Abstract—Blocks programming environments lower barriers to programming by supporting recognition over recall, reducing cognitive load by capturing structural patterns as blocks, and using block shapes to prevent errors in syntax and in static semantics [1]. However, as programmers familiarize themselves with coding, the bulk of visual code blocks can get in the way. Even relatively small blocks programs can be cumbersome to create, modify, navigate, reuse, and share compared to text programs.

To ameliorate such issues, dual-mode or hybrid environments like Tiled Grace [2]–[3], Pencil Code/Droplet [4]–[5], BlockPy [6], and Poliglot [7] support bidirectional conversion between isomorphic blocks and text representations of programs. Studies of dual-mode environments indicate that programmers commonly flip between the two modes, using the blocks as a kind of scaffolding for understanding the corresponding text [3], [8]. Users of dual-mode environments also perceive text more positively than those who transition from a pure blocks system to a pure text one [8]. These results complement studies comparing single-mode use of blocks vs. text (e.g., [9]).

I. INTRODUCTION

For novice programmers, the syntax of textual programming environments is one of the biggest obstacles to learning programming. Blocks programming environments lower the barriers to programming by supporting recognition over recall, reducing cognitive load by capturing structural patterns as blocks, and using block shapes to prevent errors in syntax and in static semantics [1]. However, as programmers familiarize themselves with coding, the bulk of visual code blocks can get in the way. Even relatively small blocks programs can be cumbersome to create, modify, navigate, reuse, and share compared to text programs.

To ameliorate such issues, dual-mode or hybrid environments like Tiled Grace [2]–[3], Pencil Code/Droplet [4]–[5], BlockPy [6], and Poliglot [7] support bidirectional conversion between isomorphic blocks and text representations of programs. Studies of dual-mode environments indicate that programmers commonly flip between the two modes, using the blocks as a kind of scaffolding for understanding the corresponding text [3], [8]. Users of dual-mode environments also perceive text more positively than those who transition from a pure blocks system to a pure text one [8]. These results complement studies comparing single-mode use of blocks vs. text (e.g., [9]).

II. PRINCIPLED DESIGN

Our project is to add a dual-mode feature to MIT App Inventor [10], a Blocks programming environment for making apps for Android mobile devices. We build upon the work of Chadha [11], who developed TAIL, a text language isomorphic to App Inventor's blocks language, and code blocks, visual blocks containing TAIL code that coexisted with and were interconvertible with regular App Inventor blocks. Chadha's system was a proof-of-concept prototype that was never released nor tested in the App Inventor community.

We are developing a text language named Venbrace as a more usable alternative to TAIL, and are planning enhancements to App Inventor in addition to code blocks that will facilitate dual-mode programming. This document summarizes the design principles behind Venbrace, some aspects of the preliminary design, and our plans for user studies to test and improve the design.

Index Terms—MIT App Inventor, blocks programming environments, dual-mode environments, evidence-based programming language design
DP3 Support easy copying/pasting from/to any text system is based on the desire for users to be able to share the textual notations via email, documents, etc. A practical consequence of this principle is that the notation does not employ Python’s indentation-based formatting, because indentation is often lost when copying between text systems. We use explicit braces to avoid this problem, but we also realize this may negatively affect reading and writing text programs, and will seek user feedback on this choice.

DP4 is Maximize flexibility by supporting alternative more concise textual representations. There are numerous applications of DP4 in Fig. 1. For example, 0 abbreviates (0); numZeros abbreviates (get numZeros); $nums abbreviates (get global nums); (list) abbreviates (create empty list); {numZeros <- ...} abbreviates (set numZeros to ...); and {ZerosLabel.Text <- ...} abbreviates (set ZerosLabel.Text to ...). The more verbose forms will also work, but we hypothesize that the more concise forms will aid reading and writing Venbrace code. These concise forms are a key difference between Venbrace and TAIL, which did not support such abbreviations.

App Inventor is used in almost every country on Earth and handles keywords in 15 languages. DP5 Support internationalization says that Venbrace should allow keywords in all handled languages and any valid App Inventor Unicode variable names. (Fig. 1 illustrates the latter but not the former.)

Fig. 1 only hints at DP6 Support bidirectional isomorphism between all aspects of an App Inventor project and text. Whereas TAIL was a textual notation for block assemblies, Venbrace also supports textual notations for (1) the entire workspace of blocks associated with a screen; (2) the Android components belonging to a screen; and (3) all the screens and meta-information associated with a project.

III. PLANNED WORK

Before we implement Venbrace, we will first evaluate various aspects of its design through a preliminary user study. This study will target App Inventor users that have built numerous projects and could benefit from the potentially more efficient textual representation, but don’t necessarily have experience with other textual programming languages. We will ask these users to (1) match Venbrace programs and App Inventor blocks programs using paper-based prototypes, (2) translate programs written in Venbrace to App Inventor blocks, and (3) manually convert App Inventor blocks to Venbrace. Following the tasks, participants will be interviewed for positive and negative views about translation details. Our first user study aims to assess whether the current design for Venbrace is intuitive and ambiguity-free and to improve the current design based on the feedback from the participants.

The next step will be to implement the revised design of Venbrace in App Inventor. We will then conduct a second round of user studies involving this implementation. This time, we will compare the programming process in (1) pure blocks, (2) pure Venbrace, and (3) the mixture of both, again tweaking the design based on the results.

When Venbrace becomes fully functional in App Inventor, we will invite experienced App Inventor programmers to evaluate whether Venbrace helps them to create, modify, and debug App Inventor programs. In our third user study, we will measure the time participants take to read and write blocks vs. and Venbrace code and how often and under what conditions users switch between blocks and their textual representation.
REFERENCES


