CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python

Majeed, Kazemitabaar
Department of Computer Science, University of Toronto, Canada
majeed@dgp.toronto.edu

Viktar, Chyhir
Department of Computer Science, University of Toronto, Canada
viktar@dgp.toronto.edu

David, Weintrop
College of Education, College of Information Studies, University of Maryland, College Park, USA
weintrop@umd.edu

Tovi, Grossman
Department of Computer Science, University of Toronto, Canada
tovi@dgp.toronto.edu

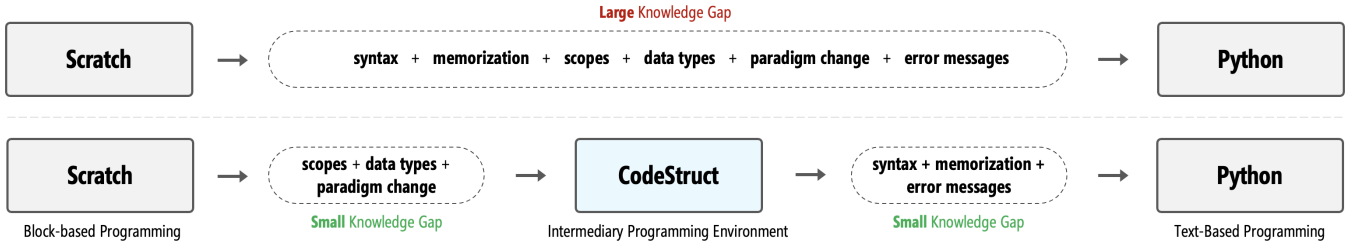

Figure 1: CodeStruct is an intermediary programming environment that eases the transition between block-based (Scratch) and text-based (Python) programming, by breaking down and reducing the knowledge gap within each transition.

ABSTRACT

Transitioning from block-based programming environments to conventional text-based programming languages is a challenge faced by many learners as they progress in their computer science education. In this paper, we introduce CodeStruct, a new intermediary programming environment for novices designed to support children who have prior experience with block-based programming to ease the eventual transition to text-based programming. We describe the development of CodeStruct and its key design features. We then present the results from a two-week long programming class with 26 high school students (ages 12-16; M=14 years) investigating how CodeStruct supported learners in transitioning from Scratch to Python. Our findings reveal how learners used the scaffolds designed into CodeStruct to support their transition from blocks to text, and that transitioning to CodeStruct reduced completion time (1.98x) and help requests (4.63x) when compared to transitioning directly to Python. Finally, learners that used CodeStruct, performed equally well (and slightly better in 10/16 programming activities) in their final transition to fully text-based Python programming.

CCS CONCEPTS

• **Human-centered computing** → Human computer interaction (HCI) → Interactive systems and tools.

KEYWORDS

high school computer science education, block-based programming, blocks-to-text transition

ACM Reference Format:
Majeed, Kazemitabaar, Viktar, Chyhir, David, Weintrop, and Tovi, Grossman. 2022. CodeStruct: Design and Evaluation of an Intermediary Programming Environment for Novices to Transition from Scratch to Python. In *Interaction Design and Children (IDC '22), June 27–30, 2022, Braga, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3501712.3529733

IDC '22, June 27–30, 2022, Braga, Portugal

ACM ISBN 978-1-4503-9197-9/22/06…$15.00
https://doi.org/10.1145/3501712.3529733

1 INTRODUCTION

Block-based programming environments (BBPEs), such as Scratch, are designed to reduce the barrier of entry to programming for young learners [7]. Research has shown that BBPEs are effective at helping K-12 students develop foundational programming and problem-solving skills [19, 37, 49], be a welcoming and fun way to introduce kids to programming [29, 36], and can support them in expressing their own ideas and interests [13, 16]. However youth may perceive block-based programming to be less powerful [47] and

wish (or need) to transition to text-based programming languages as their education progresses.

The transition from block to text-based programming can be difficult as there is a large knowledge gap of skills developed in block-based environments that do not always transfer to text-based languages [32, 46, 51] (Figure 1 Top). The difficulties associated with the transition can be grouped into three major categories: (a) removal of training wheels such as visualizing types and a browsable toolbox, (b) differences in programming style and code representation [7] between the two environments, and (c) misconceptions and difficulties that are solely associated with authoring code using a fully text-based programming language for beginners [34]. Furthermore, in theorizing about why this transition is difficult, researchers have articulated several barriers including text-based programming being more difficult to read/parse, the need for students to memorize syntax, and challenges associated with typing in commands [25] and interpreting errors [7, 14]. Finally, students are exposed to a paradigm shift in programming. Many block-based languages are event-based, allowing learners to rely on user-input or on-screen interactions (e.g., collisions) to coordinate the execution of various parts of a program [43]. At the same time, there are smaller, language-level differences, such as a **`repeat until`** block that performs the opposite behavior of a **`while`** statement. In summary, while BBPEs offer a good introduction to coding, there are substantial differences between using blocks and syntax, and between the associated programming language paradigms, that make the transition from block to text-based programming difficult.

Several designs have attempted to make the transition from block-based to text-based programming easier [28]. For example, dual-modality programming environments such as PencilCode [6] and MakeCode [2] allow the user to switch between blocks and text-based modalities with one click. Other environments seek to blend block-based and text-based features into a single environment such as the Frame-based editing approach [25] which retains some of the error avoidance and discoverability of block-based environments while attempting to support the flexibility, efficiency, keyboard control and large-scale readability of text-based editors. Finally, point-and-click editors such as TouchDevelop [1] and Grasshopper [52] almost eliminate syntax errors but lack an efficient keyboard control mechanism compared to Frame-based editing.

Although these environments can help reduce syntax errors and create executable code [33], few were designed to help young learners with the eventual transition to conventional text-based editors. At the same time, only a narrow slice of the design space of ways to support youth in transitioning from block-based to text-based programming has been explored [28, 45]. As such, there remains opportunity to further explore design approaches for supporting young learners in transitioning from block-based to text-based programming.

In this paper we introduce CodeStruct, a novel *intermediary* programming environment that aims to make it easier for K-12 learners and novices to transition from Scratch (blocks) to Python (text) (Figure 1 Bottom). CodeStruct is a browser-based Python programming environment and includes a series of design features to support novices in their transition, including: (i) a dual mode code editor that supports point-and-click insertion of code segments as well as unconstrained text-based coding (ii) a context-aware toolbox that includes basic code snippets with on-hover learning material, (iii) a structured code editor with code completion, active type-checking and fix suggestion providers, and (iv) various visual cues drawn from BBPEs to help novices know how and where commands can be used such as holes for arguments and highlighted code blocks. Unlike many other approaches for supporting this transition, CodeStruct was designed to bring effective features of block-based programming into a text-based environment designed to resemble professional text-based editors in order to prepare learners for the eventual transition to conventional and widely-used text-based development environments [9].

We then present an evaluation study of 26 K-12 students (ages 12-16; M=14 years) with no prior programming experience learning to program with CodeStruct. The study is unique in that it not only evaluates our new intermediary programming environment, but it also evaluates what impact this environment has when students subsequently transition to full text-based programming. The evaluation consisted of 11 90-minute sessions, with half of the students exposed to CodeStruct prior to their transition to text-based programming. Our results show that with CodeStruct, students spent 1.98x less time completing programming tasks in their initial environment transition and had 4.63x fewer help requests from an instructor. In a final assessment using text-based programming, students that were exposed to CodeStruct received consistently higher scores on a series of programing questions, performing better in 10 out of 16 questions.

2 RELATED WORK

2.1 Block-based Programming

While not a recent innovation (e.g., [8, 35]) block-based programing is increasingly becoming the way that youth are being introduced to the practice of programming and the field of computer science more broadly. Visual metaphors, along with user interaction and experimentation [28], are foundational elements of a BBPE. Together they provide numerous features that enable BBPEs to make programming more accessible and inviting [7, 28, 33, 47]. BBPEs use a command-as-puzzle-piece metaphor to convey information about how and where commands can be used and support a drag-and-drop interaction to help novices in assembling valid programs. Initially popularized by environments such as Scratch [36] and Alice [12], BBPEs have become widespread, with a review from 2021 identifying over 100 unique BBPEs [28]. The capabilities of BBPEs and the types of programs that can be authored with them have also expanded in recent years. It is now possible to use BBPEs to create video games [22], develop mobile apps [50], control industrial robots [42] or drones [41], query databases [23], and engage in data science [4].

Research comparing block-based programming to text-based programming has found block-based programming to be an effective introduction to programming [e.g., 27,33,49]. A meta-review of studies comparing block-based and text-based programming environments found BBPEs to outperform text-based languages with respect to cognitive learning outcomes (albeit with a small effect size), but also concluded that many open questions remain, including isolating which design features contribute to this result and

understanding how these gains translate to text-based programming [51].

Related studies have also identified drawbacks to these environments. Block-based tools fail to scale well with larger programs, and they can become cumbersome when users attempt to define commands with many components, such as mathematical formulae or complex Boolean statements [1, 47]. As youth seek to move on to create larger or more complex projects, the question of if and how to transition to text-based programming languages emerges.

2.2 Transitioning from Block-based to Text-based Programming

After studying the challenges novices face when transitioning from BBPEs to text-based programming, past research suggests that some of the features that facilitate learning also affect the difficulty of the transition. These features include the readable block structure of code, the limited need to memorize commands or syntax, and the absence of typing using drag-and-drop commands [25, 47]. As such, the challenges related to transitioning are twofold: there is both a knowledge representation problem, in terms of understanding the programming language paradigms, and an IDE problem, related to understanding how to use and interpret the functionality of the development environment and user interface. As part of their review of BBPEs, Lin & Weintrop identify three distinct approaches to supporting the block-to-text transition [28]: one-way transition, dual-modality, and hybrid.

One-way transition environments describe BBPEs that support novices in viewing their block-based programs in a text-based form or allow learners to "export" their block-based program as a text-based program for continued editing. Examples include the Blockly library [18], and the VexVR environment. *Dual-modality environments* allow youth to author their programs in either block-based or text-based forms and support them in moving back-and-forth between the two. Examples of environments that support this type of interaction include Pencil Code [6], BlockPy [5], MakeCode [2], and Tiled Grace [21]. These types of programming environments have been found to be productive in helping learners transition from block-based to text-based programming [30, 44]. *Hybrid environments* blend features of block-based and text-based programming into a single interface. A notable example is Frame-based editing [24], which was designed to have the low-threshold characteristics of block-based programming, while also retaining the expressiveness, flexibility, and keyboard-driven authoring of text-based programming languages. Pencil.cc is another example, which presented users with a text-based coding editor alongside a blocks palette, and supported learners in dragging-and-dropping block commands into a text-based program [45]. Hybrid programming tools have also been developed to allow learners to author short snippets of text-based code and embed them within a block-based program [10]. In conducting their review, Lin & Weintrop identify only 3 fully hybrid environments and conclude that "there are still fertile grounds yet to be tilled for finding new ways of supporting learners in transitioning from block-based to text-based language" [28:9]. CodeStruct was designed with this open research area in mind and introduces novel features to ease the full blocks to text transition.

2.3 Supporting Novice to Expert Transition in Desktop Software

Our work is also inspired by broader HCI research that examines the novice to expert transition in software. We refer the reader to a thorough survey of such research by Cockburn et al. [11]. Early HCI research proposed "training wheels" for user interfaces to limit the functionality of complex user interfaces for novice users [3]. Multi-layered interfaces extend on this concept and slowly expose users to more complex software functionality as their expertise develops [17, 39]. Scarr et al. discuss in detail the performance dip that occurs when transitioning from a novice to expert mode of a user interface: *"users are likely to suffer a performance dip when switching to a new modality, even if it offers a higher ultimate performance ceiling"* [38]. They propose an interactive system that minimizes the dip in performance when transitioning to expert usage of desktop software. The Blocks-to-CAD program [26] utilizes a similar concept to help users transition from novice to expert 3D modelling skills but instead uses a series of smaller interface transitions. CodeStruct follows a similar philosophy by altering the blocks to text transition into a series of two smaller transitions.

3 ITERATIVE DESIGN PROCESS

Our initial design consisted of a toolbox of insertable code snippets that was informed by prior work that showed students benefit from the browsable set of blocks used in BBPEs [47]. Afterwards, we expanded the toolbox by including tooltip style documentation and context-aware feedback (e.g., visually updating the toolbox as the learner navigates through code). For editing code, we were inspired by the syntax-error avoidance and visual affordances of BBPEs as well as the freedom in editing and code suggestions in text-based editors. Therefore, we developed a custom, structured editor that allowed freely typing in the editor while eliminating syntax errors and displaying block-shaped scopes and holes for empty expressions like BBPEs. We hypothesized that incorporating the toolbox and aforementioned visual supports inspired by BBPEs, would ease the transition from these environments. Finally, we refined the design of CodeStruct using an iterative, human-centered design process that included interview and design probe sessions with five CS educators and a pilot study with a prototype of the CodeStruct system with the above features.

The interview included five CS educators (four high school programming teachers and one after school coding instructor). These interview sessions were conducted remotely and lasted roughly 60 minutes. They included an introduction, questions about curriculum, programming language and tools that were used to teach programming, and a demonstration of the current CodeStruct interface and features. Educators were generally positive about the tool, particularly how the toolbox promoted exploration and how the editor almost eliminated syntax errors. One educator said, *"editing in this tool is similar to an actual text-editor, but with some handles"*. There were also many useful suggestions and ideas including: (i) reducing the number of things that are automatically done for the user and allowing them to make mistakes and learn from them, (ii) adding a search box in the toolbox to

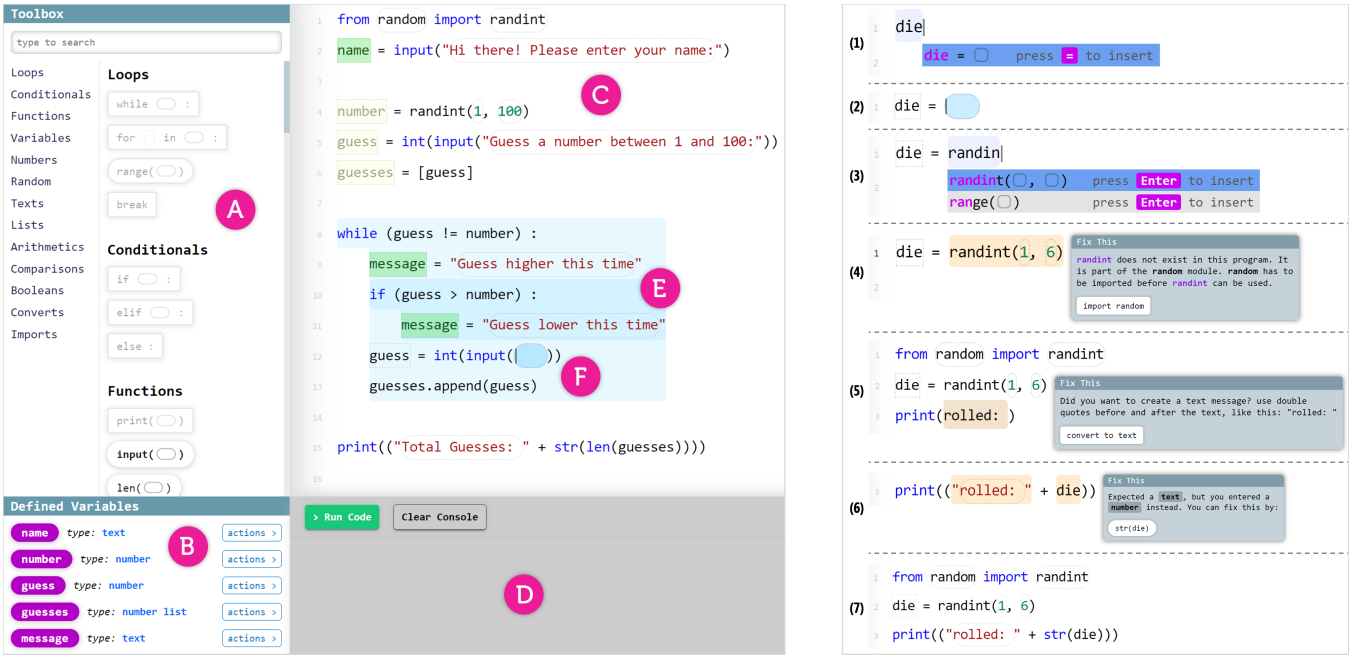

Figure 2: Left: The CodeStruct interface. A) Code Toolbox. B) Variables Toolbox. C) Text editor. D) Executaion Console. E) Scope Highlights. F) Variable Holes. Right: A sample programming sequence with CodeStruct.

improve code exploration, (iii) allow writing comments and docstrings to reduce re-textualization mental effort, (iv) including visualizations of conditions and Boolean expressions, and (v) allow users to produce visual outputs in addition to textual input and output.

In the pilot study, a 12-year-old student was taught the basics of Scratch in three 90-minute sessions, followed by three sessions of using an early version of CodeStruct to solve the same programming problems with Python. Several recurring problems emerged: (i) forgetting what each piece of code does and how to use them correctly, (ii) failing to understand how to concatenate strings and integers, (iii) missing double quotes for strings, and (iv) being unable to easily modify already inserted code. Based on these observations we iterated on the system design and added several real-time help and syntax error identification features which will be described in Section 4.

4 SYSTEM DESIGN OF CODESTRUCT

Informed by prior research and data collected as part of our interviews and pilot study, we arrived at our final design for CodeStruct. The system was designed to allow users to utilize the best of block-based programming paradigms in a text-based editor and allow the user to remove their "training wheels" as they advance. CodeStruct is designed to support two transitions: transitioning from a BBPE to CodeStruct and then transitioning from CodeStruct to a fully featured text-based programming environment. Specifically, CodeStruct was designed to serve as an intermediary between Scratch (blocks) and Python (text)—the two most common languages/environments in each modality.

4.1 CodeStruct Interface

CodeStruct is a browser-based Python code editor with built-in code execution functionality and an interface like that of modern programming environments (Figure 2 Left). Analogous to BBPE's, CodeStruct includes a context-aware toolbox from which code (A) and variables (B) can be inserted at the current cursor location. Analogous to text-based programming environments, users can also type code directly in the text editor (C) and a code-execution console is shown below (D). Subtle highlighting of scope is provided (E), as well as "holes" for inserting blocks of code (F).

4.2 Context-Aware Toolbox

The toolbox allows users to insert code, and is split into two sections: Python statements, operators, and built-in functions above, and a list of user-defined variables below. CodeStruct's toolbox is dynamic, and updates currently valid code insertions based on the current caret location or cursor selection, and the current state of the abstract syntax tree. Buttons for invalid code insertions are disabled but are not removed from the toolbox. This provides novices with opportunities to learn about syntax and general code structure by referencing the valid and invalid code buttons in the toolbox. The toolbox also allows code to be browsed by category and searched by keyword. The toolbox provides tooltip style documentation for each piece of code which is discussed further in Section 4.6. Finally, the user defined variables section of the toolbox displays all variables that are available in the current scope (based on where the cursor is located), their type, and a list of context-aware actions associated with that variable.

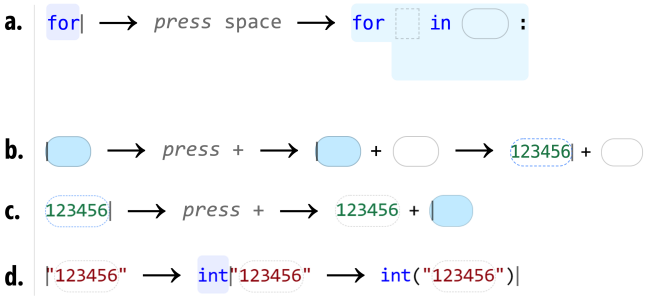

Figure 3: Structured editing in CodeStruct.

4.3 Context-Aware Code Completion

The code completion mechanism in CodeStruct is like the ones found in conventional IDEs with some enhancements, such as inserting entire statements and hiding invalid code based on the current position of the cursor. For example, if the user starts typing in **`for`** at the beginning of a line, followed by a space, the editor will insert a complete version of Python's for statement that includes two different types of holes: one with sharp edges and dashed for defining the looping variable, and one with solid rounded edges for inserting the sequence expression (Figure 3a). Similarly, the user can still type in the **`for`** inside an empty hole that requires an expression, but the context-aware code completion would not show it as one of the possible insertions. This *protected* implementation of the code completion tool lowers the barrier to entry for novices.

4.4 Structured Editing and Navigation:

CodeStruct is a structured editor and editing is mostly done on a token level instead of the character level. Code inserted from the toolbox is prefabricated and is inserted in its entirety. When typing, the tool will complete entire pieces of code for the user as outlined in the section above. Editing is completed on the character level for numbers, string literals, and variable identifiers. Navigation is also performed at a token level outside of these two situations. The user can use the arrow keys or the mouse to navigate the cursor to the closest valid location. In the cases where the cursor is attempted to be placed at an invalid location (somewhere where nothing can be inserted because it would violate syntax conventions), the cursor is placed at the closest valid and editable location.

The context awareness enabled by the abstract syntax tree allows CodeStruct to provide interactions that are typically not possible in structured editors like Genie [31] and Grasshopper [52] or BBPEs. One common example of this would be the creation of binary arithmetic expressions. In a BBPE or other structured editors, the operator is inserted first and then the operands are filled. However, in conventional editors, the most common sequence of steps is to fill the first operand and then insert the operator. CodeStruct supports *both* methods. The user can insert such expressions operator-first (similar to BBPEs) by using the toolbox or by pressing the operator's key (Figure 3b), or they can follow the conventional sequence (Figure 3c). The same applies to data type casts, where the user can perform the following sequence of steps to cast a string as an integer (Figure 3d). Our structured editor is unique by creating an experience like conventional editors, while also maintaining all the supports that a structural editor offers to a novice programmer.

4.5 Automatic Hint Suggestion

CodeStruct provides automatic hints and warnings such as type mismatch detection, undefined identifier detection, out-of-scope references, and incomplete code detection. Despite such supports being an integral part of conventional programming, it is not something that is available in BBPEs or environments such as Grasshopper [47]. Much like in a typical editor, if CodeStruct detects erroneous code, it will highlight it and display a short warning message with fix suggestions on hover. CodeStruct can identify the location of these errors accurately as it has access to the complete AST and can offer better and more personalized suggestions for fixing them. In addition, the system keeps track of whether the code is runnable or not (will it result in a compile or runtime error) and disallows the user to run code that would result in such errors, instead pointing them to where the error is and how to fix it if they try to run the code. These hints enable students to learn from their mistakes and provide much needed error resolution help to novices.

4.6 Visual Aids and Feedback

CodeStruct utilizes a combination of subtle visual aids that like those used in BBPEs but are less rigid and at a reduced level of visual prominence, to prepare users for the full transition to text editors (Figure 4). Holes are differentiated by their shape to indicate if they are for expressions or text. When selected, expression holes are completely highlighted to indicate that they may contain complete pieces of code, while the text holes show the cursor indicating that insertions will be made in the form of a single character at a time. When selecting a hole, identifiers of variables that could be referenced in the selected hole are highlighted in the editor in either green if they are valid, or pale yellow to indicate that this insertion would be invalid but can be fixed. CodeStruct displays scopes directly within the editor with a pale blue outline that gets darker as the depth of the scope increases. In the toolbox, every type of construct (statement, expression, empty placeholder/hole) has a specific outline shape in the toolbox and within the editor window. CodeStruct also draws the user's attention to any newly created variable, by highlighting it in green for a period after creation to indicate to the user where that variable is found in the toolbox.

4.7 Accessible Learning Material

By hovering over code in the toolbox the user can see enhanced tooltips that provide: a concise description of what clicking the code will do, a short explanation on how the code works, whether it can be inserted in the current context and the reason if not, and runnable example code. In the variables section, tooltips show the variable identifier and type, as well as the most common suggested actions for a variable of this type. Three additional forms of assistance are provided in the tooltips: (a) quick hints, (b) executable examples, and (c) and step-by-step example code execution (Figure 4). Quick hints have some common usages, executable examples allow the user to test and modify the code inside them, and step-by-step examples provide opportunity to learn about line-by-line execution of sample code, similar to Python Tutor [20]. This allows students

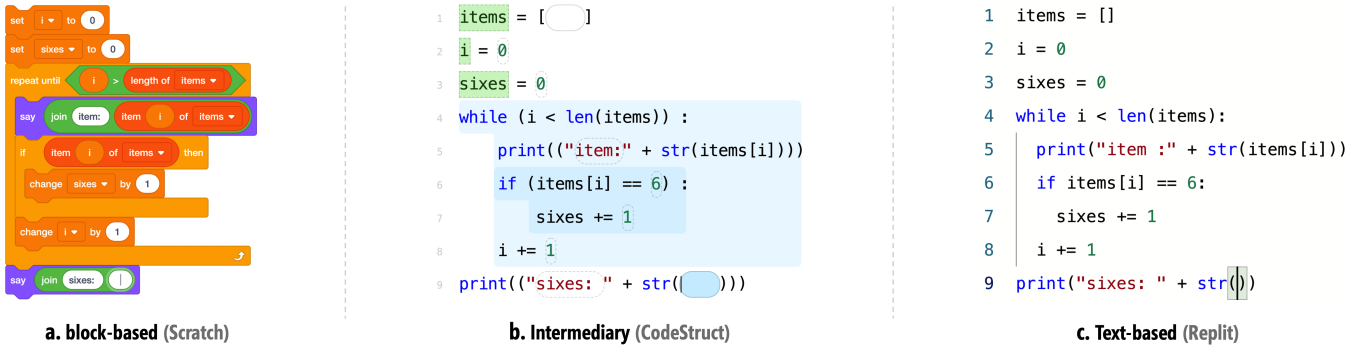

Figure 4: Visual aids of CodeStruct resemble those provided in BBPEs but are less prominent, to help with the transition.

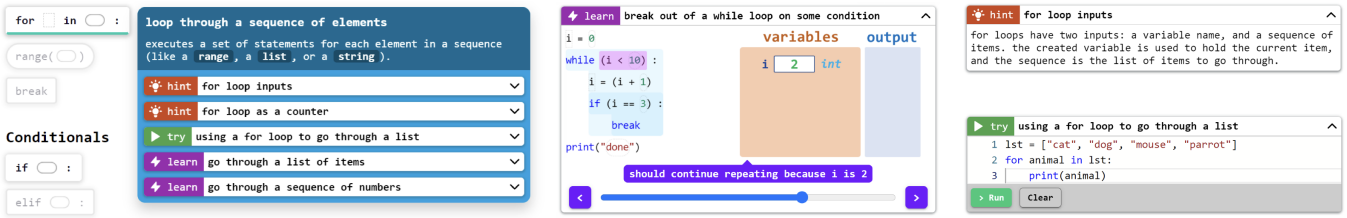

Figure 5: Left: Enhanced tooltips for elements in the code toolbox include 3 additional forms of enhanced assistance. Middle: Step by step walkthroughs allow users to move a slider to walkthrough code and see how the variables change. Right Top: Quick hints include additional tips on using the code. Right-Bottom: Users can edit and run code examples.

to learn, explore, and search for tutorials easily and with no context-switching.

4.8 Using CodeStruct: An Example

Figure 2 (right) shows how a user could utilize various features of CodeStruct to write a program that would display a random number joined with a text. The user starts typing in the editor and the only piece of code they can insert using the code completion is creating a new variable (Step 1). The user creates the variable by pressing the Enter key (to select the item in the autocomplete) or pressing the = key. The cursor immediately jumps to the right of this assignment expression which is highlighted in blue (Step 2). The user decides to enter a random integer using the **`randint`** function and can see that in the code completion menu (Step 3). CodeStruct shows a warning message about the **`randint`** function that requires an import (Step 4). After the user clicks on **`import random`** or types the correct import statement, they decide to start printing out the text **`"rolled: "`** concatenated with the random number. However, the user forgot to include double quotes around the text, so the tool immediately shows another warning message to convert what they typed to a text (Step 5). The user then adds a plus operator to the right of the string and inserts the **`die`** variable but is confronted with another error message, this time a type mismatch error with a suggestion to wrap the variable with the **`str`** function that would cast the integer to a string (Step 6). The final code is shown in Step 7.

4.9 Implementation

CodeStruct is written in TypeScript on top of the Monaco Editor [53] and uses Pyodide [54] to execute Python code in the browser. In addition, CodeStruct uses a custom abstract syntax tree configured for Python statements and keywords, however, it can easily be extended or changed to support a different language if it is runnable inside the browser. When navigating, CodeStruct checks the direction of the navigation and the next available valid caret placement/selection in that direction. Such a valid placement/selection is defined by being able to have code inserted into it without structurally breaking the AST. As a result, the navigation within CodeStruct is token-based instead of character-based.

When the caret location is changed, CodeStruct runs a series of validations on the AST to determine what code can be inserted at the new caret position. There are two types of validation that run at this point: a) structural and b) type (if the code is an expression). Structural validation deals with the syntax and general location of the code placement, while type validation looks for potential type mismatches. For each piece of code there are three outcomes, the insertion is either: valid, invalid, or draft. Invalid insertions are completely disallowed and are subsequently disabled in the toolbox and code completion. Draft insertions are updated with a pale-yellow color inside of the code completion menu and have a warning added to them in the toolbox until fixed or finished and converted to a valid insertion which is indicated by a green flash. The conversion from *draft* to *valid* is how CodeStruct allows to freely type code like a conventional editor.

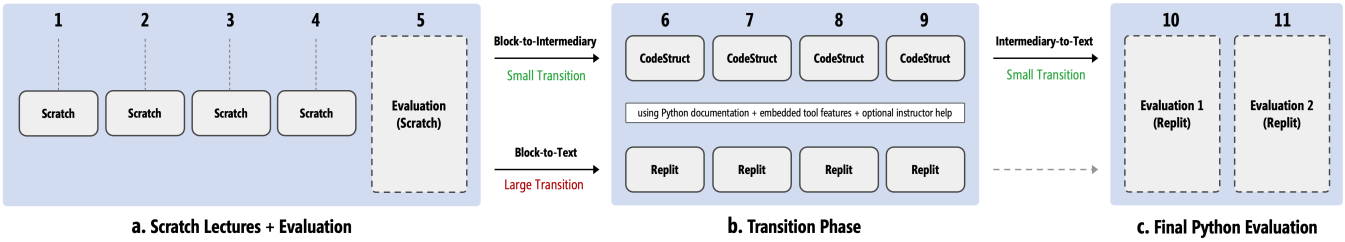

Figure 6: The study consisted of 11 sessions across three phases. The top group used CodeStruct during the transition phase.

Operands of binary expressions is an exception and runs post-insertion. This is done to make it more flexible. It is performed through a recursive walk of the expression in the AST and cross-checking operand types. In addition to this, CodeStruct performs further checks during code deletion. When deleting user-defined pieces of code such as variables, a traversal of the tree is performed to identify invalid usages of the variable (if any). These are marked with warnings and the user is suggested to delete them as they no longer reference an existing variable. Finally, when code is to be executed, a BFS traversal of the AST is performed to determine if there are any unfilled holes or pieces of code with warnings on them in the tree. If there are, the code does not run and instead the user is provided further information on potential fixes. This approach allows us to catch errors that otherwise would not be caught until compilation is attempted, coinciding with our goal of lowering the barrier of entry for text-based programming.

5 SYSTEM EVALUATION

To evaluate CodeStruct, we conducted a virtual, two-week long study comprised of 11 90-minute sessions. The study was conducted over the Google Meet platform, with the first author serving as the primary instructor and the second author as an assistant. The study was broken down into three phases: (i) a formal introduction to Scratch programming for four sessions followed by an evaluation of their Scratch learnings on the fifth session, (ii) an independent transition of participants to either CodeStruct or Replit to write Python code for four sessions, and (iii) a final evaluation of their Python programming skills and knowledge for two sessions. In splitting the participants during the second phase we can compare the results of transitioning with CodeStruct to going directly from Scratch to Replit.

5.1 Participants

Participants were recruited through a local school and an after-school program in a North American city. The study consisted of 26 students (14 female) ages 12-16 (M=14.1; SD=1.2). Students were screened to have no prior programming experience other than Scratch. Parent/guardian consent and child assent was obtained before the first session of the course and each student was given a $50 gift card at the end of the class.

5.2 Introduction with Scratch

To begin the study, all youth in the study went through the same 4-session Scratch sequence. Each session included 3-6 programming activities and 3-7 comprehension questions. All programming activities were followed by a Likert scale that asked about the difficulty of the problem. The event-driven and multi-thread programming styles in Scratch were de-emphasized in our study to foreground on core imperative programming concepts. Therefore, the first session covered the basics of Scratch programming such as sequence of code execution, how to display values, working with mathematic operators, generating random numbers, and working with variables. The second session started with a recap on the previous session by answering all the activities in the previous session in detail following with a lecture on Booleans and conditionals and new activities. The third and fourth sessions followed a similar pattern to the second session but with new topics on loops, lists, and important patterns such as the accumulator pattern or traversing through a list using a loop. Students were encouraged to ask questions if they needed help as the goal of this phase was to teach them the basics of Scratch and computational thinking.

5.3 Transitioning to Text

In the second phase of the study, participants transitioned to text using either CodeStruct or Replit, a popular Monaco-based Python editor with autocomplete and basic code analysis that flag programming errors. We used participant performance on the Scratch assessments to divide the students into two groups of 13 students that were comparable in terms of knowledge and skill. The first group used CodeStruct, while the second group used Replit. The goal of this phase was to study how they initially transition from Scratch to Python and compare their performance. Learners in both conditions were given the same set of activities to complete: (i) convert a given Scratch program to Python, (ii) the same programming activities that they did in Scratch but this time in Python. Finally, to provide additional assistance, all learners were given a documentation that included all of the Python sections of W3Schools [55]. The group that used CodeStruct also had the ability to use any of the embedded learning mechanisms inside CodeStruct.

5.4 Data Collection and Analysis

We collected a variety of measures and observations throughout the study, including video recordings, online form data, content assessments, and log data on student interactions with the programming tools and documentation. At the end of the Scratch phase (session 5) and Python phase (session 10), we administered content assessments to evaluate student understanding. At the end of phase 1, we administered a Scratch content assessment that included 19

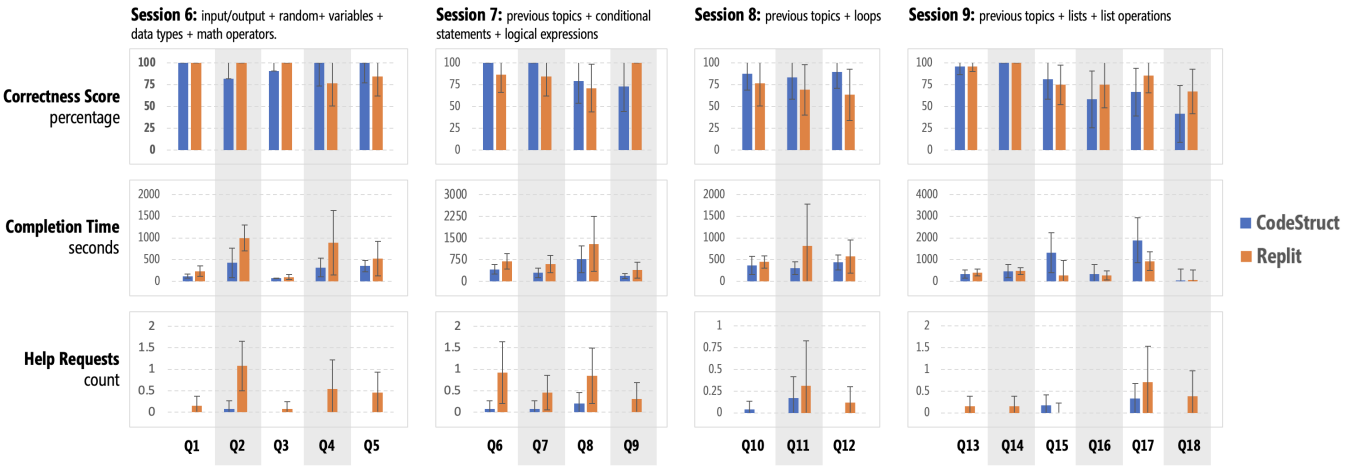

Figure 7: Top: Correctness scores, Middle: competition time, and Bottom: help request during the transition phase.

multiple choice comprehension questions, and 7 programming challenges. The programming challenges and questions were inspired from [48] and evaluated common programming misconceptions—derived from [40].

For each of the programming challenges throughout sessions 6-11, we measured each student's perceived difficulty, correctness, and completion time. To measure perceived difficulty, learners were given Google Forms that included all problems and the programming challenges within the form accompanied a five-point Likert scale on difficulty. Furthermore, all participants were asked to share their entire screen so that we could record and analyze their performance later, as well as to help them if they were struggling with a program or had a question. The video recordings were used to measure task completion time, number of help requests, number of problems/errors (by type), learning material usage, and solution correctness. To measure correctness fairly, the final solution for each task was graded independently by the first two authors using a rubrik (deducting 25% for every issue in their solutions).

At the conclusion of the second phase of the study (session 10 and 11), we administered a second assessment, this time asking questions in Python. The assessment included Python versions of the Scratch Evaluation questions, as well as additional Python programming and conceptual understanding questions. The questions were designed to evaluate (i) general Python programming ability, (ii) python-specific concepts (e.g., for loops), and (iii) Scratch to Python conceptual differences (e.g., while loop conditions and list indexing). Both conditions in this phase used the Replit environment with no help from the instructors or the instrumented Python documentation. At the end of the study all the 13 learners from the CodeStruct condition answered a questionnaire that included qualitative questions about CodeStruct and Replit. Our analysis omits data for one student from the CodeStruct group who chose not to complete the problems.

6 RESULTS

We discuss our results on a subset of the programming activities that students worked on, focusing first on the transition phase (Figure 6b) and then the final evaluation phase (figure 6c). The results are presented through visualizations of means, and confidence intervals, as opposed to Null Hypothesis Statistical Testing—an approach that is favored by those in HCI that see benefit in switching from statistical testing to reporting informative charts and offering nuanced interpretations of results [15].

6.1 Transition Phase

The transition phase looks at the transition from Scratch to CodeStruct for the CodeStruct group, and the direct transition from Scratch to Replit for the Replit group. The transition phase consisted of 33 programming activities (15 construction and 18 conversion activities) across sessions 6, 7, 8, and 9.

6.1.1 Task Performance Measures. Task performance along three dimensions (correctness, completion time, and help requests) is illustrated in Figure 7 for 4 sessions. The overall results show that both conditions had similar performance in terms of correctness score (CodeStruct: M=84.7%, SD=34.5%, Replit: M=84.5%, SD=34.8%), with CodeStruct scoring higher than Replit in session 8 (CodeStruct: M=86.8%, SD=31.9%, Replit: M=69.9%, SD=45.9%), and Replit performing better in session 9 (CodeStruct: M=73.9%, SD=42.1%, Replit: M=84.7%, SD=32.9%). Looking at completion time (the middle row of tables) learners in the CodeStruct group were able to finish problems in the first three sessions of the transition phase roughly two times faster (CodeStruct: M=340s, SD=330s, Replit: M=675s, SD=767s). However, the CodeStruct group was 30% slower in the last session of the transition phase. Furthermore, the total number of help requests was 4.63 times less for the CodeStruct group which requested help for only 19 times compared to 88 times for the Replit group. Finally, the Python documentation was used 2.6 more times in the Replit group (Replit: 156 times, CodeStruct: 60 times). These show that the CodeStruct group went through the transition phase more independently and with less context switching.

Moreover, analyzing the types of issues that learners encountered during the transition phase and comparing them between each

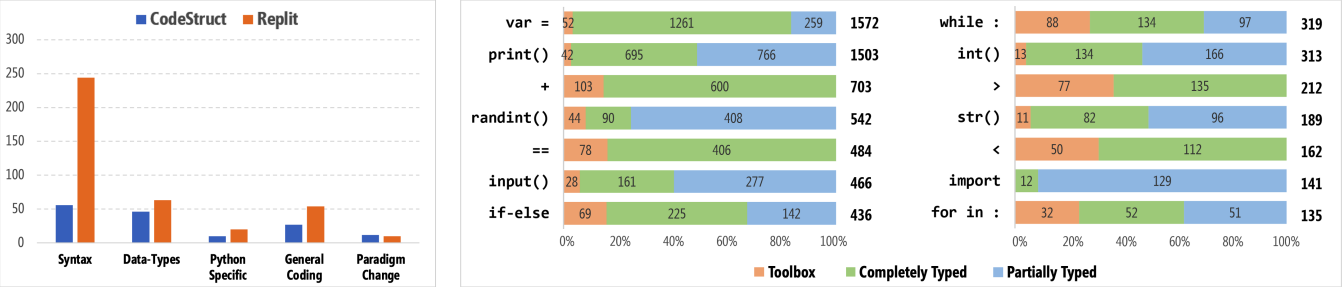

Figure 8: Left: Comparison of the number of problems encountered during the transition phase of the study, broken down by type. Right: Toolbox vs. keyboard usage in CodeStruct, ordered by frequency of use.

condition, we found that syntax errors (which are shown in the left-most pair of columns in Figure 8) were the most frequently observed coding error learners encountered and occurred more frequently in the Replit group (CodeStruct: 56 times, Replit: 244 times). The difference in syntax errors was visible in both quantity and type of error between condition. Learners from the Replit condition encountered about 40 parenthesis mismatch problems, and almost all learners (9 out of 13) forgot colons or had similar issues at the first time they wrote an if statement in Python. For the CodeStruct group, we did not see any type of issues related to missing colons and parenthesis or incorrectly calling functions, as the tool was built to ensure correct syntax. Indentation was another recurring source of issue that caused about 23 errors for the Replit condition (which could also be attributed as a semantic issue).

6.1.2 Usage of CodeStruct Features. By analyzing the log data from CodeStruct we can identify how and when learners chose to type code and when they used the toolbox to insert code segments used the toolbox to insert code segments. The 10 most common commands present in student projects were used a total of 6550 times. Of these code insertions, 9% were added to the program by clicking on the toolbox, 59% were completely typed, and 32% were partially typed and inserted using the autocomplete. Figure 8 (Right) shows the breakdown of toolbox and keyboard usage for the 14 most inserted commands in CodeStruct. It can also be seen that the autocomplete feature was used consistently across commands that were longer than a single character. Looking at the proportions, we can identify the **`while`** and **`for`** loops and the comparison operators to be the most inserted commands from the toolbox.

Analyzing CodeStruct's toolbox usage, we observed that 10 learners solely relied on the toolbox to find code, and four learners also successfully used the search bar 24 times with queries such as "repeat" or "random" to filter the results. To learn about code, learners mostly relied on non-interactive material in the tooltip menus such as reading the hover descriptions more than 5 seconds (106 times) or clicking on one of the executable examples (70 times), but the step-by-step code examples were used much less (8 times). Analyzing the automatic hint suggestions, the most common hints and warnings that were successfully used, was importing a module needed for an inserted function (28 times), followed by fixing a type mismatch error (26 times). Furthermore, analyzing the post-study questionnaires where we asked learners about their preferences between the Python Documentations and the embedded tooltips in CodeStruct, 10 favored the CodeStruct's embedded tooltips and mentioned how it was easily accessible, reliable, and concise. For example, one student wrote *"CodeStruct tooltips was used more because it is more concise, and you can see first-hand how the commands should be used"*. Finally, the user-defined variables section in the toolbox, which dynamically displayed variables and their types, was not used to insert variables, and rarely used to learn about type-specific actions.

6.2 Final Python Evaluation Phase

The final Python evaluation phase investigates the transition from CodeStruct to Replit for the CodeStruct group, which is contrasted to the Replit group, who already had exposure to Replit in sessions 6-9. This was the first time that learners from the CodeStruct group were authoring Python code in a fully text-based editor. Both groups used Replit for the evaluation and did not have access to any documentation. To understand the differences, we look at responses to the summative multiple-choice questions and final set of programming exercises.

6.2.1 Programming Tasks Results. Overall, both groups had similar performance in the final programming tasks, which is illustrated in Figure 9. Overall, scores were higher for learners in the CodeStruct condition on 10 of the 16 programming tasks. The average score across all tasks was 75% compared to 68% for the Replit group. The CodeStruct group had a slightly higher completion time the first day of the evaluation (CodeStruct: M=471s, SD=512s, Replit: M=360s, SD=409s) and a slightly lower completion time the second day of the evaluation (CodeStruct: M=321s, SD=531s, Replit: M=350s, SD=650s) but with almost no differences. The trend is also similar for number of encountered issues with the CodeStruct having slightly more issues in the first day (CodeStruct: M=1.55, SD=1.32, Replit: M=1.31, SD=1.21), and slightly less on the second day (CodeStruct: M=0.79, SD=1.02, Replit: M=1.01, SD=1.21).

Comparing completion time and the number of encountered issues of the two conditions between the first and second days of the evaluation phase, we can see that the CodeStruct group noticeably progressed through a learning curve for the first few problems. For example, comparing conceptually similar problems from the first and second days, such as Q3 and Q5 (from session 10) where learners had to write a heads-or-tails program and a program that would compare the sum of two random numbers, both

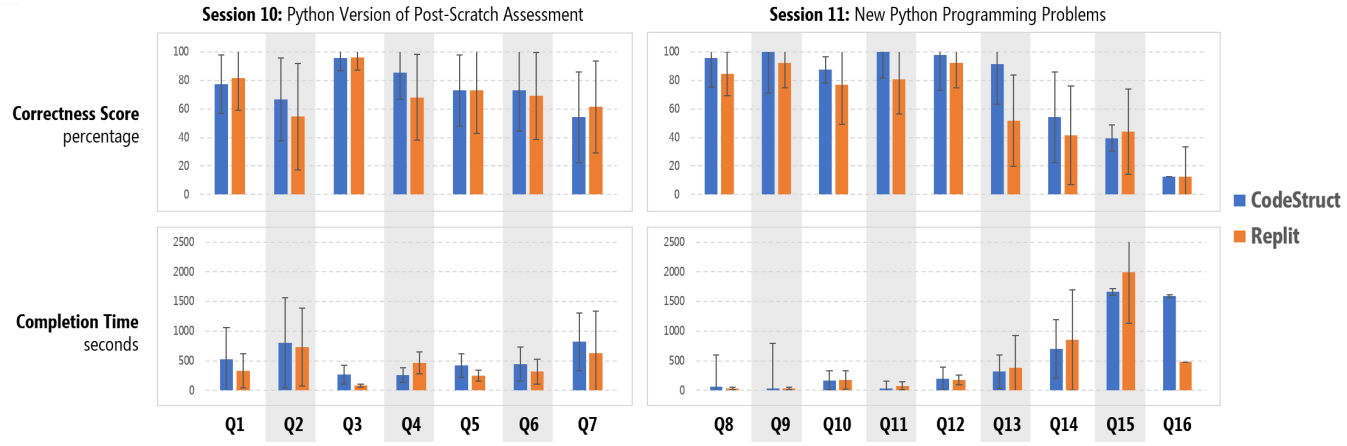

Figure 9: Results from the programming tasks in the final python evaluation phase.

conditions had similar correctness scores. However, the average number of issues per participant for the CodeStruct group was 1.68, while being only 0.5 for the Replit group. Similarly, the completion time for the CodeStruct group was more than two times higher (CodeStruct: M=336s, SD=267s, Replit: M=149s, SD=113s). This difference between the two conditions was substantially reduced as the study progressed. For example, on Q12 asked learners to write a program that would ask the user to enter two numbers and to print "yes" if their sum is greater or equal to 10, otherwise print "no". The completion times were similar (CodeStruct M=195s, SD=227s, Replit: M=177s, SD=117s), while the number of issues per student was less (CodeStruct: 0.5, Replit: 1.15).

6.2.2 Comprehension Questions Results. Overall, results from the comprehension questions were similar (CodeStruct: M=70.9%, SD=17.8%, Replit: M=68.8%, SD=22.9%). Comparing across categories of questions, scores were CodeStruct 91%, Replit 88% for variables, concatenation, and data types; CodeStruct 81%, Replit 85% for Boolean expressions; CodeStruct 61.5%, Replit 57.5% for conditionals; CodeStruct 61.5%, Replit 63.5% for While loops and nested loops with conditionals; CodeStruct 61.6%, Replit 64% for lists; and CodeStruct 98%, Replit 94% for Python specific operators. On two questions which asked to compare Python's while loops with the repeat until block from Scratch, learners in the CodeStruct condition scored 83.5% while the Replit group scored 48.5%.

6.2.3 Qualitative Results - How was CodeStruct perceived: At the end of the study, learners that experienced both Python programming environments, generally felt that CodeStruct was user-friendly. When learners were asked to tell what they like about CodeStruct, they mostly mentioned the toolbox on the left which shows the available commands and a good place to learn from, as well as how CodeStruct helped with auto filling code. One student wrote *"CodeStruct shows you exactly what you did wrong, why, and a solution to fix it"* and another student mentioned *"You could see everything that you could use, you could see how to use it, and you could understand what went wrong very easily"*. They were also asked to rate the difficulty of programming with Replit and CodeStruct separately. Comparing the results, eight learners felt that understanding errors was much easier in CodeStruct, seven felt that memorizing commands and their usages were easier, and five that typing and spelling commands was easier in CodeStruct. However, six reported that editing existing code was more difficult in CodeStruct.

7 DISCUSSION AND LIMITATIONS

Overall, our study showed promise for CodeStruct and the concept of intermediary tools. The initial transition to CodeStruct resulted in fewer programming issues than transitioning directly to Python, but learners were still able to perform equally well (and better in 10/16 tasks) in the final Python evaluation. We now provide further discussion of our results, limitations, and future considerations.

7.1 Design supports for Transitioning from Block-based to Text-based Programming

CodeStruct was designed to support two important transitions: transitioning from Scratch and transitioning to fully text-based Python. The first transition (from Scratch to CodeStruct) demands lowering the barriers to text-based programming to support novices in writing Python code and helping them draw on their prior experiences with a BBPE. This was mediated by: (i) incorporating familiar concepts from block-based programming into CodeStruct (e.g., the toolbox), familiar visual affordances (e.g., empty expression holes and block-shaped scopes), and programming style (e.g., not requiring to handle syntax, and being able to author expressions from outside to inside), and (ii) developing various automatic help and learning mechanisms into CodeStruct such as displaying immediate hint suggestions and including accessible learning material in the toolbox. Our results show that using CodeStruct successfully reduced the mental demand associated with syntax (e.g., indentations, parenthesis) while also reducing issues related to data types.

The second transition (from CodeStruct to Python) requires developing a learner's text-based programming skills and cultivating their knowledge about syntax, semantic and data-type paradigms in Python. This was mediated by incorporating familiar practices from text-based programming into CodeStruct such as being able to use the keyboard to author and edit code character-by-character, providing code completion suggestions, and syntax-highlighting

for Python. Our analysis found that CodeStruct enabled students to mostly use the keyboard and the code completion suggestions to insert code (Figure 8, Right) in a way that is almost indistinguishable from fully text-based programming environments. To cultivate python-specific paradigms, CodeStruct passively develops syntax knowledge by always displaying the correct syntax. Future work could take a more active approach by using semi-transparent punctuation that would require the learner to type over. Finally, for datatype, which caused the second most recurring issue in the transition phase (Figure 8, Left), CodeStruct relied on immediate hint suggestions incorporated in the toolbox.

7.2 CodeStruct Design Innovations

One of the main principles used in designing CodeStruct that separates it from existing environments such as Stride [25] (frame-based) was *including familiar concepts from both conventional block-based and text-based programming environments*. This can be seen in the design of the main layout, toolbox components and categorization, editing and navigation style, code completion, highlighting errors, and displaying code, scope, and empty expressions. This approach is distinct from other environments trying to accomplish the same outcome. Frame-based editors [25] are designed to be steppingstones between blocks and text. Inserting code is different for frames, expressions, and method calls in Stride. Expressions and method calls are totally text-based and mediated by maintaining the structure of parenthesis and quotes, and code completion for method calls. However, frames (high-level code blocks such as variable assignments, conditionals, or loops) are inserted using a cheat-sheet (like CodeStruct's toolbox) either by clicking or pressing a hotkey and not by typing. Stride also uses a special frame cursor to navigate between frames in addition to the simple text caret used for navigation between characters. Although, these features make the transition from blocks easier, they do not directly transfer to text-based programming. In contrast, we designed CodeStruct with visual elements, editing, and navigation styles that are almost identical to either blocks or text to reduce the transition gaps associated with the full blocks-to-text transition. Furthermore, dual-modality environments such as MakeCode [2] use a conventional text-based editor that include code completion and similar syntax highlighting in addition to having the ability of dragging blocks of textual code from a toolbox into the editor (with minimal validation). Furthermore, there are no paradigm shifts in dual-modality programming environments (as both modalities are based on the same programming language). However, these environments offer no intermediate support between the two supported modalities and no features that support the blocks-to-text transition (e.g., syntax or data-types support) beyond rendering programs in both modalities.

7.3 Limitations and Future Work

CodeStruct's approach to supporting learners showed promise, however, there are some limitations of its design and the current study the point the way towards future work on the environment. For example, this study took place as a voluntary after school course and while we tried to control as many aspects of the study as possible, there were still several aspects of the study that may have impacted the results, such as participants' prior mathematics and logic skills, and learners working at different paces given the fixed 90-minute time limit. Furthermore, the study was limited to the basic concepts of programming as we did not implement defining functions or classes in CodeStruct. Future iterations of our tool will allow this. Another limitation comes from the fact that the study included eleven 90-minute sessions and was conducted in two weeks (a session for every workday of the week). We acknowledge this is a lot of content in a short time, so this could have potentially caused some learners to be overwhelmed with new topics and lose motivation. Our results are also limited by the sample size which could be scaled with future developments of the tool. Finally, an in-depth analysis of learners' progression, qualitative analysis, and within-participant analysis is left for future work.

From a design perspective, there were also some limitations that may have impacted the results. First, several previously unknown bugs with CodeStruct forced participants to restart the editor (on average 1.9 times per participant) during 6 hours of usage. Our analysis revealed that some aspects of the environment were rarely (if ever) used, such as the user-defined variables section and the interactive tooltips in the toolbox (such as Figure 5 Middle). Future iterations of CodeStruct could use gamification techniques to motivate using the interactive learning material, dynamically display user-defined variables in-line with user code and combine it with live programming for better debugging support.

8 CONCLUSION

This paper contributes a new intermediary programming environment that helps novice users transition from block-based programming environments to fully text-based environments. Our evaluation shows that learners with no prior programming experiences were able to complete programs roughly two times faster in three out of the four sessions and about 4.6 times less help requests in the first two sessions of the transition phase. Although most learners preferred to have more freedom in a programming editor, our work proves that structured editors can be useful in easing the blocks-to-text transition. To allow more freedom, we envision building semi-structured programming environments with an extended set of immediate warnings to learn from mistakes. Finally, we believe our system and evaluation results will help guide the way to future research on intermediary programming tools to help ease the blocks-to-text transition.

9 SELECTION AND PARTICIPATION OF CHILDREN

Participants were recruited through a local school and an after-school program in a North American city. The study consisted of 26 learners (14 female) ages 12-16 (M=14.1; SD=1.2). Learners were screened to have no prior programming experience other than Scratch. Informed consent was obtained before the first session of the course, including an assent form completed by the child and a consent form completed by the parent/guardian. The assent form explained the details and procedure to the child in a simple language that the child would understand. The child was given the opportunity to ask any questions before confirming their participation. Each student was given a $50 gift card at the end of the class. The

study protocol was approved by our institution's Research Ethics Board.

REFERENCES

[1] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. 2011. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (Onward! 2011), 49–60. https://doi.org/10.1145/2048237.2048245

[2] Amjad Altadmri and Neil C.C. Brown. 2015. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE '15), 522–527. https://doi.org/10.1145/2676723.2677258

[3] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, MichałMoskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E* (SPLASH-E 2019), 7–12. https://doi.org/10.1145/3358711.3361630

[4] M Bannert. 2000. The effects of training wheels and self-learning materials in software training. *Journal of Computer Assisted Learning* 16, 4: 336–346. https://doi.org/10.1046/j.1365-2729.2000.00146.x

[5] A. Cory Bart, J. Tibau, D. Kafura, C. A. Shaffer, and E. Tilevich. 2017. Design and Evaluation of a Block-based Environment with a Data Science Context. *IEEE Transactions on Emerging Topics in Computing* PP, 99: 1–1. https://doi.org/10.1109/TETC.2017.2729585

[6] Austin Cory Bart, Eli Tilevich, Clifford A. Shaffer, and Dennis Kafura. 2015. From interest to usefulness with BlockPy, a block-based, educational environment. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE*, 87–89.

[7] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. 2015. Pencil code: block code for a text world. In *Proceedings of the 14th International Conference on Interaction Design and Children* (IDC '15), 445–448. https://doi.org/10.1145/2771839.2771875

[8] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. 2017. Learnable programming: blocks and beyond. *Communications of the ACM* 60, 6: 72–80. https://doi.org/10.1145/3015455

[9] A. Begel. 1996. LogoBlocks: A graphical programming language for interacting with the world. Electrical Engineering and Computer Science Department. MIT, Cambridge, MA.

[10] John M. Carroll and Caroline Carrithers. 1984. Training wheels in a user interface. *Communications of the ACM* 27, 8: 800–806. https://doi.org/10.1145/358198.358218

[11] Karishma Chadha. 2014. Improving App Inventor through Conversion between Blocks and Text. Wellesley College.

[12] Andy Cockburn, Carl Gutwin, Joey Scarr, and Sylvain Malacria. 2014. Supporting Novice to Expert Transitions in User Interfaces. *ACM Computing Surveys* 47, 2: 31:1-31:36. https://doi.org/10.1145/2659796

[13] S. Cooper, W. Dann, and R. Pausch. 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges* 15, 5: 107–116.

[14] Sayamindu Dasgupta and Benjamin Mako Hill. 2018. How "Wide Walls" Can Increase Engagement: Evidence From a Natural Experiment in Scratch. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, 1–11. https://doi.org/10.1145/3173574.3173935

[15] 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.

[16] Pierre Dragicevic. 2015. HCI Statistics without p-values. Inria.

[17] D Fields, M Giang, and Y Kafai. 2014. Programming in the wild: trends in youth computational participation in the online Scratch community. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, 2–11. https://doi.org/10.1145/2670757.2670768

[18] Leah Findlater and Joanna McGrenere. 2007. Evaluating reduced-functionality interfaces according to feature findability and awareness. In *Proceedings of the 11th IFIP TC 13 international conference on Human-computer interaction* (INTERACT'07), 592–605.

[19] Neil Fraser. 2015. Ten things we've learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, 49–50. https://doi.org/10.1109/BLOCKS.2015.7369000

[20] Shuchi Grover, Roy Pea, and Stephen Cooper. 2015. Designing for deeper learning in a blended computer science course for middle school students. *Computer Science Education* 25, 2: 199–237. https://doi.org/10.1080/08993408.2015.1033142

[21] Philip J. Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education* (SIGCSE '13), 579–584. https://doi.org/10.1145/2445196.2445368

[22] Michael Homer and James Noble. 2014. Combining Tiled and Textual Views of Code. In *IEEE Working Conference on Software Visualisation (VISSOFT)*, 1–10. https://doi.org/10.1109/VISSOFT.2014.11

[23] Andri Ioannidou, Alexander Repenning, and David C. Webb. 2009. AgentCubes: Incremental 3D end-user development. *Journal of Visual Languages & Computing* 20, 4: 236–251.

[24] Sven Jacobs and Steffen Jaschke. 2021. SQheLper: A block-based syntax support for SQL. In *2021 IEEE Global Engineering Education Conference (EDUCON)*, 478–481. https://doi.org/10.1109/EDUCON46332.2021.9453897

[25] M Kölling, N. C. C. Brown, and A Altadmri. 2017. Frame-Based Editing. *Journal of Visual Languages and Sentient Systems* 3: 40–67. https://doi.org/10.18293/VLSS2017

[26] Michael Kölling, Neil CC Brown, and Amjad Altadmri. 2015. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, 29–38.

[27] Ben Lafreniere and Tovi Grossman. 2018. Blocks-to-CAD: A Cross-Application Bridge from Minecraft to 3D Modeling. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (UIST '18), 637–648. https://doi.org/10.1145/3242587.3242602

[28] Colleen M. Lewis. 2010. How programming environment shapes perception, learning and goals: logo vs. scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education* (SIGCSE '10), 346–350. https://doi.org/10.1145/1734263.1734383

[29] Yuhan Lin and David Weintrop. 2021. The landscape of Block-based programming: Characteristics of block-based environments and how they support the transition to text-based programming. *Journal of Computer Languages* 67: 101075. https://doi.org/10.1016/j.cola.2021.101075

[30] J.H. Maloney, K. Peppler, Y. Kafai, M. Resnick, and N. Rusk. 2008. Programming by choice: Urban youth learning programming with Scratch. *ACM SIGCSE Bulletin* 40, 1: 367–371.

[31] Yoshiaki Matsuzawa, Takashi Ohata, Manabu Sugiura, and Sanshiro Sakai. 2015. Language Migration in non-CS Introductory Programming through Mutual Language Translation Environment. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 185–190. https://doi.org/10.1145/2676723.2677230

[32] Philip Miller, John Pane, Glenn Meter, and Scott Vorthmann. 1994. Evolution of Novice Programming Environments: The Structure Editors of Carnegie Mellon University. *Interactive Learning Environments* 4, 2: 140–158. https://doi.org/10.1080/1049482940040202

[33] D. Parsons and P. Haden. 2007. Programming osmosis: Knowledge transfer from imperative to visual programming environments. In *Procedings of The Twentieth Annual NACCQ Conference*, 209–215.

[34] Thomas W. Price, Neil C.C. Brown, Dragan Lipovac, Tiffany Barnes, and Michael Kölling. 2016. Evaluation of a Frame-based Programming Editor. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (ICER '16), 33–42. https://doi.org/10.1145/2960310.2960319

[35] Yizhou Qian and James Lehman. 2017. Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review. *ACM Transactions on Computing Education* 18, 1: 1:1-1:24. https://doi.org/10.1145/3077618

[36] A Repenning. 1993. Agentsheets: a tool for building domain-oriented visual programming environments. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, 142–143. Retrieved October 22, 2014 from http://dl.acm.org/citation.cfm?id=169119

[37] M. Resnick, Brian Silverman, Yasmin Kafai, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, and Jay Silver. 2009. Scratch: Programming for all. *Communications of the ACM* 52, 11: 60.

[38] Mitchel Resnick. 2008. Sowing the Seeds for a More Creative Society. *Learning & Leading with Technology* 35, 4: 18–22.

[39] Joey Scarr, Andy Cockburn, Carl Gutwin, and Philip Quinn. 2011. Dips and ceilings: understanding and supporting transitions to expertise in user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '11), 2741–2750. https://doi.org/10.1145/1978942.1979348

[40] Ben Shneiderman. 2002. Promoting universal usability with multi-layer interface design. *ACM SIGCAPH Computers and the Physically Handicapped*, 73–74: 1–8. https://doi.org/10.1145/960201.957206

[41] Alaaeddin Swidan, Felienne Hermans, and Marileen Smit. 2018. Programming Misconceptions for School Students. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (ICER '18), 151–159. https://doi.org/10.1145/3230977.3230995

[42] Eric Tilley and Jeff Gray. 2017. Dronely: A Visual Block Programming Language for the Control of Drones. In *Proceedings of the SouthEast Conference*, 208–211.

[43] D Weintrop, A Afzal, J Salac, P Francis, B Li, D. C Shepherd, and D Franklin. 2018. Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 366:1–12. https://doi.org/10.1145/3173574.3173940

[44] D Weintrop, A. K Hansen, D. B Harlow, and D Franklin. 2018. Starting from Scratch: Outcomes of Early Computer Science Learning Experiences and Implications for What Comes Next. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, 142–150. https://doi.org/10.1145/3230977.3230988

[45] D Weintrop and N 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* (SIGCSE '17), 633–638. https://doi.org/10.1145/3017680.3017707

[46] D Weintrop and U Wilensky. 2017. Between a Block and a Typeface: Designing and Evaluating Hybrid Programming Environments. In *Proceedings of the 2017 Conference on Interaction Design and Children* (IDC '17), 183–192. https://doi.org/10.1145/3078072.3079715

[47] D Weintrop and U Wilensky. 2019. Transitioning from introductory block-based and text-based environments to professional programming languages in high school computer science classrooms. *Computers & Education* 142: 103646. https://doi.org/10.1016/j.compedu.2019.103646

[48] .David Weintrop and Uri Wilensky. 2015. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th international conference on interaction design and children*, 199–208.

[49] David Weintrop and Uri Wilensky. 2015. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *ICER*, 101–110.

[50] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)* 18, 1: 1–25.

[51] David Wolber, Hal Abelson, Ellen Spertus, and Liz Looney. 2014. *App Inventor 2: Create Your Own Android Apps*. O'Reilly Media, Beijing.

[52] 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: 177–204. https://doi.org/10.1080/08993408.2019.1565233

[53] Grasshopper: the coding app for beginners. *Grasshopper*. Retrieved January 12, 2022 from https://grasshopper.app/

[54] Monaco Editor: a Browser-based Code Editor which Powers Visual Studio Code. Retrieved January 28, 2022 from https://microsoft.github.io/monaco-editor/

[55] Pyodide: a Python distribution for the browser and Node.js based on WebAssembly. Retrieved January 28, 2022 from https://pyodide.org/en/stable/

[56] W3Schools Free Online Web Tutorials. Retrieved January 13, 2022 from https://www.w3schools.com/