The Elephant in the Syntax: A Comparative Study of Semantics-First, Block-Based, and Textual Programming

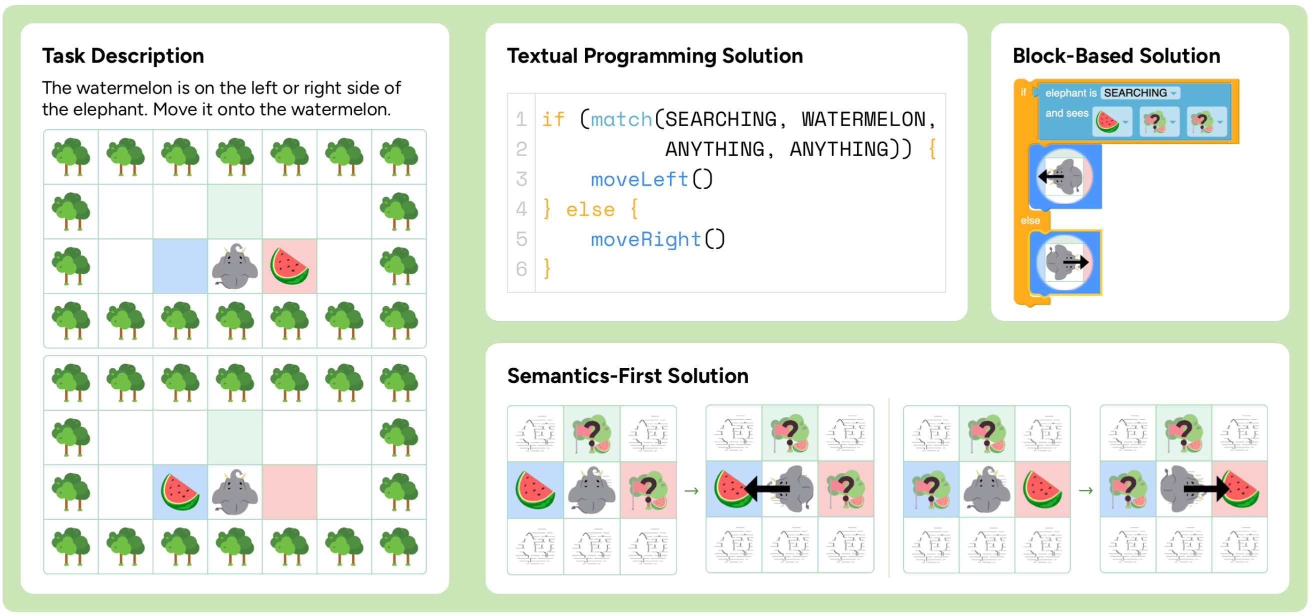
Theo B. Weidmann[*]
ETH Zürich
Switzerland
theo.weidmann@inf.ethz.ch

Sverrir Thorgeirsson[*]
ETH Zürich
Switzerland
sverrir.thorgeirsson@inf.ethz.ch

Karl-Heinz Weidmann
karl-heinz.weidmann@fhv.at
University of Applied Sciences
Vorarlberg
Dornbirn, Austria

April Yi Wang
ETH Zürich
Switzerland
april.wang@inf.ethz.ch

Zhendong Su
ETH Zürich
Switzerland
zhendong.su@inf.ethz.ch



The watermelon is on the left or right side of the elephant. Move it onto the watermelon.

```
if (match(SEARCHING, WATERMELON,
          ANYTHING, ANYTHING)) {
    moveLeft()
} else {
    moveRight()
}
```

Figure 1: The *Elephant* research platform offers three programming modes: textual, block-based, and semantics-first programming. Each of the three programs makes the elephant turn left or right depending on the watermelon's position.

Abstract

Syntax remains a major barrier for novices. Although block-based systems reduce or eliminate syntax errors, conditionals still challenge learners, likely because their semantics remain implicit. In this paper, we address this problem by introducing a semantics-first, state-visible programming approach inspired by the classic visual language Stagecast Creator. To demonstrate its usefulness, we designed *Elephant*, a unified, Karel-like research platform that supports three equally expressive programming paradigms: (i) semantics-first programming, (ii) block-based programming with the Blockly library, and (iii) text-based programming in JavaScript with domain-specific libraries. We then deployed Elephant in two within-subjects studies with secondary-school students (N = 39) to compare semantics-first programming to textual and block-based baselines, keeping the program semantics constant across modes and reducing cross-tool confounds. Results indicate, among other things, that semantics-first programming yields significantly higher task performance, suggesting that increasing the visibility of the program state during program composition could support greater outcomes in secondary computing education.

CCS Concepts

• **Social and professional topics** → **Computational thinking**; • **Human-centered computing** → *Visualization theory, concepts and paradigms*.

Keywords

secondary education, visual programming, conditional logic, block-based programming, programming by demonstration, direct manipulation

[*]Theo B. Weidmann and Sverrir Thorgeirsson are co-primary authors.

1 Introduction

Decades of empirical research suggest that syntax is a major source of difficulty for novice programmers [8, 16, 49, 69]. For more than half a century, researchers have proposed and refined a succession of simplified programming systems that can make syntax easier for beginners [29], often by attempting to dispense with textual syntax altogether. Examples of this approach include flow-chart programming languages like *RAPTOR* [14], in which programs are constructed by dragging and connecting flow-chart symbols rather than typing code, and block-based programming like *Scratch* [52], in which free-form text is replaced with interlocking blocks that preempt syntax errors by construction. In K-12 education, block-based programming is often seen as a gentle introduction to textual programming [83], and its success is evident by its popularity, as by mid-2025, the Scratch platform had attracted more than 135 million registered users who have created over 164 million projects [28].

Despite the success of block-based programming in education, conditional execution, which is a frequent source of misconceptions for novices [66], remains a persistent stumbling block for Scratch programmers. For instance, a large-scale study of 250,000 Scratch programs [2] revealed that less than 40% of them contained `if` or `if-else` blocks, compared to 77% which contained loops, implying that many beginner Scratch programmers avoid or struggle with implementing conditional logic. One plausible explanation for this disparity is that while Scratch makes the syntax of conditionals trivial, the semantics are almost entirely implicit, and the predicate's truth value remains just as hidden as in any textual language when the learner is constructing the program. Before block-based programming became widely adopted, the visual system *KidSim* [63, 64] from the 1990s—later commercialized as Stagecast Creator [65] (see Fig. 2)—attempted to address this issue by turning conditionals into graphical rules [63, 64], and the more recent visual language Algot [79] makes the semantics of conditionals observable by keeping the program state live during composition, with randomized, controlled trials versus Scratch and Python reporting promising outcomes [21, 75–77]. However, this semantics-first line of work has seen limited adoption in K-12 practice or mainstream tooling.

In this paper, we aim to revive the semantics-first, state-visible tradition in modern novice programming tooling for secondary education by instantiating the more generic paradigm from Stagecast Creator [65] for the introductory programming classroom. We present and evaluate a unified programming research platform called Elephant that allows its users to solve tasks in the same environment with three distinct but equally expressive modes of interaction: (i) a semantics-first approach, (ii) block-based programming, and (iii) traditional textual programming in JavaScript with a library of domain-specific functions. Crucially, all three modes share a single runtime and identical task semantics, meaning that the programming paradigm is the only manipulated factor, offering a methodological advantage for comparative studies on introductory visual programming languages. To our knowledge, Elephant is the first such unified system that includes a semantics-first, rule-based mode of programming that targets the learning of introductory concepts in computer programming.

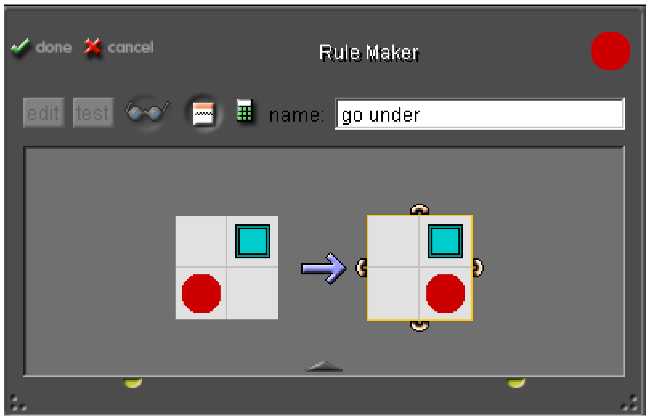


Figure 2: Screenshot from the "Rule Maker" system in Stagecast Creator. A new rule is defined such that if the circle is to the bottom left of the teal square, it will move below it.

Elephant features a classical agent-based grid world programming environment based on the programming language *Karel* [46] (referred to as *Karel-like* in the literature [24, 39]) which shares the same characteristics as the grid environment used in the popular Computational Thinking Test [20, 54], and is designed to support students' affective-motivational factors through a playful visual design. In our interaction mode of interest, the semantics-first approach, learners program conditional behavior in a rule-based way. Individual rules map the agent's local view and current state to an action and next state, and by allowing students to write and modify those rules using direct manipulation, we attempt to keep student attention on program meaning and not program form.

Our work makes three contributions, in ascending order of importance:

(1) We introduce a semantics-first interaction method that lets learners implement conditional behavior in a Karel-like environment by directly manipulating the agent's local view and state, and we apply design principles (e.g., live semantics, locality of evaluation, and action-state co-specification) in an environment that explicitly supports computational-thinking concepts (e.g., sequencing, iteration, and abstraction), updating earlier semantics-visible approaches for a modern context.

(2) We introduce Elephant, a unified environment backed by a single runtime that exposes semantics-first, block-based, and text-based programming modes within the same Karel-like programming environment, keeping task semantics constant across modes. In contrast to most earlier comparative studies on strands of visual and textual programming [27, 55, 75, 76], our environment allows one to conduct experiments in which the task domain, runtime semantics, and program state are constant by running all conditions in the same Karel-like environment. This makes programming modes (in the same sense as discussed by Weintrop and Wilensky [81]) the only manipulated factor and eliminates cross-tool confounds. While similar systems have been constructed to support comparative evaluations on textual and

block-based programming [25], Elephant is the first of its kind that also supports a semantics-first programming mode and aims to provide a solid methodological foundation for testing the semantics-first approach in today's context.

(3) We present evidence of the effectiveness of the semantics-first approach in secondary education. To do so, we conducted two comparative experiments of our approach against textual and block-based programming, respectively, with two distinct groups of secondary school students ($N = 22$ and $N = 17$), measuring task performance, cognitive load, computational thinking proficiency, perceived system usability, and perceived enjoyment. Our results indicate that the semantics-first approach supports significantly greater task performance than both textual and block-based programming.

Along with this report, we release the Elephant system itself, task examples, study instruments, and anonymized logs to support replication and classroom adoption (see the supplementary materials).

2 Background and Related Work

2.1 Programming by Demonstration and Visual Semantics

Direct manipulation is an interaction technique that was first described by Ben Shneiderman in the early 1980s [60]. The technique is characterized by the continuous visual representation of objects, where users can perform incremental actions that provide immediate feedback, enabling them to directly control on-screen objects in a way that closely resembles physical manipulation of real-world items. For example, dragging an icon across the screen resembles the experience of physically moving an object, providing a tangible sense of control. This idea has been successful across a variety of domains, for instance, in visual design [67], spreadsheets [85], and video games [62]. A follow-up paper by Shneiderman from 1983 [61] describes how direct manipulation can be seen as an enhancement of programming languages, allowing "novices [to] learn basic functionality quickly" and help "experts [carry out] work extremely rapidly." However, some problems with this technique, according to Shneiderman, are that coming up with a suitable visual representation can be difficult, with some representations being misleading or unergonomic.

Programming by demonstration (PbD) or, more generally, *demonstrational interfaces*, is a related technique that Brad A. Myers describes as a "step beyond direct manipulation" in a 1992 paper [35], allowing the user to construct abstract programs by performing actions on example objects. Similarly to direct manipulation, Myers considers it helpful because it can allow novices to quickly learn basic functionality of a system. PbD systems which do not rely on program synthesis but rather require the programmer to discretely specify different cases is an emerging approach in computer science education (CSE) in recent years, for example, with Algot, the graph-based programming language [79]. Comprehensive psychophysiological studies from 2024 indicate that programming in Algot induces lower or comparable cognitive load than textual programming [76, 77] and that the language is beneficial in secondary

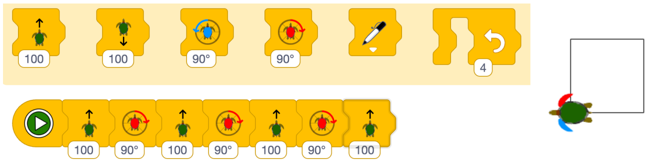


Figure 3: The XLogo [68] environment enables the creation of simple Logo turtle programs through block-based programming. A program (left) to draw a square (right) can be seen in the figure.

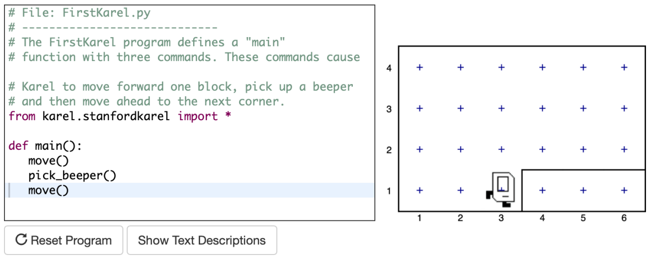
and tertiary education [21, 75]. AlgoTouch [18] is another example from tertiary education that implements direct manipulation and programming by demonstration, with an experimental study from 2019 finding that it compared well to textual programming [1].

To accomplish programming by demonstration, systems like Algot use *visual semantics*. This means that the aspects of the program state visualization can be used by themselves as building blocks for programs. For instance, when working with visualizations of tree structures, the programmer can click on tree nodes to manipulate the graph, like adding new edges or calculating the sum of a sub-tree's values. This mechanism makes such systems distinct from those that rely on code alongside algorithm visualizations, in which the programmer must constantly switch attention between textual code and a separate visual display. As a result, if an error appears in the visualization, it may not be immediately clear which specific segment of code caused it, thus slowing down debugging.

In this way, visual-semantic systems can address what Don Norman calls *the gulf of execution* and *the gulf of evaluation* [37, 38]. The gulf of execution describes the gap between a user's intention (e.g., wanting to manipulate data in a particular way) and their ability to translate that intention into system commands. By letting users directly interact with visual representations of the program's data structures, visual-semantic systems can minimize this gap, making it more obvious how to perform the desired actions. The gulf of evaluation, meanwhile, denotes the gap between the system's internal state and the user's ability to perceive and interpret that state. Because visual-semantic systems inherently integrate the interface with the program's operational logic, they present immediate and more transparent feedback, enabling users to evaluate the outcome of their actions with minimal effort. In the semantics-first mode of Elephant, we apply these principles by allowing learners to author programs through direct manipulation of the visual program state, aiming to minimize both gulfs during the composition of conditional logic.

2.2 Logo and Karel the Robot

One of the earliest notable advances in making programming accessible for novices began with *Logo*, a language inspired by the work of Seymour Papert [44], particularly through its use of turtle graphics [13] (see Fig. 3). Logo's design was shaped strongly by Papert's theory of constructionism, which extends Piaget's constructivism by positing that learning is most effective when learners actively construct personally meaningful artifacts in the world [44].



```
# File: FirstKarel.py
# ----------------------------
# The FirstKarel program defines a "main"
# function with three commands. These commands cause

# Karel to move forward one block, pick up a beeper
# and then move ahead to the next corner.
from karel.stanfordkarel import *

def main():
   move()
   pick_beeper()
   move()
```

Figure 4: A screenshot of *Karel the Robot Learns Python* [47], a modern 2019 implementation of the Karel environment based on Python. Karel is seen on the right in its grid world, which is similar to the grid in Elephant.

Under this view, programming environments should provide "microworlds," which are simplified, explorable domains where learners can experiment and gain immediate feedback, thereby refining their mental models. Logo's turtle graphics exemplified this philosophy: the turtle served as a concrete, manipulable object that made abstract mathematical concepts tangible.

Over the subsequent years, various so-called "mini-languages" emerged, for example Karel the Robot, developed by Richard Pattis [45], followed by Hamster, created by Dietrich Boles [7], and Kara, introduced by Raimond Reichert [51]. These languages typically involve programming an agent that interacts within a graphically represented world that is reduced in its complexity. Such actors have included robots, turtles, ladybugs, or other imaginative figures. Students learn programming by issuing instructions to the actor to solve predefined problems and focus thereby on the algorithmic challenges of programming. Mini-languages offer immediate visual feedback, enabling students to observe their programs' execution in real-time [13, 51].

In particular, Karel the Robot, which inspired the programming environment and tasks used in Elephant, operates in a world structured as a grid, allowing movement from one grid point to another, either horizontally or vertically (see Fig. 4). This world may contain various objects, such as walls between grid points or *beepers* placed on the grid itself. In contrast to Logo's turtle, Karel is equipped with three sensors that allow it to detect walls to its front, left, or right. [45] Karel can also sense whether it is next to a beeper and determine its own orientation with respect to the grid. It is equipped with a bag for collecting beepers and a sensor to check if the bag contains any. Karel responds to commands such as `move`, `turnleft`, `pickbeeper`, and `putbeeper` [13, 45, 51]. Like Logo, Karel is programmed textually, and adaptations of Karel the Robot for several popular textual languages, such as Java [53] and Python [19, 47] have been created. A comparative study on Karel and the block-based language Scratch found that students working in Scratch performed better while Karel students displayed higher self-regulation [55]. The world in Elephant inherits this grid-based property and local sensing model directly, with an elephant replacing the robot and watermelons and flags replacing beepers.

2.3 KidSim and Stagecast Creator

The KidSim [63] system and its successor Stagecast Creator [65], which were under active development in the 1990s, are visual programming languages that allow their users to compose programs by directly manipulating a visual representation of the program state. In these systems, users create programs by demonstrating desired behaviors directly within a grid-based simulation environment: the user selects any part of the grid (called the "spotlight") to create a "before" and "after" rule, and the system records this as a graphical rewrite rule (see Fig. 2). Furthermore, the system supports multiple agents, in which each agent maintains a list of ordered rules that are tried from top to bottom until one of them matches. Stagecast's combination of programming by demonstration with visual before-after rules allows users to express program logic through concrete manipulation of the semantic domain itself, which is why we refer to this type of programming in this paper as semantics-first (SF) programming.

Stagecast and its predecessor systems are versatile systems that can be used to construct programs with at least moderate complexity. Its developers wished to give children a tool that would allow them to construct simulations on their own, noting explicitly the constructivist idea that "children learn best when they actively create" [15]. For example, students could use the system to reimplement games, such as Breakout and Pacman, and construct science-based simulations, for example, of plant life and the emission of light in a laser [57]. However, contemporary opinions and reviews of Stagecast pointed out issues with the scalability of its paradigm [36, 59]; an otherwise positive review notes that "it becomes harder to manipulate the images on screen to see what you're doing" when the number of conditions and tiles grows [36], which may affect the viability of the paradigm for constructing simulations. In our work, our aim is not to demonstrate the use of SF programming for building games or simulations, in which scalability and open-ended complexity are significant concerns, but rather to apply it in a way that helps students master fundamental constructs such as conditionals and sequencing. In Section 3.4, after we have introduced Elephant and the way it supports SF programming, we discuss the system differences in more depth.

2.4 Programming Environments Supporting Multiple Modes

A class of programming environments allows learners to edit the same program through two or more representations, or *modes*, typically blocks and text, with the views kept in sync via a shared underlying presentation. For example, Tiled Grace [25] provides fully bidirectional, semantics-preserving switching between tiles and the textual Grace language, with animated transitions to make correspondences explicit. Similarly, Droplet [6] (an editor for Pencil Code) supports round-trip editing in blocks and JavaScript, meaning that learners can switch modes without changing the program's meaning or formatting. Empirical studies show that offering students the opportunity to switch between block-based and textual modes can be helpful [80], with a large variance in when and under what circumstances students will switch. On the other hand, we are not aware of any system that provides a unified environment with consistent semantics that supports a semantics-first mode as well as others, whether or not the system supports mode switching, which is the gap we aim to fill by presenting Elephant.

2.5 Computational Thinking

The term *computational thinking* refers to a set of mental processes and problem-solving techniques drawn from computer science, for instance, decomposition, pattern recognition, abstraction, and algorithmic thinking [82]. These competencies have increasingly been integrated into K-12 curricula worldwide, reflecting a broader push to equip students with the logical thinking and digital literacy skills necessary for the 21st century [84]. Grover and Pea [22], for example, have found that recognizing conditional patterns significantly improves programming proficiency in K-12 students, helping them develop faster debugging skills and more efficient code construction. In addition, empirical evidence from unplugged computational thinking activities [9] showed that pattern recognition is strongly correlated with problem-solving performance. As such, there is great interest in introductory languages supporting the teaching of computational thinking concepts [43, 70].

A large number of assessment instruments have been developed to test computational thinking [48], for instance, the validated Computational Thinking Test (CTt) [20], which is designed to address several computational concepts, including basic directions and sequences, loops, conditionals, and simple functions. According to the CTt authors, these items were chosen to be aligned with existing frameworks for computational thinking [3, 11]. Most of the questions on the test are framed using recognizable constructs like mazes and arrows. In the present work, we employ the CTt in an attempt to assess how different programming modes influence computational thinking acquisition.

2.6 Cognitive Load

Cognitive Load Theory (CLT), which is rooted in John Sweller's pioneering work in the 1980s [40, 71], offers critical insights into how our brains process and manage incoming information. Central to CLT is the notion that working memory has a strictly limited capacity, which directly affects the efficiency with which learners can acquire and retain new knowledge [42, 73]. When the volume or complexity of information exceeds this cognitive threshold, learning becomes less effective, making it harder to absorb and recall material. Over the past few decades, CLT has notably influenced research in educational psychology and instructional design [30, 58], underscoring the importance of carefully structuring and sequencing learning materials to optimize cognitive resources. Such considerations are essential for developing instructional strategies that bolster learning outcomes and enhance long-term knowledge retention.

Many ideas from CLT have implications for educational technology design for computing education. For instance, the *split-attention effect* [4] suggests that learners who must divide their attention across multiple, dispersed sources of information will experience higher cognitive load than those who experience a single, integrated source of information. This, in turn, will have a negative effect on their learning. In programming contexts, it may be challenging for students to simultaneously process textual code while keeping a mental image of the program state, as it requires them to constantly integrate and synthesize the separate pieces. This split-attention effect is closely related to the *redundancy effect* [72], which can emerge when learners need to handle identical or equivalent information; instead of reinforcing a student's understanding, it may contribute to higher *extraneous* cognitive load, meaning that the working memory is overloaded with information that does not contribute to a student's learning. Algorithm visualization systems that do not contribute meaningful additional information to what the student can find in the code may be liable to cause this effect. Instead, system designers may consider supporting the student in composing programs by manipulating the visualization directly. To investigate these effects empirically, our study employs validated cognitive load measures to assess the mental demands imposed by different programming modes.

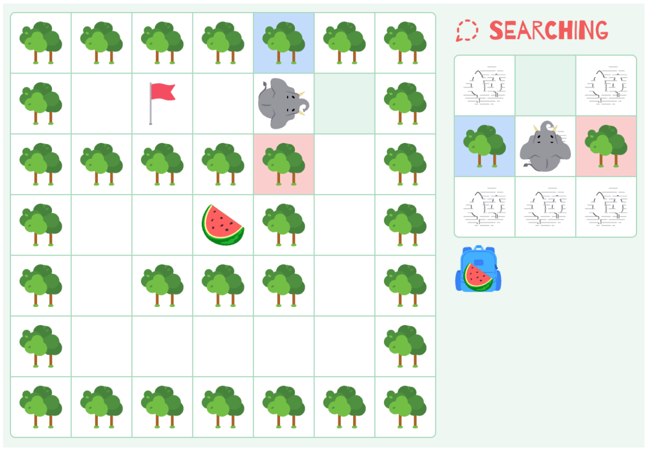


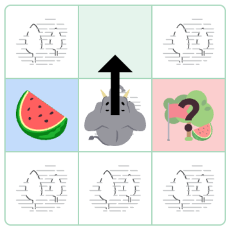
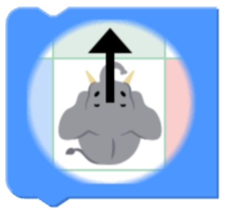
Figure 5: A screenshot depicting a grid configuration for Elephant (left). The state of the elephant is currently *SEARCHING* (top right), and its immediate environment shows a tree on the left and right and nothing to its right (state depiction on the middle right). Furthermore, the elephant has already collected one watermelon (indicated by the watermelon on the backpack on the right).

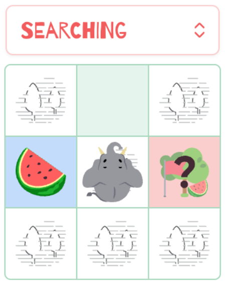
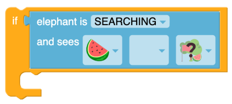
3 The Elephant System

3.1 Elephant's World

We designed Elephant such that each task naturally requires the use of conditional constructs while retaining the straightforward, engaging nature of turtle programming. To this end, we adapted design principles from the classical Karel the Robot [45, 46] educational tool, and reimagined them for a visually rich and interactive context.

Our environment features an elephant located on a grid where each cell contains either a tree, a watermelon, a flag, or nothing at all. For a given grid configuration, the user is tasked with helping the elephant reach some specific goal, such as collecting watermelons. The elephant can only see the fields directly to its left, directly ahead, and directly to its right. Fig. 5 depicts an example of this: on the right-hand side, the elephant's local view only contains a tree

	Semantics-First	**Block-Based**	**Textual (JavaScript)**
Specifying Movement	 Elephant is dragged to the target field	 Blocks that illustrate the desired movement are combined	Appropriate functions such as `moveAhead()` are called

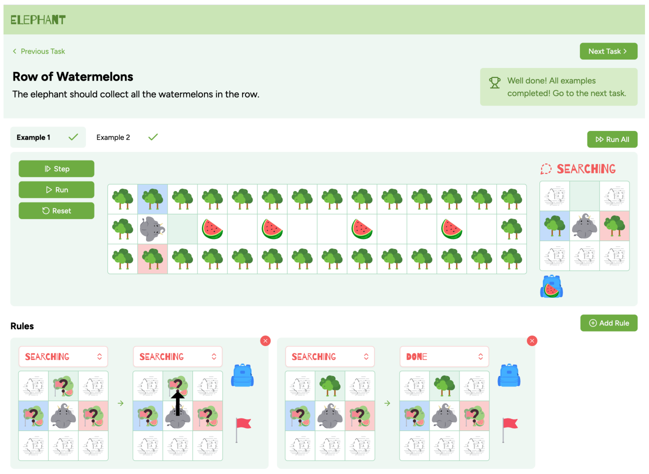
Conditional	 Condition is specified visually and within the context	 Conditions are specified using if-blocks in combination with a matching-block	`if (match(SEARCHING,`
			`    WATERMELON,`
			`    EMPTY,`
			`    ANYTHING)) {`
			`}`
			Conditions are specified using `if` in combination with our `match()` function

Setting new state	New state is chosen from select box	New state is chosen in appropriate block	Appropriate function call such as `setState(DONE)`
Prevents syntax mistakes	✓	✓	✗
Syntax visually implies semantics	✓	(✓)	✗
Conditionals become visible in context of movement	✓	✗	✗
Programming Style	Turn-based: Program is evaluated to decide one turn of the elephant		
Program Execution	For each turn, most specific rule (least wildcards) is selected and executed	For each turn, all code blocks are executed. Movement and new state can only be specified once per execution	For each turn, all code is executed. Movement and new state can only be specified once per execution

Table 1: Overview of the three programming modes in Elephant.

on its left and right side. The diagonal fields are not visible to the elephant.

The elephant can move left, forward, right, and backward, and movements are relative to the elephant's perspective on the grid. For example, in Fig. 5, the elephant is facing an empty cell, and moving forward would take it to the cell highlighted in light green. If the elephant moves into a tree, it crashes, and the program stops with a warning message. The elephant has the ability to collect watermelons found on grid cells and can place flags on the grid itself.



Row of Watermelons

The elephant should collect all the watermelons in the row.

Well done! All examples completed! Go to the next task.

Figure 6: The ELEPHANT interface: At the top is the task description. The student can choose between the different examples, which are instances of the given problem, and run their implementation. At the bottom, the semantics-first programming approach is active. This area can also show a block-based environment or a traditional JavaScript environment.

The elephant has an internal state that is represented as a string. For example, the elephant's state might be a value such as `SEARCHING` (like in Fig. 5) or `RETURNING HOME`. The programmer can use these internal states as preconditions for the elephant's actions. For each task, these states are pre-defined, and participants can only select between a certain number of available states.

3.2 Learning Environment

ELEPHANT presents learners with a progressive sequence of tasks, each increasing in complexity (see Table 2). Each task comprises a set of *examples*, analogous to test cases, that serve to illustrate the requirements of the task description. To advance, students create a program (either through semantics-first, block-based, or textual programming) that enables the elephant to achieve the specified goal across all provided examples. Progression to subsequent tasks is contingent upon successful completion of the previous task. Similar to Karel the Robot, certain tasks pose significant challenges, despite being situated within environments governed by a succinct set of rules.

To support iterative problem-solving, ELEPHANT offers three primary controls: `Step`, `Run`, and `Reset` (see Fig. 6). The `Step` function executes one movement of the elephant at a time, facilitating step-wise debugging; `Run` executes the entire program; and `Reset` restores the grid to its initial state. Upon program termination, the system displays a notification that communicates whether the attempt resulted in success or a faulty state. Additionally, ELEPHANT augments feedback through auditory cues, providing distinct signals when the elephant either succeeds or fails to achieve the intended outcome.

3.3 Overview of Programming Modes

Before we introduce the three programming modes supported by ELEPHANT in the next section, we provide an overview in the form of a comparison in Table 1.

3.3.1 Semantics-First Programming. To implement semantics-first programming, the elephant's actions are controlled by the user through the creation of one or more *rules*, which collectively define the program that is needed to solve a given task. The rules specify how the elephant should move within the grid by mapping its immediate surroundings and internal state to a particular action, such as moving, picking up watermelons, or placing flags. Formally, a rule is a mapping from

(1) its current state (a string)
(2) the elephant's local view, which is a triple representing what the elephant sees to its left, front, and right, and

to a movement (left, right, forward, back, or stay), a new state, the option to pick up a watermelon and place a flag on the next field. For example, the following rule

$$(\texttt{SEARCHING},\ \langle \texttt{L: empty},\ \texttt{F: watermelon},\ \texttt{R: empty}\rangle) \mapsto$$
$$(\textsc{move}: \uparrow,\ \textsc{state}: \texttt{DONE},\ \textsc{pick}: ✗,\ \textsc{place}: ✗)$$

would make the elephant move forward if there is a watermelon ahead and the other fields are empty. It will also change its state to `DONE`, not pick up the watermelon, and not place a flag. In case the programmer wants to specify an action regardless of the contents of a cell, a wildcard can be used. A wildcard can represent all possible cells. The rule

$$(\texttt{SEARCHING},\ \langle \texttt{L: }*,\ \texttt{F: watermelon},\ \texttt{R: empty}\rangle) \mapsto$$
$$(\textsc{move}: \leftarrow,\ \textsc{state}: \texttt{DONE},\ \textsc{pick}: ✓,\ \textsc{place}: ✗)$$

for instance, applies regardless of what the cell to the left of the elephant contains. The states the elephant can assume are predefined for each task. While some tasks will only have two states (mostly `SEARCHING` and `DONE`), other tasks have more varied states.

Rules are defined visually within ELEPHANT (see Fig. 7), similar to how KidSim [63] worked. Rules are defined on two 3x3 grids. The left-hand side grid in Fig. 7 (the *see* grid) represents the state the elephant must be in and the local view the elephant sees to apply the rule, and the right-hand side grid in Fig. 7 (the *act* grid) represents the movement, new state, and actions performed by the elephant. The user defines the rule by directly manipulating both grids with the mouse.

For example, consider the following rule:

$$(\texttt{SEARCHING},\ \langle \texttt{L: }*,\ \texttt{F: }*,\ \texttt{R: watermelon}\rangle) \mapsto$$
$$(\textsc{move}: \rightarrow,\ \textsc{state}: \texttt{DONE},\ \textsc{pick}: ✓,\ \textsc{place}: ✓)$$

To implement this rule in the ELEPHANT system, the user first adds a new empty rule, then clicks on the field to the right of the elephant in the *see* grid, and selects the watermelon (see top left in Fig. 7). This action specifies that this rule can only apply if a watermelon is to the right of the elephant. When changing the elephant's local view in the *see* grid, the *act* grid also updates immediately. In the act grid, the programmer can drag the elephant to the left, right, forward, or backward field to instruct the elephant's movements in this rule. The elephant can also be dragged to the center field, to instruct it to *not* move at all. Thus, in our example, the user drags the elephant to the right (see top center in Fig. 7). Then, to instruct the elephant to pick up the watermelon, the user drags the watermelon to the elephant's backpack on the right-hand side of the act grid (see top right and bottom left in Fig. 7). Likewise, the user can drag a flag from the right-hand side of the act grid to the elephant's target field to instruct the elephant to place a flag there (see bottom center in Fig. 7). Finally, the user updates the elephant's target state to DONE by selecting it from the select box (see bottom right in Fig. 7).

In some aspects, the rules in Elephant resemble Dijkstra's *guarded commands* [17], which are constructs that combine a boolean expression (the guard) with a command that executes only if the guard evaluates to true. However, in contrast to guarded commands where multiple true guards can lead to nondeterministic execution, Elephant is deterministic. Elephant always selects the most specific rule, i.e., the one with the least wildcards. If no single most specific rule exists, the user is informed about the issue, and the program stops.

3.3.2 Programming in JavaScript. To complement the semantics-first environment, we added a JavaScript interface to Elephant that allows learners to write textual solutions to the tasks (see Fig. 8). The API provides simple commands such as `moveLeft()` and `pickUp()`, designed to mirror the actions available in the visual rule system.

Our goal was to make the two environments comparable in both structure and execution. In JavaScript, programs are continuously re-executed until termination, just as rules are re-evaluated after every move in the semantics-first environment. Likewise, an `if`-block corresponds closely to a visual rule. To keep the system accessible for students with only two days of JavaScript instruction, we excluded general boolean logic and instead provided a `match` function. This function takes four arguments (the elephant's state and the elements to its left, front, and right), like the conditions in the visual environment.

Finally, to reduce the burden of typing, the interface offers buttons for inserting the most commonly used syntactic elements. These are purely optional: learners can write complete solutions without using them.

3.3.3 Block-Based Programming. Alongside the semantics-first and JavaScript programming modes, Elephant also integrates a block-based programming environment. This mode was implemented using the latest version of *Blockly* and styled with a color theme consistent with Scratch, thereby providing learners with a familiar visual experience. As in Scratch, blocks can be composed through drag-and-drop interactions; however, in contrast to conventional block-based systems, the environment retains the same semantics as the semantics-first and JavaScript modes of Elephant. Rules are expressed through conditional blocks that combine the agent's current state with its local perceptual context to determine the next action (see Fig. 9).

The block-based environment was designed to be able to express the same range of programs as the JavaScript and semantics-first modes, ensuring that any solution authored in one mode can be equivalently represented in the others. Internally, block-based programs are translated to the same JavaScript API used in the textual interface, guaranteeing semantic consistency across modes. This unified runtime makes it possible to compare modes directly without a loss of semantics.

Pedagogically, the block-based environment serves as an experimental condition for learners already accustomed to Scratch-like systems. By situating rule-based programming within a familiar block-based paradigm, we aim to distinguish between the effects of representation style and underlying semantics in learners' ability to engage with core computational thinking concepts. The environment was included in the empirical studies reported in this paper, enabling direct comparison between the semantics-first, block-based, and textual modes.

3.4 Relationship to Prior Systems

Although the SF programming mode in Elephant draws from the Stagecast Creator tradition of combining graphical-rewrite rules with programming by demonstration, our overall approach is different as it is aligned with contemporary instruction in computer science. First, Elephant operates within a constrained Karel-like domain rather than an open-ended simulation environment. Where Stagecast Creator allowed users to create arbitrary simulations with multiple independent agents, Elephant focuses on a single agent navigating in a turn-based way in a grid with a fixed vocabulary of objects (e.g., trees, watermelons, flags) and actions (in our study: movement, collection, and placement). These constraints are intentional: by narrowing the problem space, we aim to direct learner attention toward core computational thinking concepts such as conditionals, sequencing, and state management, rather than open-ended simulation design.

Second, unlike Stagecast, Elephant adopts an agent-relative perspective throughout the interface for all of the programming modes; it is "egocentric" rather than "allocentric". The elephant can rotate, and all movements are specified relative to the direction it is currently facing rather than absolute compass directions. This design has several advantages. Most importantly, it enables rule generalization: a single rule such as "if there is a watermelon ahead, move forward" applies regardless of the elephant's absolute orientation on the grid, whereas an allocentric system would require separate rules for each compass direction. Elephant's semantics-first approach may thus mitigate what has been called the "brittleness" of graphical rewrite rules in Stagecast, where there is "an explosion of rules whenever the user tries to create even slight variations on a theme" [59]. The egocentric view also aligns with how people naturally reason about navigation (e.g., "turn left at the corner" rather than "turn west") and ensures consistency between rule authoring and execution, as the *see* grid displays exactly what the agent perceives.

Third, the SF programming mode in Elephant employs a strict conflict-detection mechanism: when multiple rules match the current state with equal specificity, no rule is selected, and the learner is informed of the ambiguity. This contrasts with Stagecast Creator's approach, where rule ordering was managed explicitly by the user. We believe that this design has several pedagogical advantages for novice learners. For one, it encourages learners to confront logical overlaps in their rules rather than allowing the system to silently resolve ambiguities, which could mask incomplete understanding. It also simplifies debugging and may reduce the learner's cognitive

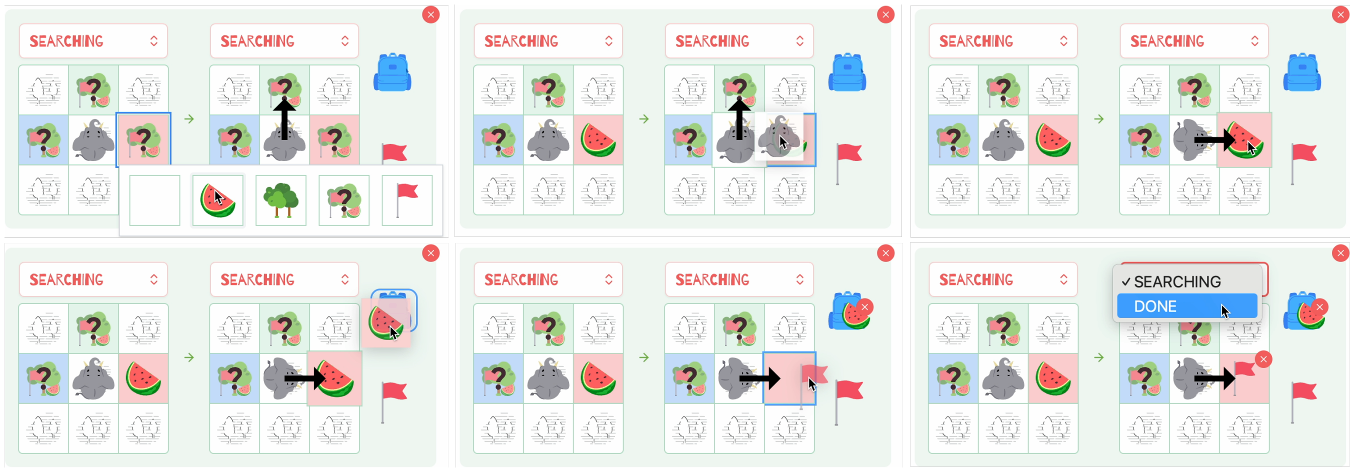
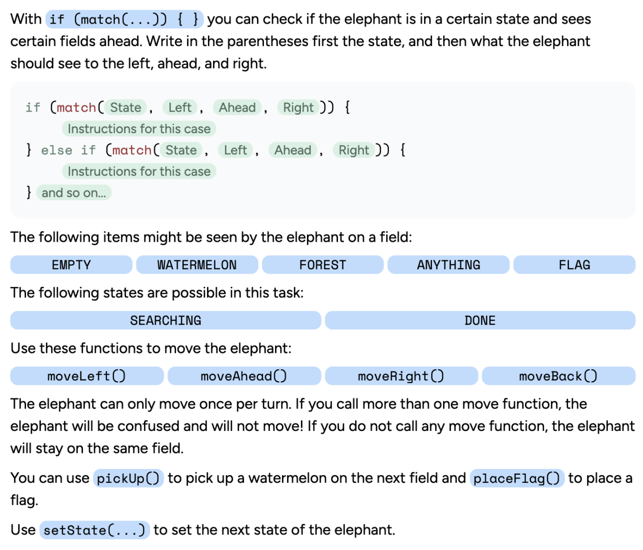


Figure 7: Sequence of screenshots showing how the programmer in ELEPHANT defines the rule (SEARCHING, ⟨**L**: ∗, **F**: ∗, **R**: watermelon⟩) ↦ (**MOVE**: →, **STATE**: DONE, **PICK**: ✓, **PLACE**: ✓)**.**



With `if (match(...)) { }` you can check if the elephant is in a certain state and sees certain fields ahead. Write in the parentheses first the state, and then what the elephant should see to the left, ahead, and right.

```
if (match( State , Left , Ahead , Right )) {
    Instructions for this case
} else if (match( State , Left , Ahead , Right )) {
    Instructions for this case
} and so on...
```

The following items might be seen by the elephant on a field:

`EMPTY` `WATERMELON` `FOREST` `ANYTHING` `FLAG`

The following states are possible in this task:

`SEARCHING` `DONE`

Use these functions to move the elephant:

`moveLeft()` `moveAhead()` `moveRight()` `moveBack()`

The elephant can only move once per turn. If you call more than one move function, the elephant will be confused and will not move! If you do not call any move function, the elephant will stay on the same field.

You can use `pickUp()` to pick up a watermelon on the next field and `placeFlag()` to place a flag.

Use `setState(...)` to set the next state of the elephant.

Figure 8: Instructions for how to work with JavaScript in the ELEPHANT system. To the left of the instructions, the students were given a code editor. Clicking on the blue code segments inserts the code into the editor for convenient code composition.

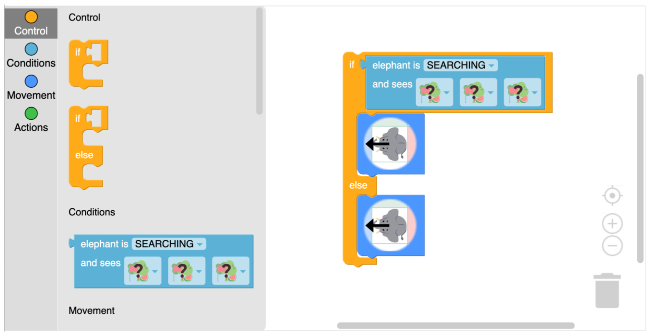


Figure 9: A screenshot showing the Blockly integration, which is part of the block-based programming mode in ELEPHANT.

load: when a rule fires, the learner knows unambiguously which rule matched and why, without needing to reason about ordering or hierarchies. In this way, our system may improve on the debugging experience reported for KidSim/Stagecast: in a classroom study with second graders, children sometimes created rules that did not behave as intended (for example, a mis-specified "move right" rule or a rocket that silently failed to move because other objects blocked its path), and some groups explicitly relied on peers or strategic prompts from the adult facilitator to diagnose such problems [26]. By making rule selection explicit, ELEPHANT aims to shift more of this diagnostic work into the interface itself, helping learners understand why a rule does or does not fire to decrease the need for external help. This design may also implicitly teach the concept of mutual exclusivity in conditional logic; learners must ensure their rule preconditions partition the space of possible situations, which could plausibly transfer to writing well-structured conditionals in conventional programming languages.

The fourth key difference is that every programming mode in ELEPHANT requires learners to solve multiple examples for each task. Examples are instances of the same problem statement sharing the same underlying goal and constraints (e.g., to collect a watermelon behind a line of trees), which the learner can toggle at will. A program only succeeds if it solves all examples, preventing learners from overfitting rules to a single configuration and encouraging them to identify the general pattern underlying the task. This design is well-aligned with the variation theory of learning [34], namely that learners can only discern an important feature when it varies across examples while other features stay mostly the same. It also has the advantage that it discourages trial-and-error strategies; when only a single example must pass, learners may succeed through ad-hoc actions that happen to work for one configuration. While this structure is well known in typical programming practice, where code must generalize across varied inputs, we are unaware of prior semantics-first systems that enforce generalization across multiple examples as a success criterion.

Lastly, whereas Stagecast Creator was a standalone environment, Elephant is designed as a research platform that supports direct comparison across programming paradigms. By embedding semantics-first, block-based, and textual modes within a single runtime with consistent task semantics, Elephant enables controlled experiments that isolate the effects of representation from confounds introduced by differing task domains or execution models. This methodological contribution distinguishes Elephant from its predecessors and positions it as a tool for both pedagogy and empirical research on novice programming.

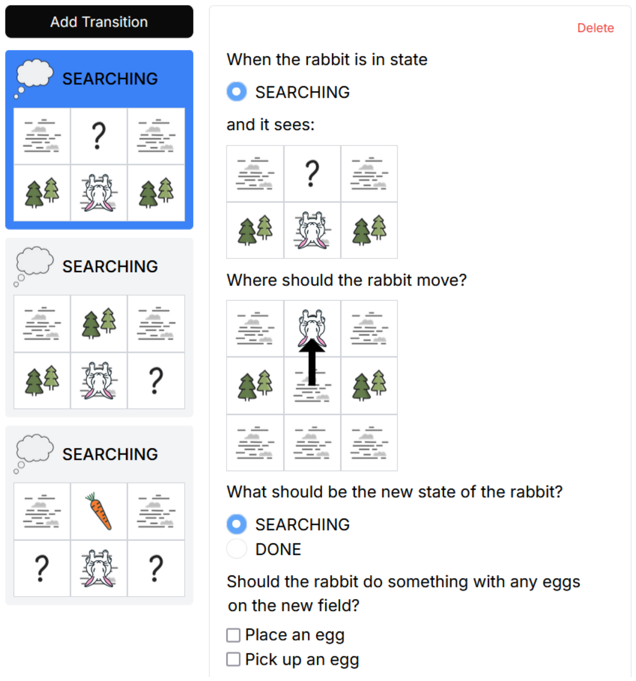
4 Experimental Evaluation

We conducted two empirical studies on the Elephant environment that took place one year apart, in 2024 and 2025. The objective of our studies was to evaluate the semantics-first programming method, both on its own and in comparison with the other modes, and how conducive the programming environment is for conducting comparisons of block-based programming, textual programming, and semantics-first programming.

For each study, we secured a separate ethics approval from our institution. All attendees and their guardians signed consent forms two days before the study took place. Participation was voluntary, and no compensation was offered. Both studies took place during a yearly recurring summer programming workshop in Austria, called Code Base Camp Vorarlberg, that was organized independently by a team unaffiliated with the authors. The workshop is specifically designed for secondary students and apprentices with limited or no prior programming experience. The organizers prioritize participants from groups with less access to digital education. According to the organizers, general computer literacy varies, with some participants requiring instruction on basics such as file management and browser navigation, and most participants' computing experience primarily coming from mobile devices. On average, approximately 5 out of 20 participants in the JavaScript-focused sessions had previously attended an introductory HTML workshop offered by the same institution, though the majority entered with no prior programming experience.

We conducted our experiments during the morning and afternoon sessions of the second day of each workshop. Prior to our study sessions, participants had completed one day of JavaScript syntax instruction (brackets, semicolons, basic drawing commands using P5.js), allowing them to create simple non-parameterized drawings.

In the first study, there were 22 participants, all of whom completed the study to its end. Their age was between 13 and 16; their average age was 13.8, and the standard deviation of their age was 0.85. Five participants identified as female and 17 as male. The pretest suggested that the participants were highly motivated and had a strong belief in their capacity to learn programming; the average agreement with the intrinsic interest and self-efficacy statements was 4.50 and 4.32, respectively, out of a maximum of 5.00.



When the rabbit is in state

● SEARCHING

and it sees:

Where should the rabbit move?

What should be the new state of the rabbit?

● SEARCHING
○ DONE

Should the rabbit do something with any eggs on the new field?

☐ Place an egg
☐ Pick up an egg

Figure 10: The first prototype of the Elephant system, shown in this screenshot, used a different set of visual assets. Most notably, the agent was changed from a rabbit to an elephant, which we believe is easier to identify visually from an overhead view. The system mechanics were unchanged, but the actions of placing and picking up items had to be configured with checkboxes. In contrast to the version of the system that is shown here, the new assets were designed by a professional illustrator with the goal that the elements in every programming mode should be more immediately recognizable and appealing for young users.

In the second study, there were 17 participants, none of whom had participated in the first study. As in the first study, all participants completed the study to its end. Their age range was 13 to 15, and their average age was again 13.8, and the standard deviation was 0.75. Two participants identified as female, while 15 identified as male. Their average response on the intrinsic interest and self-efficacy survey questions was also high, or 4.65 on both measures. No participant offered a lower rating than 4.0 on either question in either study.

The study duration was about five hours in total, excluding breaks in between. Each participant was asked to solve programming exercises in two different modes; in Study I, the two modes that we tested were semantics-first programming and textual programming; in Study II, the two modes were semantics-first programming and block-based programming.

To reduce learning effects, we implemented counter-balancing; the participants were divided into two groups, each in their own classroom, with each group working on the tasks in a given programming mode. The two groups swapped modes after a lunch break. Each session began with an approximately 15-minute tutorial on the system, followed by about 35 minutes of task solving in the first study. We expected fatigue after 35 minutes, but students stayed motivated and unfinished, so we extended task time to 50 minutes in the second study. The remaining time was allocated for surveys. All participants were given a pretest before the study began, a test following the first session, and a test following the last session. The battery of tests that we chose to use is described in Section 4.1. The tutorials for all modes followed a predefined script and were all administered by the same instructor to avoid any confounding effect of instructional quality.

Although its core functionality was not changed, the Elephant system evolved in some ways between the two evaluations. In the initial study, we tested a more bare-bones prototype (see Fig. 10), while in the later 2025 study, we had adopted the current elephant-based design, with its visual design language and custom-drawn illustrations for grid items. For the study, we modified the way in which the tasks were presented; students could no longer advance to the next task unless the current task had been solved correctly, as we had observed in the first study that students were sometimes quick to give up and move on to the next task. This implied that, although the task selection remained the same in both studies, a new grading rubric was needed; in the initial study, we manually graded students' work based on the grading rubric in Table 2. Students obtained partial credit for submitting solutions that demonstrated progress towards the correct solution, and would get bonus points for solutions that solved every example case. Each exercise was weighed the same. In the second study, we simply measured the number of levels students had successfully completed. Both sets of grading criteria were defined and decided on before each study took place.

Our research questions were as follows:

RQ1 For secondary school students, does semantics-first programming induce greater performance on Karel-like grid world tasks than block-based and text-based programming?

RQ2 For secondary school students, does semantics-first programming induce lower cognitive load than block-based and text-based programming?

Our secondary research questions were whether semantics-first programming induces greater enjoyment than text-based programming (first evaluation) and whether semantics-first programming is perceived as more usable than block-based programming (second evaluation). We hypothesized affirmative answers to all RQs.

4.1 Surveys

On the pretest of each evaluation, participants completed a short survey with free-form questions regarding age and gender, and two single-item, 5-point Likert-type measures regarding their intrinsic interest and self-efficacy in learning programming ("I am excited about learning computer programming" and "I believe in my ability to learn computer programming", respectively).

In Study I, students solved nine questions from the Computational Thinking Test (CTt) [20] (questions 1, 2, 3, 5, 6, 8, 9, 13, and 14). These questions were selected because they mapped onto the constructs supported by Elephant, namely sequencing and state.

ID	Goal (brief)	Partial Credit
1	Collect watermelon	Move forward at least one step
2	Collect watermelon (search left column)	Move adjacent to watermelon
3	Collect watermelon (turn around)	Cross the gate
4	Follow maze; stop on watermelon	Make two turns
5	Place a flag two steps ahead	Move exactly two steps
6	Collect all watermelons	Collect all watermelons but stops in error state (e.g. crashed into trees)
7	Collect exactly three watermelons	Collect two watermelons
8	Collect watermelon and return to start	Mark start and collect watermelon
9	Collect exactly three; end on a watermelon	Collect all watermelons
10	Create checkerboard of flags	Cover an entire row or column correctly

Table 2: Combined overview of the ten tasks with brief goals and partial-credit criteria. For a detailed listing of the tasks used, refer to Appendix A.

We limited the assessment to these specific items to avoid overwhelming the participants, following previous research with young learners [32, 78]. On the posttest, students were administered the same CTt questions to assess skill acquisition. Additionally, to measure cognitive load in Study I, we administered the single-item Paas scale [41] and measured enjoyment with a single 7-point Likert question.

In Study II, we adjusted the dependent measures to prioritize usability and granular load metrics over learning gains, acknowledging that longer interventions are typically required to observe significant changes in computational thinking. Consequently, we removed the CTt from Study II. This reduction in test-taking fatigue and time allocation allowed us to replace the coarse Paas scale with the more detailed NASA Task Load Index (NASA-TLX) [23] and to introduce the System Usability Scale (SUS) [12]. We measured enjoyment again with the same single 7-point Likert question.

4.2 Tasks

The participants were given the same ten tasks to solve in both studies (see Table 2 and Appendix A), which helped make the studies comparable. The tasks were designed to be increasingly challenging to solve and not to favor any mode over another. We drew inspiration from assignments designed for Karel the Robot [45, 46, 56]. The last tasks were included to ensure that we did not have a ceiling effect in the results; we anticipated that very few students, if any, would be able to solve all tasks successfully.

The tasks generally center on having the elephant find watermelons hidden on the grid, place flags on the grid in specific positions, and, in the final task, having the elephant create a check pattern of flags on the grid. Our grading criteria for the tasks can also be seen in Table 2. Students would get partial credit for submitting solutions that demonstrated progress towards the correct solution and would get bonus points for solutions that solved every example case. Each exercise was weighted the same.

4.3 Analysis

To evaluate the impact of each programming mode on performance and other metrics, we performed significance testing using two-sided, paired-samples Student's t-tests. Following established statistical recommendations [50, 74], we did not perform tests of normality, as t-tests are known to be robust against moderate normality deviations, and normality tests can produce misleading outcomes if used solely to justify subsequent significance testing. We calculated effect sizes using Cohen's d with a standard error (SE). We used the JASP statistical analysis software [33] for our calculations.

To account for the multiple hypothesis problem, we defined families by research question, following the principle that multiplicity should be controlled over the set of tests that speak to the same scientific claim. For performance (RQ1) and cognitive load (RQ2), we controlled the false discovery rate using the Benjamini–Hochberg (BH-FDR) procedure at $q = 0.05$ within each family of two planned pairwise comparisons ($m = 2$). For secondary outcomes (usability, enjoyment, and CT change), we also controlled the false discovery rate using BH at $q = 0.05$ across that set of outcomes. We report BH-adjusted q-values.

We conducted additional exploratory analyses on the regular snapshots of students' work collected in Study II. These snapshots captured the complete application state and were recorded after 10 seconds of inactivity, with a maximum interval of 15 seconds during continuous activity. First, we calculated the approximate task completion times, and second, we manually reviewed students' interactions as they were solving the tasks to identify common issues or misconceptions in the two different modes. These analyses aimed to contextualize and help explain the quantitative findings.

5 Results

The results from both studies can be found in Table 3. According to our Student's t-test of students' task performance difference, participants performed significantly stronger ($p = 0.016$) when solving the tasks with the semantics-first (SF) mode instead of code. The effect size was moderate (Cohen's $d = 0.56$). The average performance was stronger using SF than Code on nine tasks out of ten. Students also performed better on the tasks when using SF instead of block-based programming (BB); incidentally, the p-value was also $p = 0.016$ here, and also with a moderate effect size (0.76). Thus, we were able to reject the null hypothesis here in response to RQ1.

The difference in cognitive load was only significant under the SF versus BB comparison ($p = 0.026$). The effect size was moderate (Cohen's $d = 0.60$ in favor of SF). However, under the SF versus Code comparison, there was no significant difference ($p = 0.583$). Therefore, our hypothesis for RQ2 could only be partially validated. For our secondary research questions, the difference in enjoyment on the SF versus code mode was not significant, but the average scores were in favor of code. The intervention was not effective at improving computational thinking outcomes, which can likely be explained by the short duration of the study. The usability of the SF-mode was rated significantly higher ($p = 0.033$) than the usability of the block-based programming mode.

After BH-FDR control within each research-question family, RQ1 remained significant: SF outperformed Code in Study I and BB in II (both $q_{\text{BH}} = 0.016$; $d = 0.56$ and $d = 0.76$, respectively). For RQ2 (cognitive load), neither comparison met the $q = 0.05$ threshold (SF versus BB on TLX: $q_{\text{BH}} = 0.052$, $d = 0.60$; SF versus Code on the Paas scale: $q_{\text{BH}} = 0.583$, $d = 0.12$). Across secondary outcomes analyzed with BH-FDR at $q = 0.05$, none survived adjustment (usability, Study II: $q = 0.098$; enjoyment, Study II: $q = 0.098$; enjoyment, Study I: $q = 0.163$; CT Δ, Study I: $q = 0.620$).

While our study used counterbalancing and a pause between activities (a lunch break) to mitigate learning effects, we also conducted a brief sensitivity analysis by measuring the difference in performance after the first session only, i.e., when students could not have benefited from prior exposure in another programming mode. To do so, we restricted the analysis to the first session in each study, defined as the block in which participants encountered the task set for the first time in the respective pair of modes. In the comparison between SF and BB, the former mode again yielded significantly higher performance on this subset ($p = 0.046$) according to an independent samples Student's t-test, with a large effect size (Cohen's $d = 1.06$) in favor of SF. In the comparison between SF and Code, the effect size in favor of SF was also large (Cohen's $d = 0.86$), while it did not reach significance according to the t-test ($p = 0.058$). Given the reduced per-group sample size (half the original within-subject sample) and the loss of paired information in this first-session-only analysis, this pattern is compatible with the main results and, while not conclusive, offers some additional reassurance that practice or fatigue on repeated tasks are unlikely to be the driver of the observed advantage of SF.

5.1 Exploratory Analysis

Through manual review of the snapshots, we identified several misconceptions specific to block-based programming. Especially early in the session, students often failed to connect blocks properly (see top right in Fig. 11). Another common source of confusion was if and if-else blocks: We observed students apparently mixing up the if and else branches. Students also frequently placed instructions outside the intended block. For example, the "set state to DONE" instruction was sometimes placed after an if-else statement when putting it into the if or else statement would have been correct instead (see left in Fig. 11). We also observed students creating trivially faulty instructions, such as testing if there is a tree to the elephant's left and then instructing it to walk to the left (see bottom right in Fig. 11).

In contrast, the SF mode produced fewer classifiable misconceptions; difficulties primarily arose from overfitting rules to one specific example, such that the program did not solve other example instances of the same task.

The second exploratory measure was time to completion. For each task, mode, and participant, we computed the time from the snapshot when participants first opened the task to the snapshot where the task had been successfully completed. Table 4 shows average completion times by task and mode. The averages for the SF mode are lower for all but task 7. Task 7 shows a large standard

		Means (SD)				
Study	Outcome	SF	BB	Code	p	Cohen's d (SE)
I	Performance (Manual)	0.30 (0.15)	—	0.21 (0.13)	**0.016**	0.56 (0.26)
I	Cognitive load (Paas)	5.23 (1.88)	—	5.50 (1.82)	0.583	0.12 (0.27)
I	Enjoyment	6.90 (2.32)	—	7.68 (1.99)	0.122	0.34 (0.24)
I	CT skills Δ	0.00 (0.78)	—	−0.27 (1.62)	0.620	0.22 (0.43)
II	Performance (Levels)	6.06 (1.71)	4.47 (2.21)	—	**0.016**	0.76 (0.29)
II	Cognitive load (TLX)	59.16 (18.28)	66.67 (13.96)	—	**0.026**	0.60 (0.16)
II	Enjoyment	5.53 (2.81)	4.59 (3.04)	—	**0.049**	0.52 (0.39)
II	Usability (SUS)	53.24 (22.20)	46.32 (23.74)	—	**0.033**	0.57 (0.14)

Table 3: Consolidated comparative results across the two studies. Bold p-values indicate significance ($p < 0.05$). Mean values are shown for semantics-first (SF), block-based (BB), and textual code (Code) modes, with standard deviations (SD) in parentheses. All tests are Student's paired-sample t-test, except for the change in computational thinking (CT Δ), which shows an independent-samples Student's t-test. Cohen's d is reported with its standard error (SE) in parentheses. Higher values are better for performance, usability, computational thinking (CT), and enjoyment; for cognitive load, lower values indicate lower workload.

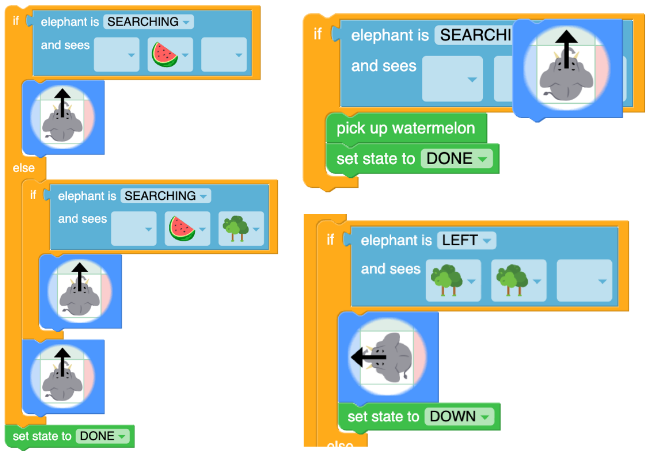


Figure 11: Three screenshots illustrating common issues in block-based mode. Left: a likely misconception about if-else logic: The state is set to DONE regardless of whether the watermelon was found. Top right: a block overlaid rather than connected, causing unexpected behavior. Bottom right: a trivially incorrect instruction directing the elephant into a tree.

Task ID	Semantics-First	Block-Based
1	295 (222.17)	498 (432.45)
2	205 (266.53)	220 (208.94)
3	252 (214.04)	507 (345.04)
4	324 (195.93)	672 (429.18)
5	180 (105.90)	308 (264.02)
6	148 (106.59)	158 (40.39)
7	1489 (612.46)	728 (150.74)
8	300 (4.99)	765 (—)

Table 4: Average time to task completion in seconds for each task and programming mode. Standard deviations are in parentheses.

deviation, which might be attributed to the task being more difficult than intended or fatigue, as it was completed only by seven students, and for most of them the last task they worked on. Task 8 was only completed by one student in the block-based mode.

Consistent with our earlier observation that students struggled with block connections, especially early in the session, the BB-SF gap was larger for Task 1 (69% longer) than for the structurally similar Task 2 (7% longer), suggesting that the mechanics of block-based syntax posed a greater hurdle than the underlying problem-solving.

6 Discussion

Our two evaluations suggest that semantics-first programming yields measurable benefits for novices in solving tasks in a Karel-like programming environment; we found that participants solved significantly more tasks successfully under this mode than with either textual JavaScript or block-based programming. The significance of the results persisted under multiple-comparison control, and the effect size in each case was moderate.

On the other hand, our question on cognitive-load differences in different modes showed less conclusive results, and after multiple-comparison control, the results were not significant. Generally, the cognitive load scores for all modes were high according to both the Paas rating and the NASA-TLX, indicating that students spent significant mental effort in solving the tasks (a finding consistent with our observations).

We hypothesize that the performance improvement induced by the semantics-first approach can be attributed to its design philosophy. When programming using traditional blocks and code, the meaning of a conditional is implicit when it is authored. In contrast, during semantics-first programming, the predicate that determines whether a rule is active is a concrete configuration on a visible grid, showing the agent's immediate context. This locality of evaluation minimizes the need to integrate spatially separated information sources (program state versus code), a known contributor to extraneous cognitive load both in novice programming and other settings (the split-attention effect). Our exploratory analysis, which indicates that students struggled with the semantics of if-blocks, also point into this direction. However, as our results did not show a significant difference in perceived cognitive load (at least not for the Code versus SF condition), this theory should be investigated further in larger studies or in studies that make use of more objective measurements of cognitive load with physiological instruments, such as eye-tracking equipment.

The results are well-aligned with the early success of older, similar demonstrational systems such as KidSim [63] and Stagecast [65], and recent evidence from the visual language Algot finding that systems that make the program state more visible can lower or maintain cognitive load while improving outcomes in novice populations [76, 77]. However, in contrast to existing studies, we believe that our work offers more robust evidence in favor of semantics-first and semantics-first-adjacent programming, the reason being that under all conditions, students were programming in the same Karel-like programming environment Elephant. Furthermore, our study demonstrates the usefulness of the Elephant research platform for conducting empirical research on different programming paradigms; future studies can utilize the environment to conduct longitudinal research on how semantics-first programming affects learning, which our study was not designed for.

For our secondary research questions, we believe that the results were impacted by the study design and the attributes of the study population. For instance, all participants were highly motivated volunteers attending an extracurricular, multiday workshop on JavaScript programming, indicating a possible bias towards JavaScript programming. We also hypothesize that the perceived utility of Elephant could play a role when compared to textual programming; for example, Krings et al. found that "most students defined programming traditionally as 'writing code"' and do not consider any alternatives [31], and a 2020 study on four visual programming languages such as Scratch and Karel the Robot found that secondary school students "perceive these environments as having only a specific, educational purpose" and do not consider them to be "real programming" [10]. As for improvements in computational thinking skills, we believe that the intervention was too short to induce measurable changes.

Prior work by Banerjee et al. [5] has also demonstrated that text-free, language-neutral programming-by-demonstration environments can be beneficial for learners who face a language barrier, as many textual and block-based systems use English words and expressions, which can put non-English speakers at a disadvantage. Elephant likewise is a programming-by-demonstration whose SF approach relies on little text. While the design goals and the technical details of the paradigms differ, we believe that our work reinforces that programming can be made more language-neutral. Beyond language barriers, visual and direct-manipulation approaches may also support learners with diverse learning needs, such as dyslexia or attention differences, who may struggle with text-heavy programming environments.

We believe that our findings motivate certain design choices in novice programming environments for strengthening conditional reasoning, for example, to support bidirectional projections of the program state. The design of such an environment does not have to involve bespoke visual programming languages, but could be layered into existing IDEs by adding live state views. By using static analysis or abstract interpretation, similar to how Algot does it [77, 79], effects of if conditions could be visualized. In classroom practice, SF could be well-suited for introducing conditionals and debugging strategies before or alongside blocks or text. One possible progression is to start with conditional semantics-first patterns using live semantics, view the same solutions in block-based or textual programming via projection, and finally transition students to working directly in blocks or text while preserving live predicate feedback. Such an approach could help students learn conditional logic without abandoning their eventual need to read and write code.

6.1 Limitations

We acknowledge several limitations. The programming tasks focus on grid-based control logic with local perception in a Karel-like grid. While this enabled experimental control, it limits external validity and whether the findings generalize to broader programming contexts with richer state spaces or more abstract problem domains. Additionally, Study I and Study II used different system versions (v1 vs. v2) and different dependent measures (Paas/CTt vs. TLX/SUS), creating confounding factors that weaken cross-study comparability and qualify claims about consistent effects across contexts.

Our evaluations were short (approximately 5 hours), used small samples (N=22 and N=17) with gender imbalance, and occurred in workshop contexts with participants who may be more motivated than typical students. The Elephant interface evolved between studies, and despite counterbalancing, novelty effects may have influenced subjective measures. Future work should examine longer-term outcomes in authentic classroom settings with more diverse samples and task domains (for example, with a richer state and more open-ended goals).

7 Future Work

7.1 System Extensions

Our current work focuses primarily on conditionals and sequencing within Elephant. However, the semantics-first approach can support more advanced computational topics in the introductory computer science curriculum. To demonstrate this, we designed an extension with two new abilities that enable learners to reason about higher-level algorithmic concepts.

In the existing system, the elephant has the ability to place flags on the grid to record previously visited locations (see Ⓐ in Fig. 12). This mechanic supports *search*; the learner can use the flags to implement greedy strategies to navigate unexplored areas on the grid. However, greedy search strategies have limitations that the learner will quickly discover when coming across cycles or dead ends. To address this, our extension adds a new ability to Elephant that allows the elephant to place teleportation platforms and later teleport back to the most recently placed teleportation platform (see Ⓑ in Fig. 12). This functionality can support diverse algorithms. For example, it allows the student to carry out systematic backtracking,

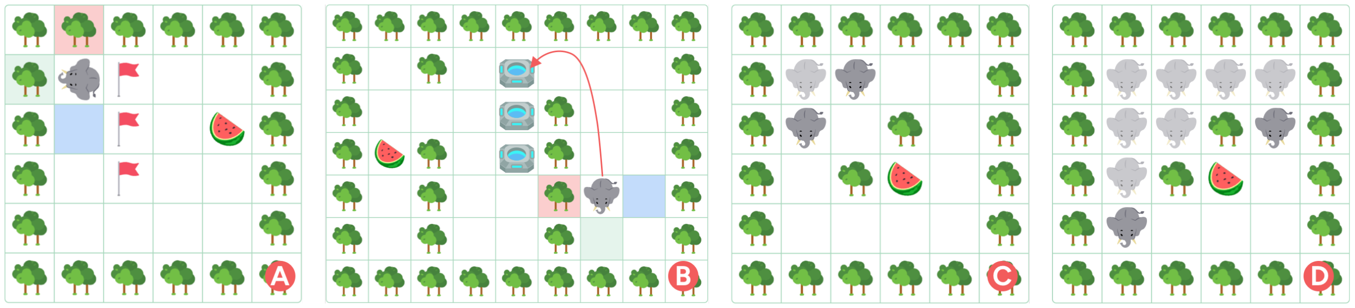


Figure 12: Elephant extensions for teaching graph search. (A) The elephant can place flags to mark visited cells, enabling greedy strategies that prioritize unexplored areas, though this may still lead to loops. (B) The elephant can place a teleportation platform and teleport back to the most recently placed platform, supporting systematic backtracking as in depth-first search. (C–D) The elephant can clone itself to explore multiple branches simultaneously, serving as a visual metaphor for breadth-first search, where multiple frontier nodes are expanded in parallel.

which is critical for depth-first search (DFS). From a computational thinking perspective, this ability also embeds a nested structure into the system; the elephant can teleport to a given location, conduct several tasks (e.g., collect watermelons), and then teleport back to its initial location. Second, our extension allows the elephant to clone itself into multiple instances, allowing the user to explore different areas of the grid in parallel (see Ⓒ and Ⓓ in Fig. 12). In addition to supporting breadth-first search, this ability can support advanced topics like recursion and a variety of divide-and-conquer algorithms.

To support this, Elephant grids can contain a pre-placed teleportation portal, in addition to any that are created by the user during runtime. These portals may be positioned in areas inaccessible to the original elephant, requiring transportation to reach them. Accessing a portal will move the elephant to the most recently placed portal, should more than one exist. In the SF mode, portals appear as a new cell type in the see grid, allowing rules to match when the elephant is adjacent to a portal. The act grid gains a "Create Teleport" action that the learner can select to specify that the elephant should create a new portal after making a movement. In block-based mode, a block for creating portals is added, while the textual API provides a `createPortal()` function. The act of cloning an elephant is conceptually similar, with the learner specifying clone spawning locations in the act grid, with corresponding blocks and API functions in the other modes. Clones share the same rule set but operate independently based on their local context, enabling parallel exploration of multiple branches.

Together, these two additions transform the environment from a semantics-first programming system for introductory computational thinking into a more general resource for computing education, and can help (i) make advanced topics more accessible to young learners, and (ii) improve on textual representations by making the program state more visible. To validate these extensions, qualitative feedback should be gathered (for example think-aloud sessions or interviews with students) to assess the user experience of the teleportation and cloning mechanics, which may introduce unforeseen complexities or usability issues. Subsequently, controlled empirical studies should examine whether the semantics-first representations of these extensions improves learners' understanding of search algorithms. Possibly, broader outcomes could be investigated, such as whether it reduces misconceptions about graph traversal, or whether it can strengthen computational thinking skills and support transfer to conventional text-based programming environments. Such directions might be appropriate for upper secondary school students.

7.2 Semantics-First as a Primer for Textual Syntax

Another avenue for future work is the idea that semantics-first programming could serve as a preparatory activity before students encounter textual syntax. Rather than teaching syntax and semantics simultaneously, a "semantics-first, syntax-second" progression might allow students to first develop robust mental models of program behavior through state-visible environments before taking on the additional cognitive demands of textual syntax. Such a reordering could potentially reduce early frustration and attrition while building stronger conceptual foundations.

However, our studies were not designed to test this hypothesis directly. A compelling validation would require demonstrating that students who first work in the SF mode subsequently perform better when transitioning to text-based programming, compared to students who begin with the text-based mode directly. If such transfer effects can be demonstrated, it would strengthen the case for SF programming as a principled entry point in introductory CS curricula rather than merely an alternative modality.

8 Conclusion

In this paper, we presented Elephant, a unified Karel-like grid-based programming environment that offers three modes of programming within a single runtime: block-based, textual programming, and *semantics-first* programming. The last of those is a new direct-manipulation programming method inspired by classical but underexplored visual approaches. Our environment is explicitly designed to teach secondary school students to work with conditionals. Across two five-hour studies (N=22 and N=17), the semantics-first mode yielded higher task performance than both textual and block-based programming, while differences in perceived cognitive load and subjective ratings were smaller and less consistent. We release the system, tasks, and study materials to enable replication and classroom use. Overall, the tool and findings support adding semantics-visible interactions as a practical complement to block and textual tools in early programming instruction, and motivate longer classroom deployments that track transfer and retention.

Acknowledgments

We thank the organizers of the Code Base Camp for allowing us to conduct our studies as part of their workshops; especially Sebastian Rangger and Christiane Valtiner, for supporting our interventions. We thank OMICRON, who hosted the workshop, for generously allowing us to use their infrastructure. Finally, we thank Interreg Alpenrhein-Bodensee-Hochrhein (ABH020) for supporting our work in popularizing more accessible programming-by-demonstration systems.

References

[1] Michel Adam, Moncef Daoud, and Patrice Frison. 2019. Direct manipulation versus text-based programming: An experiment report. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 353–359.

[2] 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* (Melbourne, VIC, Australia) *(ICER '16)*. Association for Computing Machinery, New York, NY, USA, 53–61. doi:10.1145/2960310.2960325

[3] Computer Science Teachers Association et al. 2011. K-12 computer science standards. *Retrieved November* 21 (2011), 2016.

[4] Paul Ayres and John Sweller. 2005. The split-attention principle in multimedia learning. *The Cambridge handbook of multimedia learning* 2 (2005), 135–146.

[5] Rahul Banerjee, Leanne Liu, Kiley Sobel, Caroline Pitt, Kung Jin Lee, Meng Wang, Sijin Chen, Lydia Davison, Jason C Yip, Amy J Ko, et al. 2018. Empowering families facing english literacy challenges to jointly engage in computer programming. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–13.

[6] David Bau. 2015. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges* 30, 6 (2015), 138–144.

[7] Dietrich Boles. 1999. *Programmieren spielend gelernt mit dem Java-Hamster-Modell*. Teubner.

[8] Jeffrey Bonar and Elliot Soloway. 1985. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human–Computer Interaction* 1, 2 (1985), 133–161. doi:10.1207/s15327051hci0102_3

[9] Christian P. Brackmann, Marcos Román-González, Gregorio Robles, Jesús Moreno-León, Ana Casali, and Dante Barone. 2017. Development of computational thinking skills through unplugged activities in primary school. In *Proceedings of the 12th workshop on primary and secondary computing education*. 65–72.

[10] Gert Braune and Andreas Mühling. 2020. Learning to program: The gap between school world and everyday world. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 1–9.

[11] Karen Brennan and Mitchel Resnick. 2012. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American educational research association, Vancouver, Canada*, Vol. 1. 25.

[12] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

[13] Peter Brusilovsky, Eduardo Calabrese, Jozef Hvorecky, Anatoly Kouchnirenko, and Philip Miller. 1997. Mini-languages: a way to learn programming principles. *Education and Information Technologies* 2, 1 (March 1997), 65–83. doi:10.1023/A:1018636507883

[14] Martin C. Carlisle, Terry A. Wilson, Jeffrey W. Humphries, and Steven M. Hadfield. 2005. RAPTOR: a visual programming environment for teaching algorithmic problem solving. *SIGCSE Bull.* 37, 1 (Feb. 2005), 176–180. doi:10.1145/1047124.1047411

[15] Allen Cypher and David Canfield Smith. 1995. KidSim: End user programming of simulations. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 27–34.

[16] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 208–212.

[17] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM* 18, 8 (1975), 453–457.

[18] Patrice Frison. 2015. A teaching assistant for algorithm construction. In *Proceedings of the 2015 ACM conference on innovation and technology in computer science education*. 9–14.

[19] Ioannis Giachos, Evangelos-Diomidis Sagias, Christos Papakitsos, Evangelos Papakitsos, and Nikolaos Laskaris. 2024. Educational Robotics: The Implementation of Karel in 3D. *Mediterranean Journal of Basic and Applied Sciences (MJBAS)* 8, 3 (2024), 70–81.

[20] M Román González. 2015. Computational thinking test: Design guidelines and content validation. In *EDULEARN15 proceedings*. IATED, 2436–2444.

[21] Oliver Graf, Sverrir Thorgeirsson, and Zhendong Su. 2024. Assessing Live Programming for Program Comprehension. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1* (Milan, Italy) *(ITiCSE 2024)*. Association for Computing Machinery, New York, NY, USA, 520–526. doi:10.1145/3649217.3653547

[22] Shuchi Grover and Roy Pea. 2013. Computational thinking in K–12: A review of the state of the field. *Educational researcher* 42, 1 (2013), 38–43.

[23] Sandra G. Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage publications Sage CA: Los Angeles, CA, 904–908.

[24] Poul Henriksen and Michael Kölling. 2004. greenfoot: combining object visualisation with interaction. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 73–82. doi:10.1145/1028664.1028701

[25] Michael Homer and James Noble. 2017. Lessons in Combining Block-based and Textual Programming. *J. Vis. Lang. Sentient Syst.* 3, 1 (2017), 22–39.

[26] Jane Howland, Jim Laffey, and Linda M. Espinosa. 1997. A computing experience to motivate children to complex performances. *Journal of Computing in Childhood Education* 8, 4 (1997), 291–311.

[27] Anna A. Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H. Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. 2020. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife* 9 (dec 2020), e58906. doi:10.7554/eLife.58906

[28] JetLearn. 2025. Scratch Statistics: Examining the Popularity of Scratch 2025. https://www.jetlearn.com/blog/scratch-statistics Accessed: 2025-08-07.

[29] Caitlin Kelleher and Randy Pausch. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.* 37, 2 (June 2005), 83–137. doi:10.1145/1089733.1089734

[30] Paul A. Kirschner, John Sweller, and Richard E. Clark. 2006. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist* 41, 2 (2006), 75–86.

[31] Kevin Krings, Nino S. Bohn, Nora Anna Luise Hille, and Thomas Ludwig. 2023. "What if everyone is able to program?"–Exploring the Role of Software Development in Science Fiction. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–13.

[32] Katrin Kunz, Korbinian Moeller, Manuel Ninaus, Ulrich Trautwein, and Katerina Tsarava. 2023. Making the Transition to Text-Based Programming: The Pilot Evaluation of a Computational Thinking Intervention for Primary School Students. In *Proceedings of the 18th WiPSCE Conference on Primary and Secondary Computing Education Research*. 1–6.

[33] Jonathon Love, Ravi Selker, Maarten Marsman, Tahira Jamil, Damian Dropmann, Josine Verhagen, Alexander Ly, Quentin F. Gronau, Martin Šmíra, Sacha Epskamp, et al. 2019. JASP: Graphical statistical software for common statistical designs. *Journal of Statistical Software* 88 (2019), 1–17.

[34] Ference Marton, Amy BM Tsui, Pakey PM Chik, Po Yuk Ko, and Mun Ling Lo. 2004. *Classroom discourse and the space of learning*. Routledge.

[35] Brad A. Myers. 1992. Demonstrational interfaces: A step beyond direct manipulation. *Computer* 25, 8 (1992), 61–73.

[36] Matt Neuburg. 1999. All the World's a Stagecast. *TidBITS* (14 June 1999). https://tidbits.com/1999/06/14/all-the-worlds-a-stagecast/

[37] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[38] Donald A Norman. 1986. Cognitive engineering. *User centered system design* 31, 61 (1986), 2.

[39] Giorgio Olimpo. 1988. The Robot Brothers: An environment for learning parallel programming oriented to computer education. *Computers & Education* 12, 1 (1988), 113–118. doi:10.1016/0360-1315(88)90064-4

[40] Elizabeth Owen and John Sweller. 1985. What do students learn while solving mathematics problems? *Journal of educational psychology* 77, 3 (1985), 272.

[41] Fred GWC Paas. 1992. Training strategies for attaining transfer of problem-solving skill in statistics: a cognitive-load approach. *Journal of educational psychology* 84, 4 (1992), 429.

[42] Fred GWC Paas and Jeroen JG Van Merriënboer. 1994. Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review* 6 (1994), 351–371.

[43] Zacharoula Papamitsiou, Michail Giannakos, Simon, and Andrew Luxton-Reilly. 2020. Computing education research landscape through an analysis of keywords. In *Proceedings of the 2020 ACM conference on international computing education research*. 102–112.

[44] Seymour Papert. 1980. *Mindstorms: children, computers, and powerful ideas.* Basic Books, Inc., USA.

[45] Richard E. Pattis. 1984. *Karel the Robot: A Gentle Introduction to the Art of Programming* (1st ed.). IBM Corp., John Wiley & Sons, Inc., USA.

[46] Richard E. Pattis. 1994. *Karel the robot: a gentle introduction to the art of programming.* John Wiley & Sons.

[47] Chris Piech and Eric Roberts. 2019. Karel the Robot: Learns Python. https://compedu.stanford.edu/karel-reader/docs/python/en/intro.html Accessed: 2025-03-12.

[48] Emmanouil Poulakis and Panagiotis Politis. 2021. Computational thinking assessment: Literature review. *Research on e-learning and ICT in education: Technological, pedagogical and instructional perspectives* (2021), 111–128.

[49] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Trans. Comput. Educ.* 18, 1, Article 1 (Oct. 2017), 24 pages. doi:10.1145/3077618

[50] Dieter Rasch, Klaus D. Kubinger, and Karl Moder. 2011. The two-sample t test: pre-testing its assumptions does not pay off. *Statistical papers* 52 (2011), 219–231.

[51] Raimond Reichert. 2003. *Theory of computation as a vehicle for teaching fundamental concepts of computer science. A dissertation submitted to ETH Zürich.* Dissertation. ETH Zürich. https://api.semanticscholar.org/CorpusID:263547326

[52] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. doi:10.1145/1592761.1592779

[53] Eric Roberts. 2005. *Karel the Robot Learns Java.* Stanford University. Available at https://cs.stanford.edu/people/eroberts/karel-the-robot-learns-java.pdf.

[54] Marcos Román-González, Juan-Carlos Pérez-González, and Carmen Jiménez-Fernández. 2017. Which cognitive abilities underlie computational thinking? Criterion validity of the Computational Thinking Test. *Computers in human behavior* 72 (2017), 678–691.

[55] Alexander Ruf, Andreas Mühling, and Peter Hubwieser. 2014. Scratch vs. Karel: impact on learning outcomes and motivation. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education.* 50–59. doi:10.1145/2670757.2670772

[56] Mehran Sahami. 2022. Handout 5 - Assignment 1: Karel the Robot. https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1226/handouts/05-assignment1.html Based on problems by Nick Parlante and Eric Roberts, modified by the CS106A staff.

[57] Kurt Schmucker. 1999. A taxonomy of simulation software. *Learning Technology Review* (1999), 40–75.

[58] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19 (2007), 469–508.

[59] Cheryl Denise Seals. 2004. *A Framework for Learning and Reuse in Visual Programming Environments: Supporting Novice Programmer Development of Educational Simulations.* Doctoral dissertation. Virginia Polytechnic Institute and State University, Blacksburg, Virginia. http://hdl.handle.net/10919/28766 Archived at https://web.archive.org/web/20250813043258/https://vtechworks.lib.vt.edu/items/b493e487-8d44-40fa-858c-703c88ba19eb..

[60] Ben Shneiderman. 1982. The future of interactive systems and the emergence of direct manipulation. *Behaviour & Information Technology* 1, 3 (1982), 237–256.

[61] Ben Shneiderman. 1983. Direct manipulation: A step beyond programming languages. *Computer* 16, 08 (1983), 57–69.

[62] Ben Shneiderman. 2024. About Ben Shneiderman. https://www.cs.umd.edu/users/ben/about.html. Accessed: 2024-09-10.

[63] David Canfield Smith and Allen Cypher. 1995. KidSim: Child constructible simulations. In *Proceedings of the Imagina'95 Conference. Monte-Carlo.* 87–99.

[64] David Canfield Smith, Allen Cypher, and Jim Spohrer. 1994. KidSim: programming agents without a programming language. *Commun. ACM* 37, 7 (July 1994), 54–67. doi:10.1145/176789.176795

[65] David Canfield Smith, Allen Cypher, and Larry Tesler. 2000. Programming by example: novice programming comes of age. *Commun. ACM* 43, 3 (March 2000), 75–81. doi:10.1145/330534.330544

[66] Rebecca Smith and Scott Rixner. 2019. The Error Landscape: Characterizing the Mistakes of Novice Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 538–544. doi:10.1145/3287324.3287394

[67] James Somers. 2017. The coming software apocalypse. *The Atlantic* 26 (2017), 1.

[68] Jacqueline Staub. 2016. *xLogo online-a web-based programming IDE for Logo.* Master's thesis. ETH Zurich. doi:10.3929/ethz-a-010725653

[69] Andreas Stefik and Susanna Siebert. 2013. An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)* 13, 4 (2013), 1–40.

[70] William Stewart and Kwanwoo Baek. 2023. Analyzing computational thinking studies in Scratch programming: A review of elementary education literature. *International Journal of Computer Science Education in Schools* 6, 1 (2023), 35–58.

[71] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[72] John Sweller, Paul Ayres, Slava Kalyuga, John Sweller, Paul Ayres, and Slava Kalyuga. 2011. The redundancy effect. *Cognitive load theory* (2011), 141–154.

[73] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31 (2019), 261–292.

[74] Thomas Douglas Victor Swinscow, Michael J. Campbell, et al. 2002. *Statistics at square one.* Number Ed. 10. Bmj London.

[75] Sverrir Thorgeirsson, Lennart C. Lais, Theo B. Weidmann, and Zhendong Su. 2024. Recursion in Secondary Computer Science Education: A Comparative Study of Visual Programming Approaches. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1.* 1321–1327. doi:10.1145/3626252.3630916

[76] Sverrir Thorgeirsson, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. Comparing Cognitive Load Among Undergraduate Students Programming in Python and the Visual Language Algot. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1.* 1328–1334. doi:10.1145/3626252.3630808

[77] Sverrir Thorgeirsson, Chengyu Zhang, Theo B. Weidmann, Karl-Heinz Weidmann, and Zhendong Su. 2024. An Electroencephalography Study on Cognitive Load in Visual and Textual Programming. In *Proceedings of the 2024 ACM Conference on International Computing Education Research. In press. (ICER '24)*. ACM, Melbourne, VIC, Australia.

[78] Katerina Tsarava, Luzia Leifheit, Manuel Ninaus, Marcos Román-González, Martin V. Butz, Jessika Golle, Ulrich Trautwein, and Korbinian Moeller. 2019. Cognitive correlates of computational thinking: Evaluation of a blended unplugged/plugged-in course. In *Proceedings of the 14th workshop in primary and secondary computing education.* 1–9.

[79] Theo B. Weidmann, Sverrir Thorgeirsson, and Zhendong Su. 2022. Bridging the Syntax-Semantics Gap of Programming. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* 80–94. doi:10.1145/3563835.3567668

[80] David Weintrop and Nathan Holbert. 2017. From Blocks to Text and Back: Programming Patterns in a Dual-Modality Environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA) *(SIGCSE '17)*. Association for Computing Machinery, New York, NY, USA, 633–638. doi:10.1145/3017680.3017707

[81] David Weintrop and Uri Wilensky. 2018. How block-based, text-based, and hybrid block/text modalities shape novice programming practices. *International Journal of Child-Computer Interaction* 17 (2018), 83–92.

[82] Jeannette M. Wing. 2006. Computational thinking. *Commun. ACM* 49, 3 (2006), 33–35.

[83] Zhen Xu, Albert D. Ritzhaupt, Fengchun Tian, and Karthikeyan Umapathy. 2019. Block-based versus text-based programming environments on novice student learning outcomes: a meta-analysis study. *Computer Science Education* 29, 2-3 (2019), 177–204. doi:10.1080/08993408.2019.1565233

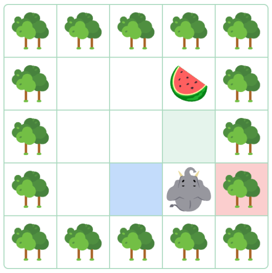
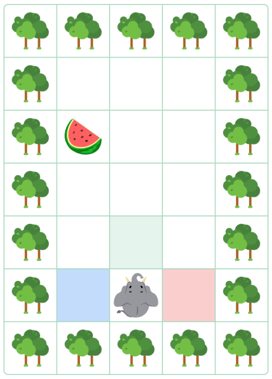
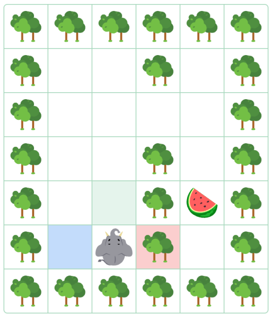
[84] Aman Yadav, Hai Hong, and Chris Stephenson. 2016. Computational thinking for all: Pedagogical approaches to embedding 21st century problem solving in K-12 classrooms. *TechTrends* 60 (2016), 565–568.

[85] JE Ziegler and K-P Fähnrich. 1988. Direct manipulation. In *Handbook of human-computer interaction.* Elsevier, 123–133. doi:10.1016/B978-0-444-70536-5.50011-7

A Tasks

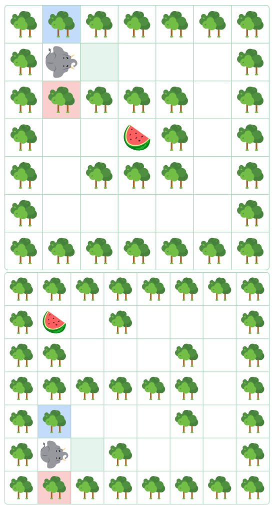
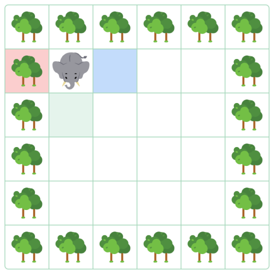
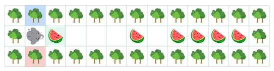
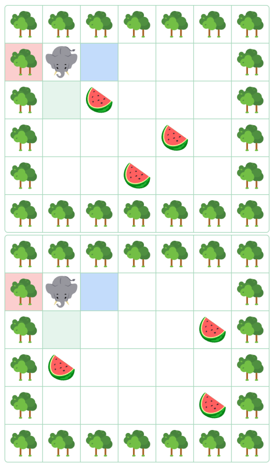
Table 5 lists the tasks we used in both studies. While Elephant does not offer direct support for counting, counting in task 9 can be achieved through changing the state. The last tasks were included to ensure that we did not have a ceiling effect in the results; we anticipated that very few students, if any, would be able to solve all tasks successfully.

Table 5: Overview of the ten programming tasks presented to participants. Each task included several examples, the number of which is specified in the column *# Examples*. Depending on the task, we show one or two representative examples in the column *Selected Example(s)*. The table also shows the states available for programming the elephant and the task description shown to participants.

Task ID	Selected Example(s)	# Examples	States	Description Shown To Participants
1		2	`SEARCHING, DONE`	The watermelon is in front of the elephant. Stop the elephant on the watermelon.
2		2	`SEARCHING, DONE`	The watermelon is somewhere in the column to the left of the elephant. Find the watermelon and stop the elephant on the watermelon.
3		2	`SEARCHING, DONE`	The watermelon is behind the right wall. The elephant should turn around and stop on the watermelon.

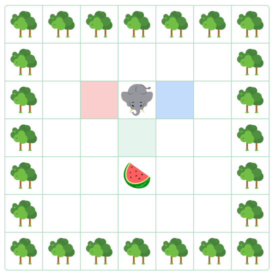
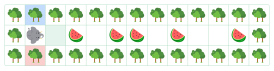
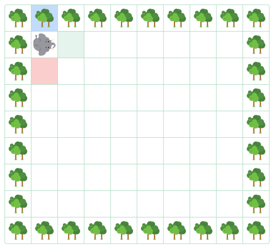
Continued on next page

Table 5 – *Continued from previous page*

Task ID	Selected Example(s)	# Examples	States	Description Shown To Participants
4		3	`SEARCHING, DONE`	The elephant should follow the maze and stop on the watermelon. The maze has only one path that the elephant can follow, and the elephant will face the entrance to the maze at the beginning.
5		1	`ONE, TWO, DONE`	The elephant should place a flag two fields ahead.
6		2	`SEARCHING, DONE`	The elephant should collect all the watermelons in the row.
7		3	`DOWN, UP, LEFT, RIGHT, DONE`	The elephant should collect all three watermelons.

Continued on next page

Table 5 – *Continued from previous page*

Task ID	Selected Example(s)	# Examples	States	Description Shown To Participants
8		3	`STARTING, SEARCHING, FOUND_WATERMELON, TURNING, GOING_HOME, DONE`	There is a watermelon ahead in the column of the elephant. Make the elephant collect the watermelon and then return to the field where it started (hint: you must mark the field yourself!). There is always an empty field behind the watermelon, allowing the elephant to turn around.
9		4	`COLLECTING1, COLLECTING2, COLLECTING3, SEARCHING, DONE`	Make the elephant collect exactly three watermelons and then stop on the watermelon.
10		2	`A, B, C, D, E, F, G, H, DONE`	Make the elephant draw a checkered pattern of flags (like on a chessboard), exactly as seen in the image.