \Rightarrow x\ x$ signifies a function that outputs two copies of its input. Note that this is one point of difference from JavaScript, where one cannot write $(x) \Rightarrow x\ x\ x$ to construct a function with multiple return values.

Toolbox

To further limit the player’s ability 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.

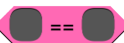
Concept	Concrete Variant	Metaphor	Abstract Variant	JavaScript ES2015
Primitives		Shapes as types		Strings e.g. “star,” “rect”
Booleans		Key / Broken key		true and false
Equality		Reflecting glass		Equal to ==
Conditional		Lock and Key		Ternary if ? :
Function		Hole and Pipe		One-parameter arrow function (x) =>
Collection		Bag of items		Bracket syntax e.g. [1, 2, 3]
Map over a set		Factory		Array.prototype.map()
Null	(does not appear)			null

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



Figure 5: From left to right: Pieces of the game at progressive stages of concreteness fading, starting from the most concrete and ending at the most abstract.

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:

```
[star, rect, tri].map((x) => (x == star) ? star : null)
```

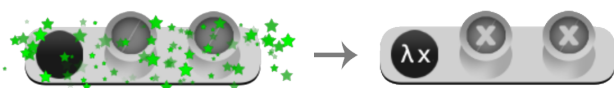


Figure 6: Concreteness fading in practice. When players mouse over the green sparkles, the piece fades to a more abstract representation.

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 $(x) \Rightarrow x$ and a star 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

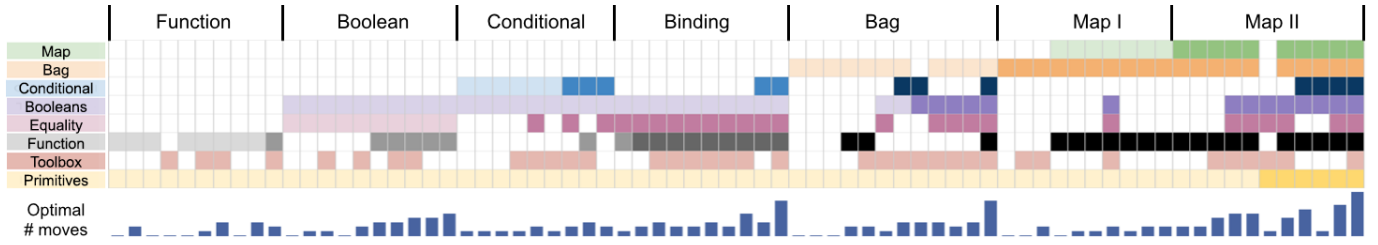


Figure 7: The progression of concepts as they appear in the game, where concepts are mapped to colors. A box indicates the appearance that concept in the level. Change in color strength marks the point at which that concept was faded.

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 question to isolate an item in a collection (Figure 8d). Participants in the CF condition also

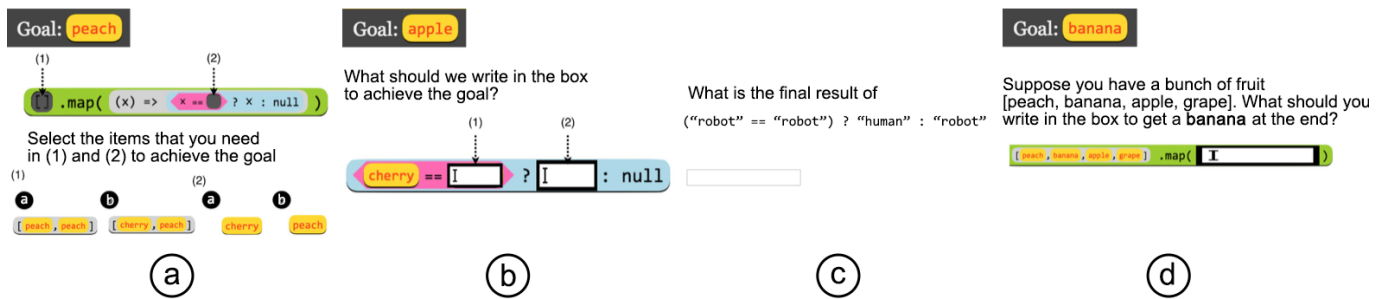


Figure 8: Example questions from our post-test, one per question type: (a) recognition, (b) recall, (c) near-transfer, and (d) optional.

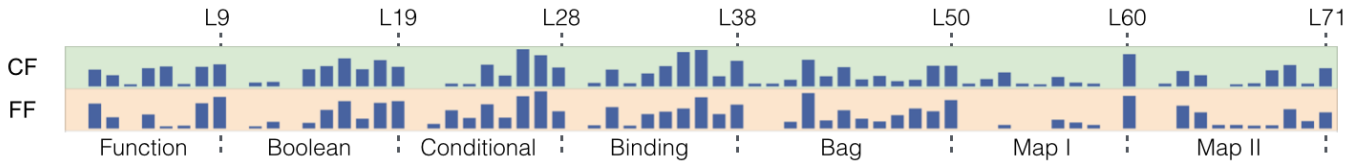


Figure 9: Progression of difficulty in each concept. A larger bar indicates a higher degree of average effort (see text for details). First level data was removed for confounding reasons.

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 5min 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 ($F(6, 154) = 1.168$, $p = 0.326$, $\eta_p^2 = 0.044$) or Concreteness ($F(1, 154) = 0.551$, $p = 0.459$, $\eta_p^2 = 0.004$). There were no interactions between the two variables ($F(6, 154) = 0.551$, $p = 0.769$, $\eta_p^2 = 0.021$). 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 ($(\text{actual_moves} - \text{optimum}) / \text{optimum}$). 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 two-way ANOVA on this dependent variable shows no effect of Concreteness ($F(1, 154) = 0.069$, $p = 0.794$, $\eta_p^2 = 0.0004$) but a main effect of Concept ($F(6, 154) = 20.878$, $p < 0.01$, $\eta_p^2 = 0.449$) with Map and Map2 being significantly lower than the other level ($p < 0.001$). These interactions between the two variables did not reach significance ($F(6, 154) = 1.953$, $p = 0.08$, $\eta_p^2 = 0.071$). We further remark that the η_p^2 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 eventuality, 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 distinction).

We evaluated three primary type of learning outcomes using our post-questionnaire: recognition, recall, and near transfer. This data shown Figure 11 was analyzed using a two-way LearningOutcome X Concreteness mixed-design ANOVA. This test shows an effect of LearningOutcome with the near transfer being the weakest, as can be expected, but no main effect of Concreteness ($F(1, 66) = 0.404$, $p = 0.527$, $\eta_p^2 = 0.006$) and no interaction between Concreteness and LearningOutcome. Overall, participants answered about

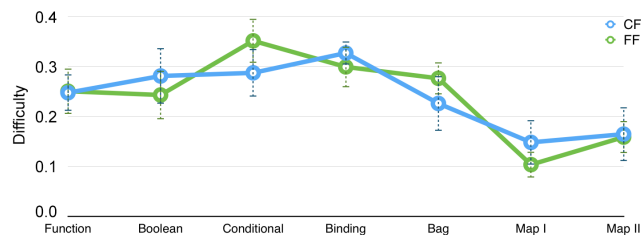


Figure 10: Game difficulty per section, plotted as relative effort (See text for details).

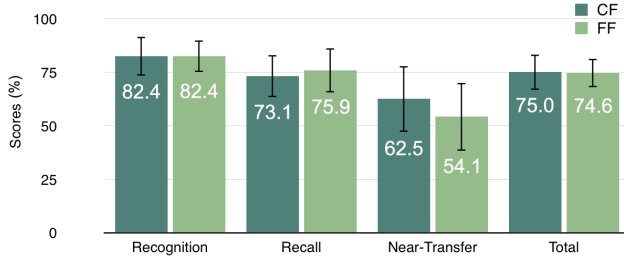


Figure 11: Mean percentage post-test scores.

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 mirrored in the online deployment (presented below).

Player perception

We now look at our participants’ perception of the game. As shown on Figure 12, players believed that both versions of the game were easy to understand (average rating here), fun (average rating here), and overall they would like to play more of the game. A two way ANOVA showed no effect of type of game ($F(1, 110) = 0.04$, $p = 0.843$, $\eta_p^2 = 0.0003$) but a main effect of measures ($F(4, 110) = 4.395$, $p = 0.002$, $\eta_p^2 = 0.138$) with difficulty measure being significantly lower than easy-to-understand measure. There were no interactions between the type of game and these measures ($F(4, 110) = 0.560$, $p = 0.692$, $\eta_p^2 = 0.02$). These findings were reinforced by direct feedback from users. One participant said “Please publish the game on App Store. I will totally go for it.” Another participant reported that “if I want to learn more about programming concepts, I definitely will play this game more.”

Without being prompted, 9 participants (CF P6, 13, 19, 23; FF P12, 14, 16, 18, 24) particularly mentioned that they like the difficulty progression of the game. They said that they got a strong sense of satisfaction when they completed the difficult

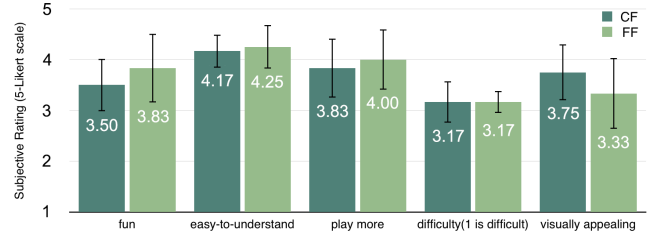


Figure 12: Mean of 5-Likert scale subjective rating on fun, easy to learn, play more, difficulty, and visual appeal.

puzzles with the knowledge they had learned in the earlier, simpler levels.

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 (x) => $x + 5$ 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

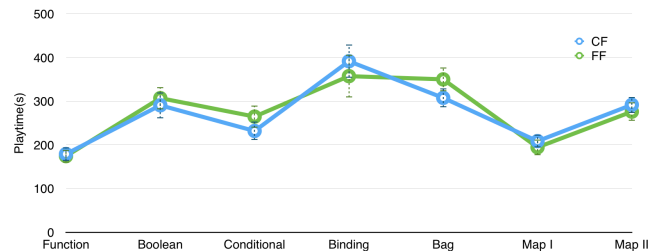


Figure 13: Average playtime per section for players that completed every level in the online study.

two-way Concept X Concreteness mixed design ANOVA showed no effect for CF ($F(1, 2471) = .689, p = .407, \eta_p^2 = .0003$). The analysis also showed a main effect of Concept ($F(6, 2471) = 104.793, p < 0.01, \eta_p^2 = .203$) and a small interaction ($F(1, 2471) = 4.837, p < 0.01, \eta_p^2 = 0.012$). These 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 $\langle \rangle$ 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 “ $2 + 2 \rightarrow 4$ ” and “ $3 + 4 \rightarrow 7$.” 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 β reduction. Using the syntax of *Reduct*, this rule appears as follows:

$$\overline{((x) \Rightarrow e) e' \rightarrow e\{e'/x\}}$$

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

1. Samson Abramsky. 1993. Computational interpretations of linear logic. *Theoretical computer science* 111, 1 (1993), 3–57.
2. ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. Computer science curricula 2013. <http://www.acm.org/education/CS2013-final-report.pdf>. (2013).
3. Deborah Adshead, Charles Boisvert, David Love, and Phil Spencer. 2015. Changing Culture: Educating the Next Computer Scientists. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 33–38.
4. Efthimia Aivaloglou and Felienne Hermans. 2016. How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, 53–61.
5. Erik Andersen, Yun-En Liu, Richard Snider, Roy Szeto, Seth Cooper, and Zoran Popović. 2011. On the harmfulness of secondary game objectives. In

- Proceedings of the 6th International Conference on Foundations of Digital Games*. ACM, 30–37.
6. Erik Andersen, Eleanor O'Rourke, Yun-En Liu, Rich Snider, Jeff Lowdermilk, David Truong, Seth Cooper, and Zoran Popovic. 2012. The impact of tutorials on games of varying complexity. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 59–68.
 7. John Robert Anderson. 2000. Learning and memory. (2000).
 8. John R Anderson, Frederick G Conrad, and Albert T Corbett. 1989. Skill acquisition and the LISP tutor. *Cognitive Science* 13, 4 (1989), 467–505.
 9. John R Anderson, Albert T Corbett, Kenneth R Koedinger, and Ray Pelletier. 1995. Cognitive tutors: Lessons learned. *The journal of the learning sciences* 4, 2 (1995), 167–207.
 10. Mark H Ashcraft. 2002. Math anxiety: Personal, educational, and cognitive consequences. *Current directions in psychological science* 11, 5 (2002), 181–185.
 11. Titus Barik, Emerson Murphy-Hill, and Thomas Zimmermann. 2016. A Perspective on Blending Programming Environments and Games: Beyond Points, Badges, and Leaderboards. In *Proceedings of the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing*.
 12. David Bau, D Anthony Bau, Mathew Dawson, and C Pickens. 2015. Pencil code: block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children*. ACM, 445–448.
 13. Piraye Bayman and Richard E. Mayer. 1983. A Diagnosis of Beginning Programmers' Misconceptions of BASIC Programming Statements. *Commun. ACM* 26, 9 (Sept. 1983), 677–679. DOI : <http://dx.doi.org/10.1145/358172.358408>
 14. David W Braithwaite and Robert L Goldstone. 2013. Integrating formal and grounded representations in combinatorics learning. *Journal of Educational Psychology* 105, 3 (2013), 666.
 15. Karen Ann Brennan. 2013. *Best of both worlds: Issues of structure and agency in computational creation, in and out of school*. Ph.D. Dissertation. Massachusetts Institute of Technology.
 16. Neil CC Brown, Jens Mönig, Anthony Bau, and David Weintrop. 2016. Panel: Future Directions of Block-based Programming. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 315–316.
 17. Jerome Seymour Bruner. 1966. *Toward a theory of instruction*. Vol. 59. Harvard University Press.
 18. Shalina Chatlani. 2016. Challenges persist when gamifying education. (December 2016). <http://www.educationdive.com/news/challenges-persist-when-gamifying-education/430817/> [Online; posted 5-Dec-2016].
 19. Alonzo Church. 1941. *The calculi of lambda-conversion*. Number 6. Princeton University Press.
 20. Michael Clancy. 2004. Misconceptions and attitudes that interfere with learning to program. *Computer science education research* (2004), 85–100.
 21. Codecademy. 2011. Codecademy. <http://www.codecademy.org>. (2011).
 22. CodeCombat. 2014. Code Combat. PC Game. (2014).
 23. Code.org. 2013. Code.org. <http://www.code.org>. (2013).
 24. Stephen Cooper, Wanda Dann, and Randy Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges*, Vol. 15. Consortium for Computing Sciences in Colleges, 107–116.
 25. Mihaly Csikszentmihalyi. 1991. *Flow: The psychology of optimal experience*. Vol. 41. HarperPerennial New York.
 26. Mihaly Csikszentmihalyi. 1996. Flow and the psychology of discovery and invention. *New Yprk: Harper Collins* (1996).
 27. Sayamindu Dasgupta, William Hale, Andrés Monroy-Hernández, and Benjamin Mako Hill. 2016. Remixing as a pathway to computational thinking. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*. ACM, 1438–1449.
 28. Stuart E Dreyfus and Hubert L Dreyfus. 1980. *A five-stage model of the mental activities involved in directed skill acquisition*. Technical Report. DTIC Document.
 29. Sarah Esper, Stephen R Foster, and William G Griswold. 2013. CodeSpells: embodying the metaphor of wizardry for programming. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 249–254.
 30. Anne L Fay and Richard E Mayer. 1988. Learning LOGO: A cognitive analysis. (1988).
 31. Emily R Fyfe, Nicole M McNeil, and Stephanie Borjas. 2015. Benefits of “concreteness fading” for children’s mathematics understanding. *Learning and Instruction* 35 (2015), 104–120.
 32. Emily R Fyfe, Nicole M McNeil, Ji Y Son, and Robert L Goldstone. 2014. Concreteness fading in mathematics and science instruction: A systematic review. *Educational Psychology Review* 26, 1 (2014), 9–25.
 33. Robert L Goldstone and Ji Y Son. 2005. The transfer of scientific principles using concrete and idealized simulations. *The Journal of the Learning Sciences* 14, 1 (2005), 69–110.

34. Google. 2013. Blockly. <https://developers.google.com/blockly/>. (2013).
35. Lindsey Ann Gouws, Karen Bradshaw, and Peter Wentworth. 2013. Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. ACM, 10–15.
36. Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 579–584.
37. Kyle J Harms, Noah Rowlett, and Caitlin Kelleher. 2015. Enabling independent learning of programming concepts through programming completion puzzles. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on*. IEEE, 271–279.
38. Michael S Horn, Erin Treacy Solovey, R Jordan Crouser, and Robert JK Jacob. 2009. Comparing the use of tangible and graphical programming languages for informal science education. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 975–984.
39. Hansen Hsu. 2015. *The Appsmiths: Community, Identity, Affect And Ideology Among Cocoa Developers From Next To Iphone*. Ph.D. Dissertation. Cornell University.
40. Lisa C Kaczmarczyk, Elizabeth R Petrick, J Philip East, and Geoffrey L Herman. 2010. Identifying student misconceptions of programming. In *Proceedings of the 41st ACM technical symposium on Computer science education*. ACM, 107–111.
41. Ken Kahn. 1996. Toontalk: An animated programming environment for children. *Journal of Visual Languages & Computing* 7, 2 (1996), 197–217.
42. Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1455–1464.
43. Jordana Kerr, Mary Chou, Reilly Ellis, and Caitlin Kelleher. 2013. Setting the scene: scaffolding stories to benefit middle school students learning to program. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 95–98.
44. Khan Academy. 2006. <https://www.khanacademy.org/>. (2006).
45. Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*. ACM, 29–38.
46. Dave Krebs, Alexander Conrad, and Jingtao Wang. 2012. Combining visual block programming and graph manipulation for clinical alert rule building. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2453–2458.
47. Deepak Kumar. 2014. Digital playgrounds for early computing education. *ACM Inroads* 5, 1 (2014), 20–21.
48. Alyson La. 2015. Language trends on GitHub. <https://github.com/blog/2047-language-trends-on-github>. (2015).
49. Michael J Lee, Faezeh Bahmani, Irwin Kwan, Jilian LaFerte, Polina Charters, Amber Horvath, Fanny Luor, Jill Cao, Catherine Law, Michael Beswetherick, and others. 2014. Principles of a debugging-first puzzle game for computing education. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 57–64.
50. Michael J Lee and Andrew J Ko. 2015. Comparing the effectiveness of online learning approaches on cs1 learning outcomes. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM, 237–246.
51. Jonathan Liu. 2012. Dragonbox: Algebra beats angry birds. *Wired*, June (2012).
52. Yun-En Liu, Christy Ballweber, Eleanor O'rourke, Eric Butler, Phonraphee Thummaphan, and Zoran Popović. 2015. Large-Scale Educational Campaigns. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 8.
53. Yanjin Long and Vincent Alevan. 2014. Gamification of Joint Student/System Control over Problem Selection in a Linear Equation Tutor. In *Intelligent Tutoring Systems*. Springer, 378–387.
54. John H Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. 2008. *Programming by choice: urban youth learning programming with scratch*. Vol. 40. ACM.
55. Nicole M McNeil and Emily R Fyfe. 2012. “Concreteness fading” promotes transfer of mathematical knowledge. *Learning and Instruction* 22, 6 (2012), 440–448.
56. Timothy S McNerney. 2004. From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing* 8, 5 (2004), 326–337.
57. Indrani Medhi, Aman Sagar, and Kentaro Toyama. 2006. Text-free user interfaces for illiterate and semi-literate users. In *2006 International Conference on Information and Communication Technologies and Development*. IEEE, 72–82.
58. Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2011. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. ACM, 168–172.

59. Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2013. Learning computer science concepts with Scratch. *Computer Science Education* 23, 3 (2013), 239–264.
60. Marvin Minsky. 1986. Introduction to LogoWorks. (1986).
61. Seymour Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
62. Seymour Papert. 1986. Seymour Papert: On Logo. (1986). [Video series].
63. Roy D Pea and Karen Sheingold. 1987. *Mirrors of minds: Patterns of experience in educational computing*. ERIC.
64. Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin* 39, 4 (2007), 204–223.
65. Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).
66. Cyndi Rader, Cathy Brand, and Clayton Lewis. 1997. Degrees of comprehension: children’s understanding of a visual programming environment. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, 351–358.
67. Vennila Ramalingam, Deborah LaBelle, and Susan Wiedenbeck. 2004. Self-efficacy and mental models in learning to program. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 171–175.
68. C Reigeluth and R Stein. 1983. Elaboration theory. *Instructional-design theories and models: An overview of their current status* (1983), 335–381.
69. Mitchel Resnick. 2007. All I really need to know (about creative thinking) I learned (by studying how children learn) in kindergarten. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*. ACM, 1–6.
70. Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and others. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.
71. Hong Kian Sam, Abang Ekhsan Abang Othman, and Zaimuarifuddin Shukri Nordin. 2005. Computer self-efficacy, computer anxiety, and attitudes toward the Internet: A study among undergraduates in Unimas. *Educational Technology & Society* 8, 4 (2005), 205–219.
72. Katharina Scheiter, Peter Gerjets, and Julia Schuh. 2010. The acquisition of problem-solving skills in mathematics: How animations can aid understanding of structural problem features and solution procedures. *Instructional Science* 38, 5 (2010), 487–502.
73. James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.
74. Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and learning programming and software engineering via interactive gaming. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1117–1126.
75. Ingo J Timm, Tjorben Bogon, Andreas D Lattner, and René Schumann. 2008. Teaching distributed artificial intelligence with RoboRally. In *Multiagent System Technologies*. Springer, 171–182.
76. Tomorrow Corporation. 2015. Human Resource Machine. PC Game. (2015).
77. Jeroen JG Van Merriënboer. 1990. Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of educational computing research* 6, 3 (1990), 265–285.
78. We Want To Know. 2013. DragonBox. PC Game. (2013).
79. Wizards of the Coast. 1995. RoboRally. Board Game. (1995).
80. Sharon Zhou, Ivy J Livingston, Mark Schiefsky, Stuart M Shieber, and Krzysztof Z Gajos. 2016. Ingenium: Engaging Novice Students with Latin Grammar. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 944–956.