A Design for Bidirectional Conversion between Blocks and Text for App Inventor

Ruanqianqian Huang
Department of Computer Science
Wellesley College
Wellesley, MA, USA
rhuang2@wellesley.edu

Franklyn Turbak
Department of Computer Science
Wellesley College
Wellesley, MA, USA
fturbak@wellesley.edu

***Abstract*—Blocks programming environments lower barriers to programming using visual code fragments that prevent syntactic and static semantic errors and reduce cognitive load. However, visual blocks notations can make programs more difficult to read, write, and share. Dual-mode environments that allow bidirectional conversion between isomorphic blocks and text representations can leverage the benefits of both modes. We are designing a dual-mode system for MIT App Inventor that supports textual representations for blocks, workspaces, screens, and whole projects. We present a preliminary design for Venbrace, a fully-braced textual syntax for App Inventor, based on a collection of design principles. Since a key principle is employing evidence-based programming language design, we will follow an iterative process that interleaves user studies with implementation. We describe planned user studies involving individuals of various levels of programming experience that test the usability of Venbrace and explore possible improvements.**

***Index Terms*—MIT App Inventor, blocks programming environments, dual-mode environments, evidence-based programming language design**

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]).

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.

II. Principled Design

Our design of Venbrace is guided by several design principles (DPs) that we developed. Here we discuss the design principles in the context of the concrete example of converting between App Inventor blocks and Venbrace text in Fig. 1.

The most important principle is **DP1 Employ evidence-based language design**. The design exemplified in Fig. 1 is just an initial one in a sequence of designs that will be part of an iterative design process featuring several rounds of user studies (described in the next section). This principle is inspired by the work of Stefik and others [12]–[13].

The principle **DP2 Treat blocks as primary** is motivated by the fact that App Inventor began as a pure blocks system without any associated textual notation. This distinguishes it from Tiled Grace, Droplet, and BlockPy, visual versions of (respectively) the existing textual languages Grace, CoffeeScript, and Python: in these systems, text is primary. A practical consequence of **DP2** is that there should be a simple strategy for converting any App Inventor block into its textual equivalent based on the shape of the block and the text written on it. Our basic strategy is that expression blocks (which have plugs on their left, sockets on their right, and compose horizontally) correspond to text delimited by parentheses; statement blocks (which have notches on the top, nubs on the bottom, and compose vertically) correspond to text delimited by curly

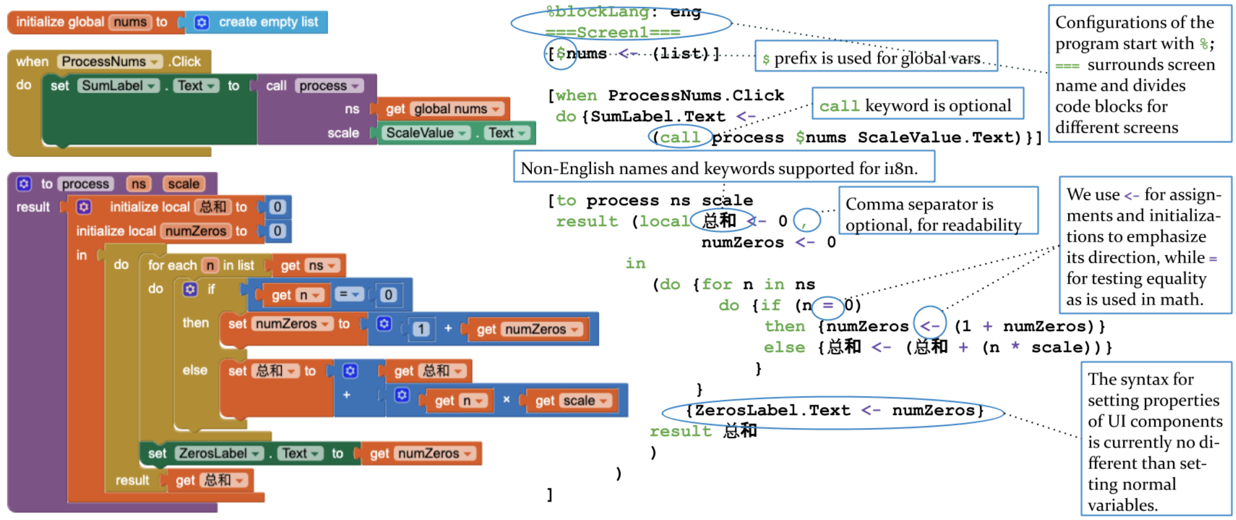

Fig. 1. Example of App Inventor blocks and the corresponding Venbrace textual notation in the current design. Due to space limitations, only some of the blocks in only one screen (`Screen1`) are shown, and declarations for components (like `SumLabel` and `ZeroLabel`) are not shown.

braces; and top-level declarations (event handlers, procedure declarations, and global variable declarations) correspond to text delimited by square brackets. The Venbrace language is thus fully braced (more in **DP3**), with different conventions (parentheses for expressions, braces for statements) chosen for consistency with traditional text languages. As described below, some abbreviations are allowed, but this fully-braced syntax highlights the correspondence between visual blocks and braced textual notation.

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 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

[1] D. Bau, J. Gray, C. Kelleher, J. Sheldon, and F. Turbak, "Learnable programming: Blocks and beyond," *Communications of the ACM*, vol. 60, no. 6, pp. 72–80, Jun. 2017.

[2] M. Homer and J. Noble, "A tile-based editor for a textual programming language," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2013, pp. 1–4.

[3] M. Homer and J. Noble, "Lessons in combining block-based and textual programming," *Journal of Visual Languages and Sentient Systems*, vol. 3, pp. 22–39, 07 2017.

[4] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil Code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*, Jun. 2015, pp. 445–448.

[5] D. A. Bau, "Droplet, a blocks-based editor for text code," *Journal of Computing Sciences in Colleges*, vol. 30, no. 6, pp. 138–144, Jun. 2015.

[6] A. C. Bart, J. Tibau, E. Tilevich, C. A. Shaffer, and D. Kafura, "BlockPy: An open access data-science environment for introductory programmers," *Computer*, vol. 50, no. 5, pp. 18–26, May 2017.

[7] Z. Leber, M. Črepinek, and T. Kosar, "Simultaneous multiple representation editing environment for primary school education," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '19)*, Oct. 2019, pp. 175–180.

[8] J. Blanchard, C. Gardner-McCune, and L. Anthony, "Effects of code representation on student perceptions and attitudes toward programming," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '19)*, Oct 2019, pp. 127–131.

[9] D. Weintrop and U. Wilensky, "Comparing block-based and text-based programming in high school computer science classrooms," *ACM Transactions on Computing Education*, vol. 18, no. 1, pp. 3:1–3:25, Oct. 2017.

[10] MIT App Inventor website. [Online]. Available: http://appinventor.mit.edu/

[11] K. Chadha, "Improving the usability of app inventor through conversion between blocks and text," Bachelor's thesis, Wellesley College, Wellesley, MA, 2014.

[12] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Transactions on Computing Education*, vol. 13, no. 4, pp. 19:1–19:40, Nov. 2013.

[13] A. Stefik and S. Hanenberg, "The programming language wars: Questions and responsibilities for the programming language community," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, 2014, pp. 283–299.