The Design and Implementation of Venbrace, a Text Language for App Inventor

Ruanqianqian "Lisa" Huang

Submitted in Partial Fulfillment of the Prerequisite for Honors in Computer Science under the advisement of Franklyn A. Turbak

May 2020

© 2020 Ruanqianqian Huang
Abstract

Blocks programming environments (BPEs) lower barriers to programming using visual code fragments that prevent syntactic and static semantic errors and reduce cognitive load. App Inventor, for example, provides a BPE that helps programmers of all levels to build mobile device apps easily. However, visual blocks notations can make programs more difficult to read, write, edit, and share. Dual-mode environments that allow bidirectional conversion between isomorphic blocks and text representations can leverage the benefits of both modes.

Having recognized the advantages of allowing textual editing as a plug-in in BPEs, I have designed Venbrace, a fully-braced textual syntax isomorphic to the blocks-based programming language in App Inventor, based on a collection of design principles. I implemented a lexer, a parser, and a syntax editor for Venbrace. To ensure the usability of Venbrace, I also developed, conducted, and did a preliminary analysis of the first round of a user study (with 17 subjects) to evaluate its initial syntax design, determine some design choices, and explore possible improvements.

This project aims to (1) increase App Inventor’s general usability, (2) maximize the usability of Venbrace through user studies, and (3) confirm the benefits of evidence-based programming language design.

Keywords: App Inventor, Blocks Programming, Evidence-Based Programming Language Design, Human-Language Interaction (HLI)
Acknowledgments

First and foremost, I would like to express my sincerest gratitude to my advisor, Professor Lyn Turbak. Despite being on sabbatical for the year, he had graciously agreed to advise me on this thesis and has spent countless meetings and discussions with me to fine tune the project. Had I not taken CS 251 on Principles of Programming Languages with Lyn, I wouldn’t have discovered my academic passion for programming languages. I could not have arrived at this point without his guidance and support for me, my academic career, and my life.

Next, I would like to thank my thesis committee, Professors Eni Mustafaraj, Catherine Delcourt, and Robbie Berg, for taking an interest in this work and putting in the time to provide insightful ideas and helpful feedback. I would also like to thank Dr. Andreas Stefik from the University of Nevada, Las Vagas for answering my questions on his user study on programming language syntax evaluation, which sets the foundation for the user study in this project.

Thanks to Professor Sohie Lee, my major advisor, for all the mentorship since the beginning of my journey in Computer Science. Thank you to everyone at the Computer Science department who has taught me so much, supported my endeavors, and encouraged my academic pursuits.

Thanks to the Jerome A. Schiff fellowship that funded this project.

I especially would like to thank Huihan Li, the best friend, roommate, and partner-in-crime I could ever ask for. I am grateful for having her always by my side during moments of ups and downs throughout my four years at Wellesley. I am also thankful for the 24/7 support from all my friends around the globe when I encountered problems.

Diane A Yi, and the rest of the Boggess family – I cannot thank them more for providing me with a shelter & home-away-from-home in the past four years and especially during the current pandemic of COVID-19. I could not have focused entirely on this project since the beginning of the pandemic had it not been for their kindness.

Lastly and most importantly, I am thankful for my family, for supporting me in my academic pursuits and always having my best interest at heart.

On a concluding note, my final acknowledgement goes to Yuzuru Hanyu, the greatest figure skater of all time. I thank him for being my hero and pushing me to always challenge the unknown.
2.3 Dual-Mode Programming Environments: Bidirectional Isomorphism

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>Existing DMPEs</td>
<td>26</td>
</tr>
</tbody>
</table>

2.4 Ad Hoc uses of Text With Blocks Languages 28

2.5 Translating App Inventor into Text

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.1</td>
<td>App Inventor to Python</td>
<td>29</td>
</tr>
<tr>
<td>2.5.2</td>
<td>App Inventor to TAIL</td>
<td>29</td>
</tr>
</tbody>
</table>

3 Design Principles 32

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Evidence-Based Language Design</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Bidirectional Isomorphism</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Blocks Are Primary</td>
<td>33</td>
</tr>
<tr>
<td>3.4</td>
<td>Concise Alternatives</td>
<td>34</td>
</tr>
<tr>
<td>3.5</td>
<td>Support Copy/Paste</td>
<td>35</td>
</tr>
<tr>
<td>3.6</td>
<td>Internationalization</td>
<td>36</td>
</tr>
</tbody>
</table>

4 Initial Design 37

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>VENBRACE Syntax</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Expressions</td>
<td>38</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Statements</td>
<td>43</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Declarations</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>Translations that are not Obvious</td>
<td>47</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Multi-Word Key Phrases</td>
<td>47</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Labels on Expression Sockets</td>
<td>48</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Labels on Statement Sockets</td>
<td>49</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Event Handlers Parameters</td>
<td>51</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Procedure Names and Procedure Parameter Names</td>
<td>51</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Local Variable Declarations</td>
<td>52</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Global Variable Declarations, Getters, and Setters</td>
<td>53</td>
</tr>
<tr>
<td>4.2.8</td>
<td>Undecided Abbreviations for Setters</td>
<td>54</td>
</tr>
</tbody>
</table>
## 5 Implementing Venbrace

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Overview</td>
<td>55</td>
</tr>
<tr>
<td>5.2 Parser Generator</td>
<td>55</td>
</tr>
<tr>
<td>5.3 Lexing Venbrace</td>
<td>56</td>
</tr>
<tr>
<td>5.4 Parsing Venbrace</td>
<td>57</td>
</tr>
</tbody>
</table>

## 6 A Syntax Editor for Venbrace

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Motivation</td>
<td>59</td>
</tr>
<tr>
<td>6.2 Implementation</td>
<td>61</td>
</tr>
<tr>
<td>6.2.1 Brace Auto-Completion</td>
<td>61</td>
</tr>
<tr>
<td>6.2.2 Brace Matching and Balancing</td>
<td>61</td>
</tr>
<tr>
<td>6.2.3 Keyword and Block Extent Highlighting</td>
<td>62</td>
</tr>
<tr>
<td>6.2.4 Keyword-Based Formatting</td>
<td>62</td>
</tr>
</tbody>
</table>

## 7 Online Venbrace Syntax User Study

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Overview</td>
<td>64</td>
</tr>
<tr>
<td>7.2 Goals</td>
<td>66</td>
</tr>
<tr>
<td>7.3 Methods</td>
<td>67</td>
</tr>
<tr>
<td>7.3.1 Participants</td>
<td>67</td>
</tr>
<tr>
<td>7.3.2 Procedure</td>
<td>68</td>
</tr>
<tr>
<td>7.3.3 Instruments</td>
<td>70</td>
</tr>
<tr>
<td>7.3.4 Data Overview and Analysis</td>
<td>86</td>
</tr>
<tr>
<td>7.3.5 Pilot Study</td>
<td>87</td>
</tr>
<tr>
<td>7.4 Results</td>
<td>87</td>
</tr>
<tr>
<td>7.4.1 Pre-survey</td>
<td>87</td>
</tr>
<tr>
<td>7.4.2 Translation Typing</td>
<td>89</td>
</tr>
<tr>
<td>7.4.3 Preference Ranking</td>
<td>93</td>
</tr>
<tr>
<td>7.4.4 Preference Typing</td>
<td>95</td>
</tr>
<tr>
<td>7.4.5 Post-survey</td>
<td>97</td>
</tr>
<tr>
<td>7.5 Discussion</td>
<td>99</td>
</tr>
<tr>
<td>7.5.1 Evaluation of Current Design</td>
<td>99</td>
</tr>
</tbody>
</table>
8 Conclusion and Future Work

8.1 Current State

8.1.1 VENBRACE Syntax

8.1.2 VENBRACE in App Inventor

8.1.3 Support for VENBRACE

8.2 Immediate Future

8.2.1 An Improved Syntax Design

8.2.2 More Thorough Data Analysis

8.2.3 More User Studies on VENBRACE Syntax

8.3 Near Future

8.3.1 VENBRACE Support for All App Inventor Components

8.3.2 Handling Known Ambiguities Better

8.3.3 VENBRACE in App Inventor

8.3.4 Automatically Updating the VENBRACE Lexer and Parser

8.3.5 Support of Internationalization

8.3.6 Improving the Syntax Editor

8.3.7 Simple Error Reporting

8.3.8 Importing/Exporting App Inventor as VENBRACE

8.3.9 Testing and User Studies on VENBRACE-in-App Inventor

8.4 Far Future

8.4.1 Preserving All Editing Features of App Inventor in Text Mode

8.4.2 More User Studies on Using App Inventor Incorporated with VENBRACE

8.4.3 Error Reporting

A Round 1 Study Materials

A.1 Pre-study Survey (Activity 01)
A.2 Tutorial/Task Section 1 (Activity 02) ........................................... 123
A.3 Tutorial/Task Section 2 (Activities 03-05) ................................. 123
A.4 Tutorial/Task Section 3 (Activities 06-13) ................................. 125
A.5 Tutorial/Task Section 4 (Activities 14-18) ................................. 128
A.6 Tutorial/Task Section 5 (Activities 19–22) ............................... 130
A.7 Tutorial/Task Section 6 (Activities 23–27) ............................... 132
A.8 Tutorial/Task Section 7 (Activities 28–30) ............................... 133
A.9 Tutorial/Task Section 8 (Activities 31–36) ............................... 135
A.10 Tutorial/Task Section 9 (Activities 37–41) ............................. 137
A.11 Post-study Survey (Activity 42) .............................................. 140

B Round 1 Study Data ................................................................. 145
  B.1 Translation Typing .............................................................. 145
  B.2 Preference Ranking ........................................................... 155
  B.3 Preference Typing .............................................................. 170

C Venbrace.g4 ................................................................. 184

D Implementation of TAM in JavaScript ........................................ 191
1 Introduction

Computer programming can have a steep learning curve. Learning to use a programming language, for example, is often one of the biggest obstacles faced by many novice programmers. For people who are new to programming, learning to interact with a new (and their first) programming language to convey ideas and express creativity can be hard, especially when the programming language is in no way similar to a human language. Even programmers with more experience may still suffer from learning to use a programming language whose syntax and semantics are new to them.

To address this problem, blocks programming environments (BPEs) (e.g. MIT App Inventor [MIT], Scratch [Scr], Snap! [Sna], Blockly [Blo]) have been developed to ameliorate such issues. Implemented with visual syntactic fragments instead of plain text to construct computer code, BPEs lower the barriers to learning and using programming languages by supporting recognition over recall, reducing cognitive load by capturing structural patterns as blocks, and using block shapes to prevent common errors [BGK+17].

However, many novices believe that BPEs are more of a pedagogic tool for kids due to their visual-based nature, and that BPEs are less useful for their long-term plans with programming. As a result, they find it inauthentic for learning “real” programming [WW15, WW17]. Even when they choose to learn coding with blocks programming, as they familiarize themselves with programming concepts, they might start to feel that the visual-based nature of blocks programming compromises their coding efficiency. For example, the bulk of visual code blocks can get in the way as programs grow with size and complexity. Unlike in traditional (text) programming environments, where one can create, modify, and share programs with a few cursor/mouse moves and strokes on the keyboard, it is hard to reproduce the same procedure in blocks programming environments due to high viscosity [GP96] – the difficulty of making local changes. Programming involves not only computer code
generation, but also changes and revisions. In BPEs, however, even relatively small blocks programs can be cumbersome to create, modify, navigate, reuse, and share compared to text programs.

1.1 MIT App Inventor

MIT App Inventor is a browser-based BPE for designing and building mobile apps. One of the world’s most popular BPEs, it has more than 10 million registered users across 195 countries and regions (as of April 2020). It introduced a new era of mobile computing by allowing people, regardless of their experience in programming, to design and create mobile apps \[PV13, WAF15\], thus allowing novice programmers to build fully functional mobile apps even with little programming experience. Its initial version, App Inventor Classic, was previously maintained by Google and then migrated to MIT CSAIL. The MIT App Inventor team later redesigned the BPE’s architecture to its current version, MIT App Inventor\(^1\), by adopting Google’s Blockly library in its blocks editor (shown in Figure 1-1).

\[Insert Figure 1-1: App Inventor’s blocks editor.\]

\(^1\)In the rest of the document, "App Inventor" refers to the current version of MIT App Inventor.
A mobile app can involve multiple screens of content. In App Inventor, the user first indicates which components are part of a screen (in the Designer), and then they specify the behavior of these components in the blocks editor. The way they do that is by selecting blocks out of menus and connecting them to form programs. Figure 1-2 presents a cluster of blocks that illustrates how the content and text color of a text box (component) is changed (behavior) when a click event is triggered on a button (component).

![Figure 1-2: A cluster of blocks that handles a click event on a button named Button1.](image)

### 1.2 Challenges Facing App Inventor Users

This section summarizes barriers faced by App Inventor programmers. Some of the challenges are already eased by existing mechanisms, which will be introduced later in Section 1.3. This thesis project proposes an even better solution, a textual representation for App Inventor, that further lowers these barriers. For each of the challenges facing current App Inventor users, I will briefly explain why such a challenge would be less problematic if a textual representation existed in App Inventor, before further introducing the textual representation in Section 1.4.

#### 1.2.1 Code Reading and Navigation

App Inventor supports many components that mobile app designers want in simple apps by providing a full library of blocks that are related to specifying behavior for the components. However, even a small mobile app with basic functionality can involve a sequence of procedure and variable definitions and event handlers and thus numerous blocks, let alone an app of a bigger size. The large number of blocks soon fill up the blocks editor, and it is often impossible to examine all blocks at once without scrolling.
through the blocks editor. Figure 1-3 shows the large number of blocks involved in a simple app that tells whether a campus event provides food. These difficulties caused by the numerous clusters of blocks make it hard for App Inventor programmers to read, navigate, and search for items in App Inventor programs.

![blocks editor screenshot]

Figure 1-3: Even a simple app could involve numerous clusters of blocks that are hard to navigate in App Inventor.

If there existed a text representation of an App Inventor project, code reading and navigation would become much less labor-intensive, especially with the help of keyboard shortcuts and syntax highlighting for specific keywords.

### 1.2.2 Viscosity

The **viscosity** of a programming language describes the amount of effort the user has to put in making a change [GP96]. Like in many other BPEs, viscosity exists in App Inventor. Users may encounter such an issue in many aspects of App Inventor, including adding new blocks, making changes to the arguments of an existing block, moving the position of a specific block among a sequence of blocks, and removing an existing block. Figure 1-4 presents the complex process of changing a conditional from \( x > 15 \) to \((x+1) > 15\), which can be made through few keystrokes within seconds if the same conditional is written in a traditional text programming language. In this scenario, if there existed a *text programming mode* in App Inventor that is based on
text representations for App Inventor blocks, making the change in a text representation for corresponding App Inventor blocks would be as simple as in a traditional text language.

1. The original cluster of blocks
2. Detach "get x"

3. Place a new addition block
4. Drag the getter block inside the addition block

5. Create a new number block
6. Drag the number block inside the addition block

Figure 1-4: Making a small change can take many steps in App Inventor.

1.2.3 Sharing A Whole Project, or Just A Part of It

For code scripts written in textual programming languages, sharing a script with collaborators and interested programmers can be as simple as sending an email whose body is a copy of the textual content of the script. Even for projects composed of several scripts and other components, it is also straightforward to compress a project to a zip file and share it through file transfer. The sender and receiver of a shared text-based project can often open, read, and edit the project using different operating systems and integrated development editors (IDEs).

Sharing App Inventor projects requires a lot more effort. To share an App Inventor project under the current version of App Inventor, the sender has to first export the project to a .aia file on their computer, and then share the .aia file through email or other means of file transfer. To work on an shared project, the receiver has to first
go to App Inventor’s IDE, then import the received .aia file from their computer, and eventually start working on the project within App Inventor’s IDE. A complete workflow of sharing an App Inventor project and working on an shared project is illustrated in Figures 1-5 and 1-6.

Compared to the process of project sharing for projects written in text-based programming languages, it requires both the sender and the receiver to open, read, and edit an App Inventor project in the App Inventor IDE. Exporting an App Inventor project as an .aia file only allows sharing an entire project but not part of it. To share a part of a project, the user needs to seek for an alternative way, such as PNG exports (introduced in Section 1.3.3). Still, there is no one mechanism in App Inventor that allows sharing both whole projects and clusters of blocks within a project, and the overall user experience with App Inventor in terms of project sharing could be enhanced if there existed one.

In a text programming language, it is always possible to share snippets of code within a script, several (but not all) scripts, or even all of the scripts within a project written in that language. A text representation for App Inventor would thus inherit the benefits of a text language and introduce easier ways of project sharing in App Inventor.

1.3 Existing Mechanisms to Solve These Challenges

1.3.1 Block Operations for Better Navigation

The block condensing feature in App Inventor allows users to condense the visual volume of a cluster of blocks to render more space in the blocks editor. It can be done by right clicking and selecting the Collapse Blocks option. To view the complete form of the cluster, simply right click again and select the Expand Blocks option. Figure 1-7 illustrates such a process. Note that you cannot see the details within a collapsed block, so while collapsing reduces clutter, it impedes searching.

Mutable blocks in App Inventor refer to blocks whose number of inputs can vary,
Figure 1-5: Export an App Inventor project and share it via email.
Figure 1-6: Import an App Inventor project from a local .aia file in App Inventor’s IDE.
Figure 1-7: Condensing and expanding blocks in App Inventor.
such as the addition block, the multiplication block, and the local variable declaration expression and statement blocks. Users have the option of viewing the sequence of inputs of an App Inventor expression block inline (all in one line, horizontally) or external (vertically), especially in mutable blocks that could involve a long sequence of inputs. For users who would like more vertical space in their blocks editors, the *inline inputs* function would be helpful and could be done by right clicking and selecting the *Inline Inputs* option. To view the inputs externally, they can select the *External Inputs* option after the right click. Figure 1-8 demonstrates using this feature.

Figure 1-8: Viewing inputs of mutable blocks inline or externally in App Inventor.
1.3.2 Support for Faster Block Creation

Drop-down Menus

Drop-down menus in App Inventor introduced an easier way for naming [TWM14], allowing users to change the variable name or the component property which a block uses efficiently. A drop-down menu list only names that are valid in the current scope, and further prevents users from referencing variables or components outside of the scope (which will cause error). Figure 1-9 shows an example of a drop-down menu in an event handler.

Figure 1-9: Add a block by typing in its name rather than looking up the block and dragging it out from the blocks navigation bar to the left.

Keyboard Support

Although users rely a lot on mouse movements to interact with BPEs due to their visual nature, App Inventor provides several keyboard support for a better user experience. Typeblocking, for example, introduced a faster way to add a block to the blocks editor once the user knows the name of the block to be added. They can switch to the Typeblocking mode by clicking on an blank area in the blocks editor and typing the name of the wanted block, and they will be presented with a list of blocks with the matching titles to the typed input. Figure 1-10 presents working with Typeblocking.

There are more basic but essential keyboard support for blocks editing as well. Deletions can be done by clicking on the unwanted block and hitting the delete or backspace key. For blocks to be reused, copying, pasting, and cutting are similarly efficient through the use of the copy/paste/cut shortcuts on the keyboard.
1.3.3 PNG Exports for Partial Project Sharing

App Inventor now allows sharing clusters of blocks in a project by converting the clusters into .png files for further distribution. To export a cluster as a .png file, right click the cluster and select the "Download Blocks as PNG" option, and App Inventor-related information of a cluster will be stored in the downloaded .png file. To reuse the .png file in the same or a different blocks editor, drag the file from the local directory where it is stored into the blocks editor in the browser, and a same cluster as illustrated by the .png file will be replicated.

1.4 Venbrace: An Even Better Solution

The few features discussed in the previous section hardly address at all the challenges for creating, modifying, navigating, searching, and sharing App Inventor programs, which I mentioned in Section 1.2. It is still a complex process to make a small change as illustrated in Figure 1-4 in the current version of App Inventor despite affordances like drop-down menus and Typeblocking. Users still rely on a browser to access the App Inventor blocks editor to import/export and work on a shared project. A text-based language for App Inventor can go a long way to addressing many of the problems I have discussed so far.
Here I propose **Venbrace**, a textual language for App Inventor. Venbrace allows for text-based editing of App Inventor programs, and aims to support code aggregation of a multi-screen project into one single file and editing App Inventor programs outside of App Inventor’s browser-based IDE in the future.

Note that in 2014, Chadha designed and implemented TAIL [Cha14], which is also a text representation for App Inventor. Details of TAIL are reviewed in Chapter 2, Section 2.5.2. Although driven by similar motivations, which I stated above, and sharing some (but not all) design principles (more in Chapter 3), Venbrace and TAIL are different in their designs for syntax (described and compared in Chapter 4). Venbrace also follows an iterative design process (more in Chapters 3, 4, and 7), while most design decisions for TAIL were arbitrarily made by the designer. In addition, I implemented a syntax editor for Venbrace, while similar support was absent in the development of TAIL.

### 1.4.1 Venbrace: A Text-based Language for App Inventor Programs

Venbrace is a textual language designed to be *isomorphic* to the blocks language used in App Inventor. In other words, any App Inventor blocks, clusters of blocks, workspaces, screens, and whole projects will eventually be represented in pure text with all original features kept. Such isomorphism will enable bidirectional translation between any App Inventor entity and its Venbrace representation, making it possible to edit any App Inventor program in Venbrace without losing its functionality. Due to the text-based nature of Venbrace, it will be possible for a multi-screen App Inventor project to be condensed into one Venbrace file when translated into text, and the Venbrace translation of any App Inventor project can be edited in any text editor outside of App Inventor’s browser-based IDE, from a plain text editor to a highly-implemented IDE. Figure 1-11 illustrates how App Inventor blocks are translated into Venbrace based on a tentative design for Venbrace published in October, 2019 [HT19].
Figure 1-11: Example of App Inventor blocks and the corresponding VENBRACE textual notation in the current design.

Note that the current design for VENBRACE has some differences from this earlier, tentative design. For more details on the current design of VENBRACE, including design choices not described here, see Chapter 4.

1.4.2 The Design of VENBRACE

VENBRACE is a text language designed in a principled way, following six design principles:

- **DP1**: Employ evidence-based language design.

- **DP2**: Support bidirectional isomorphism between all aspects of an App Inventor project and text.

- **DP3**: Treat blocks as primary.

- **DP4**: Maximize flexibility by supporting alternative more concise textual representations.

- **DP5**: Support copying/pasting from/to any text system.

- **DP6**: Support internationalization.
Details of these design principles are described in Chapter 3.

The name of VENBRACE indicates its essence, a fully-braced textual language for App Inventor. Three types of braces, the parentheses ( ), the curly braces { }, and the squared brackets [ ], are used to represent App Inventor expression blocks, statement blocks, and top-level blocks, respectively. The decision to adopt braces instead of whitespace or indentations to indicate the boundary of App Inventor blocks in VENBRACE is the result of two design principles, DP2 and DP5 – to employ bidirectional isomorphism between blocks and text, and to support VENBRACE’s easy copying/pasting from/to any text system. This design principle ensures that the potential loss of whitespace and indentations during copying and pasting will not affect the parsibility of a VENBRACE program. Following several other principles defined prior to the initial design, the design and implementation of VENBRACE is principled. Chapter 3 details the formation and reasons behind each principle.

The design and implementation of VENBRACE is also empirical by integrating results from at least one user study to iteratively improve the design. During an iterative programming language design process, (1) the designer proposes a design of the language, (2) the design is evaluated by end-users of the language through rounds of user studies, and (3) the designer implements the next design iteration by improving the current design based on user feedback from the user studies. Chapter 7 describes the first user study for evaluating the keyword and symbol choices, understanding users’ preferences on handling App Inventor-specific helper words that are not often seen in traditional textual programming languages, and seeking for signs of unfavorable design decisions that trigger potential syntactic errors.

The driving force behind the design of VENBRACE is the need for a user-preference-informed block editing tool that increases the general usability of App Inventor. Integrating VENBRACE in App Inventor will take advantage of both textual programming and blocks programming. VENBRACE will not interfere with the existing methods for block editing (mouse-based creation, modification, deletion, and arrangement, and keyboard support) and import/export (.aia files for full-project import/export and .png files for block(s) import/export). Instead, it will serve as an add-on to the
platform.

Because App Inventor precedes VENBRACE, I believe it is a good idea to design VENBRACE in a principled way on the basis of the current version of App Inventor. To maintain its isomorphism to App Inventor, VENBRACE has to support all features that App Inventor supports, and VENBRACE cannot introduce new features that App Inventor is unable to handle. To increase the general usability of App Inventor, the functionality of VENBRACE should not diminish the current usability of App Inventor, and addition of VENBRACE to App Inventor is not meant to further complicate App Inventor.

Because VENBRACE is designed for an increased usability of App Inventor, feedback from the potential end-users, i.e., actual users of App Inventor, will matter. VENBRACE is thus designed and constantly improved empirically to ensure all the design choices are understood and favored by the end-users. The evidence-based design approach further guarantees that VENBRACE replicates expected App Inventor behaviors correctly and follows the declared design principles.

As a result, development for VENBRACE happened in 4 stages: (1) Designing the syntax, including choosing keywords and symbols; (2) Implementing the language; (3) Developing its front-end interaction, i.e., a syntax editor for the language; and (4) Conducting an online user study to evaluate the language and building related technology to the study. Each of the stages, their technological details, and challenges will be presented in Chapters 4 - 7.

1.5 Contributions

In this thesis, I made the following contributions:

- I developed six design principles that guide the development of the VENBRACE language.

- I designed the first iteration of VENBRACE, a textual language isomorphic to the blocks language used in App Inventor. It bears similarity to TAIL [Cha14].
in many respects, but there are some key differences in terms of symbol choices, handling keywords, and, in particular, the incorporation of user feedback during language development.

• I implemented a lexer and a parser for **Venbrace** using ANTLR 4 [Par] that handles most of App Inventor’s blocks for a single screen.

• I implemented a syntax editor for **Venbrace** that supports keyword highlighting and brace matching using CodeMirror [Cod]. This is a JavaScript-based editor that is embeddable in web browsers, and will replace the simple text areas used in Chadha’s code blocks system for TAIL [Cha14].

• Prof. Franklyn Turbak (the advisor of this thesis project) and I jointly designed and implemented an online user study that evaluates the initial syntax design of **Venbrace**, determines critical design decisions, and obtains feedback from the end-users based on their interaction with the language.

### 1.6 Road Map

The rest of the document is organized as follows:

• Chapter 2 reviews past related work on dual-mode programming environments, the Textual App Inventor Language (TAIL), and user studies on programming language design and implementation.

• Chapter 3 describes the principles that were followed before the initial design of **Venbrace**.

• Chapter 4 shows the initial design of **Venbrace** following the principles described in Chapter 3, including its choice of symbols and keywords.

• Chapter 5 details the development of **Venbrace** and technical details of the lexing and parsing processes.
• Chapter 6 presents an overview of the implementation of a syntax editor designed specifically for VENBRACE to be used in the later user studies.

• Chapter 7 discusses an online user study that evaluates the syntax of the initial design of VENBRACE in detail, including its goals, design, related technology, analysis techniques, results, and implications.

• Chapter 8 concludes with a description of the current state of the project, ideas that are yet to be finished and potential work that could be done, and suggests future work to improve embedding VENBRACE into App Inventor.
2 Related Work

2.1 Empirical Studies of Programming Languages

2.1.1 Background and Motivation

Stefik and Hanenberg are among the first few who point out a long-lasting problematic phenomenon in the programming language community: programming language inventors have created language products "for a variety of reasons" [SH14], researchers and engineers have claimed and believed to have made language improvements that are beneficial; however, few of them have provided a formal proof of the proper functioning of their product, let alone engaging the actual users of the tools they have created and enhanced during the process. The development of programming languages, like other software technologies or mathematical theorems that follow the scientific disciplines, also demands evidence [SH14]. They concluded that language design "needs a stronger foundation of evidence," [SH14] and that empirical methods should play an important role in programming language design and implementation. Stefik’s proposal further inspired Kaijanaho’s thesis work on evidence-based programming language design [Kai15].

In 2016, Myers et al. called for attention on appropriate methods for programming language evaluation and on empirical evidence-guided programming language design [MSH+16]. Myers, LaToza, and Burnett further proposed that human-computer interaction (HCI) methods should be applied to research on programming languages and tools, because they are the user interfaces through which programmers interact with computers [MKLY16]. A strong emphasis is especially posed on designing programming tools through an iterative design process using human-centered methods. Myers et al. [MKLY16] then introduced the HCI methods they have used over the last 30 years to improve the usefulness and usability of such tools including the design
of programming languages.

Among the HCI methods they introduced, Exploratory Lab Studies and Surveys are most often adopted in studies that evaluate programming languages and tools involving human participants. Natural-Programming Elicitation is also involved in my design for the Preference tasks in the Venbrace syntax evaluation study, which will be discussed in detail in Chapter [7]. The next two sections review prior studies evaluating the designs of programming language syntax and the use of visual programming tools.

2.1.2 Investigation into Programming Language Syntax

Stefik and Siebert conducted four user studies investigating into programming language syntax, not only seeking insights for improving Quorum, an empirically-designed textual programming language, but also examining the syntax of textual languages that are commonly seen. The first two studies were based on surveys to understand participants’ preferences for programming language keyword and symbol choices. Surprisingly, some of the keywords and symbols widely used in mainstream textual languages were not perceived as intuitive by the participants as they should have been [SS13].

The results from Studies 1 and 2 implied potential improvements to keywords and symbols of Quorum, leading to part of Studies 3 and 4. These two studies fell under the Exploratory Lab Studies category as they had pre-defined tasks involving the language(s) to be evaluated, and they were conducted in a lab setting. For each pre-defined tasks that has only one correct answer, researchers adopted the Token Accuracy Map (TAM) algorithm to understand whether participants could produce an expected Quorum keyword at the right position, and to derive the intuitiveness for each keyword based on the overall accuracy of all responses. Based on the results, the researchers were able to determine unnecessary keywords as well as keywords that can be improved, and consequently modified the syntax for a later version of Quorum.
2.1.3 Studies Comparing Blocks and Text Languages

As Homer and Noble were developing Tiled Grace, they conducted a tutorial-guided user experiment to evaluate their design at the time [HN14]. The experiment, also an Exploratory Lab Study, involved primarily undergraduate students with some experience in computer science at their home institute. The researchers recorded the participants' interaction with Tiled Grace, their interpretation of the system, and their general feedback on user experience, hoping to examine the usability of the switch views (between blocks and text), the usefulness of the error reporting and handling mechanisms, user satisfaction with the system, and reveal unanticipated problems or successes. The researchers found positive evidence about their current design based on all measurements. They were also able to identify possible improvements on user interaction in Tiled Grace based on user feedback, potentially migrating the "drag-and-drop" mechanism at that time to a different interaction mechanism, such as a point-and-click arrangement that is "less error-prone" [HN14]. In general, they successfully evaluated some aspects of usability of Tiled Grace through an exploratory lab experiment.

Weintrop and Wilensky [WW15, WW17] explored the effects of using blocks and text in introductory computer programming classes on novices through in-lab experiments. To understand the perception of blocks- and text-based programming among high school students (often novice programmers), they conducted a study comparing participants who used blocks-only, hybrid-read-only, and hybrid-read/write environments in a high school introductory programming course [WW15]. They found that participants considered blocks easier to read, navigate, compose and debug, and more useful in recognizing programming structures and concepts. However, the participants still perceived blocks programming as less powerful, efficient, and authentic than text programming. The researchers further investigated whether the modality of programming in high school introductory computer science class impacted students learning outcomes, attitudes, and future interest in the field [WW17] by assessing students using blocks- and text-only modes, and received more positive responses from students.
using blocks. For novice programmers, as both of the studies revealed, the use of blocks in learning computer programming would lead to more benefits than the use of text.

## 2.2 Hybrid Systems: Improving BPEs by Incorporating Text

Hybrid systems are programming environments that are somewhere on the spectrum between pure BPEs and pure text-based environments. Some have evolved as BPEs that incorporate aspects of text editing to make editing more efficient, while the others have evolved by adding blocks-based elements to text editors.

### 2.2.1 Using Text in GP

Mönig et al. [MOM15] noticed the issues with code navigation and viscosity in BPEs as they were developing GP. To tackle these issues, they conducted three experiments involving text editing in GP.

In the first experiment, GP blocks were converted directly to text with pre-defined syntax. The direct conversion allowed for more space in the editor and faster constructions of input blocks than its blocks counterpart, but it introduced potential text editing-specific errors including invalid syntax, spelling, and input values. It also required users to acquire the pre-defined syntax of the text representation, and further disabled internationalization, which Scratch blocks originally supports.

The second experiment translated GP blocks to textual contents that retain all original features of the blocks, including the wording, the drop-down menus, and the option of hiding inputs, while users were able to directly interact with the text through their keyboard. Compared to the first experiment, this experiment solved the screen space problem to a less extent due to the presence of more helper words, but it prevented potential typing errors more effectively, and users did not have to learn a new syntax rule to use the mechanism.
The third experiment introduced a mechanism very similar to App Inventor’s Typeblocking (see Section 1.3.2), such that users were able to insert a block by typing out the name of the block using the keyboard. It did not solve the screen space issue, but it completely prevented typing errors (invalid inputs would not be mapped to any valid blocks), and also support internationalization well.

Reflecting on the three experiments, the researchers concluded that direct editing to the blocks using the keyboard (i.e., Typeblocking in App Inventor), rather than completely introducing text-based programming into BPEs, might be the most optimal solution to the two problems to date. However, the proposed solution still could not increase the efficiency of blocks reproduction (see Section 1.2.3), which is an issue often faced by users of BPEs.

2.2.2 Frame-Based Editing

Kölling et al., on the other hand, explored hybrid programming from the other direction by developing frame-based editing [KBA17]. Frame-based editing is extended from text programming, because the researchers recognized the strong preference of text-based systems to blocks-based systems among proficient programmers. Still, it incorporates the benefits of blocks-based editing and combines the advantages of both programming modes indeed. As a result, frame-based editing improves program representation (benefit of text) and error avoidance (benefit of blocks) for beginners and can speed up program manipulation (benefit of applying text to blocks) for experts. In addition, an implementation of such a system for a new Java-like language, Stride, can also serve as an ideal transition from blocks programming to text programming in an educational context.

Although not a BPE, frame-based editing is one of the most important of the hybrid systems because (1) it is the basis of the popular Greenfoot environment [1] for introductory programming in Java, and (2) it is aimed not only at improving the program editing experience of novices but of experts as well.

[1]https://www.greenfoot.org/door
2.2.3 Typeblocking

As is introduced in Section 1.3.2, Typeblocking in App Inventor helps with easier and faster blocks creation. The idea of Typeblocking originates from Corey McCaffrey’s masters thesis work, in which Typeblocking was first introduced to StarLogo TNG, an early BPE [McC06]. With Typeblocking, users can construct block code through the use of a keyboard in a blocks programming environment. In 2019, Terrence Liang extended Typeblocking to StarLogo Nova, a successor of StarLogo TNG, and proved its improvement to the programming experience in BPEs like StarLogo Nova [Lia19].

Daniel Wendel [Wen19] believed that there should be room for improvement to the link between keyboard and the underlying abstract syntax tree (AST) of blocks-based programs, which would eventually aid blocks creation with higher efficiency. In his position paper, he further discussed Liang’s initial evaluation of the use of Typeblocking in StarLogo Nova to revealed that Typeblocking can create as efficient keyboard-based editing experience in blocks-based programs as editing in plain text (i.e., text-based programming). Compared to plain text editing, Typeblocking can also reduce the number of keystrokes required, and is less error-prone. Wendel proposed the inclusion of Typeblocking-like features in BPEs, because they could (1) continue to encourage the use of AST representations for blocks programs, and (2) enable editing experiences with efficiency mirroring that of typing.

2.3 Dual-Mode Programming Environments: Bidiirectional Isomorphism between Blocks and Text

Dual-mode programming environments (DMPEs) are BPEs in which traditional (text) programming is simultaneously available. Text programming in DMPEs are examined to determine what efforts have been made in applying text programming in BPEs. Past studies that evaluated the effects of using traditional programming languages, BPEs, and DMPEs in educational settings concluded that DMPEs has more benefits for novice programmers, who happen to be the major users of App
Bau et al. designed Pencil Code to explore the possibility of better preparing novice programmers for traditional (text) programming with the help of blocks programming [BBDP15]. The tool had been used and tested in classrooms ranging from elementary to undergraduate. The researchers found that students frequently switched between blocks and text when using Pencil Code, and they tended to use blocks more often towards the beginning of the course while mostly text later on, although they were never required to use the text. Most of the subjects reported equally positive experience of both using blocks and text. The authors concluded that DMPEs like Pencil Code could bridging the gap between blocks- and text-based programming and helping novices transition from introductory blocks programming to further study in computer programming, where traditional (text) programming is predominant.

Homer and Nobel reached a similar conclusion as they were evaluating Tiled Grace, a DMPE they were developing, through a user experiment [HN14] that measured user engagement with the blocks (tiles) and their corresponding text. They found that the participants, especially those with less programming experience, appreciated having both the option of viewing the code in text while editing in blocks and the ability to get a visual "overview" of the program when editing in text. Their later study in 2017 also confirmed the same conclusion, and even indicated that users frequently switch between blocks and text back and forth when they programmed in Tiled Grace [HN17].

The study by Blanchard et al. [BGMA19] also investigated how DMPEs affect novices transitioning from blocks to text. The study was conducted in a public high school with six classes (n=129) using Python (text programming language) and a Python variant for Pencil Code (BPE with hybrid option) [BBDP15] developed by the researchers. Participants were all students of an introductory programming class, and they were assigned to three course tracks: text-only (as control group), in which only Python was taught; blocks-to-text, in which they first learned to program in the Python-version Pencil Code, with the hybrid option turned off, and then in Python;
and blocks-to-hybrid-to-text, in which students programmed in the Python-version Pencil Code, first without the hybrid option and then with the hybrid option, and later in Python. They found that participants in the blocks-to-hybrid-to-text track, although perceived text as more frustrating than blocks, considered text as easier and more favorable than those in the blocks-to-text track. Their findings indicated that hybrid environments help novices transition from blocks to text, given that text will be more widely used in their study in programming beyond, while the blocks help them recognize and understand the errors made in text.

2.3.1 Existing DMPEs

Tiled Grace

Tiled Grace [HN14] is the first DMPE. It is a BPE that visualizes the textual programming language Grace, in which a "Tile" is the block editor component. Tiled Grace allows for bidirectional switch between the textual Grace code and its corresponding tiles with error reporting mechanisms. Targeted towards novice programmers, Tiled Grace is intended to relieve syntactic burden of programming like all the other existing BPEs, while concurrently introducing the concepts of a textual language for an easier transition into traditional (text) programming.

Pencil Code

Bau et al. [BBDP15] created Pencil Code, a BPE designed specifically for novice programmers, aiming at building their confidence in programming early on. It uses Droplet as its block editor component, which is the exact visual representation of CoffeeScript, an existing text programming language. Consequently, any legal CoffeeScript program could be converted to a visual Pencil Code view and back. Pencil Code further allows writing and converting textual JavaScript code to Droplet blocks and back. However, direct translation between blocks representation of CoffeeScript/JavaScript and the actual JavaScript/CoffeeScript code is not allowed.
**BlockEditor**

Built by Matsuzawa et al. [MOSS15], BlockEditor is an editor for a visual language called Block that can save the program to Java. The textual Java code can be edited and further reimported into BlockEditor in the form of Block. In this way, BlockEditor allows viewing/editing a program in both the blocks mode and the text mode to some extent. However, BlockEditor is not a true DMPE: although semantically equivalent, the Block visual language is not syntactically the same as the Java textual language.

**BlockPy**

BlockPy is a DMPE for Python that intends to support introductory programming for novices in a data-science context [BTT+17, BTK+20]. It supports bidirectional, seamless transitions between block and text programming: users can switch between the block and the text views, and any change they make in either view will be reflected in the other view simultaneously. Note that BlockPy uses Blockly [Blo] for its block language interface, and the Blockly blocks preserve keywords and notations that are nowhere to be found in Python syntax. While BlockPy is in no doubt a DMPE, its lack of isomorphism between its blocks and text receives some criticisms.

**Poliglot**

Leber et al. [LCK19] created Poliglot, a DMPE that aims to bridge the gap between blocks- and text-based programming and, for novice programmers who learned programming through blocks, smooth the transition into text-based programming. The development of Poliglot was still in its initial stage. Still, through a preliminary user study on primary school students, the researchers evaluated the DMPE and found that users gradually shifted from frequently editing in block to mostly editing in text-based notations, confirming the benefits of helping novices transition from blocks to text programming provided by the DMPEs.
CodeMirror Blocks

In 2019, Schanzer et al. [SBK19] presented CodeMirror Blocks (CMB), a toolkit for building fully-accessible, browser-based programming environments for multiple languages. CMB is a special contribution to the DMPE area, because it extends the accessibility of blocks programming to visually-impaired programmers by generating a block editor familiar to sighted users while communicating the structure of a program with visually-impaired users through audio assistance. For navigation, it also provides accessible keyboard shortcuts for such users.

2.4 Ad Hoc uses of Text With Blocks Languages

Outside of DMPEs, text is also used in BPEs that are not dual-mode environments.

- PicoBlocks² is a blocks programming language for programming the PicoCricket. It has a text mode with a Logo-like language for writing code that cannot be expressed in blocks and defining new blocks in the blocks language, although there is no automated translation between the text and blocks languages.

- Scratch 3.0 [Scr] has a way of expressing extensions in JavaScript³.

- The micro:bit PXT editor⁴ allows expressing micro:bit programs in both blocks and JavaScript in a DMPE. Because some JavaScript constructs cannot be represented by blocks, there are special “JavaScript” blocks⁵ in the blocks mode to keep track of the “otherwise untranslatable” JavaScript code.

- The Blockly Playground⁶ is a popular Block site for creating simple Blockly programs and seeing how they can be translated into languages like JavaScript and Python. Note that these are all unidirectional translations – Blockly programs

---

³https://github.com/LLK/scratch-vm/blob/develop/docs/extensions.md
⁴https://makecode.microbit.org/#editor
⁵https://makecode.microbit.org/blocks/javascript-blocks
⁶https://blockly-demo.appspot.com/static/tests/playground.html
are translated to existing text languages that may not represent all features of Blockly, such that there is not way to go from text back to blocks.

- Text Blocks\textsuperscript{7} are a convention for writing Scratch blocks in text on Scratch forums when the normal block images are avoided or tedious to upload. It is just a convention, and no aspect of it is automated. Still, it implies that any block can be represented by text with defined rules.

### 2.5 Translating App Inventor into Text

#### 2.5.1 App Inventor to Python

Philip Guo proposed a pseudo-Python translation of App Inventor Classic blocks \[Guo\]. It intended to improve the readability of App Inventor Classic programs, especially the ones built with complex blocks, and to help users (often novice programmers) to transition to future study in programming, which involved reading and composing textual code.

As Guo pointed out, the purpose of the pseudo-Python translation would be "read-only, ...merely illustrative... [and] not meant to be executable" \[Guo\]. Therefore, it could potentially solve only one of the challenges faced by App Inventor users described in Section 1.2 with easier navigation of the blocks. It would be impossible to edit the blocks in text or to convert the translation back to App Inventor blocks.

#### 2.5.2 App Inventor to TAIL

Karishma Chadha developed TAIL, the Textual App Inventor Language, intending to solve the problems addressed in Section 1.2 \[Cha14, CT14\]. TAIL is isomorphic to App Inventor’s blocks language and thus allows for bidirectional conversion between blocks and text \[Cha14\]. By introducing bidirectional conversion between blocks and text to App Inventor, TAIL would increase App Inventor’s general usability as it addressed users concerns about reading, composing, editing, and sharing App Inventor blocks.

\textsuperscript{7}https://scratch.fandom.com/wiki/Text_Blocks
blocks, and potentially ease the transition from blocks to text programming for novice programmers.

TAIL adopts a fully-braced syntax to prevent the possible loss of whitespace and indentations during copying and pasting. Since the three categories of App Inventor blocks have different shapes, TAIL uses three different kinds of braces to maintain the feature in text: { } for expressions, [ ] for statements, and ( ) for top-level blocks, based on the similarity of the shapes of these braces to the shapes of the blocks they tend to represent.

Beyond the syntax design and language implementation, Chadha also prototyped the bidirectional conversion mechanism in App Inventor that converts TAIL to XML-based App Inventor blocks and back [CT14]. Chadha further extended the App Inventor blocks library and implemented a set of code blocks in which users are able to type in TAIL code. In particular, there are three types of code blocks that respectively represent App Inventor expressions, statements, and declarations by preserving the original shape of the corresponding App Inventor blocks. In this way, TAIL code could be embedded into App Inventor programs in App Inventor’s IDE and coexist with regular App Inventor blocks, which turns the proposed DMPE into a truly hybrid one by enabling the coexistence of the textual TAIL code and the rest of App Inventor blocks. In addition, any code block containing valid TAIL code can be converted to corresponding App Inventor blocks, and any regular App Inventor block can be transformed into a code block that contains the block’s TAIL translation. Figures 2-1 – 2-3 [Cha14] illustrate the conversion between App Inventor blocks and TAIL code blocks.

![Figure 2-1: Converting an App Inventor event handler to TAIL.](image-url)
The feature of TAIL code blocks was innovative because they provided fine-grained conversion between blocks and text for any block clusters (not just whole screens). However, TAIL is never integrated into App Inventor’s current framework. While its technical implementation is near to complete, TAIL has been work-in-progress with several App Inventor features yet to be developed. It has not gone through user testing as well.
3 Design Principles

The design of Venbrace is guided by several design principles (DPs), first presented in the 2019 Blocks and Beyond Workshop [HT19] but discussed in detail in the following sections. Note that the principles are reordered in this chapter by priority, different than they were initially presented in the Workshop.

3.1 Evidence-Based Language Design

The most important principle is DP1 Employ evidence-based language design. The initial design of Venbrace, completed in Spring 2019 and detailed in Chapter 4, is the first in a sequence of designs that will be part of an iterative design process featuring several rounds of user studies (described in Chapter 7). In traditional programming language design, feedback from actual end-users is often absent when design choices are made for the language syntax. This project takes a different programming language design approach: influenced by Stefik and Hanenberg [SS13, SH14] and Myers et al. [MSH+16], who emphasized the importance of and demand for evidence in programming language development, I designed the user studies following the guidelines by Myers et al. [MKLY16] and Ko et al. [KLB15] to confirm whether the design choices would be positively perceived and seek for possible improvements.

3.2 Bidirectional Isomorphism

Bidirectional isomorphism, also one of the design principles adopted by TAIL [Cha14], means the following: any App Inventor block should be able to be represented in a textual language, while any valid code written in that textual language should be able to be converted back to the visual syntax of App Inventor blocks. Bidirectional
isomorphism between an App Inventor block and its textual representation guarantees bidirectional conversion between the block and the text.

As is briefly mentioned in Section 2.5.2, whereas TAIL was only a textual notation for block clusters, VENBRACE will ultimately (but does not currently) support 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 any App Inventor project. **DP2 Support bidirectional isomorphism between all aspects of an App Inventor project and text** ensures that any App Inventor program could be converted to VENBRACE, and any valid VENBRACE composition, whether abbreviated or not, should be able to translated back to its App Inventor original. Such bidirectional isomorphism guarantees the code integrity before and after the conversion of any direction, and can potentially turn App Inventor into a DMPE.

A decision has to be made, however, on whether the most verbose or the most concise translation will be the default when an App Inventor fragment is converted into VENBRACE, given that abbreviations of App Inventor helper words are potentially allowed. If the most verbose version is arbitrarily adopted, I foresee a potentially conflicted situation in which the user writes their VENBRACE code in a rather abbreviated mode, converts the code to App Inventor blocks with no trouble, but obtains a fully verbose VENBRACE representation that is completely different from their original composition when they convert the App Inventor blocks back to VENBRACE. It is worth exploring the opinions from the actual end-users in user studies using the Natural-Programming Elicitation technique [MKLY16]: what they perceive as the most natural implementation may shed lights on solving this conflict.

### 3.3 Blocks Are Primary

The principle **DP3 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 [HN14] and Droplet/Pencil Code [BBDP15], which
are visual versions of the existing textual languages Grace and CoffeeScript. In these systems, text is primary; the blocks were designed to match the text, not vice versa.

A practical consequence of DP3 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. The 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 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 Section 4.1), with different conventions (parentheses for expressions, braces for statements) chosen for consistency with traditional text languages. This fully-braced syntax thus highlights the correspondence between visual blocks and braced textual notation, although some abbreviations of App Inventor keywords (mostly helper words) are allowed and lead to alternative VENBRACE representations of App Inventor blocks, which are further described in Section 3.4.

### 3.4 Concise Alternatives

App Inventor and other BPEs have been perceived as easy to use and understand by novices [BGK+17, BBDP15, WW15, WW17] due to the adoption of helper words, natural language-like words and phrases in the blocks that describe their usage. However, requiring users to type out all the helper words in VENBRACE text is rather labor intensive and error prone. Therefore, I consider it important to allow for abbreviations for complete helper word phrases in VENBRACE using either shorter keywords or symbols, because it contributes to the programming efficiency and, eventually, usability of VENBRACE.

DP4 is thus Maximize flexibility by supporting alternative more concise textual representations. For example, I imagined abridging variable setters using
the assignment symbol "<-", such that \{numZeros <- \ldots\} abbreviates \{set numZeros to \ldots\}. However, arbitrarily abbreviating the helper words and phrases leads to a tension between this design principle and **DP3 Blocks Are Primary**: while the flexibility of composing VENBRACE is maximized with the support of alternative more concise textual representations, the resulting abbreviations might not be similar to their counterparts in the original App Inventor blocks, which might cause confusion among users. Therefore, a decision must be made determining the priority between the two principles.

There are also abbreviations of multi-word key phrases that cannot be determined at this moment. For example, I cannot decide whether the long key phrase in (create empty list) should be abbreviated using underscores (create_empty_list), camelCase (createEmptyList), or one simple word that is commonly see in LISP-like languages (list).

To solve these unanswered design questions and make a favorable design decision, I will seek for insights from user feedback on the abbreviations.

### 3.5 Support Copy/Paste

**DP5 Support 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, without being impacted by a potential loss of whitespace and indentation during the copying/pasting process. A practical consequence of this principle is that the notation does **not** employ Python-like indentation-based formatting, because indentation is often lost when copying between text systems. I use explicit braces in VENBRACE to avoid this problem, and also to emphasize the boundary of each block.

A downside of this design choice is that it may negatively affect reading and writing text programs, especially when users compose VENBRACE representations for complex blocks that involve nested blocks, which will further result in nested braces. To ameliorate this issue, I also implemented a syntax editor that helps with brace completion and balancing (more in Chapter [6]). I will also seek user feedback on this
choice in the study described in Chapter 7.

## 3.6 Internationalization

App Inventor is used in almost every country on Earth and handles keywords in 15 languages. **DP6 Support internationalization** says that Venbrace should allow keywords in all handled languages and any valid App Inventor Unicode variable names. As Mönig et al. [MOM15] noted when they were developing GP, the textual representation for a blocks language should support all original features that come with the blocks to maximize its usability, and in this case internationalization comes as an original feature with App Inventor. As a result, the support of internationalization also complies with **DP2 Bidirectional Isomorphism**.
4 Initial Design

4.1 VENBRACE Syntax

The VENBRACE syntax is systematic and summarized in the following rules:

1. A pair of **braces** are used to indicate the extent of an App Inventor block. There are three types of braces in VENBRACE: parentheses ( ), curly braces { }, and squared brackets [ ].

2. Parentheses are used to represent expression blocks: ( ... ).

3. Curly braces are used to represent statement blocks { ... }.

4. Squared brackets are used to represent declaration blocks: [ ... ].

5. All braces should be balanced. An open brace has to be matched with a closed one.

6. Block extents are indicated by the three types of brackets mentioned above, and a closed bracket indicates the end of an extent.

7. Doubles-slashes indicate the beginning of single-line comments: // ....

8. Slash-asterisk marks the beginning of a multi-line comment, which is ended by asterisk-slash: /* ... */.

9. The pound sign represents project-specific configurations, including block language environment and custom configuration for block positions: # .... Such configurations should be placed at the top of a file.

10. Whitespace characters between tokens are ignored and do not affect the syntactic structure: <token1> <whitespace>* <token2>.
These rules are described in detail in the following sections. Please note that angle brackets < > are regarded only as signs for mathematical comparisons in VENBRACE, and such a case of <text> is not valid in VENBRACE. Therefore, as is used in the last syntax rule and more in the following sections, <text> will be used as place holders, in which text specifies the meaning of each place holder. In addition, pseudo-code samples are written in teletype characters using actual keywords and place holders written as regular expressions.

I also use the notations of parent block and child block in explaining the VENBRACE syntax. I refer to blocks that accept the attachment from other blocks as parent blocks, and blocks that are to be connected to other blocks as child blocks. Most App Inventor blocks can be parent blocks and child blocks simultaneously, although there are blocks that can serve only as child blocks (see Section 4.1.1 "Atomic Expressions") or as parent blocks (see Section 4.1.3 "Declarations").

4.1.1 Expressions

In App Inventor, expression blocks are blocks that have plugs to their left. The shape of expression blocks in App Inventor indicates that they carry values to be attached to other blocks, which have sockets that match the shape. There are various kinds of expression blocks in App Inventor, such as math expressions, logical expressions, strings, lists, colors, and screen components. Figure 4-1 shows some examples of App Inventor expression blocks.

I further divide App Inventor expressions into multiple categories, each category translated into VENBRACE differently.

Atomic Expressions

There are certain expression blocks in App Inventor that do not have any keywords and can only be child blocks, i.e., they do not have any sockets to be plugged in with other blocks. I define such expressions as atomic expressions.

Atomic expressions form the basis of any App Inventor program because they
Figure 4-1: Different kinds of expression blocks in App Inventor. They all have plugs to their left.

can only be child blocks. In other words, any parent block could potentially end up having one or more atomic expressions as its child block(s). Therefore, atomic expressions are commonly seen and widely used in App Inventor. Examples of atomic expressions include numbers, strings, boolean expressions, colors, and some of the screen component properties (Figure 4-2).

Figure 4-2: Examples of atomic expression blocks.

As noted earlier, an expression is surrounded by a pair of parentheses in VENBRACE. Writing and pairing parentheses in VENBRACE for atomic expressions can be cumbersome given their frequent appearances. Therefore, I decided to make the surrounding parentheses for atomic expressions optional. In such a case, both (17) and 17 are considered as valid VENBRACE translations for an App Inventor block of
number 17 (Figure 4-3).

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Venbracer (Complete)</th>
<th>Venbracer (Abbreviated)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Number Block" /></td>
<td>(17)</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 4-3: A number block, which is an atomic expression, its complete Venbracer translation, and the abbreviated translation.

**Simple Expressions**

Simple expressions are expressions with all child blocks atomic. For example, an addition of two numbers and a logical negation of a boolean value are both considered as simple expressions (Figure 4-4). A simple expression must have its own pair of surrounding parentheses in Venbracer. In fact, Venbracer text has to be written in exactly the same way as its corresponding App Inventor block(s). All child blocks of a simple expression have to be translated into Venbracer following their original order; changing the ordering of child blocks in App Inventor will result in a different Venbracer translation accordingly (Figure 4-5).

**Nested Expressions**

For a better explanation, I refer to child blocks of an expression block as its subexpressions. Nested expressions are expressions with subexpressions. A nested expression must have its own pair of surrounding parentheses, and all its subexpressions have to be translated in order, following their corresponding Venbracer translation rules.
Figure 4-5: Although these two expressions evaluate to the same value, they are not the same expression block.

Most nested expressions have infix operators, and their VENBRACE translations are normally ordered translations of the subexpressions connected by the operators. Prefix operations, on the other hand, are a special common kind of nested expressions. They are translated by writing the operation name before the ordered translations of their subexpressions.

Figure 4-6: A nested expression with infix operators (upper) and one with a prefix operator, min (lower).

Special Expressions

Empty Expressions

Empty expressions refer to sockets of an expression block that do not yet contain any subexpressions. However, some notation is needed to indicate an unfilled socket
to distinguish cases like the position of the unfilled socket in the \texttt{min} examples in Figure 4-7.

Figure 4-7: Two \texttt{min} blocks with unfilled sockets at different positions.

In her design for TAIL, Chadha [Cha14] used \{\} for such an empty socket (note that curly braces are used for expressions in TAIL). The TAIL translations for the two examples in Figure 4-7, respectively, would be \{\texttt{min} \{3\} \{\}$\} and \{\texttt{min} \{\} \{3\}\}.

In \textsc{Venbrace}, empty expressions are instead represented by a pair of parentheses. The examples in Figure 4-7 would be \texttt{(min 3 ()}) and \texttt{(min () 3)}, respectively.

\textbf{Variable Getters}

Variable getters are another kind of special expressions. Like atomic expressions, they can only serve as child blocks and cannot have their own child blocks, but they do not qualify as atomic expressions because they come with the keyword \texttt{get} in front of the variable name in App Inventor. See Figure 4-8 for details. Nevertheless, I believe it is necessary to introduce some flexibility in translating variable getters in \textsc{Venbrace} because they are widely used like atomic expressions, but it is also important to emphasize their difference from atomic expressions. In \textsc{Venbrace}, I allow variable getters to be translated into their complete form \texttt{(get <variable>)} or an abbreviated form \texttt{<variable>}. Note that the abbreviated form has neither the \texttt{get} keyword nor the pair of parentheses typically required for other expressions.

\begin{table}[h]
\centering
\begin{tabular}{ccc}
Blocks & \textsc{Venbrace (Complete)} & \textsc{Venbrace (Abbreviated)}
\hline
\texttt{get x} & \texttt{(get x)} & \texttt{x} \\
\texttt{get num + 1} & \texttt{((get num) + (1))} & \texttt{(num + 1)}
\end{tabular}
\caption{Variable getter expressions, their complete \textsc{Venbrace} translations, and abbreviated translations.}
\end{table}
4.1.2 Statements

Statements perform actions. A sequence of statements compose vertically with each other to form a stack of statements using the nubs and notches. Their shape indicates that statements are composed and processed in order in App Inventor.

Common App Inventor statements include component setters, variable setters, component method calls, procedure calls, and control statements. In VENBRACE, statements are always represented by a pair of curly braces.

Many statement blocks look for expression blocks as their child blocks to perform certain actions. Rules of translating expressions still apply when expression blocks are involved in translating statement blocks. See Figure 4-9.

![Figure 4-9: Simple statement blocks and their VENBRACE translations.](image)

Statements normally appear in a sequence. When multiple statements appear in a sequence (i.e., “a stack of statements”), they are translated to VENBRACE one after another following their original order, resulting a sequence of pairs of curly braces. Each statement has its own pair of curly braces.

When a statement is nested within another statement, the inner statement needs its own pair of curly braces, so does the outer statement. Similar to how the blocks are visually nested in App Inventor, the corresponding pairs of curly braces are written in a nested way in VENBRACE. See Figures 4-10 and 4-11.

An empty statement socket in an App Inventor block indicates an empty sequence of statements, which is semantically "do nothing" or "pass." Just as in TAIL, the empty sequence translates into nothing in VENBRACE. Optionally, it could be represented by a pair of curly braces {}. Such an option is added because it might be
Figure 4-10: Nested statements involving conditionals.

Figure 4-11: An example in which the two conditional branches each has a sequence of statements.

too confusing for programmers to go with the default option (with something translated into nothing), and {} is also a statement analog to () for an empty expression. Figure 4-12 illustrates both empty expressions and empty sequences. In App Inventor, procedure or component method calls are statement blocks that start with the keyword call. For a more efficient programming experience, call keyword is optional in Venbrace.

4.1.3 Declarations

Declarations create event handlers, procedures, and global variables. Each declaration is independent from one another, and they do not have any external connectors. As previously mentioned, declaration blocks can only parent blocks in App Inventor, so they are also addressed as top-level blocks.

In Venbrace, all top-level declarations are surrounded by a pair of squared brackets. Since they are parent-only blocks, they always involve statements and
expressions as child blocks. Rules of translating expressions and statements still apply when these blocks are involved in translating declarations.

Event handlers always have the keywords `when <component_event> do <statement(s)>`. These keywords are needed in VENBRACE as well (Figures 4-13 and 4-14).

Procedure declarations have two structures: `to <procedure_name> <parameter>* do <statement(s)>` and `to <procedure_name> <parameter>* result <expression>`. The keywords `to`, `do`, and `result` are used in VENBRACE as well, depending on the type of the procedure declaration.

Procedure declarations can have zero or more parameters. When a procedure declaration has parameters, which are shown in the orange areas following the procedure name in App Inventor, the parameters are translated into VENBRACE following their original order without any modification. See Figures 4-15 to 4-17.

Global variable declarations come with the structure of `initialize global`
Figure 4-14: A more complicated event handler.

Figure 4-15: A simple procedure without parameters.

Figure 4-16: Procedure with parameters that do not return results.

Figure 4-17: Procedure with parameters that return results.
<variable> to <value>. When the global variables are referred after they are declared, they are always prefixed with the keyword global.

4.2 Translations that are not Obvious

In App Inventor, helper words are used in a lot of blocks to help explain the usage and function of such a block. Despite the initial design for VENBRACE follows the principles described in Chapter 3, translations for blocks with helper words are not obvious. This section describes six kinds of such blocks, and compare current design decisions/ideas with design decisions made in TAIL [Cha14].

Any arbitrary decision on these translations without evidence support will go against DP1 Evidence-based Language Design (see Chapter 3.1). Therefore, all VENBRACE design ideas presented in this section will be further evaluated in user studies (such as the one described in Chapter 7) before they are finalized.

4.2.1 Multi-Word Key Phrases

Many App Inventor blocks begin with multi-word key phrases containing helper words that explain the purpose of the block and distinguish it from other blocks. (e.g., Figure 4-18). For traditional text languages, the lexer/parser design is geared

![convert_radians_to_degrees](convert_radians_to_degrees)

Figure 4-18: One of many App Inventor blocks beginning with a multi-word key phrase

at single-word keywords, as single-word keywords are used to distinguish constructs. A multi-word key phrase such as convert radians to degrees could be converted to a single-word keyword following some general rule, such as replacing spaces by underscores (convert_radians_to_degrees), using camelCase (convertDegreesToRadians), or some notation involving delimiters (e.g. <convert degrees to radians>, |convert degrees to radians|). To represent multi-word key phrases in App Inventor blocks
in Venbrace, should the multi-word phrase be converted to a single token by some general rule used for traditional text language design, or should the spaces be kept? Should we also allow abbreviations involving one or more of the multiple words in Venbrace?

In TAIL, underscores are used to turn multi-word key phrases into single words, such as `convert_radians_to_degrees` and `add_items_to_list`. However, certain special-case abbreviations are allowed, such as `list` for both `make a list` and `create empty list`.

In my very initial design, I chose to use camelCase instead of underscores to convert multi-word key phrases into single-word keywords out of personal preference. However, there has been a debate between the camelCase and underscore strategies [BDLM09]. Instead of making an arbitrary decision like the one in TAIL, I would like to see what users prefer and review the user study results (see Section 7.4) to guide the decision.

### 4.2.2 Labels on Expression Sockets

Helper words directly to the left of expression sockets, such as `text`, `start`, and `length` shown in Figure 4-19, are labels on expression sockets. Such labels are technically unnecessary, since all argument expressions are unambiguously determined by position. However, they are useful as annotations to help remind which arguments belong in which positions. For example, there are 8 arguments in `Canvas1.DrawArc` (see Figure 4-20), so the 8 labels for these arguments become very helpful. If such labels are included in Venbrace, should they appear unchanged (`text`, `start`, `length`), or should they be distinguished by some marker (e.g., a trailing colon, as in `text:`,

![Figure 4-19: A block with labels on expression sockets.](image)
Figure 4-20: Labels on expression sockets are especially helpful in blocks like the Canvas DrawArc method that have many arguments.

\[ \text{start:}, \text{length:} \] that make it more obvious that they are expression socket labels rather than other constructs (e.g. abbreviated variable references)?

TAIL uses trailing colons on all expression socket labels. At the current stage, Venbrace includes all socket labels unchanged, but does not allow any alternative abbreviations (such as allowing socket labels to be omitted). This simplifies the translation rules, since programmers can just write down the labels they see on the blocks.

### 4.2.3 Labels on Statement Sockets

Statement socket labels are similar to expression socket labels, but unlike with expression socket labels, which are normally used to remind positions of arguments and are completely optional, statement socket labels are sometimes necessary to avoid ambiguity. When there is only one statement socket label in a block (as in Figure 4-21), there can never be ambiguity by omitting it. However, when two or more labeled

Figure 4-21: A block with labels on statement sockets.

statement sockets appear next to each other (as in Figure 4-22) ambiguity is possible.
Figure 4-23 shows that without the `else` keyword as a separator, it would not be clear how many of the 3 setters are in the `then` branch and how many are in the `else` branch. Many text languages have an `else` keyword for this reason, even though they may also support special notation for sequences of statements (like curly braces in C/Java-like languages) that makes the `else` keyword technically unnecessary. Note that such a special notation statement grouping is missing in the current design for Venbrace, however.

TAIL treats statement socket labels just like expression socket labels, marking them with a trailing colon, so TAIL’s translation of the block in Figure 4-22 is `[if {${} then: else:]`. Note that TAIL statements are delimited by square brackets `[]` rather than curly braces as in Venbrace, expressions are delimited by curly braces, and nothing is written for an empty sequence of statements. In Venbrace, all statement labels remain unchanged, so the translation would be `{if () then {} else {}}` if `{}` is used for the empty statement sequences (as described in Section 4.1.2) or `{if () then else }` if the occurrences of `{}` are omitted.
4.2.4 Event Handlers Parameters

Many event handlers have fixed parameter names as part of their contract. As Figure 4-24 shows, Canvas.Dragged is an extreme case, with 7 such parameters. Since a user has no right to edit such parameter names in App Inventor, it is tedious and error prone to require a user to type them all in when editing in the text mode. Nevertheless, not writing them down explicitly makes variable scoping less clear, because the declaration of these names would not be shown.

Chadha did not specify how such cases would be handled in TAIL. In Venbrace, no design decision has been made, either. Similar to the multi-word key phrases (see Section 4.2.1), translations for event handlers with parameters cannot be determined until I obtain initial user feedback from the user study.

4.2.5 Procedure Names and Procedure Parameter Names

In App Inventor procedure declaration blocks, the places to type in the procedure name and parameter names are editable fields that look different than other block words (Figure 4-25).

TAIL uses angle brackets to delimit names that are editable. So the TAIL translation for the example in Figure 4-25 would begin (to \texttt{average} \texttt{a} \texttt{b} result: \ldots). In the current design of Venbrace, on the other hand, all names on blocks are
written down without any special notation for editable names like procedure names and parameter names. In this case, the Venbrace translation for the example block begins \[\text{to average a b result }\ldots\].

Mathematics and many text languages have special notations for declaring procedures/functions with parameters. One commonly seen notation is to first write out the procedure name, then wrap the parameters with parentheses and separate the parameters by commas, such as \texttt{average(a, b)}. But this notation is inconsistent with Venbrace's use of parens and does not match the words on the blocks.

### 4.2.6 Local Variable Declarations

In addition to procedure parameter declarations, local variables can also be declared in AI in loop blocks (e.g., Figure 4-26) and in local variable initialization blocks (e.g., Figure 4-27).

![Figure 4-26: Declaring a local variable through an editable field in a loop block.](image)

Neither of the two types of blocks appears to have an obvious text representation. Local variable declarations written in editable fields look different than other words in an App Inventor block. Many text languages have special notation for declaration of local variables, and it remains unknown whether a special notation should be used/developed for local variable declarations in App Inventor similarly.
As with procedure parameters, TAIL uses angle brackets to denote editable local variable name declarations. In this case, the loop in Figure 4-26 is written as `[for_each <num> from: {$} to: {$} by: {$} do: ]`. However, TAIL translations for variable initialization blocks, such as the one in Figure 4-27, have not been specified. So far, no design decision has been made for either type of blocks in VENBRACE, and I seek for guidance from the user studies.

4.2.7 Global Variable Declarations, Getters, and Setters

Similar to other blocks with helper words, App Inventor blocks that have global variables do not have obvious textual representations due to the extra helper word `global` in front of a variable name, including global variable declarations, getters, and setters (Figure 4-28). The helper word `global` results in multi-word key phrases in all these related blocks. What remains unclear is whether `global` should be part of a key phrase (combined with other neighboring keywords) or part of the variable name.

For global variable declarations, TAIL is consistent with other design decisions mentioned above: `initialize_global <angle> to: 30`. For getters and setters, however, TAIL treats `global` as a special extra keyword that is not incorporated into the `get` or `set` keyword with an underscore but rather as an extra modifier on the variable name, such as `{get global sum}`, `[set global n to: 5]`.

My current design for VENBRACE requires users to always write out the keyword `global` before the variable name when referencing global variables in VENBRACE, no matter in declarations, setters, or getters: `[initialize global angle to 30]`, `{get global sum}`, `{set global n to 5}`. However, like local variables, the getters and setters for global variables are very common in App Inventor programs, so shorthands are worth considering. Although not integrated into the current VENBRACE implementation, I propose using the variable prefix symbol `$` to replace the keyword `global` as an abbreviation to improve programming efficiency:
[initialize $angle$ to 30], (get $sum$) (or just $sum$, {set $n$ to 5}). This will be further evaluated in the user studies before being finalized.

4.2.8 Undecided Abbreviations for Setters

In App Inventor, setters are used for variable and component assignments. Text representations for setters are rather obvious. The most verbose VENBRACE translation for the setter block in Figure 4-29, for instance, would be {set Label1.Text to ("Yes")}. The fact that setters are commonly used in App Inventor suggests designing abbreviations for the setter blocks; however, there doesn’t seem to be an obvious VENBRACE abbreviation for a setter.

![Figure 4-29: A variable setter.](image)

In many textual programming languages, the equal sign = is widely used to denote assignments, and it is also conceived as the most straightforward operator for assignments by both programmers and non-programmers [SS13]. However, it conflicts with the algebraic meaning of = (to test equality). In R, another widely-used textual programming language, although = is also used as the assignment operator, the left arrow <- is more recommended [Goo], as it clearly states in which direction the assignment is made.

TAIL does not specify any abbreviation for setters. In VENBRACE, no design decision has been made, either. The equal sign = cannot be used as an abbreviated assignment symbol in VENBRACE given that it is already a reserved notation for the equality-testing operator in App Inventor. The left arrow sign, though, could potentially be used in abbreviated translations for setter blocks.
5 Implementing VENBRACE

5.1 Overview

Creating VENBRACE, like creating any other textual programming language, requires two major steps: lexing and parsing. From the computer’s perspective, code written in any textual programming language is read as a stream of pure text. To make the stream of text "meaningful," i.e., understood by the computer according to the syntactic rules of a programming language, a lexer and a parser are needed.

A lexer takes the text of a program as input and breaks it into a stream of lexical tokens, the smallest units of the programming language, based on rules specified by the lexical syntax. Tokens generally include numbers, strings, whitespace characters, identifiers (meaningful names recognized as variables and procedures), symbols (such as braces in VENBRACE), and language-specific keywords. The VENBRACE lexer is detailed in Section 5.3.

Code written in a programming language can be represented by a syntax tree, a tree-like representation of the abstract syntactic structure of the code according to the syntax of the language. A parser takes the tokens generated by a lexer as its input to generate a syntax tree for the original input code. Consequently, a parser requires parsing rules specified by the syntax to determine whether a sequence of tokens could generate a valid syntax tree and thus a valid program. Because it requires a sequence of tokens, or the tokenized text, as input, a parser has to work together with a lexer. The VENBRACE parser is further described in Section 5.4.

5.2 Parser Generator

Rather than writing a lexer and a parser from scratch, I used ANTLR, Another Tool for Language Recognition, to generate the VENBRACE lexer and parser. ANTLR
is a parser generator and, like other parser generators, it automates the creation of lexer and parser using a user-generated grammar file as its input. The grammar is written in ANTLR’s own grammar syntax, and involves the lexer grammar and the parser grammar, which include lexer rules and parser rules and to generate the lexer and the parser, respectively [Par13].

I kept the lexer grammar and the parser grammar for VENBRACE within one ANTLR grammar file. ANTLR distinguishes lexer rules from parser rules based on the rule names: lexer rule names must begin with an uppercase letter, while parsing rules have to have their rule names in lower cases. The lexical and parsing rules are detailed in the following sections.

### 5.3 Lexing VENBRACE

Some lexer rules recognize string literals as tokens and assigned them with token types directly. The code snippet below shows an example of specifications of arithmetic operators in VENBRACE.

```
PLUS: '+';
MINUS: '-';
MUL: '*' ;
DIV: '/' ;
POW: '^' ;
```

Another type of lexer rules do not specify tokens directly. Instead, they use regular expression to recognize other lexer rules which lead to token specifications or even some other rules. Below I presents how the VENBRACE token `COMPONENT_PROPERTY` consists of tokens `ID` and `DOT`.

```
COMPONENT_PROPERTY: ID DOT ID;
ID : (ALPHA | '_') (ALPHA | '_') | DIGIT)*;
```

There are also lexer rules that do not specify tokens but rather aid in the recognition of tokens. In other words, they cannot assign token types to the stream of text
or tokens they recognize, they cannot be referenced in parser rules (more in Section 5.4), and they can only be used in other lexer rules. The ANTLR syntax requires that the keywords fragment has to be used before the rule name of such a rule. For example, VENBRACE lexer rules DIGIT and HEX_DIGIT help with the definition of rule NUMBER.

```antlr
fragment DIGIT : ('0'..'9');
fragment HEX_DIGIT : (DIGIT | 'a'..'f' | 'A'..'F');
NUMBER : ((DIGIT* DOT DIGIT+) | (DIGIT+ (DOT)?) | ('0x' (HEX_DIGIT)+));
```

### 5.4 Parsing VENBRACE

The VENBRACE parser grammar is context-free, and its parser rules specify parts of the syntax tree of a VENBRACE program using regular expressions. The grammar starts with a top-level nonterminal rule, which recognizes other rules that are not top-level. A VENBRACE syntax tree is built by repeating the process recursively until a terminal rule is reached, i.e., a parser rule that consists of purely lexer rules. The code snippets below illustrate a top-level rule, a rule that is neither top-level nor terminal, and a terminal rule in the VENBRACE parser grammar.

```antlr
program: (decl_block)* EOF;

A top-level nonterminal parser rule.

decl_block: LSQR decl RSQR;

A non-top-level-nonterminal parser rule.

atom:

NUMBER
| NEG_NUM
| STRING
| TRUE
```
A terminal parser rule.

For a full version of the VENBRACE grammar, see Appendix C. Note that only blocks with decided translations described in Chapter 4 are included in this Appendix; all other components that are still work-in-progress are not included.
6 A Syntax Editor for VENBRACE

6.1 Motivation

In Spring 2019, when the design of VENBRACE was still in its initial stage, I imagined an Integrated Development Environment for VENBRACE and App Inventor that would support the following features:

**VENBRACE-specific**

- Auto-completion for keywords, identifiers, and braces;
- Brace matching and balance checking;
- Keyword and block extent highlighting;
- Automatic helper word insertion and deletion;
- Error detection, reporting, and handling;
- Typeblocking as in App Inventor (see Section 1.3.2);
- Display of variable dependencies that allow automatic renaming when a variable name is changed.

**VENBRACE and App Inventor**

- Partial VENBRACE-to-App Inventor and App Inventor-to-VENBRACE conversion;
- Real-time collaboration in both text and blocks.

As I finished the initial syntax design for VENBRACE and was planning for a user study to evaluate the syntax, I realized that VENBRACE would be new to any
potential participant, and that the study would hence become an exploratory study in which participants would learn to use the new technology by interacting with the technology itself [MKLY16]. To ease the cognitive burden brought by the learning process, I believed that a text editor more powerful than a plain text editor would be beneficial in any study requiring subjects to enter VENBRACE code.

Considering the scope of this project, I determined that a lightweight, JavaScript-based and HTML-embeddable syntax editor with some of the basic features of the imagined IDE would be ideal for improving user interactions with VENBRACE in the study. The simplified syntax editor could do the following:

- Auto-completion for braces;
- Brace matching and balance checking;
- Keyword and block extent highlighting;
- Keyword-based formatting (for readability).

Because I wanted the syntax editor to be lightweight and embeddable in a web page that could run in any major web browser, I looked for existing open-source, JavaScript-based tools with APIs that dealt with syntax highlighting and were extendable with my customizations for VENBRACE. Ace [Aja] and CodeMirror [Cod] are both widely-used JavaScript-based source code editors that support syntax highlighting and are configurable with user-specified features. I chose to implement the VENBRACE syntax editor based on CodeMirror because I could extend on its existing modes that already handled part of the core features I wanted for the VENBRACE editor. The following section demos how the VENBRACE editor realizes the four features above.
6.2 Implementation

6.2.1 Brace Auto-Completion

CodeMirror’s pre-defined mode closebrackets automatically fills in a closed bracket } (I refer to it as an curly brace in this document) when an open one is typed into the editor. In the constructor of the mode configuration, the instance variable pairs specifies that the mode applies to curly braces. I extended the closebrackets mode to auto-close VENBRACE braces (parentheses, curly braces, and squared brackets) by adding all of these symbols to the pairs variable. Auto-closing VENBRACE braces could save users some time typing closing braces and potentially improve their programming efficiency.

6.2.2 Brace Matching and Balancing

CodeMirror also comes with a mode matchbrackets that matches a closed curly brace with a closest open curly brace in its front. The matchbrackets mode further adds CSS to the matched curly braces that may be customized. A local variable matching in the implementation of matchbrackets maps a brace to its match and tags the direction (open vs. closed) of the match. I extended the matchbrackets mode to match VENBRACE braces by adding all of these symbols, their matches, and directions of the matches to the matching variable.

In addition to allowing me to customize the CSS of matched VENBRACE braces, CodeMirror also allowed me to add special CSS for braces with no match, i.e., unbalanced braces. In the VENBRACE syntax editor, the pair of matched braces closest to the cursor will be highlighted in green, while the brace closest to the cursor that does not have a match will be highlighted in red. The contrasting coloring of matched and unmatched braces could assist users with maintaining the balance among all braces in a fully-braced language like VENBRACE.
6.2.3 Keyword and Block Extent Highlighting

CodeMirror already supports syntax highlighting for more than 130 programming languages, including Scheme, a fully-parenthesized LISP-like programming language. CodeMirror recognizes keywords and symbols for a programming language through language modes, which are in fact mini-lexers written in JavaScript. Following the structure of the Scheme language mode in CodeMirror, I created the Venbrace syntax mode so that the editor would recognize and further highlight all Venbrace keywords and symbols.

The extent of an App Inventor block is indicated by a pair of matching braces in Venbrace, as described in Section 4.1. Inspired by DrRacket [Rac], which highlights scopes within a Racket program with a special background color, the Venbrace syntax editor also highlights the extent of a block, i.e., the textual contents between a pair of matching braces. I implemented this feature by further extending CodeMirror’s matchbrackets mode: whenever the user moves the cursor next to an open brace, if there is a matching closed brace, the text between the open and the closed brace is highlighted in light grey; if the user moves the cursor next to an closed brace with a matching open brace, the text between the pair of matching braces will be highlighted similarly. I believe highlighting the block extent would constantly remind the user of the proper usage of braces in Venbrace and also suggest the connection between braced entities in Venbrace and blocks in App Inventor.

6.2.4 Keyword-Based Formatting

Unlike an indentation-based programming language like Python, where whitespace characters affect a program’s syntactic structure, whitespace characters are ignored in Venbrace and do not affect the parsing process. In other words, a valid Venbrace program can be written in either one single line or in multiple lines full of indentations and spaces. Whitespace does affect code readability, however. I believe that a well-formatted textual program comes at a high readability and thus positively affect programming efficiency. Therefore, in the Venbrace syntax mode for
CodeMirror, I marked **VENBRACE** keywords such as **to** and **when** as **indentKeys**, so that an indentation would be placed at the beginning of the new line following any of these keywords. Note that although the **VENBRACE** syntax editor uses **VENBRACE**

![Figure 6-1: A cluster of App Inventor blocks translated in **VENBRACE** using the **VENBRACE** syntax editor.](image)

keywords and symbols to recognize text patterns and highlight syntactic components, the **VENBRACE** lexer and parser are not yet embedded in the editor. In the user study that evaluates the **VENBRACE** syntax, I combined them all to optimize the functionality of the editor, which further enables simple error detection and reporting. In addition, to observe how users use and interact with **VENBRACE** in the study, I also implemented keystroke tracking functionality in the editor, such that the editor would listen to all keystroke and cursor events triggered by the user. See the next chapter for more details.
7 Online VENBRACE Syntax User Study

7.1 Overview

VENBRACE was designed and implemented to improve the general usability of App Inventor and bridge the gap between blocks and text programming. To comply with the design principle DP1 Evidence-based Language Design (see Chapter 3.1), the design and implementation of VENBRACE must be user-preference-informed, and has to follow an iterative design process featuring several rounds of user studies. Through the user studies, we would like to understand the following questions:

1. How easy/hard are our design choices for the design of the VENBRACE syntax to learn/use?
2. When VENBRACE is integrated into App Inventor, how easy is it to use to (i) modify existing block clusters, and (ii) create new structures (top level blocks, whole screens, whole programs)?
3. When App Inventor becomes a DMPE with VENBRACE incorporated, how do users actually use it in practice?

Initially, we imagined three phases of user studies [HT19].

Phase 1: Prior to the implementation of VENBRACE, we would first evaluate various aspects of its design through a preliminary user study targeting people who could potentially benefit from programming in text inside App Inventor to ensure that the design would be intuitive and ambiguity-free. Since the study would be before any working prototype of VENBRACE, it would consist of matching and translation tasks between blocks and VENBRACE text from both directions, following one-on-one interviews for opinions on the translation details.

Phase 2: The second phase of study would be based on an implementation of
VENBRACE on the improved design that incorporates feedback from the previous study. By this phase, VENBRACE would be integrated into App Inventor with part of (if not all) the features implemented. We would like to compare the programming processes in (1) pure App Inventor, (2) pure VENBRACE, and (3) the mixture of both, again tweaking the design based on the results.

**Phase 3:** After VENBRACE becomes fully functional in App Inventor, we would invite experienced App Inventor programmers to evaluate whether VENBRACE could help them create, modify, and debug App Inventor programs. We intended to measure (1) the time participants take to read and write App Inventor blocks vs. VENBRACE code and (2) how often and under what conditions users switch between blocks and their textual representation.

While the first imagined phase of study would involve no implementation of VENBRACE at all, the second and the third phases would be based on an implementation already integrated into App Inventor. We later realized that a phase of study/studies that evaluates aspects of VENBRACE syntax design *without* any integration in App Inventor at the early stages of development should be necessary yet missing. Considering the scope of this thesis project, we decided to conduct studies of such a kind that would focus on the syntax design of VENBRACE *with* an implementation of the language but *without* using it in App Inventor at all. In the future, we still would love to have at least one study of VENBRACE-in-App Inventor being used to modify App Inventor programs.

To evaluate the current design of VENBRACE syntax and its user experience, I planned three rounds of user studies, with the number of subjects approximately doubling between rounds. This thesis project only includes the first round of user study with 17 participants (see details below in Section [7.3.1]). This chapter describes the study design, which applies to all three rounds of studies, summarizes findings from the first round, and discusses the implications from the initial results.
7.2 Goals

Inspired by the structure of online learning platforms such as Coursera and edX, this study was designed to be conducted online in a manner similar to a mini-course that interleaves tutorial videos explaining how to translate App Inventor blocks into the Venbrace text language with tasks that test how well the subjects can perform these translations (more in Section 7.3).

The implementation of the study materials and the related technology was a joint effort by me and Prof. Franklyn Turbak (supervisor of this thesis project), with the help of students from the TinkerBlocks group at Wellesley College in terms of web design, content proof-reading, and pilot testing of the study (more in Section 7.3.5). Together we developed the consent protocol and overall design of the study. In particular, I developed the tutorial materials, the Translation Typing tasks, and the Pre- and Post-study Surveys. Prof. Turbak took the lead on implementation of the Translation Selection tasks, the Preference Ranking tasks, and the Preference Typing tasks with help from the TinkerBlocks research students.

Goals of the first round of study include testing the technology and collecting some preliminary data, which could help improve future rounds of studies and the next design(s) of Venbrace. The end goal for the entire syntax study is to evaluate Venbrace design choices before it is actually integrated into App Inventor. It involves the following aspects:

1. Evaluating the initial design of Venbrace and acknowledging aspects of Venbrace that are easy/hard to learn/use, especially in the Translation Typing tasks (more in Section 7.3.3);

2. Understanding users’ preferences on handling various aspects of the translation whose initial design is tentative, i.e., the ones detailed in Section 4.2 through the Preference Ranking and Preference Typing tasks (more in Section 7.3.3); and

3. Exploring potential syntactic errors that people tend to make to help with the
development of error reporting systems for Venbrace.

We expected the first round of study to take less 90 minutes to complete (verified through pilot testing described in Section 7.3.5), while the later rounds might be less time-consuming if there is going to be any changes to the structure of the study. More details regarding the design of the study are described in following section.

7.3 Methods

7.3.1 Participants

Based on advice from Dr. Andreas Stefik at the University of Nevada, Las Vegas (who has run many user studies involving programming language design [SS13, SH14]), I planned three rounds of studies: first (small, about 15 participants), second (medium, about 30 participants), and third (large, about 60 participants), doubling the number of subjects between rounds. I adopted the successive doubling strategy (suggested by Dr. Stefik) because both the design of Venbrace and the overall design of the study will become more fine-tuned along the iterated process, which will give me more confidence in enlarging the size of the study. Conducting three rounds of studies will allow for changes and improvements, if any, to details of the Venbrace syntax and/or aspects of the tutorials and study materials between rounds based on results from prior rounds.

For participants, we are targeting high school students and undergraduate/graduate college students who have had significant experience in programming using App Inventor, because recruitment within this participant group is easier. We expect that most high school participants will be US high school students who are currently taking an AP Computer Science Principles (CSP) course that uses the Mobile CSP curriculum (which uses App Inventor). They will be recruited with the help of teachers who teach the Mobile CSP curriculum. Undergraduate and graduate participants will be recruited through the MIT App Inventor team and through the researchers’ personal connections. During the recruitment process, potential partic-
Participants will be asked to provide an email address that is expected to be a valid App Inventor account with properties consistent with someone knowledgeable about App Inventor (in terms of the number of projects they have created and the time period during which those projects have been created). For the small round, it was not necessary to verify these account properties, since subjects were recruited by just a few teachers known to the researchers. For the medium and large rounds, as more potential participants will be involved, a goal is to automate this verification process.

In the first round, participants with all levels of education mentioned above will be included. For the medium and large rounds, the plan is to focus more on high school students taking the Mobile CSP course, because (1) they are knowledgeable about App Inventor and (2) many will have limited or no experience with text-based programming languages, and so they will have fewer preconceived notions of what a text language is “supposed” to look like. This plan may be modified depending on how well the recruitment works, though. We may reach out to a broader population if the smaller target does not produce enough subjects.

7.3.2 Procedure

We learned from pilot studies that they study takes about 90 minutes of dedicated time. The study is entirely online and every individual will participate in the study by interacting with a web interface whenever they find it convenient to participate in the study. Once the study runs, an email that includes a URL for the welcome page of the study will be sent to recruited subjects. On this welcome page, they will be informed of the content, purpose, duration, and compensation details of the study, as well as what knowledge/background is expected of them. For the first round of the study, all the potential subjects are provided with this information during the recruitment phase as well, so they know whether they want to participate or not before the study starts. To simplify our consideration of consent procedures, all participants are expected to attend schools in the US.

To comply with guidelines from the Institutional Review Board (IRB) at Wellesley College, subjects less than 18 years old are considered an at-risk population and are
required to follow a different protocol (involving a parent or guardian) than those who are 18 or older. We developed an online consent protocol (detailed below) that was approved by the Wellesley College IRB. Potential subjects are given the choice to click one of two buttons in the interface to indicate whether they are less than 18 years old or whether they are 18 or older, and they will be further guided to different consent collection process based on their choice. Their consent (or their assent and their guardian’s consent, if they are less than 18 years old) will be collected before any remaining steps in the study.

The study is structured as follows:

• A pre-survey;

• The main body of the study that consists of nine sections, where each section begins with a video tutorial describing some aspect of Venbrace, and in all but the first section this tutorial is followed by tasks that test the understanding of material presented in the video; and

• A post-survey.

A pre-survey at the beginning of the study will collect information that includes a participant’s age, gender, the full name of their school or affiliation, the state in which their school/affiliation is, and their previous experience with (1) programming in App Inventor, (2) programming in other blocks programming environments, and (3) programming in any text languages the subjects may have used (the subjects are explicitly informed that they are not expected to have any prior experience with text programming languages). The purpose of collecting school and state information is to help with verifying that high school participants are indeed at a school where the Mobile CSP curriculum is being taught.

The main body of the study contains various tasks that involve seeing how accurately a subject can apply blocks-to-text translation rules learned from the tutorial videos, and their preferences for blocks whose translation rules we have not yet determined because they involve complications (as described in Section 4.2). A
subject’s responses to these tasks will be collected, which include open-ended translations of blocks into Venbrace, answers to selecting valid/invalid translations of blocks into Venbrace, and Likert rankings that express preferences for translation options (more details in Section 7.3.3). Optional feedback on the tasks from the participants throughout the study will also be collected.

At the end of the study, there will be a post-survey asking for the subjects’ general thoughts about Venbrace and its utility in a future version of App Inventor.

Upon the completion of the study, a subject’s responses will be manually examined by the researchers. If the entire study has been completed and the researchers determine that the subject has made a serious attempt to answer all the questions in a meaningful way, an electronic Amazon gift card for $20 will be emailed to the subject’s App Inventor account email address provided at the before the study within 48 hours of completion.

The study involves no planned intervention from nor interaction with the researchers (I and Prof. Turbak) and developers of the online study, beyond reminder emails to take the study before it closes. However, there is interaction and feedback from the online system in terms of (1) indicating whether a blocks-to-text translation answer is right or wrong and (2) error messages indicating why a blocks-to-text translation is incorrect. Additionally, subjects are provided with the study email address venbrace-syntaxstudy@wellesley.edu and the wellesley.edu email addresses of the two researchers so that they can request for assistance with technical issues, if any, with the online study.

7.3.3 Instruments

Pre-survey

In a web interface, participants will be presented with the survey questions. Their responses will be collected through the form submission functionality embedded in the interface.

Following the pre-survey, participants will go through a sequence of sections, each
involving a number of tasks, in the same web interface. For the first round of study, there will be 9 sections with a total of 40 tasks of various kinds as described below. Not all tasks will appear in all sections.

**Tutorial**

A brief (ranging between 2-6 mins) self-paced tutorial video that covers the background information of the section will be presented at the beginning of each section (Figure 7-1). A summary of the key concepts covered in the tutorial video will be available as a document linked from the interface (Figures 7-2 and 7-3). There will also be a page that contains all video and document information linked from all tasks throughout the study, to which the participants can refer back anytime during each corresponding section.

For the first round of study, no information is collected for this kind of task. In later rounds, the researchers might collect information on whether a participant watched the video, how much time they spent watching the video, whether they consulted the written documentation, and how much time they spent consulting the written documentation.

For the first round of study, I developed all the tutorial slides and videos based on these slides that totaled 30 minutes. Many slides and videos were developed in an iterative process, incorporating feedback from Prof. Turbak and students in the TinkerBlocks group.

**Translation Selection**

The purpose of this kind of task is two-fold: (1) to test the subject’s understanding of the translation rules and (2) to help the subject learn the translation rules better for the more challenging subsequent Translation Typing tasks. In each task of this type, the participants will be presented with a valid cluster of App Inventor blocks and given a list of Venbrace text translations, some of which are valid, and others of which are invalid, given that Venbrace’s flexibility allows many different valid translations which might involve abbreviations (Figure 7-4). The subjects are then asked to
Tutorial: Simple Math Expressions

Please watch the tutorial video before continuing to the next page. Most tutorials are followed by tasks related to the tutorial.

Here is an optional document summarizing the key points in the video that some people might find helpful.

(Optional) Use the area below to give feedback to the study team about this tutorial. E.g.: Were any concepts confusing? Do you have ideas for improving it?

Continue

Figure 7-1: Example tutorial page with embedded video and link to summary document. Every page also has an optional feedback area near the bottom of the page above the Continue button; this area and button will be omitted in images of subsequent pages.
Overview: Expressions

Math expression:

String expression:

Logical expression and boolean expression:

Local variable getter:

In Venbrace, all expressions are translated using a pair of parens with stuff in between:

( ...

Simple Math Expressions

1. Blocks with a plug on the left are "expressions" and you should translate the corresponding text with a single pair of matched parentheses.

(5)

(-42)

2. All parentheses have to be balanced. The code editor used in this study for translation purposes will help you determine the balance of parentheses.

(17) // balanced
(17 // missing a right paren

17) // missing a left paren

3. Mathematical operations are expressions, leading to nested parenthesized text. This expression block for subtraction below has two subexpressions, 17 and 8.

Figure 7-2: Example summary document for Simple Math Expressions tutorial (page 1).
both \((5) + (23)\) and \((5 + 23)\) are acceptable.

7. The computer views the text as a sequence of tokens. For example, this expression \((17 - 8)\) has five tokens: \((, 17, -, 8, )\)

8. Spaces are ignored between tokens, but NOT within atomic expressions. All of \((17 - 8)\), \((17-8)\), \((17-8)\), \((17 - 8)\), \((\quad 17\quad -\quad 8\quad )\) ... are legal

\(17\) and 17 are legal, but \((1\ 7)\) and \(1\ 7\) are NOT legal

9. Venbrace text has to be written in exactly the same way as its corresponding App Inventor block(s).

\(2 + 7\) can be written as \((2 + 7)\) but \(7 + 2\) is NOT legal

\(7 + 2\) can be written as \((7 + 2)\) but \(2 + 7\) because they are DIFFERENT blocks!
indicate the validity of each of the translations (via radio buttons for Valid/Invalid) using the information from the tutorial video and documentation (Figure 7-5). When they are done, they will submit their response by clicking a **check my answers** button below the list (Figure 7-6).

Form validation will be used to guarantee that the subject has selected an answer (Valid or Invalid) for all translations before any other action is taken. Once all answers have been chosen, user answers will be marked correct or incorrect, and explanations for the correctness of each answer will appear next to each of the translations. Symbols and font colors will be used to indicate the correctness of the participant’s answers to each of the translations. The participant is prevented from resubmitting another answer for any translation once all translation responses are submitted and the explanations are revealed; instead, they will be prompted to continue with the remaining part of the study. Subsequently, we will collect the Valid/Invalid answers submitted for this kind of tasks.

![Selection Task: Nested Expressions](image)

*Selection Task: Nested Expressions*

For each text notation below, indicate whether it is a **valid or invalid** Venbrack translation for the above blocks. (You may wish to view the page of all tutorials while doing this and subsequent tasks.)

<table>
<thead>
<tr>
<th>Translation</th>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + (4 * 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((1 + 4) * 7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1+(4*7))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1+4*7)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1+4*7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(((1) + (4 * (7)))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>((4 * 7) + 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 + (7 + 4))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 + (1 + 7)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once you have selected an answer from each row, click **Check My Answers** to check which answers are right and wrong.

**Check My Answers**

Figure 7-4: Example selection task, before selecting answers.
Selection Task: Nested Expressions

For each text notation below, indicate whether it is a valid or invalid \textit{Venbrack} translation for the above blocks. (You may wish to view the \texttt{page of all tutorials} while doing this and subsequent tasks.)

<table>
<thead>
<tr>
<th>Translation</th>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 + (4 * 7)$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} + 4) * 7$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} - (4*7))$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} - 4*7)$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$1 - 4*7$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} - (4 * (7)))$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} - (4 * 7))$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$(\texttt{cl} + (7 * 4))$</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>$4 (1 + (7 * )$</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Once you have selected an answer from each row, click \textbf{Check My Answers} to check which answers are right and wrong.

Check My Answers

Figure 7-5: Example selection task, after selecting answers but before checking them.

Selection Task: Nested Expressions

For each text notation below, indicate whether it is a valid or invalid \textit{Venbrack} translation for the above blocks. (You may wish to view the \texttt{page of all tutorials} while doing this and subsequent tasks.)

<table>
<thead>
<tr>
<th>Translation</th>
<th>Valid</th>
<th>Invalid</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 + (4 * 7)$</td>
<td>✔</td>
<td>✔</td>
<td>Outer parens for + block are missing</td>
</tr>
<tr>
<td>$(\texttt{cl} + 4) * 7$</td>
<td>✔</td>
<td>✔</td>
<td>Top-level operation should be +, not *</td>
</tr>
<tr>
<td>$(\texttt{cl} + 4*7)$</td>
<td>✔</td>
<td>✔</td>
<td>Has correct paren structure, and spaces aren't necessary</td>
</tr>
<tr>
<td>$(\texttt{cl} - 4*7)$</td>
<td>✔</td>
<td>✔</td>
<td>Paren for * are missing; precedence rules in math do not apply</td>
</tr>
<tr>
<td>$(\texttt{cl} - (4 * (7)))$</td>
<td>✔</td>
<td>✔</td>
<td>Paren for * and + are missing; precedence rules in math do not apply</td>
</tr>
<tr>
<td>$(\texttt{cl} - (4 * 7))$</td>
<td>✔</td>
<td>✔</td>
<td>A single pair of parens around any number is OK</td>
</tr>
<tr>
<td>$(\texttt{cl} + (7 * 4))$</td>
<td>✔</td>
<td>✔</td>
<td>Order of + operands matters</td>
</tr>
<tr>
<td>$4 (1 + (7 * )$</td>
<td>✔</td>
<td>✔</td>
<td>Order of * operands matters</td>
</tr>
<tr>
<td>$4 (1 + (7 )$</td>
<td>✔</td>
<td>✔</td>
<td>This crazily formatted sequence of tokens is correct as far as the computer is concerned, but is very hard for humans to read, so should be avoided in practice</td>
</tr>
</tbody>
</table>

Once you have selected an answer from each row, click \textbf{Check My Answers} to check which answers are right and wrong.

Check My Answers

After studying the explanations, click on the \textbf{Continue} button below.

Figure 7-6: Example selection task, after checking answers.
Translation Typing

The purpose of this kind of task is (1) to evaluate the design choices of a selected number of App Inventor concepts for Venbrace and (2) to help the subject learn the translation rules better with the help of a syntax-highlighted code editor and a basic error reporting system.

In each task of this kind, the participants will be presented with a valid cluster of App Inventor blocks, and then asked to convert the blocks into textual Venbrace code in a syntax-highlighted code editor, using the information from the tutorial video and documentation (Figure 7-7). When they are done, they will submit the response by clicking the Translate! button below the editor.

![Translation Task: Nested Math Expression 1](image)

Figure 7-7: Example Translation Typing Task at beginning.

If the participant submits a valid translation, a congratulations message will be displayed (Figure 7-8). The subject will have the option to read the solution for any alternative solutions, or to continue with the rest of the study.

If the participant enters a textual translation that is incorrect, an error message containing information indicating incorrect aspects of the translation will appear, and the participant will have the option to either redo the task or give up (Figure 7-9). If they indicate that they are redoing the task, they will be allowed to edit their translation in the code editor and submit a modified or new translation if they have not reached the maximum number of attempts. If they indicate that they are giving up on the task, or they have submitted the maximum number of incorrect responses,
Figure 7-8: Example Translation Typing Task when a correct translation is submitted.

Figure 7-9: Example Translation Typing Task when an incorrect translation is submitted.
they will be presented prompted to read the correct answer and to continue with the rest of the study (Figure 7-10). For the first round of study, participants are allowed up to five (5) attempts for each Translation Typing task.

Figure 7-10: Example Translation Typing Task where the maximum number of attempts has been reached, after which the subject has clicked the Show me the solution button.

**Error Detection and Reporting**

To determine the correctness if a subject’s response to a Translation Typing task, I applied the Token Accuracy Map (TAM) approach proposed by Stefik [SSI3] and Daleiden [Dal16]. Each Translation Typing task has one and only correct VENBRACE translation in its most verbose form. Within the study page for each Translation Typing task, the most verbose solution to the task is hidden in an HTML field and converted to tokens by the VENBRACE lexer once the page is loaded. After a subject submits a response through the code editor, their response will be similarly tokenized by the VENBRACE lexer. I first expand the response tokens to their most verbose form by filling in omitted tokens at correct positions, then compare the expanded response tokens to the solution tokens using TAM.
The TAM algorithm is based on the Needleman-Wunsch (NW) \cite{Dal16} sequence alignment algorithm on token arrays. The NW algorithm is a dynamic programming algorithm widely used in DNA sequencing that tries to align two strings (broken into token arrays) based on their longest common substring components. In my case, the token arrays are compared and aligned based on the literal content of the tokens. To find the optimal alignment, the tokens are either shifted (if a match occurs with an offset) or their positions remain unchanged (if there is a match, or a match cannot be made even with shifting). For a token that will be shifted to meet a match, the target position will be filled in with a null token to indicate the shift. Simultaneously, a 2-D matrix containing scores calculated for each token in each way of attempted alignment will be generated. An optimal alignment can be further identified by backtracing the score matrix from its last cell. The result of the TAM algorithm is two possibly shifted token arrays with same size, the subject’s response and the solution, that are optimally aligned from backwards. For example, the most verbose VENBRACE translation for the block below is \((10 \ ^\ (3)\)\). Figure \ref{fig:alignment} illustrates why a correct, abbreviated translation \((10 \ ^\ 3)\) is determined as correct. Since the alignment result renders two identical token arrays, the algorithm determines that the most verbose version of the submitted response matches the solution.

![Alignment Diagram](image)

<table>
<thead>
<tr>
<th>Step</th>
<th>Response</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tokenization</td>
<td>[,(, 10, ^, 3, ,)]</td>
<td>[,(, (10, ), ^, (3, ), ,)]</td>
</tr>
<tr>
<td>Expansion</td>
<td>[,(, (10, ), ^, (3, ), ,)]</td>
<td>[,(, (10, ), ^, (3, ), ,)]</td>
</tr>
<tr>
<td>Alignment</td>
<td>[,(, (10, ), ^, (3, ), ,)]</td>
<td>[,(, (10, ), ^, (3, ), ,)]</td>
</tr>
</tbody>
</table>

Figure 7-11: Figure illustrating how a response is determined as correct.

If the user submits an incorrect translation \((10 \ ^\ (3)\), Figure \ref{fig:incorrect} illustrates why it is determined as incorrect. The two resulted token arrays after alignment are obviously not identical. The code editor thus decides that the subject’s response is invalid whenever the alignment algorithm renders two nonidentical token arrays.
Five types of errors in the user’s response will be detected and reported to the user through the interface: the number of missing braces and their types, the number of unexpected braces and their types, missing non-brace tokens, unexpected non-brace tokens, and any token expected in the solution that is not in the correct position. A null token in the response token array after alignment indicates a possibly missing token, while null in the solution token array after alignment indicates an unexpected token at that position, regardless of the token type. If the response contains a token that is required by the solution, the error checking mechanism also makes sure that it appears at a correct position. For example, for the example above, although a response such as \((3 ^ 10)\) provides all required tokens, tokens 10 and 3 appeared at wrong positions, and the response will thus be considered as incorrect.

**Logging Keystrokes and Counting Attempts**

The code editor used in this kind of tasks is also equipped with keystroke-logging functionality, which listens and records any typing behavior and the key pressed or cursor moving behavior and the cursor position, as well as the timestamps of such behaviors. It also counts and records the number of attempts a subject has tried on a task. For each attempt of a task of this kind, the start and finish timestamps, the subject’s submitted translations (as raw text and as tokens processed by the Venbrace lexer) as well as the their keyboard editing behaviors through the code editor will be collected.

**Preference Typing**

The purpose of this kind of tasks is to see how subjects would translate blocks whose translation rules have not been explained by a previous tutorial. They will help us
decide which translations are most natural for blocks whose translation rules are not obvious, such as the undecided abbreviations described in Section 4.2.

In each task of this kind, the participants will be presented with a valid cluster of App Inventor blocks and told they have not been given rules for translating the blocks. They will be asked to translate the blocks into textual VENBRACE code in a regular text area (not a syntax-highlighted code editor as used in the Translation Typing tasks). Participants will type in whatever they think to be a good translation and submit the response by clicking a “submit” button below the editor. Figure 7-14 gives an example of a Preference Typing task.

Form validation will be used to ensure that the text area contains a nontrivial answer: in the first round of study, the shortest possible valid solution for a Translation Typing task involves 6 characters, so any solution with fewer than 6 characters will be considered as being too short. Re-submission will be required until the text area contains a nontrivial answer. Other than form validation, they will not be given a second chance of submission, nor will they be given any feedback on their translation. Clicking on the submit button will bring them to the next page in the study. The only data collected for this task will be the final translation submitted.
Preference Ranking

Tasks of this kind intend to see how subjects rank various options to translate blocks whose translation rules have not been explained by a previous tutorial, and whether they can come up with better options (if any). They will help us decide which translations are most natural for blocks whose translation rules are not obvious, such as the ones described in Section 4.2.

In each task of this kind, the participants will be presented with a valid cluster of App Inventor blocks and told they have not been given rules for translating the blocks. They will be given a list of suggested translation options, and will be asked to rank each option on a 5-point Likert scale from Strongly Dislike (1) to Strongly Like (5) using radio buttons. Optionally, they are encouraged to provide their own translation in the text area below the ranking list. They will submit their rankings and their ideas (if any) all at once by clicking on a submit button. Figure 7-15 gives an example of a Preference Ranking task.

Form validation will be used to ensure that all options have been ranked; re-submission will be required until all options have been ranked. Like in the Preference typ
Preference Ranking Task: Degree Conversion

This is a task for which there is no correct answer. There are various options for translating the blocks on this page, but we don’t know which is best. We need input from you and the other study participants to decide which is best. (You may wish to view the page of all tutorials while doing this and subsequent tasks.)

Below are some possible options for translating the above blocks. For each row below, indicate how much you like the option in that row on the scale that goes from Strongly Like to Strongly Dislike.

**Note:** In this and many other preference tasks, we’re looking for your input on a general issue using a particular example. In this task, we want to know how to handle the many App Inventor blocks that begin with multiple words separated by spaces. The goal is to have a strategy for handling such blocks that can be expressed by a simple rule that can be followed in all cases, rather than having different special cases for particular blocks.

<table>
<thead>
<tr>
<th>Translation</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert%degrees%to%radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert%degrees%to%radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert%degrees%to%radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>(convert degrees to radians 60)</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

(Optional) If you have an idea for an option you like better than the above choices, enter it below.

**Note:** While there are much nicer special-case translations like (degrees -> radians 60) in this particular case, we’re really looking for translations that handle the general problem of blocks that begin with multiple words separated by spaces.

Figure 7-15: Example preference ranking task.
Typing tasks, clicking on the submit button will bring them to the next page in the study. The data collected for this task will be their rankings for each translation option and any optional suggestion they may have entered in the text area.

Post-study Survey

At the end of the study, participants will be asked to give feedback on the design of Venbrace and their experience of using the language through Likert-style ranking questions, as well as open-ended suggestions on future improvements to the language and/or to the user study through text areas. Their responses will be similarly collected through the form submission functionality embedded in the interface.

Other types of data that will be collected throughout the study include (1) the starting and ending timestamps for each task, and (2) optional open-ended feedback for each task, if any, submitted by the participants through an attached text area in each task.

In general, the online survey is constructed in a way that attempts to prevent the subject from (1) going back to previous tasks to redo them or (2) skipping through tasks without doing them. To implement (1), each submitted response for all tasks is tagged with its completion timestamp, and only the response of one task that has the earliest completion timestamp will be kept. In fact, we keep track of the current task of each subject on the server used for this study. Whenever the subject accesses the study via a web page request, they are brought to the first task in the task sequence for which no response has been recorded. If they use the Go Back button in their browser to go back to a previous study page or have old study pages accessible in other browser tabs/windows, any resubmission of the old tasks is ignored. To realize (2), the survey is built with care such that the participants are prevented from going to future tasks before they finish the current task.
7.3.4 Data Overview and Analysis

Out of the 19 people who signed up for the first round of study, 17 completed the study and submitted valid responses, which I automated quantitative analysis through Python scripts and conducted qualitative analysis through manual inspection. When necessary, I also manually examined the quantitative results to look for data trends.

Quantitative Analysis

To quantify subjects’ familiarity with BPEs other than App Inventor and experience with text languages (Pre-survey), average number of attempts spent on a task (Translation Typing), overall ranking an option received (Preference Ranking), subjects’ evaluations for the Venbrace language and opinions for having a text language in App Inventor (Post-survey), I developed Python scripts that convert related data into data frames to conduct statistical analyses.

For the Translation Typing tasks, I also evaluated the Venbrace tokens of current design by calculating the overall accuracy of each token for each task using the results rendered by the TAM algorithm. For each task, I used the final attempts submitted by the subjects to determine the rate at which a token was accurately used by the subjects. A low accuracy of a token strongly suggests redesigning Venbrace to address this issue.

Qualitative Analysis

For some parts of the study, I also manually examined specific responses. For each Translation Typing task, for subjects who failed to submit a correct response in their last attempt, I reviewed such incorrect last attempts to record common mistakes, which could also indicate possible language redesigns.

To classify user preference for Venbrace translations of blocks that are not covered in the tutorials and the rest of the study (Preference Ranking and Preference Typing), I looked at subjects’ responses to discover commonalities. Some subjects submitted their own Venbrace translations for certain blocks in the Preference
Ranking tasks as "better options," and I reviewed them to look for any common ideas as well.

As for the Post-study Survey, subjects were asked whether they would like to have a text mode in App Inventor, and I associated their responses and reasons with the information they provided in the Pre-survey to look for potential correlations. I reported any suggestions on improving the Venbrace language and the design of the study submitted through the Post-survey as well.

7.3.5 Pilot Study

Some informal testing of the online study interface was completed upon the release of the first round of study. Students in the TinkerBlocks group, not knowing the study materials even after helping out with implementing the interface, took the alpha version of the study and returned with valuable feedback. They suggested improvements to the tutorial materials, identified a number of bugs in the system, and requested a series of features to enhance the overall interaction with the interface. A member from the MIT App Inventor team took the beta version of the study before the formal release, and the feedback was generally positive.

7.4 Results

7.4.1 Pre-survey

Demographics

The study involved 17 students (10 female, 6 male, and 1 non-binary) with an average age of 19.41 years old. 9 are high schoolers currently taking the AP CSP course, 6 are college students, and 2 are graduate students. (The original data showed that there were 3 graduate students, but 1 undergrad marked themselves as graduate by mistake). They all go to school in Massachusetts, USA. On average, the subjects have less than 10 months of experience with App Inventor: 3 participants reported no more
than 4 months of experience, 10 participants reported 5-9 months of experience, no one reported with 10-20 months of experience, and 4 reported more than 20 months of experience.

**Familiarity with App Inventor**

The subjects reported familiarity with almost all common App Inventor program components, except the following: generics, declarations/calls of procedures that do return a value (expression), declarations/calls of procedures that have one or more parameters, local variable declarations and usage with local variable initialization blocks, TinyDB, and CloudDB. See Table 7.1 for more information.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Mean</th>
<th>Median</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditionals</td>
<td>2.00</td>
<td>2.0</td>
</tr>
<tr>
<td>Global Variables</td>
<td>1.94</td>
<td>2.0</td>
</tr>
<tr>
<td>Loops</td>
<td>1.76</td>
<td>2.0</td>
</tr>
<tr>
<td>Lists</td>
<td>1.76</td>
<td>2.0</td>
</tr>
<tr>
<td>Screen Components</td>
<td>1.59</td>
<td>2.0</td>
</tr>
<tr>
<td>Procedure - No Return</td>
<td>1.53</td>
<td>2.0</td>
</tr>
<tr>
<td>Local Variable Getters</td>
<td>1.53</td>
<td>2.0</td>
</tr>
<tr>
<td>Procedure - No Params</td>
<td>1.41</td>
<td>2.0</td>
</tr>
<tr>
<td>Screens</td>
<td>1.41</td>
<td>2.0</td>
</tr>
<tr>
<td>Generics</td>
<td>1.35</td>
<td>2.0</td>
</tr>
<tr>
<td>Procedure - Return</td>
<td>1.35</td>
<td>1.0</td>
</tr>
<tr>
<td>Procedure - With Params</td>
<td>1.29</td>
<td>1.0</td>
</tr>
<tr>
<td>TinyDB</td>
<td>1.12</td>
<td>1.0</td>
</tr>
<tr>
<td>Initialize Local Variables</td>
<td>1.06</td>
<td>1.0</td>
</tr>
<tr>
<td>CloudDB</td>
<td>0.88</td>
<td>1.0</td>
</tr>
</tbody>
</table>

The numbers in the table refer to the subjects’ degree of familiarity with an App Inventor programming concept, where 0 is Unfamiliar, 1 is Somewhat Familiar, and 2 is Very Familiar.

**Familiarity with other BPEs**

Most participants have no experience with BPEs other than App Inventor except for Scratch (about 70% reported prior experience) and Code.org (about 41% with
prior experience). A few of them have used Alice, geojson, and/or CodeHS; note that none of these BPEs are DMPEs, i.e., BPEs with text mode. Table 7.2 details the breakdown of subjects’ prior experience with such BPEs.

Table 7.2: Familiarity with Other BPEs

<table>
<thead>
<tr>
<th>BPE</th>
<th>&gt;10 Programs</th>
<th>5-10</th>
<th>&lt;5</th>
<th>No Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Code.org</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Alice</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>geojson</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>CodeHS</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Looking Glass</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>MakeCode</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Pencil Code</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Snap!</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>17</td>
</tr>
</tbody>
</table>

The integers in the table refer to number of subject responses in each category.

Experience with Text Languages

11 out of 17 participants reported prior experiences with text languages. Among the reported text languages, Python, Java, and JavaScript are the most popular. See Table 7.3 for more details.

7.4.2 Translation Typing

Participants were allowed to submit to five (5) attempts for each Translation Typing task before continuing onto the next task. Some participants were able to accomplish a valid response within five attempts, while others used up all five attempts or gave up midway without a correct answer. Throughout the study, those who succeeded managed to come up with a correct answer within three (3) attempts on average, and those who failed often spent 3 or more attempts. Table 7.4 details the relationship between the correctness of the last attempt submitted and the average number of attempts participants spent on each task.
### Table 7.3: Familiarity with Text Languages

<table>
<thead>
<tr>
<th>Language</th>
<th>Total</th>
<th>&gt;10 Programs</th>
<th>5-10</th>
<th>&lt;5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>10</td>
<td>6</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Java</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>JavaScript</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C#</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Matlab</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>APL</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bash</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C++</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Delphi</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Go</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HTML</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pascal</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Processing</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Racket</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Scala</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Swift</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The integers in the table refer to number of subject responses under each category.

### Table 7.4: Average attempts spent on a task, based on whether the last attempt was correct.

<table>
<thead>
<tr>
<th>Task</th>
<th>Attempts (correct)</th>
<th>Attempts (incorrect)</th>
</tr>
</thead>
<tbody>
<tr>
<td>05-TR-Simple Expression</td>
<td>1.53 (100%)</td>
<td>-</td>
</tr>
<tr>
<td>09-TR-Nested Math Expression 1</td>
<td>1.31 (94%)</td>
<td>4.00 (6%)</td>
</tr>
<tr>
<td>10-TR-Nested Math Expression 2</td>
<td>1.56 (94%)</td>
<td>4.00 (6%)</td>
</tr>
<tr>
<td>11-TR-Nested Math Expression 3</td>
<td>1.47 (88%)</td>
<td>4.00 (12%)</td>
</tr>
<tr>
<td>15-TR-Expression with Booleans and Strings</td>
<td>1.60 (59%)</td>
<td>4.00 (41%)</td>
</tr>
<tr>
<td>21-TR-Simple Statement</td>
<td>1.33 (53%)</td>
<td>4.38 (47%)</td>
</tr>
<tr>
<td>24-TR-Nested Statement</td>
<td>1.20 (29%)</td>
<td>3.58 (71%)</td>
</tr>
<tr>
<td>29-TR-Event Handler</td>
<td>2.38 (47%)</td>
<td>3.56 (53%)</td>
</tr>
<tr>
<td>32-TR-SizeUp Procedure</td>
<td>1.42 (71%)</td>
<td>2.60 (29%)</td>
</tr>
<tr>
<td>33-TR-ConcatString Procedure</td>
<td>1.42 (71%)</td>
<td>4.40 (29%)</td>
</tr>
</tbody>
</table>

The "correctness" depends on the last attempt submitted by a subject for a task.
Among all the unsuccessful last attempts, I identified three main types of common mistakes committed by the subjects throughout all Translation Typing tasks: **using braces, referencing variables, and handling helper words.**

**Using braces**

In expression-only Translation Typing tasks where multiple expressions were involved, whether or not the expression blocks are nested, participants often missed the first or last required parenthesis, and they sometimes misplaced the inner parentheses.

The various types of braces reserved for different blocks in *Venbrace* started to confuse participants when statements and declarations were involved: some used the wrong type of braces, while some others stuck to parentheses (used for expressions) when translating non-expression blocks. The outermost pair of braces were sometimes missing, too. Some participants forgot about the required pair of parentheses for the non-atomic conditional expression following *if* or *else if* in a conditional block as well. The use of curly braces for statement blocks didn’t seem well-perceived by some participants, either: for a sequence of statements that require a sequence of pairs of curly braces, they would use only one pair of curly braces for all statements, or mistakenly nest the curly braces.

**Referencing Variables: the Getters**

Only two forms of translations for a getter block are accepted: the verbose form with both the *get* keyword and the required pair of parentheses for the entire block (e.g., *(get a)*), or the abbreviated form that only comes with the variable name itself (e.g., *a*). Participants who failed to compose a valid translation either placed the *get* keyword outside of the pair of parentheses for a getter block (e.g., *get(a)*), or they attempted to translate a getter block in its most verbose form but forgot about the required pair of parentheses (e.g., *get a*).
Translating Helper Words

Participants tended to omit required keywords in their VENBRACE translations when there were multiple helper words in a task. For example, when translating a setter block that is part of another block of a different type, some participants forgot about the set keyword or the to keyword, but not both. The to keyword at the beginning of a procedure declaration block was sometimes omitted, too. In a conditional statement block, then after else if was missed by some participants, while the then after if in the same block was never missed.

The TAM analysis of each task corresponds to the common mistakes noted above. The following example (Figure 7-16) is the fifth Translation Typing task in the study which contains a nested expression.

<table>
<thead>
<tr>
<th>Block</th>
<th><img src="image" alt="Solution" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution</td>
<td>[((, , not, (, Button1.Enabled, ), ), or, (, (, length, (, &quot;cat&quot;, ), ), &gt;=, (, get, num, ), ), )]</td>
</tr>
</tbody>
</table>

Figure 7-16: The fifth Translation Typing task, and its solution tokens (verbose).

The token accuracy analysis for this task renders the result shown in Figure 7-17. The result is displayed as an array of tuples, in which the first element of a tuple is a solution token, while the second element of a tuple is its overall accuracy, i.e., the ratio at which a token was correctly represented and positioned among all submitted last attempts for this task. The result shows that the first left parenthesis has a significantly lower accuracy (0.82), because it was omitted in many failed last attempts, which corresponds to the observation that the first required parenthesis for a nested expression was often missed.

For token accuracy analysis results of other Translation Typing tasks, see Appendix 3 for more details.
7.4.3 Preference Ranking

In the Preference Ranking tasks, we asked the subjects for their preferences for blocks of which the VENBRACE translations are not yet finalized (as described in Section 4.2), including complicated math expression blocks that contain multi-word key phrases, list blocks, color blocks, event handlers with parameters, procedure calls with arguments, and global variables (declarations, getters, and setters). We quantified users preferences using a Likert scale (5 - Strongly Like, 1 - Strongly Dislike). We also collected preferences on alternative translations of blocks already covered in the current VENBRACE design using the same Likert scale. Meanwhile, we gathered subjects’ own translations for the blocks in the tasks, if any.

In general, for complicated math expression blocks, list blocks, color blocks, procedure call blocks with arguments, and all blocks related to global variables, the top ranked option involves the most verbose form of translation with textual content identical to the block and receives a preference score greater than 4 on average. Options that involve symbols which represent keywords that are intended to provide easier, more abbreviated translations are, unexpectedly, not as welcomed as the verbose translations. For example, the symbol $ I proposed to represent global variables only received an average preference score of 2.82, which falls in the range between "2 - Somewhat Dislike" and "3 - Meh" (Table 7.5).

No translation for event handlers with built-in parameters seemed to be significantly favored by the majority of participants, however (See Table B.7).

In terms of shorthands for existing VENBRACE translations, the single equal sign = was especially welcomed by the participants (53% voted "Strongly Like", 23.5% voted
Table 7.5: Preference ranking for Getting a Global Variable

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(get global sum)</td>
<td>4.29</td>
<td>4.0</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(global sum)</td>
<td>4.29</td>
<td>4.0</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(getGlobal sum)</td>
<td>3.47</td>
<td>3.0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(&lt;get global&gt; sum)</td>
<td>3.41</td>
<td>3.0</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(get_global sum)</td>
<td>3.24</td>
<td>3.0</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>('get global' sum)</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$sum</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>@sum</td>
<td>2.71</td>
<td>3.0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(getglobal sum)</td>
<td>2.65</td>
<td>3.0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>($get global$ sum)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(#get global# sum)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>#sum</td>
<td>2.41</td>
<td>3.0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(</td>
<td>get global</td>
<td>sum)</td>
<td>2.41</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>%sum</td>
<td>2.18</td>
<td>2.0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

"Somewhat Like"), but not the left arrow sign <- that is part of the initial design. Most alternatives submitted by the participants were too creative to be acceptable alternatives (i.e., they did not completely comply with the principles that guided the current design for VENBRACE), but, surprisingly, three subjects proposed using the double equal sign == as an abbreviation for variable setters.
### Table 7.6: Preference ranking for Event Handler with Parameters

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
</table>
| [when Canvas1.TouchDown <x> <y> do 
{set Label1.Text to (join x "," y)}] | 3.76 | 4.0 | 4 | 6 | 6 | 1 | 0 |
| [when Canvas1.TouchDown x y do 
{set Label1.Text to (join x "," y)}] | 3.59 | 4.0 | 5 | 5 | 2 | 5 | 0 |
| [when Canvas1.TouchDown |x| |y| do 
{set Label1.Text to (join x "," y)}] | 3.47 | 3.0 | 4 | 4 | 5 | 4 | 0 |
| [when Canvas1.TouchDown do 
{set Label1.Text to (join x "," y)}] | 3.06 | 3.0 | 2 | 5 | 4 | 4 | 2 |
| [when Canvas1.TouchDown #x #y do 
{set Label1.Text to (join x "," y)}] | 2.82 | 3.0 | 2 | 1 | 8 | 4 | 2 |

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

### 7.4.4 Preference Typing

The Preference Typing tasks opened up more room for creativity and exploration, allowing the participants to propose their ideal text translations for blocks that do not yet have *Venbrace* translations. Even so, I was able to identify the following patterns after examining all responses by hand:

- Rather than coming up with creative abbreviated translations, most participants chose to keep the text translation of a block identical to the textual content within the block, including any use of keywords. For example, 12 out of 17 participants kept the keywords *test* within the text representation for a while block, even if the keyword appears to be redundant. For a long built-in operation like *add items to list* which consists of multiple keywords in a sequence, 10 out of 17 participants chose to retain all the keywords in the text.
Table 7.7: Preference ranking for Infix Abbreviation for Setters

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>{num = 3}</td>
<td>4.18</td>
<td>5.0</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>{num := 3}</td>
<td>3.35</td>
<td>3.0</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>{num -&gt; 3}</td>
<td>3.29</td>
<td>4.0</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>{num &lt;- 3}</td>
<td>2.88</td>
<td>3.0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>{num =: 3}</td>
<td>2.65</td>
<td>3.0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>{num &lt;-&gt; 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>{num :: 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>{num :=: 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>{num -- 3}</td>
<td>2.41</td>
<td>2.0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>{num -=- 3}</td>
<td>2.00</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

• For concepts that were introduced earlier, most participants still followed the rules. In one of the tutorials, we taught the participants that parameters are translated to VENBRACE without any modification; if there are multiple parameters, they should be translated one following another according to their original order. On average, 14 out of 17 participants recognized this concept and applied it to translating unfamiliar blocks that involve one or more parameters.

• I observed more diversity in responses to translating blocks that were covered in the Preference Ranking tasks, with some participants remembering and re-
applying ideas introduced in those tasks. For example, we only introduced as one of the abbreviated symbols for global variable references, and 2 participants continued to use in a Preference Typing task that involved translating global variables.

For the complete analysis for data of this type of tasks, see Appendix B.

7.4.5 Post-survey

Similar to processing other Likert scales, we quantified subjects’ evaluations for features of Venbrace (1 - Very Easy, 5 - Very Hard). The majority of the participants found Venbrace OK to use, with translating expressions perceived as the easiest. Most participants had little trouble matching the built-in keywords in text, however verbose the keywords were. Participants rated coming up their own text translation of a given assembly of blocks as the most difficult, with a third of participants ranked it within the "hard" (Fairly Hard/ Very Hard) range. Table 7.8 details the results for the evaluations.

When asked to reflect on their experience of using Venbrace, 16 out of 17 participants agreed that Venbrace represents App Inventor precisely, 14 believed that having a text language like Venbrace in App Inventor would be helpful for building programs, and 12 reported that they would be more likely to edit their App Inventor programs using such a text language if it were available. Overall, the feedback was mostly positive.

15 out of 17 participants reported willingness to have a text language such as Venbrace in App Inventor in the future, 4 of whom had no prior experience with text programming. They believed that having a text language in App Inventor would help with reading and visualizing the structure of a program (3), reducing the viscosity of creating blocks and making changes (5), improving programming efficiency especially for those who have had some experience with App Inventor (6), and bridging the gap between blocks and text programming (4). For the two participants who disagreed,
Table 7.8: Post-survey: VENBRACE Evaluation

<table>
<thead>
<tr>
<th>Statement</th>
<th>Avg</th>
<th>Very Easy</th>
<th>Fairly Easy</th>
<th>OK</th>
<th>Fairly Hard</th>
<th>Very Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating mathematical expressions from blocks to text</td>
<td>1.53</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Translating non-mathematical expressions from blocks to text</td>
<td>2.18</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Matching the built-in keywords in text</td>
<td>2.29</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Translating statements from blocks to text</td>
<td>2.35</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Matching the parentheses/ braces/ brackets in text</td>
<td>2.59</td>
<td>2</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Translating event handlers from blocks to text</td>
<td>2.65</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Translating variable declarations from blocks to text</td>
<td>2.71</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Translating variable references from blocks to text</td>
<td>2.71</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Coming up with your own text translation of a given assembly of blocks</td>
<td>3.24</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

The numbers in the Avg column are evaluation scores according to a Likert scale, where Very Easy counts as 1 and Very Hard counts as 5. The numbers in other columns refer to the number of responses submitted under each column header.

they were both high schoolers with no prior experience in text programming, stating that a text language like VENBRACE is not better than the current App Inventor blocks (1) and will be harder to learn (1).

Some participants also suggested future improvements to the VENBRACE language, mostly in the following aspects:

- **Braces**: Using and balancing three types of braces could be confusing sometimes. Some participants suggested allowing braces to be optional under more circumstances (like the atomic expressions), while others would love help from the system with balancing the braces.

- **Symbols, keywords, and identifiers**: Two participants wished there were better ways handling long sequences of keywords, as they might become tedious sometimes (e.g. if ... then ... else if ... then ...). One participant encouraged us to use different symbols to represent and distinguish local and global variables, while another participant suggested that we should not use any
Table 7.9: Evaluation: Text in App Inventor

<table>
<thead>
<tr>
<th>Statement</th>
<th>Strongly Agree</th>
<th>Somewhat Agree</th>
<th>Unsure</th>
<th>Somewhat Disagree</th>
<th>Strongly Disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Venbrace language precisely represents the App Inventor blocks language that I am already familiar with.</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Having a textual language like Venbrace in App Inventor would be helpful for building programs.</td>
<td>11</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>If a textual language like Venbrace were available in regular App Inventor, I would be more likely to make changes to my apps using such a language.</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

The numbers in the columns refer to the number of responses submitted under each category (column header).

I reflect on the findings in the following ways: I first relate what I discovered to the design principles, then propose possible improvements to solve the conflicts, and conclude by speculating on what the next design iteration may be perceived if it is tested in future rounds of studies.

7.5 Discussion

Since the design of Venbrace is led by design principles described in Chapter 3, I reflect on the findings in the following ways: I first relate what I discovered to the design principles, then propose possible improvements to solve the conflicts, and conclude by speculating on what the next design iteration may be perceived if it is tested in future rounds of studies.

7.5.1 Evaluation of Current Design

Text Mode in App Inventor

The majority of our study respondents expressed great interest in seeing a text language for App Inventor and actually using such a language to edit their App Inventor programs in the text mode. As they suggested in the Post-survey, having a text
language and a text mode in App Inventor improves program navigation, block creation and editing, and programming efficiency especially for experienced App Inventor users, which exactly addresses the current challenges facing App Inventor users (as described in Section 1.2) and matches with the motivations driving the design and creation of Venbrace (described in Section 1.4.2).

Having a text language and a text mode in App Inventor turns App Inventor into a DMPE. Some respondents considered App Inventor along with the embedded text language to further bridge the gap between blocks and text programming if App Inventor eventually becomes a DMPE. Such an evaluation corresponds with the findings of Blanchard et al. [BGMA19] and the motivation behind existing efforts on incorporating text in BPEs [BBDP15, HN14]. In general, the overall positive feedback received from the subjects on text mode in App Inventor shed lights on the continuing development and improvements to Venbrace and the incorporation of Venbrace in App Inventor.

**Braces**

Venbrace was initially designed to be fully-braced based mainly on DP5 Support Copy/Paste. However, many study respondents reported trouble with using braces in Venbrace.

The responses to the Translation Typing tasks as well as individual feedback submitted by the participants revealed their frustration with balancing the braces. In general, expressions beginning with two parentheses were commonly missing a parenthesis, nested statements ending with two curly braces were sometimes missing a closing curly brace, even with the help of syntax editor. As a subject pointed out in their comment to the design of Venbrace, "[they] would love for it to be easier to keep track of { and ." Such responses and feedback imply improvements to the error reporting and handling mechanism and even an implementation of a proper form of error recovery in future rounds of study.

On the other hand, the design decision to represent different types of blocks (expressions, statements, and declarations) using different types of braces caused confu-
sion to the participants. The Translation Typing tasks began with only expression blocks, but incorporated statements and declarations as the study proceeded, involving more types of braces in the translation process as a result. One subject continued to use parentheses to represent all types of blocks, and I speculate that subjects in general had a better understanding and more confidence in using the parentheses to which they had been exposed early on in the study. The confusion was reported in direct feedback as well: one subject wrote, "The distinctions between /, {, and ( get confusing at times." For a better user experience, a natural solution for the next design would be having only one type of brace. Using only one instead of three types of braces, nevertheless, contradicts with DP3 Blocks Are Primary. App Inventor states the distinction between expression, statement, and declaration blocks explicitly through the difference in their block shapes, and such distinction could not be inherited and represented in the text language if only one type of brace is adopted, which violates DP3. Since the design of VENBRACE is first principled then user-preference-informed, the design principles should be more prioritized than user preference. Still, we could seek for alternative solutions to ease the confusion of using multiple types of braces in future rounds of study, such as automatically correcting incorrect use of braces through error recovery.

Abbreviations

I implemented alternative, abbreviated VENBRACE translations for some App Inventor blocks following DP4 Flexible Alternatives to allow for more flexibility. Overall, subjects were comfortable with the current support for abbreviations throughout the study, especially with atomic expressions, procedure calls, and local variable getters. Fewer participants continued to use explicit parentheses for atomic expressions as they got more familiar with the VENBRACE language. Two thirds of the participants always wrote out the call keyword at the beginning of a procedure call block, while the rest omitted it in their translations. Whether or not to translate the local variable getter block in its abbreviated form was more of a mix, because the abbreviation was slightly more complicated: the user has to omit both the parentheses and
the get keyword at the beginning of the block for a valid abbreviation.

For blocks whose abbreviated translations are yet determined, I noticed more diverse opinions. For a procedure call block that requires arguments and has parameter names to the arguments explicitly specified in the visual block, half of the participants would like to include the parameter names in their original form in the Venbrace translation, while others prefer not to include them. The results indicate that the parameter names could be optional in the next design.

Surprisingly, there was a commonly strong preference for translating helper words in their original form without any abbreviations or symbol substitutes. In general, participants preferred to keep the Venbrace translation for a block identical to its original visual form in App Inventor. They gave some consideration to alternative textual representations for a sequence of helper words, such as the camel case and the use of underscores. However, they were more against the use of arbitrary symbols in abbreviated translations of this kind. The feedback on handling long sequences of helper words from subjects of this round of study, in fact, helps us comply with DP2 Bidirectional Isomorphism. Considering the feedback from this round of study, it might be wise to translate the helper words as in the next design iteration of Venbrace.

7.5.2 Going Forward

The Venbrace Language

Based on the study results and the discussion above, the next design iteration of Venbrace should focus on handling abbreviated translations. Existing support for abbreviations of atomic expressions, procedure calls, and variable getters will continue according to participants’ positive feedback. As for implementing new abbreviations, the plan is (1) to make explicit parameter name labels in a procedure call block optional, and (2) to represent long sequences of helper words in their original form.
The Future Rounds of User Studies

Results from this round of study suggest improvements to the error reporting and recovery mechanism for the study. To ease the cognitive burden of balancing braces, the mechanism could automatically fill in missing braces under circumstances in which the errors are obvious. To reduce the confusion of using multiple types of blocks, the mechanism could automatically recover incorrect usage of braces (e.g. using parentheses for statements).

In addition, we received several comments asking if abbreviated and non-abbreviated translations could appear together, e.g. whether both \((3 + (\text{get a}))\) and \((3) + a\) could be considered as valid alternatives to their most verbose representation \((3) + (\text{get a})\). Apparently, the answer is affirmative. Participants had such a concern probably because we used either the most verbose translations or the most abbreviated ones throughout the tutorials, but not something in between. In future studies, it should be stated clearer in the tutorial material that abbreviated and non-abbreviated translations can be arbitrarily interleaved.

7.6 Limitations

A goal of the first round of study is to collect some preliminary data that could help improve future rounds of studies. Ideally, the participants for all three rounds of studies should be chosen from the same pool, i.e., people who have a decent amount of experience using App Inventor but not too many preconceived notions of textual programming languages. However, with concerns of recruiting enough participants especially due to the global pandemic of COVID-19, subjects outside of this group, such as App Inventor users with nontrivial programming experience with text languages have been included in the first round of study. Although such subjects qualify for the study given their abundant programming experience with App Inventor, they may have been significantly exposed to or involved with textual programming languages even before using App Inventor for the first time. Their experience with textual programming languages might lead to cognitive biases when evaluating the design.
of **Venbrace**, because they might rely on their existing experience with other text languages to make their judgments, rather than from the perspectives of fresh eyes or professional programming language designers.

In addition, the analysis on the duration of each task might not be as reliable, since subjects are allowed to take a break from the study before a final completion. Although the study is designed to keep track of starting and ending timestamps for tasks throughout the study, the time during which a subject takes a break is not recorded. The researchers rely on the timestamps data to determine whether future rounds of studies should be shortened or restructured and will remove any outliers, i.e., subjects that have rather prolonged completion time due to breaks during the study. Given the limited sample size of the first round of study, however, the remaining data might be insufficient to conclude on any necessary change to the structure of the study for future rounds.

Another issue is that the error reporting mechanism implemented for the Translation Typing tasks is solution-based. For each of such tasks, there is absolutely one **Venbrace** translation in its complete form, and the error reporting system is based on the comparison between the solution and the user response, both tokenized by the **Venbrace** lexer. The **Venbrace** parser was not included in this study, because the main focus of this study is to investigate whether the design choices for tokens used in the Translation Typing tasks are intuitive enough for participants to get accurate responses. As a consequence, the study emphasizes the tokens more than the syntactic structure of **Venbrace**, leading me to have decided to only use the **Venbrace** lexer. As described in Chapter 5, the lexer and parser work closely, and the lexer could not get enough information about the correctness of a program. Even when a program consists of all valid tokens, the tokens might be misplaced, causing a malformed structure, the correctness of which could only be determined by the parser. What I did in this study is close to context-sensitive lexing: I report missing tokens, unexpected tokens, and expected tokens that yet appear at where they are not supposed to be based on the relative positions of the tokens. Brace balancing and pairing is well handled by this mechanism, too. Still, the system might fail to catch
errors that appear in edge cases, and the error reporting messages generated by this system might not reveal the real cause behind the caught errors and thus could be confusing to the participants. An ideal solution would be implementing a "real" error reporting system by modifying both the VENBRACE lexer and the parser through more exploration in ANTLR's error handling strategy (more details in Section 8.4).

Lastly, there were study data which were collected but not yet prioritized for analysis that could lead to significant findings. The Translation Selection tasks, for example, were only considered as an extra step that reinforced subjects' understanding of VENBRACE grammar rules introduced in the tutorials for this round of study. However, responses to this type of tasks might be correlated with responses to the Translation Typing tasks, such as the common translation errors. On the other hand, the common errors committed in the Translation Typing tasks were summarized based on merely the incorrect last attempts. For subjects who were able to submit a correct answer after several unsuccessful attempts, their previous failed attempts could further indicate necessary redesigns to the language. Keystrokes collected in the Translation Typing tasks might reveal the causes of incorrect translations, but they were not yet included in the analysis, either. I plan to conduct a more thorough data analysis that would contain all data collected, as is further discussed in Section 8.2.

7.7 Conclusion

The first round of user study returns with evidence that the initial design of VENBRACE is successful, and that the idea of improving the language through iterations of implementation -> feedback -> improvements is effective. The study shows great interest in having a text language and editing blocks in a text mode in App Inventor from its current users, and also presents room for enhancement of the next design for VENBRACE as well as of future rounds of user studies.

Participants were generally satisfied with the current design of VENBRACE, except for complaints about using braces. Considering the design principles that guide the
design of Venbrace, I decide not to change the current design decisions on braces, but instead to improve the error reporting, handling, and recovery mechanism used in the future rounds of studies to provide the users with easier interactions with the braces. Braces may become easier to use with such improvements, which are briefly mentioned in Section 7.5.2 and detailed in Section 8.4.3.

The more concise, abbreviated Venbrace translations an alternative were appreciated by the participants. All current decisions on Venbrace abbreviations received positive feedback, so the next design of Venbrace should continue to support such abbreviations. In addition, the study respondents also helped with making decisions on translating blocks that have long sequences of help words, as is explained in Section 7.5.2. Involving more participants, future rounds of studies will continue to gather user feedback on the design of abbreviations.

By examining responses to the study tasks and subjects’ individual comments, I also identified features that should be implemented in the error handling mechanism. A more thorough error handling mechanism will be used in future rounds of user studies and eventually for the Venbrace language, aiming to flatten the curve of learning to use Venbrace and make interactions with the language more enjoyable.
8 Conclusion and Future Work

8.1 Current State

8.1.1 Venbrace Syntax

Initial Design

In its current state, Venbrace has support for App Inventor blocks of which the design for translations have been finalized. For blocks whose translations are tentative or have not been decided (see Section 4.2), we will make the design decisions/changes based on results from the first round of user study. There are features of App Inventor blocks that are not yet designed nor implemented in Venbrace, such as disabled vs. enabled blocks, internal vs. external inputs of an expression block, and block position on screen. As for App Inventor components other than blocks, no Venbrace implementation has been done so far: they include comments, component specs, full screens, and full programs, although these components are discussed in the initial design.

Known Ambiguities

In many programming languages, it is standard that keywords are reserved and cannot be used as the names of variables, procedures, etc. However, the current version of App Inventor does not prevent users from doing using keywords as variable and procedure names. As a blocks language, App Inventor does not have a notion of keyword entered by the user, since keywords are already baked into blocks. The only names users enter are variable and procedure names. Because names chosen by users cannot interact badly with the names baked into blocks, it is fine to have variable and procedure names that look like what might be keywords in traditional languages,
such as true and neg. When it comes to text, such flexibility could potentially cause trouble.

Here are some examples of such ambiguities:

• [to negate neg return (neg neg)]: neg is a parameter name, but neg could also be mathematical operator, which is a keyword. In App Inventor, this neg will never be considered as a keyword, because the keyword neg only appears at the beginning of the math negation block, which has a completely different color and shape than a procedure declaration block. In Venbrace, such a distinction originally made by visual notations is now lost, and it is impossible to tell whether neg is a keyword or a parameter name without additional information.

• [to true true do if true ...]: the first true is the procedure name, the second true is the name of the parameter required by this procedure, and the third true is ambiguous, because it could either be the parameter name true, or the abbreviated form of (true), in which true is the keyword for the boolean value.

• [to true true do if (true) ...]: this example is similar to the previous one, except that the third true leads to another ambiguity: does (true) mean the verbose form of the boolean literal, or the abbreviated form of (call true)? In either this or the previous example, a distinction between all the occurrences of true is easily made in the App Inventor editor, because they are represented by different blocks even with different colors. However, such a distinction no longer exists in a text language like Venbrace.

These known ambiguities are not covered in the current design for Venbrace. In the current Venbrace implementation, these ambiguities will likely trigger syntax (parsing) errors. In fact, ANTLR lexer rules are written in order of priority, and the current Venbrace grammar (written in ANTLR syntax, see Appendix C) determines that tokens preserving keyword-like literal content will only be lexed as keyword tokens. In other words, even if a user writes a variable name using an existing keyword and places the variable name at a position at which a variable is expected,
the VENBRACE lexer will still lex the token as a keyword, which will potentially cause some parsing error. To satisfy DP2 Bidirectional Isomorphism, this problem has to be fixed, and it will be fixed in the next iteration of the implementation in the near future (more in Section 8.3).

8.1.2 VENBRACE in App Inventor

I have partially implemented the VENBRACE-to-App Inventor conversion. Since ANTLR can generate outputs alongside the Syntax Tree of a language, the most recent VENBRACE grammar written in ANTLR syntax can produce XML code for each syntax rule that matches the XML code for the corresponding App Inventor block(s) (Figure 8-1 illustrates how VENBRACE can be converted to an XML tree structure to be used in App Inventor). Since the implementation is still under development, it is not included in Appendix C, i.e. the code for VENBRACE grammar.

Figure 8-1: It can be specified in an ANTLR4 grammar file that an XML tree structure should be built at the end of a successful parsing process apart from the parse tree.

8.1.3 Support for VENBRACE

A syntax-highlighted code editor has been implemented to support basic syntax highlighting for VENBRACE. The code editor and the VENBRACE parser together provide basic error reporting.
The first round of user study has been conducted on the current design of the Venbrace syntax. Videos and tasks from the first round (with small modifications if necessary) can be used for future rounds of studies. Their results will drive future design iterations of Venbrace and improvements to the design of the user studies.

8.2 Immediate Future

This section discusses work that I plan to do in the short term future (i.e., about a month).

8.2.1 An Improved Syntax Design

Based on the results from the first round of study, the next design iteration of Venbrace will involve the following changes:

- Multi-word key phrases will be handled by having all words separated by spaces in their verbose translations. For key phrases like make a list, some unambiguous prefix or subset of words in the phase (e.g., list) will be allowed as an abbreviation.

- Socket labels will be written in their original form (i.e., no colons or other annotations/ additional symbols), but will be optional.

8.2.2 More Thorough Data Analysis

Data analysis for the first round of user study (see Section 7.4) does not cover all data collected. In the near future, I plan to analyze all the data thoroughly, including:

- Data collected from the Translation Selection tasks (now treated as a learning opportunity for participants), to explore possible improvements to be made to Venbrace that are not covered in other tasks;

- TAM analysis for nonfinal attempts in the Translation Typing tasks, to better understand the causes of syntax errors made by the participants and unveil
more issues that subjects found difficult; and

- The correlation, if any, between results from TAM analysis on individual participant and their prior experience of programming reported in the Pre-study Survey.

8.2.3 More User Studies on VENBRACE Syntax

Two more rounds of studies evaluating the syntax of VENBRACE have been planned. The structure and interface of the future rounds will be improved based on feedback from previous rounds of the study. When an improved syntax design for VENBRACE is available, the improved version will be adopted in future rounds of studies to be further evaluated, typically involving modified tutorial videos/documentation and the associated tasks.

8.3 Near Future

This section discusses work that can be done within the next few months. It is likely that I will help out with some of these items.

8.3.1 VENBRACE Support for All App Inventor Components

An implementation of VENBRACE in the near future should support all App Inventor components that are not yet supported in the current version (as described in Section 8.1), including comments, disabled/enabled blocks, internal vs external inputs, block position on screen, and notations for component specs, full screens, and full programs.

8.3.2 Handling Known Ambiguities Better

The fact that the current implementation of VENBRACE does not treat keywords as identifiers causes the ambiguities mentioned in Section 8.1.1. An initial step to solve this issue, as Parr suggests in ANTLR4 Reference [Par13], is to additionally create a parser id rule in the parsing grammar that matches regular identifier ID tokens and
any of the keyword tokens. For example, the neg keyword currently has NEG token type, the true keyword has token type TRUE; to treat them as identifiers under certain circumstances, an additional rule can be added to the VENBRACE parser:

```
  id: NEG | TRUE | ...<other keyword tokens> ... | ID;
```

This tentative solution could potentially lead to other parsing issues. Still, it is worth trying as a first step in the near future.

### 8.3.3 VENBRACE in App Inventor

As is detailed in Section 2.5.2 for her work on the TAIL language, Chadha implemented a TAIL code block that is a new type of App Inventor block. A TAIL code block (1) allows users to type in textual TAIL code directly within the block inside of the App Inventor editor [Cha14], and (2) can be automatically converted to/back from regular App Inventor blocks. The TAIL code blocks are XML-based and allow the coexistence in App Inventor of regular blocks and code blocks that contain text code. I plan to embed VENBRACE in App Inventor following the architecture Chadha built, which is already work-in-progress as stated in Section 8.1.2 so that VENBRACE can be used in App Inventor in the same way as TAIL.

When VENBRACE is incorporated in App Inventor, support for at least two translation modes should be available: a concise mode that uses all VENBRACE abbreviations and a verbose mode that uses no abbreviations. It would also be ideal to have an additional, intermediate mode that is somewhere in between.

### 8.3.4 Automatically Updating the VENBRACE Lexer and Parser

A text mode for App Inventor is not viable until changes to core App Inventor and extensions can be handled. Consequently, it is necessary to automate the generation of the ANTLR4-generated VENBRACE lexer and parser from a specification for the full App Inventor language, rather than actual keywords, that can handle future changes to the core-language as well as user-specified extensions.
Unforeseeable App Inventor Changes

All design decisions for VENBRACE are informed by the current App Inventor design, and they might need to be changed if new features are introduced to App Inventor. For example, first-class functions are currently unavailable in App Inventor, and \(<\text{name}>\) cannot be interpreted as \(\text{call (get <name>)}\) in App Inventor at this moment. Future changes to App Inventor might make the current design for abbreviations ambiguous. So long as the ambiguity caused by the changes only affects the abbreviated VENBRACE translations, previous designs for abbreviations can always be removed to avoid the ambiguity. However, what if the ambiguity further affects the verbose translation? A fundamental redesign of VENBRACE might be necessary in this scenario.

Supporting VENBRACE in App Inventor Extensions

In addition, App Inventor has a notion of extensions, which allow third parties to add new components and blocks to the system. Presumably, the extensions could impact VENBRACE, as we cannot predict what components and blocks will be introduced by third-party developers. How do we handle such scenarios? In the short term, maybe VENBRACE could be turned off in App Inventor for projects with extensions.

While a hand-crafted, ANTLR4-generated VENBRACE lexer/parser suits the need of the current VENBRACE design and implementation, it is unsustainable going forward, as the lexer/parser has to be modified by hand every time an App Inventor feature is redesigned or a new feature is introduced. To implement a truly viable text mode for App Inventor, a text language like VENBRACE is a nonstarter until a more automated lexer/parser system is developed. In the near future, a fundamental redesign of the VENBRACE lexer/parser system based on a specification for the App Inventor architecture rather than a collection of App Inventor keywords would be necessary.
8.3.5 Support of Internationalization

App Inventor’s support of internationalization (“i18n”) allows users around the world, especially programmers whose native language is not English, to program in a language with which they are more comfortable. Currently, the platform provides programming environment in 12 different languages. It may be difficult to support multi-lingual programming in a similar way in a text language like Venbrace. The current implementation of Venbrace only supports tokens composed of alphabetic letters, numbers, and punctuation and symbols commonly seen in the English language.

To comply with DP6 Internationalization, however, supporting i18n in Venbrace is necessary. I foresee possible solutions as follows.

On the one hand, the original App Inventor language environment can be specified in Venbrace using a system configuration command at the beginning of the textual file that begins with #. An environment variable blockLang is reserved to denote the App Inventor language environment using an alpha-2 or alpha-3 language code [ISO]. For example, #blockLang: eng means that the original App Inventor blocks program of this Venbrace code file was generated in English.

There are many but a limited number of reserved keywords in App Inventor even under different language environments. Therefore, it would be possible to maintain a data structure that maps all App Inventor keywords, regardless of the language in which they are written, to the unique keyword identifiers used by App Inventor behind the scene. The lexer and parser could thus be modified in the following way: instead of directly recognize the content of a token, they should build Venbrace’s syntactic structure on these unique keyword identifiers. The tokens in a Venbrace program written in any language will be first interpreted as such unique keyword identifiers, and then go through the lexing and parsing process.

To convert such a Venbrace program to an App Inventor project, the language code specified by blockLang will be used to determine the display language of App Inventor blocks, and the embedded unique keyword identifiers in the Venbrace program will be translated accordingly. To convert an App Inventor project to a
VENBRACE program, the unique keyword identifiers will be first translated into the language used in the App Inventor editor by referencing to the keyword-to-identifier map, so that VENBRACE text file the user actually see would be in the language of their preference.

On the other hand, 12 VENBRACE grammars that generate 12 different lexers and parsers targeting each of the 12 languages supported by App Inventor could even be built to directly recognize Unicode characters (used in many non-English languages) as tokens. The lexer and the parser to be used will be thus depended on the language code carried in blockLang.

8.3.6 Improving the Syntax Editor

In the near future, I foresee the following improvements to be added to the syntax editor for VENBRACE:

- **Keyword highlighting.** Currently, all VENBRACE keywords are highlighted in blue, which is the default configuration of CodeMirror. In the future, I consider adding bolder styling to the VENBRACE keywords in the editor to further distinguish them from the non-keyword tokens.

- **Auto-completion.** For better user experience, it would be ideal if the syntax editor could support auto-completion on keywords, multi-word key phrases, and keyword structures (e.g. including a then with an if and a do with when) in the future. Beyond keywords, it would be great to also automatically flesh out parameter names for event handlers and procedure calls that have parameters.

- **Visually distinguishing optional items,** including optional braces, keywords, and less important words in multi-word key phrases.

- **Brace coloring.** To further reinforce VENBRACE’s isomorphism to App Inventor, we can possibly have the color of a pair of braces match the color of the App Inventor block to which the braced item corresponds.
• Parsing **VENBRACE**. So far, the **VENBRACE** lexer and parser are not yet embedded in the syntax editor (see Chapter 6). Going forward, embedding the **VENBRACE** lexer and parser could allow for more context-based syntax highlighting (which requires more information returned by the lexer and the parser), error reporting (where and why an error occurs), and error recovery within the syntax editor.

The features described in this section are based on tokens lexed by the **VENBRACE** lexer and syntax trees generated by the **VENBRACE** parser. When the generation of the **VENBRACE** lexer and parser becomes automated (detailed in Section 8.3.4), the features that the syntax editor supports will be similarly synchronized with any future improvements or feature releases within App Inventor.

**8.3.7 Simple Error Reporting**

A thorough error reporting and handling mechanism for **VENBRACE** is itself a whole other project in the far future (more in Section 8.4.3). Still, it is necessary to explore ANTLR’s own API for error reporting and handling to implement basic but informative error reporting in the near future. Considering that **VENBRACE** will soon be used and tested in App Inventor, the **VENBRACE** parser should at least be able to report the error to the user – what the error is, and where it occurs.

**8.3.8 Importing/Exporting App Inventor as **VENBRACE**

As described in Section 1.2.3, App Inventor projects can be imported with or exported as .aia files. Once **VENBRACE** becomes part of the App Inventor architecture, an App Inventor project should also be allowed to be imported/exported as one single text file that contains **VENBRACE** code. A new **VENBRACE** file format for the entire App Inventor program, e.g. a .ven file, should be created to enable such export options.
8.3.9 Testing and User Studies on VENBRACE-in-App Inventor

Once VENBRACE-in-App Inventor is implemented, we should invite members of the App Inventor community to agree to test alpha/beta versions of VENBRACE in code blocks and constructing VENBRACE scripts (.ven files) from scratch to constantly improve VENBRACE-in-App Inventor before its formal release. This testing process should be similar to how the App Inventor team tests a new feature and to other software testing processes.

Following the testing phase, there should also be at least one preliminary user study of VENBRACE-in-App Inventor that examines (1) how easy it is to use and (2) how do users actually use it in practice (as we initially proposed in Section 7.1).

8.4 Far Future

This section describes interesting long-term projects that are related to this work. I am not likely to help out with any of the items described in this section.

8.4.1 Preserving All Editing Features of App Inventor in Text Mode

After VENBRACE supports representing all App Inventor components in the text mode, the next step would be to inherit existing editing features in App Inventor. For example, in the current version of App Inventor, changing a parameter name of a procedure automatically changes it in all getter/setters in the procedure body; similarly, changing the name of a procedure changes it in all callers of that procedure. Such auto-renaming actions would not naturally happen if some of those getters/setters or callers are in text in a code block, let alone an App Inventor program completely written in VENBRACE.

A Real-Time Collaboration (RTC) version of App Inventor is currently experimental. When it becomes available in App Inventor, we should extend the use of VENBRACE in the RTC mode as well, possibly allowing one collaborator to program in
blocks while another to use VENBRACE in the text mode and keep their programming progress in sync.

To preserve App Inventor editing features like auto-renaming, a bridge has to be built between App Inventor and VENBRACE to keep all program structs and macros in sync. I imagine keeping the VENBRACE/App Inventor synchronization a whole other project in the far future.

8.4.2 More User Studies on Using App Inventor Incorporated with VENBRACE

As is originally proposed [HT19] and also stated in Section 7.1, there should be user studies in the future that involve a working implementation of VENBRACE-in-App Inventor. The studies will further investigate the usability of VENBRACE in App Inventor and understand when people tend to use the text mode vs. the blocks mode when both modes are available.

8.4.3 Error Reporting

As briefly mentioned in Section ??, a more thorough error reporting and handling mechanism should be added to VENBRACE, which is the topic for a big future project. Building upon ANTLR’s existing Error Handling Strategy, the lexer and parser will be able to report more meaningful error messages and even recover from the errors properly for any detected errors. The designs of error messages and of the interaction with the error handling system, on the other hand, refer to whole different areas of research [MFK11a] [MFK11b].
Bibliography


[MOM15] Jens Mönig, Yoshiki Ohshima, and John Maloney. Blocks at your fingertips: Blurring the line between blocks and text in GP. In 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), pages 51–53, Atlanta, GA, USA, October 2015. IEEE.


Snap! Build Your Own Blocks. https://snap.berkeley.edu/.


Daniel Wendel. Position: Meeting the promise of blocks-as-AST-nodes editing with “Typeblocking”. In 2019 IEEE Blocks and Beyond Workshop (B B), pages 23–26, October 2019.


David Weintrop and Uri Wilensky. Comparing Block-Based and Text-Based Programming in High School Computer Science Classrooms, October 2017.
A  Round 1 Study Materials

This appendix contains a summary of all the materials used in the online Venbrace syntax user study. There were 42 activities in the summary organized into a pre-study survey, nine sections of tutorials and tasks, and a post-study survey.

A.1 Pre-study Survey (Activity 01)

The pre-study survey is presented in Figures A-1 and A-2.

A.2 Tutorial/Task Section 1 (Activity 02)

Activity 02: Tutorial: Introduction

Section 1 contains only a tutorial video that motivates Venbrace as a textual language for App Inventor and explains that different braces will be used for different kinds of blocks: parenthesis for expression blocks, curly braces for statement blocks, and square brackets for top-level blocks (event handlers, procedure declarations, and global variable declarations).

This tutorial video and its associated summary document can be found with all Venbrace syntax study videos and summary documents at http://cs.wellesley.edu/~venbrace/syntax-study-round1/allTutorials.html.

Section 1 is the only section that consists of a tutorial with no associated tasks.

A.3 Tutorial/Task Section 2 (Activities 03-05)

Activity 03: Tutorial: Simple Math Expressions

This tutorial introduces the Venbrace parenthesized notation for number blocks and math operator blocks, along with the option to omit parens for atomic expressions.
Activity 04: Selection Task: Multiplication

This task asks which of the Venbrace translations in Table A.1 are valid for:

Table A.1: Valid and invalid Venbrace translations for

<table>
<thead>
<tr>
<th>Venbrace</th>
<th>Valid?</th>
<th>Explanation of answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(8 x 3)</td>
<td>false</td>
<td>Multiplication symbol is *, not x</td>
</tr>
<tr>
<td>(8 * 3)</td>
<td>true</td>
<td>Valid expression for blocks; numbers need not be parenthesized</td>
</tr>
<tr>
<td>(8*3)</td>
<td>true</td>
<td>Spaces around operators aren’t necessary</td>
</tr>
<tr>
<td>( 8 *3 )</td>
<td>true</td>
<td>Extra spaces in this expression are ignored</td>
</tr>
<tr>
<td>8*3</td>
<td>false</td>
<td>Outer parens are required for non-atomic expressions</td>
</tr>
<tr>
<td>8 * 3)</td>
<td>false</td>
<td>Missing open paren</td>
</tr>
<tr>
<td>((8) * (3))</td>
<td>true</td>
<td>Pares around numbers are allowed, but not required</td>
</tr>
<tr>
<td>(8 * (3))</td>
<td>true</td>
<td>Pares around numbers are allowed, but not required</td>
</tr>
<tr>
<td>(8 * (3)</td>
<td>false</td>
<td>Missing close paren</td>
</tr>
<tr>
<td>(8 * ((3)))</td>
<td>false</td>
<td>((3)) is invalid. Each parenthesized phrase stands for one expression block. Two pairs of parens would stand for two expression blocks, but there is only one: the number 3</td>
</tr>
<tr>
<td>((8) * 3)</td>
<td>true</td>
<td>Pares around numbers are allowed, but not required</td>
</tr>
<tr>
<td>(3 * 8)</td>
<td>false</td>
<td>(3 * 8) multiplies the same two numbers as (8 * 3), but stands for a multiplication block whose operands are in a different order</td>
</tr>
<tr>
<td>(6 * 4)</td>
<td>false</td>
<td>(6 * 4) means the same number as (8 * 3), but stands for a multiplication block with different operands</td>
</tr>
</tbody>
</table>

Activity 05: Translation Typing Task: Simple Expression

This task asks for the Venbrace translation of:

124
A.4 Tutorial/Task Section 3 (Activities 06-13)

Activity 06: Tutorial: More Math Expressions

This tutorial covers nested math expressions, infix math operators with more than two operands, prefix math operators, and representing an empty socket as ()

Activity 07: Selection Task: Nested Expressions

This task asks which of the Venbrace translations in Table A.2 are valid for:

<table>
<thead>
<tr>
<th>Venbrace</th>
<th>Valid?</th>
<th>Explanation of answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 + (4 * 7)</td>
<td>false</td>
<td>Outer parens for + block are missing</td>
</tr>
<tr>
<td>((1 + 4) * 7)</td>
<td>false</td>
<td>Top-level operation should be +, not *</td>
</tr>
<tr>
<td>(1+(4*7))</td>
<td>true</td>
<td>Has correct paren structure, and spaces aren’t necessary</td>
</tr>
<tr>
<td>(1+4*7)</td>
<td>false</td>
<td>Parens for * are missing; precedence rules in math do not apply</td>
</tr>
<tr>
<td>1+4*7</td>
<td>false</td>
<td>Parens for * and + are missing; precedence rules in math do not apply</td>
</tr>
<tr>
<td>((1) + (4 * (7)))</td>
<td>true</td>
<td>A single pair of parens around any number is OK</td>
</tr>
<tr>
<td>((4 * 7) + 1)</td>
<td>false</td>
<td>Order of + operands matters</td>
</tr>
<tr>
<td>(1 + (7 * 4)))</td>
<td>false</td>
<td>Order of * operands matters</td>
</tr>
<tr>
<td>4 (((1 + (4 * (7)))))</td>
<td>true</td>
<td>This crazily formatted sequence of tokens is correct as far as the computer is concerned, but is very hard for humans to read, so should be avoided in practice</td>
</tr>
</tbody>
</table>

Table A.2: Valid and invalid Venbrace translations for
Activity 08: Selection Task: 3-Operand Summation

This task asks which of the Venbrace translations in Table A.3 are valid for:

Table A.3: Valid and invalid Venbrace translations for

<table>
<thead>
<tr>
<th>Venbrace</th>
<th>Valid?</th>
<th>Explanation of answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(9 + 21)</td>
<td>false</td>
<td>Venbrace needs to show two additions with 3 operands.</td>
</tr>
<tr>
<td>(9 + + 21)</td>
<td>false</td>
<td>Can’t just have nothing between the two + operands. Must use the empty socket notation () between them.</td>
</tr>
<tr>
<td>(9 + () + 21)</td>
<td>true</td>
<td>This correctly uses () to represent the empty socket for the middle of the three operands.</td>
</tr>
<tr>
<td>(9+21+)()</td>
<td>false</td>
<td>() must be the second operand, not the third.</td>
</tr>
<tr>
<td>9+()+21</td>
<td>false</td>
<td>Outermost parens are missing.</td>
</tr>
<tr>
<td>(9 + (() + 21))</td>
<td>false</td>
<td>This represents two 2-operand + blocks, not a single + block with 3 operands.</td>
</tr>
<tr>
<td>(21 + () + 9)</td>
<td>false</td>
<td>9 and 21 are in the wrong operand positions.</td>
</tr>
</tbody>
</table>

Activity 09: Translation Typing Task: Nested Math Expr. 1

This task asks for the Venbrace translation of:

Activity 10: Translation Typing Task: Nested Math Expr. 2

This task asks for the Venbracer translation of
Activity 11: Translation Typing Task: Nested Math Expr. 3

This task asks for the Venbrace translation of

![Math expression image]

Activity 12: Preference Ranking Task: Degree Conversion

This task asks for preference rankings on the following blocks and options:

![Preference ranking image]

Note: In this and many other preference tasks, we’re looking for your input on a general issue using a particular example. In this task, we want to know how to handle the many App Inventor blocks that begin with multiple words separated by spaces. The goal is to have a strategy for handling such blocks that can be expressed by a simple rule that can be followed in all cases, rather than having different special cases for particular blocks.

- (convert degrees to radians 60)
- (convertdegreestoradians 60)
- (convert_degrees_to_radians 60)
- (convert%degrees%to%radians 60)
- (convertDegreesToRadians 60)
- (<convert degrees to radians> 60)
- ('convert degrees to radians' 60)
- ($convert degrees to radians$ 60)
- (#convert degrees to radians# 60)
- (\convert degrees to radians\ 60)

Note for Optional Better Option Area: While there are much nicer special-case translations like (degrees->radians 60) in this particular case, we’re really looking for translations that handle the general problem of blocks that begin with multiple words separated by spaces.
Activity 13: Preference Ranking Task: Modulo

This task asks for preference rankings on the following blocks and options:

- (modulo 17 5)
- (modulo 17 / 5)
- (modulo 17 % 5)
- (17 % 5)
- (% 17 5)

Note: In this task, the focus is on whether/how to translate the ÷ sign.

A.5 Tutorial/Task Section 4 (Activities 14-18)

Activity 14: Tutorial: More Expressions

This tutorial covers string and logic expressions.

Activity 15: Translation Typing Task: Expression with Booleans and Strings

This task asks for the Venbrace translation of:
Activity 16: Preference Ranking Task: Color Red

This task asks for preference rankings on the following block and options:

Note: In this task, the focus is on how to translate any color block, with red just being one example.

- red
- #red
- (color red)
- (red)
- <red>
- |red|
- color.red
- Color.red

Activity 17: Preference Ranking Task: Make a List

This task asks for preference rankings on the following block and options:

- (make a list 5 8 4)
- (makealist 5 8 4)
- (make_a_list 5 8 4)
- (makeAList 5 8 4)
- (<make a list> 5 8 4)
- ('make a list' 5 8 4)
- (#make a list# 5 8 4)
- |(make a list| 5 8 4)
- (list 5 8 4)
Activity 18: Preference Ranking Task: Empty List

This task asks for preference rankings on the following block and options:

- (create empty list)
- (createemptylist)
- (create_empty_list)
- (createEmptyList)
- (empty list)
- (emptylist)
- (empty_list)
- (emptyList)
- (list)
- (empty)

A.6 Tutorial/Task Section 5 (Activities 19–22)

Activity 19: Tutorial: Simple Statements

This tutorial covers simple single statements, including setters for components and variables, component method calls, and procedure calls. Venbrace statements are delimited by curly braces.

Activity 20: Selection Task: Component Setter

This task asks which of the Venbrace translations in Table A.4 are valid for:
Table A.4: Valid and invalid Venbrace translations for

<table>
<thead>
<tr>
<th>Venbrace</th>
<th>Valid?</th>
<th>Explanation of answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set Canvas1.Height to (2 * Canvas1.Width))</td>
<td>false</td>
<td>Outer parens need to be changed to braces to represent the set statement.</td>
</tr>
<tr>
<td>{set Canvas1.Height (2 * Canvas1.Width)}</td>
<td>false</td>
<td>Right braces/parens, but missing to keyword.</td>
</tr>
<tr>
<td>{set Canvas1.Height to (2 * (Canvas1.Width))}</td>
<td>true</td>
<td>Optional parens in (Canvas1.Width) are OK.</td>
</tr>
<tr>
<td>{set Canvas1.Height to {2 * Canvas1.Width}}</td>
<td>false</td>
<td>Inner braces need to be changed to parens to represent the * expression.</td>
</tr>
<tr>
<td>{set Canvas1.Height to(2*Canvas1.Width)}</td>
<td>true</td>
<td>Valid, but lack of spaces impedes readability.</td>
</tr>
<tr>
<td>{ set Canvas1.Height to ( 2 * Canvas1.Width }</td>
<td>false</td>
<td>Close brace missing on statement.</td>
</tr>
<tr>
<td>{ set Canvas1.Height to ( 2 * Canvas1.Width )</td>
<td>false</td>
<td>Close paren missing on expression, something that the crazy formatting makes difficult to determine.</td>
</tr>
</tbody>
</table>

Activity 21: Translation Typing Task: Simple Statement

This task asks for the Venbrace translation of:

Activity 22: Preference Ranking Task: Infix Abbreviation for Setters

This task asks for preference rankings on the following blocks and options:

Note: You’ve seen this block can be translated to {set num to 3}. But setters are
so common that it would be nice to have an abbreviation for them that uses an infix symbol. Below are some options for you to rank.

- \{num <- 3\}
- \{num -> 3\}
- \{num <-> 3\}
- \{num := 3\}
- \{num ::= 3\}
- \{num :: 3\}
- \{num = 3\}
- \{num -- 3\}
- \{num -=- 3\}

A.7 Tutorial/Task Section 6 (Activities 23–27)

Activity 23: Tutorial: Nested Statements

This tutorial explains the nesting of statements in conditionals. Sequences of statements are written simply by putting one statement after another.

Activity 24: Translation Typing Task: Nested Statement

This task asks for the Venbrace translation of:
Activity 25: Preference Typing Task: While Loop

This task asks the subject for their preferred Venbrace translation of:

![While Loop Diagram]

Activity 26: Preference Typing Task: For Loop

This task asks the subject for their preferred Venbrace translation of:

![For Loop Diagram]

Activity 27: Preference Typing Task: Blocks with Lots of Words

This task asks the subject for their preferred Venbrace translation of:

![Blocks with Lots of Words Diagram]

A.8 Tutorial/Task Section 7 (Activities 28–30)

Activity 28: Tutorial: Event Handlers

This tutorial shows how to write event handlers, one of three kinds of top-level blocks in App Inventor. All top level blocks have translations delimited by square brackets.
Activity 29: Translation Typing Task: Event Handler

This task asks for the Venbrace translation of:

\[
\text{when Canvas1.Touched x y do}
\{\text{set Label1.Text to (join x }","\ y)}\}
\]

Activity 30: Preference Ranking Task: Event Handler with Parameters

This task asks for preference rankings on the following blocks and options:

Note: Many event handlers have unchangeable local parameter names, like \(x\) and \(y\) for the TouchDown event handler of a Canvas. How should these be represented (if at all) in Venbrace? Should they be omitted? Optional?"

- \([\text{when Canvas1.Touched x y do}]
  \{\text{set Label1.Text to (join x }","\ y)}\]\n- \([\text{when Canvas1.Touched }<x>\ <y> \text{ do}]
  \{\text{set Label1.Text to (join x }","\ y)}\]\n- \([\text{when Canvas1.Touched }|x|\ |y| \text{ do}]
  \{\text{set Label1.Text to (join x }","\ y)}\]\n- \([\text{when Canvas1.Touched }#x\ #y \text{ do}]
  \{\text{set Label1.Text to (join x }","\ y)}\]\n- \([\text{when Canvas1.Touched do}]
  \{\text{set Label1.Text to (join x }","\ y)}\]
A.9 Tutorial/Task Section 8 (Activities 31–36)

Activity 31: Tutorial: Procedure Declarations

This tutorial procedure declarations, the second kind of top-level block in App Inventor.

Activity 32: Translation Typing Task: SizeUp Procedure

This task asks for the Venbrace translation of:

Activity 33: Translation Typing Task: ConcatString Procedure

This task asks for the Venbrace translation of:

Activity 34: Preference Ranking Task: Calling the average Procedure

This task asks for preference rankings on the following blocks and options:

Note: Assume an abbreviated version the call and get keywords omitted. The key question is whether/how to include the socket labels a and b. Remember, these are
specified as parameters in the declaration of the **average** procedure and might change in the future. If the socket labels *are* included, can they be written in the opposite order with the two argument expressions appropriately swapped?

- (average a num b 10)
- (average a: num b: 10)
- (average #a num #b 10)
- (average <a> num <b> 10)
- (average num 10)
- (average b 10 a num)
- (average b: 10 a: num)
- (average #b 10 #a num)
- (average<b> 10 <a> num)

**Activity 35: Preference Typing Task: addPoint Procedure**

This task asks the subject for their preferred Venbrace translation of:

![Activity 35: Preference Typing Task: addPoint Procedure](image)

**Activity 36: Preference Typing Task: A Local Variable Declaration**

This task asks the subject for their preferred Venbrace translation of:

![Activity 36: Preference Typing Task: A Local Variable Declaration](image)
A.10 Tutorial/Task Section 9 (Activities 37–41)

Activity 37: Tutorial: Global Variables

This tutorial discusses global variable declarations (App Inventor’s third kind of top-level block), as well as global variables getters and setters. The translations for these have not yet been determined, but study subject feedback will guide the future design process.

Activity 38: Preference Ranking Task: Declaring a Global Variable

This task asks for preference rankings on the following blocks and options:

Note: This block is so common that it’s worth considering special-case shorthands that might differ from the standard way to handle blocks that begin with multiple words separated by spaces.

Unfortunately, the Venbrace preference options posted during the Round 1 study were buggy: the wrong kind of braces (parens rather than square brackets) were used for the setter statements, and the abbreviated options were missing any kind of braces entirely. Table A.5 shows the buggy options during the Round 1 study and how they have been corrected for a future study (if the task is used again).
Table A.5: Buggy and Corrected Venbrace preference options for

<table>
<thead>
<tr>
<th>Buggy Venbrace in Round 1</th>
<th>Corrected Venbrace for Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>(initialize global angle to 30)</td>
<td>[initialize global angle to 30]</td>
</tr>
<tr>
<td>(initializeGlobal angle to 30)</td>
<td>[initializeGlobal angle to 30]</td>
</tr>
<tr>
<td>(initialize_global angle to 30)</td>
<td>[initialize_global angle to 30]</td>
</tr>
<tr>
<td>(&lt;initialize global&gt; angle to 30)</td>
<td>[&lt;initialize global&gt; angle to 30]</td>
</tr>
<tr>
<td>(#initialize global angle to 30)</td>
<td>[#initialize global angle to 30]</td>
</tr>
<tr>
<td>('initialize global' angle to 30)</td>
<td>['initialize global' angle to 30]</td>
</tr>
<tr>
<td>(initialize $angle to 30)</td>
<td>[initialize $angle to 30]</td>
</tr>
<tr>
<td>(initialize #angle to 30)</td>
<td>[initialize #angle to 30]</td>
</tr>
<tr>
<td>(initialize @angle to 30)</td>
<td>[initialize @angle to 30]</td>
</tr>
<tr>
<td>(initialize %angle to 30)</td>
<td>[initialize %angle to 30]</td>
</tr>
<tr>
<td>global angle &lt;- 30</td>
<td>[global angle &lt;- 30]</td>
</tr>
<tr>
<td>$angle &lt;- 30</td>
<td>[$angle &lt;- 30]</td>
</tr>
<tr>
<td>#angle &lt;- 30</td>
<td>[#angle &lt;- 30]</td>
</tr>
<tr>
<td>@angle &lt;- 30</td>
<td>[@angle &lt;- 30]</td>
</tr>
<tr>
<td>%angle &lt;- 30</td>
<td>[%angle &lt;- 30]</td>
</tr>
</tbody>
</table>

Activity 39: Preference Ranking Task: Getting a Global Variable

This task asks for preference rankings on the following blocks and options:

Note: This block is so common that it's worth considering special-case shorthands that might differ from the standard way to handle blocks that begin with multiple words separated by spaces.

- (get global sum)
- (getglobal sum)
- (get_global sum)
- (getGlobal sum)
- (&lt;get global&gt; sum)
• ($get global$ sum)
• (#get global# sum)
• (#get global$ sum)
• ('get global' sum)
• (get global sum)
• $sum$
• sum
• #sum
• @sum
• %sum

Activity 40: Preference Ranking Task: Setting a Global Variable

This task asks for preference rankings on the following blocks and options:

Note: This block is so common that it’s worth considering special-case shorthands that might differ from the standard way to handle blocks that begin with multiple words separated by spaces.

Unfortunately, the Venbrace preference options posted during the Round 1 study were buggy: the wrong kind of braces (parens rather than curly braces) were used for the setter statements, and the abbreviated options were missing any kind of braces entirely. Table A.6 shows the buggy options during the Round 1 study and how they have been corrected for a future study (if the task is used again).
Table A.6: Buggy and Corrected Venbrace preference options for Buggy Venbrace in Round 1 and Corrected Venbrace for Future

<table>
<thead>
<tr>
<th>Buggy Venbrace in Round 1</th>
<th>Corrected Venbrace for Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set global n to 5)</td>
<td>{set global n to 5}</td>
</tr>
<tr>
<td>(setglobal n to 5)</td>
<td>{setglobal n to 5}</td>
</tr>
<tr>
<td>(set_global n to 5)</td>
<td>{set_global n to 5}</td>
</tr>
<tr>
<td>(setGlobal n to 5)</td>
<td>{setGlobal n to 5}</td>
</tr>
<tr>
<td>(&lt;set global&gt; n to 5)</td>
<td>{&lt;set global&gt; n to 5}</td>
</tr>
<tr>
<td>($set global$ n to 5)</td>
<td>{$set global$ n to 5}</td>
</tr>
<tr>
<td>(#set global# n to 5)</td>
<td>{#set global# n to 5}</td>
</tr>
<tr>
<td>(</td>
<td>set global</td>
</tr>
<tr>
<td>('set global' n to 5)</td>
<td>{'set global' n to 5}</td>
</tr>
<tr>
<td>$n &lt;- 5</td>
<td>{$n &lt;- 5}</td>
</tr>
<tr>
<td>#n &lt;- 5</td>
<td>{#n &lt;- 5}</td>
</tr>
<tr>
<td>@n &lt;- 5</td>
<td>{@n &lt;- 5}</td>
</tr>
<tr>
<td>%n &lt;- 5</td>
<td>{%n &lt;- 5}</td>
</tr>
</tbody>
</table>

Activity 41: Preference Typing Task: A Local Variable Declaration

This task asks the subject for their preferred Venbrace translation of:

A.11 Post-study Survey (Activity 42)

The post-study survey is presented in Figures A-3 and A-4.
Pre-Study Survey

Please begin the study by filling out all parts of this pre-study survey form.

Your age (in years): 

Your gender:  Female  Male  Nonbinary  Other  Prefer not to say

School/institution type:  choose a school type

If you selected "other" for school type, elaborate:

Name of your school/institution:

US State/territory in which your school/institution is located:  choose a state

Class year:  choose a class year

How long have you worked with App Inventor, in months? 

For each of the following App Inventor features, indicate your familiarity with that feature:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Unfamiliar</th>
<th>Somewhat Familiar</th>
<th>Very Familiar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Getting/setting component properties</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Conditionals (if/then/else)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loops (for each/while)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lists</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring/calling procedures that do not return a value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring/calling procedures that do return a value</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring/calling procedures that have zero parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring/calling procedures that have one or more parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Getting local variables (e.g., referring to procedure or event parameters)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declaring new local variables with an initialize local block</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generic components and methods</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switching between screens</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TinyDB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CloudDB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A-1: Pre-study survey (page 1).
Please indicate how familiar you are with the following environment(s)

<table>
<thead>
<tr>
<th>Environment</th>
<th>I've never used it</th>
<th>I've written 1 to 5 programs in it</th>
<th>I've written 5 to 10 programs in it</th>
<th>I've written &gt; 10 programs in it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scratch</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Snap!</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>MakeCode</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Code.org</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Pencil Code</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Alice</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Looking Glass</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

If there are blocks languages you have used that are not listed above, please list them here and indicate how familiar you are with them:

<table>
<thead>
<tr>
<th>Environment</th>
<th>I've written 1 to 5 programs in it</th>
<th>I've written 5 to 10 programs in it</th>
<th>I've written &gt; 10 programs in it</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

If applicable, below list any textual programming languages (e.g., Python, Java, JavaScript) you have used and indicate how familiar you are with them
(Note: you are not expected to have any text language experience for this study.)

<table>
<thead>
<tr>
<th>Language</th>
<th>I've written 1 to 5 programs in it</th>
<th>I've written 5 to 10 programs in it</th>
<th>I've written &gt;10 programs in it</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td></td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

(Optional) Use the area below to give feedback to the study team about this page. E.g.: Were any concepts confusing? Do you have ideas for improving it?

Submit this Survey

Figure A-2: Pre-study survey (page 2).
Post-Study Survey

Please complete the study by filling out all parts of this post-study survey form.

Reflecting on the textual language you used today, Venbrace, on a scale from easiest to hardest, please rate the ease of performing each of these tasks:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Very Easy</th>
<th>Fairly Easy</th>
<th>OK</th>
<th>Fairly Hard</th>
<th>Very Hard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translating mathematical expressions from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translating non-mathematical expressions from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translating statements from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translating variable declarations from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translating event handlers from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Translating variable references from blocks to text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching the parentheses/braces/brackets in text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching the built-in keywords in text</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coming up with your own text translation of a given assembly of blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reflecting on using the Venbrace language, on a scale from strongly agree to strongly disagree, please rate the following statements:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Strongly Agree</th>
<th>Somewhat Agree</th>
<th>Unsure</th>
<th>Somewhat Disagree</th>
<th>Strongly Disagree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Venbrace language precisely represents the App Inventor blocks language that I am already familiar with.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Having a textual language like Venbrace in App Inventor would be helpful for building programs.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If a textual language like Venbrace were available in regular App Inventor, I would be more likely to make changes to my apps using such a language.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A-3: Post-study survey (page 1).
Would you like App Inventor to have a textual language like Venbrace such that you could edit your program in the text mode?

◯ Yes  ◯ No

Please explain your answer from the previous question.


Please give any suggestions you have for how the Venbrace language could be improved in the future:


(Optional) Use the area below to give feedback to the study team about this page. E.g.: Were any concepts confusing? Do you have ideas for improving it?


Submit this Survey

Figure A-4: Post-study survey (page 2).
B  Round 1 Study Data

This appendix contains data collected from the Translation Typing tasks, the Preference Ranking tasks, and the Preference Typing tasks in the online user study. Data from the pre-study survey and post-study survey are already detailed in Section 7.4. The Translation Selection tasks, as stated in Section ??, are only considered an extra help to the participants for a better understanding of the Venbrace grammar in this round of study, so data collected from such tasks will not be included in this appendix.

B.1  Translation Typing

For this type of tasks, we report the results of TAM analysis, as well as the failed last attempts, i.e. last submitted responses to each task that were still invalid.

Task: Simple Expression

TAM Results

\[([', 1.0), (', 1.0), ('10', 1.0), (', 1.0), ('-', 1.0), (', 1.0), ('3', 1.0), (', 1.0), (', 1.0])\]

Failed Last Attempts

None
Task: Nested Math Expression 1

TAM Results

\[((',', 0.94), ('(', 1.0), ('(', 1.0), ('4', 1.0), ('', 1.0), ('*', 1.0), ('(', 1.0), ('9', 1.0), ('', 1.0), ('', 1.0), ('/', 1.0), ('(', 1.0), ('(', 1.0), ('35', 1.0), ('', 1.0), ('-', 1.0), ('(', 1.0), ('29', 1.0), ('', 0.94), ('', 1.0), ('', 1.0))\]

Failed Last Attempts

- ((4)*(9))/((35)-(29))

Task: Nested Math Expression 2

TAM Results

\[((', 1.0), ('(', 1.0), ('45', 1.0), ('', 1.0), ('*', 1.0), ('(', 1.0), ('absolute', 1.0), ('(', 1.0), ('18', 1.0), ('-', 1.0), ('31', 1.0), ('', 0.94), ('*', 1.0), ('6', 1.0), ('', 1.0), ('', 1.0))\]

Failed Last Attempts

- (45*(absolute(18-31)*6))
Task: Nested Math Expression 3

TAM Results

$[(('(', 1.0), ('max', 1.0), ('(', 1.0), ('(', 1.0), ('6', 1.0), (')'),
→ 1.0), ('*', 1.0), ('(', 1.0), ('4', 1.0), (')', 1.0), (')', 1.0),
→ ('(', 1.0), ('57', 1.0), (')'), 1.0), ('(', 0.88), ('(', 1.0),
→ (')'), 1.0), ('/', 1.0), ('(', 1.0), ('3', 1.0), (')'), 0.94), (')',
→ 1.0), (')', 1.0)]$

Failed Last Attempts

- $((\text{max}(6*4) (57) ()/3))$
- $(\text{max} (6*4) (57) (())/(3))$

Task: Expression with Booleans and Strings

TAM Results

$[(('(', 0.82), ('(', 1.0), ('not', 1.0), ('(', 1.0),
→ ('Button1.Enabled', 1.0), (')', 0.94), (')', 0.94), ('or', 1.0),
→ ('(', 0.94), (')', 0.94), ('length', 1.0), ('(', 1.0), ('"cat"',
→ 1.0), (')', 0.94), (')', 1.0), (')=', 0.94), ('(', 0.94), ('get',
→ 0.88), ('num', 0.94), (')', 0.76), (')', 1.0), (')', 1.0)]$

Failed Last Attempts

- $((\text{not Button1.Enabled}) \text{ or } ((\text{length}) ("cat") \geq (\text{get num})))$
• ((not(Button1.Enabled)) or ((length"cat")>=get(num)))
• ((not(Button1.Enabled)))or((length"cat")==(getnum))
• ((not (Button1.Enabled)) or ((length "cat") >= (get(num))))
• (not Button1.Enabled) or ((length ("cat"))>=)(get num))
• (not (Button1.Enabled) (or) (length "cat")(>=)(get num))
• (not (Button1.Enabled) or ((length "cat") => num))

Task: Simple Statement

TAM Results

[[('(', 0.94), ('set', 1.0), ('Label1.Text', 1.0), ('to', 1.0), ('(', 1.0), ('(', 1.0), ('3', 1.0), (')', 1.0), ('*', 1.0), ('(', 0.82), ('(', 0.88), ('TextBox1.Text', 0.88), (')', 0.94), ('+', 1.0), ('(', 0.88), ('get', 0.94), ('score', 0.94), (')', 0.71), (')', 0.94), ('(', 1.0), (')', 0.94)]]

Failed Last Attempts

• {set Label1.Text to (3*{TextBox1.Text + (get score)})}
• {set Label1.Text to (3*{TextBox1.Text + get(score)})}
• {set (Label1.Text) to (3*{TextBox1.Text + get score})}
• {set (Label1.Text) to (3*{TextBox1.Text}+ get(score))}
• set Label1.Text to ((3*TextBox1.Text)+score)
• {set(Label1.Text)to(3* (TextBox1.Text)+(get score))}
• {set Label1.Text to (3 * (TextBox1.Text + (get score)))}
• {set Label1.Text to ((3*)((TextBox1.Text)+(get score)))}
Task: Nested Statement

TAM Results

```plaintext
[(['', 0.88), ('if', 0.82), ('(', 0.76), ('(', 0.94), ('get', 1.0),
  ('num', 1.0), (''), 0.88), ('>', 1.0), ('(', 1.0), ('3', 1.0),
  (')', 0.82), (''), 0.94), ('then', 0.88), ('{', 0.88), ('set',
  0.88), ('Button1.Text', 1.0), ('to', 0.88), ('(', 1.0),
  ('"Greater"', 0.94), (',', 1.0), (',', 0.76), (',', 0.76), (',set',
  0.82), ('num', 0.88), ('to', 0.71), (',', 0.82), (',', 0.94),
  ('get', 1.0), ('num', 1.0), (''), 0.94), ('-', 0.94), ('(', 1.0),
  ('1', 0.94), (''), 1.0), (''), 0.94), (',', 0.88), ('else if',
  0.88), (',', 0.65), (',', 1.0), ('get', 0.94), ('num', 0.94),
  (''), 0.82), ('<', 0.94), (',', 0.94), (',3', 0.88), (',', 0.88),
  ('', 0.82), ('then', 0.82), (',', 0.76), ('set', 0.88),
  ('Button1.Text', 0.94), ('to', 0.88), (',', 1.0), ("Less",
  0.88), (',', 1.0), (',', 0.88), ('else', 0.88), (',', 0.76),
  ('set', 0.94), ('Button1.Text', 1.0), ('to', 0.94), (',', 1.0),
  ("Equal", 0.94), (',', 1.0), (',', 0.71), (',', 0.94)]
```

Failed Last Attempts

- if ((get num) > 3) then
  ```plaintext
  {set Button1.Text to "Greater"}
  {set num to ((get num) - 1)}
  ```
else if ((get num) < 3)
    {set Button1.Text to "Less"}
else
    {set Button1.Text to "Equal"}

• {if (num > 3) then {set Button1.Text to "Greater"}
    {set num (num - 1)}
else if (num < 3) then {set Button1.Text to "Less"}
    {set num to ((get num) - 1)}
else {set Button1.Text to "Equal"}}

• {if (num > 3) then
    {set Button1.Text to "Greater"}
    {set num (num - 1)}
else if (num < 3) then
    {set Button1.Text to "Less"}
else
    {set Button1.Text to "Equal"}
}

• {if num > 3 {set Button1.Text to 'Greater'} {set num to num - 1}}
  else if num < 3 {set Button1.Text to 'Less'}
  else set Button1.Text to 'Equal'

• if {(num > 3)}
    then {set Button1.Text to "Greater" set num to (num - 1)}
  else if {(num < 3)}
then {set Button1.Text to "Less"}
else {set Button1.Text to "Equal"}

• {if(num > 3)
  (then {{Button1.Text "Greater"}
       {num (num - 1)}})
  (else if (num < 3))
  (then (Button1.Text "Less"))
  (else Button1.Text "Equal")}

• {if get num>3 then
  (Button1.Text to"Greater")
  (set num to get num-1)
else if (get num <3)
  then (set Button1.Text to"Less")
else (set Button1.Text to"Equal")

• {if (get num)>3)
  {then (set Button1.Text to "Greater")set num to ((get num)-1)}
  {else if (get num )<(3)}
  {then (set Button1.Text to "Less"))}
  {else set Button1.Text to "Equal"}

• {if num > 3 then {set Button1.Text to "Greater"}{set num to (num
  \rightarrow -1}) else if {num < 3} then {set Button1.Text to "Less" }
  \rightarrow else {set Button1.Text to "Equal"})

• {If (get num>3)
  then {set Button1.Text to "Greater"}
  {Set num to(get num - 1})
  else if {{(get num<(-1))}}
  then {set Button1.Text to "Less"}
  else{set Button1.Text to "Equal"} }
• {if ((get num) > (3))
  
  {set Button1.Text to "Greater"}
  {set num to ((get num) - (1))}
  
  else if ((get num) < (3))
  
  then {set Button1.Text to "Less"}
  
  else {set Button1.Text to "Equal"}}

Task: Event Handler

TAM Results

[['[', 0.94), ('when', 1.0), ('Switch1.Changed', 0.94), ('do', 0.94),
  → ('{', 0.94), ('if', 1.0), ('(', 0.71), ('(', 0.88), ('length',
  → 0.94), ('(', 1.0), ('TextBox1.Text', 0.88), ('')', 0.65), ('')',
  → 1.0), ('<', 1.0), ('(', 1.0), ('0', 1.0), ('')', 0.94), ('')',
  → 0.88), ('then', 0.94), ('{', 0.94), ('if', 0.94), ('(', 1.0),
  → ('Switch1.On', 0.88), ('')', 0.94), ('then', 0.88), ('{', 0.88),
  → ('set', 1.0), ('Switch1.On', 1.0), ('to', 0.94), ('(', 1.0),
  → ('false', 0.94), ('')', 1.0), ('}', 0.76), ('}', 0.71), ('}',
  → 0.94), ('][', 0.94)],

Failed Last Attempts

• [when Switch1.Changed do {if ((length TextBox1.Text) < 0) to {if
  → Switch1.On then {set Switch1.On to false}}}]
• [when Switch1.Changed do { if (length(TextBox1.Text)<0)
    then {if (Switch1.On) then
    → {set(Switch1.On)to false}}}]}

• [when Switch1.Changed
    do {if length TextBox1.Text<0
    then {if Switch1.On
    then{set Switch1.On to false}}}]

• [when Switch1.Changed [if (length TextBox.Text < 0)
    then [if Switch1.On then {set Switch1.On false}]][]

• [when Switch1.Changed
    do {if ({length TextBox1.Text<0})
    then {(set Switch1.On to (false))}]

• [when Switch1.Changed do {if (length TextBox1.Text) <(0)} then
    → {if (Switch1.0n)} {then set (Switch1.0n) to (false)}]

• {when Switch1. Changed
    do {if (length TextBox1.Text < 0)}
    then {if Switch1.0n}
    then (set Switch1.0n to "false")}

• [when Switch1.Changed do
    {if(length (Textbox1.Text)<0}) then
    {if Switch1.on}
    then {set Switch1.on to false}]

• [when Switch1.Changed
    do {if ((length) (TextBox1.Text)< 0)
    then {if (Switch1.0n)
    then {set Switch1.0n to false}}}]
Task: SizeUp Procedure

TAM Results

[('[', 0.94), ('to', 0.88), ('SizeUpButton', 1.0), ('scale', 0.94),
  ('do', 1.0), ('{', 0.94), ('set', 0.94), ('Button1.Height', 1.0),
  ('to', 0.94), ('(', 0.94), ('(', 1.0), ('Button1.Height', 1.0),
  (')', 1.0), (')', 1.0), ('*', 1.0), ('(', 0.94), ('get', 0.94), ('scale',
  0.94), (')', 0.94), (')', 1.0), (')', 1.0), ('{', 0.94), ('set',
  1.0), ('Button1.Width', 1.0), ('to', 0.94), ('(', 0.88), ('(',
  1.0), ('Button1.Width', 1.0), (')', 1.0), (')', 1.0), (')', 0.94),
  (get', 0.94), ('scale', 0.94), (')', 0.88), (')', 1.0), (')',
  1.0), (')', 0.94)]

Failed Last Attempts

- [SizeUpButton scale do {set Button1.Height to (Button1.Height *
  scale)}
  {set Button1.Width to Button1.Width * scale}]

- [SizeUpButton scale do {set Button1.Height to (Button1.Height *
  scale)} {set Button1.Width to (Button1.Width * scale)}]

- [to SizeUpButton scale do
  {(set Button1.Height to (Button1.Height * get scale))}
  {(set Button1.Width to (Button1.Width * get scale))}]

- [to SizeUpButton {do set (Button1.Height) to
  (Button1.Height)*(get scale)}{do set (Button1.Width) to
  (Button1.Width)*(get scale)}]
• {to SizeUpButton scale
do {set Button1.Height to (Button1.Height * get.scale)}
set Button1.Width to (Button1.Width * get.scale)}

Task: ConcatString Procedure

TAM Results

[['', 0.94], ('to', 0.94), ('ConcatString', 0.94), ('s1', 0.88),
  ('s2', 0.88), ('result', 0.94), ('''', 0.94), ('join', 0.88), ('''', 0.94),
  ('get', 0.94), ('s1', 1.0), ('''', 0.94), ('''', 0.94),
  ('get', 0.94), ('s2', 1.0), ('''', 1.0), ('''', 0.88), ('''', 0.94)]

Failed Last Attempts

• [to ConcatString s1 s2 result
  ((get s1) join (get s2))]

• [to ConcatString s1 s2 result (join get(s1) get(s2))]

• [to ConcatString s1 s2 result
  (join (get s1 s2))]

• [{to ConcatString} {result {(join) (get s1) (get s2)}}]

• {{to ConcatString [s1] [s2]}
  result {join get s1 get s2}}

B.2 Preference Ranking

For this type of tasks, statistics about the rankings will be reported in tables. The numbers in the Mean and Median columns are evaluation scores transcribed from a
Likert scale, while the numbers in other columns refer to the number of responses submitted under each column header.

We optionally asked the subjects to submit ideas of better options for translations. For those who submitted such ideas, their raw responses are reported as well.
### Task: Degree Conversion

#### Table B.1: Preference ranking for Degree Conversion

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(convert degrees to radians 60)</td>
<td>4.35</td>
<td>4.0</td>
<td>7</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(&lt;convert degrees to radians&gt; 60)</td>
<td>3.65</td>
<td>4.0</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(convertDegreesToRadians 60)</td>
<td>3.53</td>
<td>4.0</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(convert_degrees_to_radians 60)</td>
<td>3.35</td>
<td>4.0</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>('convert degrees to radians' 60)</td>
<td>2.88</td>
<td>3.0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>($convert degrees to radians$ 60)</td>
<td>2.65</td>
<td>3.0</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(convertdegreestoradians 60)</td>
<td>2.59</td>
<td>3.0</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>(</td>
<td>convert degrees to radians</td>
<td>60)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>(#convert degrees to radians# 60)</td>
<td>2.41</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>(convert%degrees%to%radians 60)</td>
<td>1.65</td>
<td>2.0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

#### Better Options

- I don’t have an option for something I like better than the above choices, just that I like options which use easy to type/common delimiters if they’re used at all. I guess the only improvement is to use common abbreviations so that in all the cases "degrees to radians" could become "deg to rad".
- (convert (degrees to radians) 60)
• I wouldn’t be opposed to (and in fact if this had been an option, I would have strongly liked it)

(convert _DegreesToRadians 60)

which I think nicely separates the beginning "convert" directive from the conversion that’s happening, while also avoiding the problem of dealing with MULTIPLE words in the beginning separated by spaces.

• {convert to radians 60}
Task: Modulo

![modulo of](17 + 5)

Table B.2: Preference ranking for Modulo

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(modulo 17 / 5)</td>
<td>3.94</td>
<td>4.0</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>(17 % 5)</td>
<td>3.59</td>
<td>4.0</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>(modulo 17 5)</td>
<td>3.06</td>
<td>3.0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>(modulo 17 % 5)</td>
<td>2.94</td>
<td>3.0</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(% 17 5)</td>
<td>2.53</td>
<td>3.0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

Better Options

- This could be a little confusing since it would break the rules of having items around how expressions always have ()’s around them. I want to put (modulo (17 / 5)) or (mod (17 / 5)).
- Is there a reason why the "of" was dropped?: (modulo_of 17 5) or (modulo_of 17 / 5)
  Seems a bit inconsistent with other Venbrace syntax.
  I somewhat prefer the options with / since it follows the convention of translating divide signs to /.
- (mod 17%5)
- (modulo (17/5 r))
- (5 modulo 17)
Task: Color Red

Table B.3: Preference ranking for Color Red

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(color red)</td>
<td>4.06</td>
<td>4.0</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>color.red</td>
<td>3.94</td>
<td>4.0</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>#red</td>
<td>3.88</td>
<td>4.0</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>3.88</td>
<td>4.0</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(red)</td>
<td>3.76</td>
<td>4.0</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Color.red</td>
<td>3.47</td>
<td>4.0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>&lt;red&gt;</td>
<td>3.47</td>
<td>3.0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>red</td>
<td></td>
<td>3.00</td>
<td>3.0</td>
<td>1</td>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

Better Options

- colorRed might be nice too.
Task: Make a List

Table B.4: Preference ranking for Make a List

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(list 5 8 4)</td>
<td>3.88</td>
<td>4.0</td>
<td>8</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(make a list 5 8 4)</td>
<td>3.82</td>
<td>4.0</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(&lt;make a list&gt; 5 8 4)</td>
<td>3.47</td>
<td>3.0</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(make_a_list 5 8 4)</td>
<td>3.41</td>
<td>3.0</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>(makeAList 5 8 4)</td>
<td>3.29</td>
<td>3.0</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(makealist 5 8 4)</td>
<td>2.88</td>
<td>3.0</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>('make a list' 5 8 4)</td>
<td>2.82</td>
<td>3.0</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(#make a list# 5 8 4)</td>
<td>2.35</td>
<td>2.0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>(</td>
<td>make a list</td>
<td>5 8 4)</td>
<td>2.35</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

Better Options

- I also strongly like (5 8 4).
- have (make a list 5 8 4) (or whatever version ends up being most popular) and (list 5 8 4) be synonyms?
Task: Empty List

Table B.5: Preference ranking for Empty List

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(create empty list)</td>
<td>4.00</td>
<td>4.0</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(empty list)</td>
<td>3.94</td>
<td>4.0</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(createEmptyList)</td>
<td>3.59</td>
<td>4.0</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(emptyList)</td>
<td>3.53</td>
<td>4.0</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(create_empty_list)</td>
<td>3.53</td>
<td>3.0</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>(emptylist)</td>
<td>3.41</td>
<td>3.0</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(empty_list)</td>
<td>3.35</td>
<td>3.0</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(list)</td>
<td>3.00</td>
<td>3.0</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>(createemptylist)</td>
<td>2.82</td>
<td>3.0</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>(empty)</td>
<td>2.18</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

Better Options

- it would be nice if there were options, like (create empty list) [and the many equivalent options] and (empty list) [and its many equivalent options] essentially being synonyms.
Task: Infix Abbreviation for Setters

Table B.6: Preference ranking for Infix Abbreviation for Setters

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>{num = 3}</td>
<td>4.18</td>
<td>5.0</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>{num := 3}</td>
<td>3.35</td>
<td>3.0</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>{num -&gt; 3}</td>
<td>3.29</td>
<td>4.0</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>{num &lt;- 3}</td>
<td>2.88</td>
<td>3.0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>{num =: 3}</td>
<td>2.65</td>
<td>3.0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>{num &lt;-&gt; 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>{num :: 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>{num :=: 3}</td>
<td>2.47</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>{num -- 3}</td>
<td>2.41</td>
<td>2.0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>{num -=- 3}</td>
<td>2.00</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

Better Options

- I strongly like num == 3
- num == 3
- num == 3
**Task: Event Handler with Parameters**

![Diagram of Canvas1.TouchDown event handler](image)

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>[when Canvas1.TouchDown &lt;x&gt; &lt;y&gt; do</td>
<td>3.76</td>
<td>4.0</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>{set Label1.Text to (join x &quot;&quot;,&quot; y))}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[when Canvas1.TouchDown x y do</td>
<td>3.59</td>
<td>4.0</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>{set Label1.Text to (join x &quot;&quot;,&quot; y))}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[when Canvas1.TouchDown</td>
<td>x</td>
<td></td>
<td>y</td>
<td>do</td>
<td>3.47</td>
<td>3.0</td>
<td>4</td>
</tr>
<tr>
<td>{set Label1.Text to (join x &quot;&quot;,&quot; y))}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[when Canvas1.TouchDown do</td>
<td>3.06</td>
<td>3.0</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>{set Label1.Text to (join x &quot;&quot;,&quot; y))}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[when Canvas1.TouchDown #x #y do</td>
<td>2.82</td>
<td>3.0</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>{set Label1.Text to (join x &quot;&quot;,&quot; y))}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

**Better Options**

- [when Canvas1.TouchDown $x$ $y$ do
  
  {set Label1.Text to (join x "," y)}]
### Task: Calling the average Procedure

#### Table B.8: Preference ranking for Calling the average Procedure

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(average a: num b: 10)</td>
<td>4.06</td>
<td>4.0</td>
<td>6</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(average a num b 10)</td>
<td>3.65</td>
<td>4.0</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(average &lt;a&gt; num &lt;b&gt; 10)</td>
<td>3.35</td>
<td>3.0</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(average num 10)</td>
<td>3.29</td>
<td>3.0</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>(average #a num #b 10)</td>
<td>2.94</td>
<td>3.0</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>(average b 10 a num)</td>
<td>2.29</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>(average b: 10 a: num)</td>
<td>2.24</td>
<td>2.0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>(average &lt;b&gt; 10 &lt;a&gt; num)</td>
<td>2.00</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>(average #b 10 #a num)</td>
<td>1.82</td>
<td>2.0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

#### Better Options

- (average $a$ num $b$ 10)
- (average a:= num b:= 10)
Task: Declaring a Global Variable

The options below that were given to the subjects were buggy: the parens should have been square brackets, and all of the abbreviated options are missing brackets.

Table B.9: Preference ranking for Declaring a Global Variable

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(initialize global angle to 30)</td>
<td>4.12</td>
<td>4.0</td>
<td>7</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(initializeGlobal angle to 30)</td>
<td>3.65</td>
<td>4.0</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(initialize_global angle to 30)</td>
<td>3.53</td>
<td>4.0</td>
<td>1</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(&lt;initialize global&gt; angle to 30)</td>
<td>3.35</td>
<td>3.0</td>
<td>5</td>
<td>1</td>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(global angle &lt;- 30)</td>
<td>3.00</td>
<td>3.0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>([initialize global] angle to 30)</td>
<td>2.94</td>
<td>3.0</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>('initialize global' angle to 30)</td>
<td>2.88</td>
<td>3.0</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(initialize $angle to 30)</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(initializeglobal angle to 30)</td>
<td>2.76</td>
<td>3.0</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>(initialize #angle to 30)</td>
<td>2.71</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>(#initialize global# angle to 30)</td>
<td>2.65</td>
<td>3.0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>($initialize global$ angle to 30)</td>
<td>2.59</td>
<td>3.0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>(initialize @angle to 30)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>(initialize %angle to 30)</td>
<td>2.41</td>
<td>2.0</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>@angle &lt;- 30</td>
<td>2.29</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>$angle &lt;- 30</td>
<td>2.24</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>%angle &lt;- 30</td>
<td>2.12</td>
<td>2.0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>#angle &lt;- 30</td>
<td>2.06</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.
Better Options

- global angle = 30 or global angle = 30

**Task: Getting a Global Variable**

Table B.10: Preference ranking for Getting a Global Variable

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(get global sum)</td>
<td>4.29</td>
<td>4.0</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(global sum)</td>
<td>4.29</td>
<td>4.0</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(getGlobal sum)</td>
<td>3.47</td>
<td>3.0</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(&lt;get global&gt; sum)</td>
<td>3.41</td>
<td>3.0</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(get_global sum)</td>
<td>3.24</td>
<td>3.0</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>('get global' sum)</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>$sum</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>@sum</td>
<td>2.71</td>
<td>3.0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>(getglobal sum)</td>
<td>2.65</td>
<td>3.0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>($get global$ sum)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(#get global# sum)</td>
<td>2.53</td>
<td>3.0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>#sum</td>
<td>2.41</td>
<td>3.0</td>
<td>0</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>(</td>
<td>get global</td>
<td>sum)</td>
<td>2.41</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>%sum</td>
<td>2.18</td>
<td>2.0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.
Better Options

- I extra like the @sum option.
- possibly global.sum
### Task: Setting a Global Variable

The options below that were given to the subjects were buggy: the parens should have been curly braces, and all of the abbreviated options are missing curly braces.

Table B.11: Preference ranking for Setting a Global Variable

<table>
<thead>
<tr>
<th>Option</th>
<th>Mean</th>
<th>Median</th>
<th>Strongly Like</th>
<th>Somewhat Like</th>
<th>Meh</th>
<th>Somewhat Dislike</th>
<th>Strongly Dislike</th>
</tr>
</thead>
<tbody>
<tr>
<td>(set global n to 5)</td>
<td>4.29</td>
<td>4.0</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(setGlobal n to 5)</td>
<td>3.53</td>
<td>4.0</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>(&lt;set global&gt; n to 5)</td>
<td>3.41</td>
<td>3.0</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(set_global n to 5)</td>
<td>3.18</td>
<td>3.0</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(setglobal n to 5)</td>
<td>2.82</td>
<td>3.0</td>
<td>1</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>(</td>
<td>set global</td>
<td>n to 5)</td>
<td>2.76</td>
<td>3.0</td>
<td>0</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>('set global' n to 5)</td>
<td>2.65</td>
<td>3.0</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>@n &lt;- 5</td>
<td>2.59</td>
<td>3.0</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>$n &lt;- 5</td>
<td>2.59</td>
<td>3.0</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>($set global$ n to 5)</td>
<td>2.59</td>
<td>3.0</td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(#set global# n to 5)</td>
<td>2.47</td>
<td>3.0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>#n &lt;- 5</td>
<td>2.24</td>
<td>2.0</td>
<td>0</td>
<td>1</td>
<td>7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>%n &lt;- 5</td>
<td>2.12</td>
<td>2.0</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

The numbers in the Mean and Median columns are evaluation scores according to a Likert scale, where Strongly Like counts as 5 and Strongly Dislike counts as 1. The numbers in other columns refer to the number of responses submitted under each column header.

**Better Options**

- @n to 5
\[ n = 5 \]

\[ \text{(setGlobal $n$ to 5)} \]

- set global.n to 5

- G'n <- 5

### B.3 Preference Typing

For this type of tasks, raw responses submitted by the participants are presented.

**Task: While Loop**

```
{while (x < 100)
do {set x to (2*x)}}
```

```
{while ((get x) < 100)
{set x to (2 * (get x))}}
```

```
{while test (x < 100)
do {set x to (2 * x)}}
```

```
{while (x<100) do {x = (2*x)}}
```

```
(while test (x<100) do (set x to (2*x))
```

```
{while ((get(x)) < 100) do
set{x to (2*(get x))}}
```
{while test (x < 100) do
    {set x to (2 * x)}
}

{while (x < 100) do {set x (2 * x)}}

{while test (x<100) do {set x to (2*x)}

do{set x to (2*x)

{while test(x<100)
}{set x to (2*x)} }

{while test (x < 100) do [set (x) to (2 * get (x))]

{while test (get x<100)}
do (set x to (2*get x)

{while test (get x)<(100)}
{do (set x) to (2 * get x)}

{while test (x < 100) do {set x to (2 * x)}}

{if while test((get num)<(100))
then do {set x to ((2)*(get x))}}

{while test ((get x) < (100))
   Do {set (x) to ((2) * (get (x)))}
Task: For Loop

{for each n from 1 to limit by 5
do {Canvas1.DrawPoint x=n y=(n*n)}}

{for each "n" from 1 to (get limit) by 5
    {Canvas1.DrawPoint (get n) ((get n) * (get n))}}

{for each n from 1 to limit by 5
do {Canvas1.DrawPoint (n) (n * n)}}

{for each n from 1 to (limit) by 5
do {Canvas1.DrawPoint (x is n) (y is (n*n))}}

{for each n from (1) to (get limit) by (5) do {call Canvas1.Drawpoint
    (x=n) (y= n*n)}}

{for each (n) from (1) to (get limit) by (5) do
    {call Canvas1.DrawPoint x = (get n) y = (get n) * (get n)}}

{for each n from 1 to limit by 5
do {Canvas1.DrawPoint n (n * n)}}

{for n from (1 ; limit) by (5) do
for each n (from 1 to limit) by 5
    do { call Canvas1.DrawPoint(x get n)((y get n) * (get n))}

for each n from 1 to limit by 5
    do { call Canvas1 {x(n)} {y(n*n)}}

    { for each n (1:limit:5) do
        { call Canvas1.DrawPoint (x=n) (y = (n*n))} }
{For each n from (1) to (get limit) by(5)
do {call Canvas1.DrawPoint x(get n) y ((get n)*(get n))}}

{ for each “n” 1 limit 5
  Do {call Canvas1.DrawPoint n , (n * n)} }
\{add \ items \ to \ list \ (get \ L) \ 7 \ 4\} \ OR
\{add \ items \ to \ list \ list:(get \ L) \ item:7 \ item:4\}

OR
\{add \ items \ to \ list \ list \ (get \ L) \ item \ 7 \ item \ 4\}

(though the last two seem overly verbose, and then we would not be
able to have item as a variable name)

\{add \ (7 \ 4) \ to \ (list \ L)\}

\{add \ items \ to \ list \ list\(get \ L\)(item \ 7)(item \ 4)\}
Task: addPoint Procedure

[to addPoint y
do {Canvas1.DrawPoint y (2*y)}]

[to addPoint $y do
   {Canvas1.DrawPoint (get y) (2 * (get y))}]

[to addPoint y do
   {call Canvas1.DrawPoint y (2*y)}]

[to addPoint y do
   {call Canvas1.Draw x:(y) y:(2*y)}
]

[to addPoint y do
   {call Canvas1.DrawPoint x=(y) y=(2*y)}]

[to addPoint y do
   {call Canvas1.DrawPoint (x (get y)) (y (2* get y))}]

[to addPoint y do
   {call Canvas1.DrawPoint y (2 * y)}]
[to addPoint y do {call Canvas1.DrawPoint <x> y <y> (2*y)}]

[to addPoint y do {call Canvas1.DrawPoint(x (get (y)) y (get (2 * get(y)))}]

[to addPoint y do {call Canvas1.DrawPoint <x>(y) <y> (2*y)}]

[to addPoint y do {Canvas1.DrawPoint (x=y) (y=(2*y))}]

[to addPoint <y> do {call Canvas1.DrawPoint x (get <y>) y (2 * (get <y>)}

[to"addPoint" y do
call Canvas1.DrawPoint
x get y
y (2*get y)]

[to add point (y) do {call Canvas1.DrawPoint (x get y) (y 2*gety)

{to addPoint (y)
do {call Canvas1.DrawPoint
x = get y
y = (2 * y)}

[to addPoint y
do{call canvas1.drawPoint

x(get y)
y((2)*(get y))

[to addPoint y do
  {call Canvas1.DrawPoint x:y y:(2 * y)}]

Task: A Local Variable Declaration

[to squaredSum n m
result (sum*sum) where (initialize local sum to (n+m))]}

[to squaredSum $n $m result
  |initialize local sum to ((get n) + (get m)) in
  ((get sum) * (get sum))]}

[to squaredSum n m result
  (initializeLocal sum to (n + m) in (sum * sum))]

[to squaredSum n m
result (initialize local sum to (n+m) in sum*sum)
]

[to squaredSum n m result
{initialize local sum to (n+m)
in (sum*sum)}]
[to squaredSum n m
result
  {initialize local(sum) to ((get n) + (get m))
in ((get sum)*(get sum))}]

[to squaredSum n m result
  (initialize local sum to (n + m) in
   (sum * sum))]
or
[to squaredSum n m result
  (initialize local sum to (n + m) in
   (sum * sum))]

[to squaredSum n m result {initialize local sum to (n + n) in (sum * sum)}]

[to squaredSum n m result {initialize local sum to (get (n)+ get(m) in
   (get(Sum)* get(sum))}]

[to squaredSum n m
result {initialize localsum to (num + m) in (sum * sum)}]

[to squaredSum n m result
  {initialize local sum to (n+m)
in (sum*sum)}]
]

{to squaredSum <n> <m>
result (initialize local <sum> to (<n> + <m>) in (<sum> * <sum>)}
[to "squaredSum" n m
result "initialize local "sum" to (get n + get m)
{in (get sum * get sum)}]

[to squaredSum {result initialize local sum to ((get n)+(get m)) in
~ (get sum)*(get sum)]

{to squaredSum n m
result {initialize localsum -> (n + m)
in (sum * sum)}

[to squaredsum n m
result(initializes localsum to(get n)+(get m)
in((get sum)*(get sum))]

[to squaredSum n m result
   [initialize local sum to (get n + get m)
   In (get sum * get sum)]]

Task: Globals with Procedures

(initialize global total to 0)

[to addToTotal num
do {set global total to (num + global total)}]
[@total to 0]

[to addToTotal $num do
   {@total to ((get num) + @total)}]

(init global total to 0)
[to addToTotal num do
   {set global total to (num + global total)}]

[initialize global.total to 0]
[to addToTotal num do {set global.total to (num + global.total)}]

(initialize global total to 0)
[to addtoTotal num do {set global total to (num+global total)}]

{initialize global total to 0}
[to addToTotal num do {set global total to ((get num)+(get global total))}]

[total = 0]

[to addToTotal num do {set (global total) to (num + (global total))}]

[global total <- 0]
[to addToTotal num do {set @total to (num + @total)]
initialize global total to 0
[ to addToTotal num do { set global total to (get (num)+ get(global total))}]

(initialize global total to 0)
[to addToTotal num do {set global total to (num + global total)}]

(<initialize global> total to 0)
[to addToTotal num do
  {<set global> total to (num + (<get global> total))}
]

[%total <- 0]
{to addToTotal <num> do {set %total to (num + %total)}

[initialize global "total" to 0
 [{to "addToTotal "num" do
  {set global total to (get num + get global total)}}]]

initialize global 'total' to (0)
[to addToTotal do {set golbal total to ((get num)+(get global total))}]

{initialize (total) to (0)}
{addToTotal "num"
do {set "global total" to (num / global total)}}

initialize global total to 0
[to addToTotal num
do[set global total to ((get num)+(get global total))]

[initialize global total to 0]

[to addToTotal num do
 {set global total to ((get num) + (get global total))}]]
C    Venbrace.g4

This is the ANTLR4 specification file for the VENBRACE parser and lexer. It only includes syntax rules for which a finalized/tentative design decision has been made. Other work-in-progress elements not covered in the body of this thesis document are removed from the code presented in this appendix.

/*
Venbrace grammar (simplified version)

Author: Ruangiangian Huang
Modified: May 4, 2020

FEATURES:
- Text representations for *some* App Inventor blocks

NOT INCLUDED:
- Venbrace-to-XML (only for testing purposes, still work-in-progress)
- New rules currently under development that are not covered in the thesis body
*/

// ===========GRAMMAR MACROS==========
grammar Venbrace;

options
{
    language = 'JavaScript';
    tokenVocab=VenbraceLexer;
}

@lexer::members{
    var idType = "label";
    var errors = [];
    VenbraceLexer.prototype.emitErrorMessage = function(error) {
        //var hdr = getErrorHeader(e);
        //var msg = getErrorMessage(e, tokenNames);
        errors.push(error);
    }
}
VenbraceLexer.prototype.getErrors = function() {
    return errors;
}

VenbraceLexer.prototype.setTokenFactory = function(newFactory) {
    factory = newFactory;
}

@parser::members{

    var errors = [];

VenbraceParser.prototype.emitErrorMessage = function(error) {
    //var hdr = getErrorHeader(e);
    //var msg = getErrorMessage(e, tokenNames);
    errors.push(error);
};

VenbraceParser.prototype.getErrors = function() {
    return errors;
};

VenbraceParser.prototype.recoverFromMismatchedToken = function(input, ttype, follow){
    throw new antlr4.runtime.MismatchedTokenException(ttype, input);
}

document.createVenbraceElement = function (tagName) {
    var element = document.createElement(tagName);
   element.appendChildReal = element.appendChild; // save original appendChild method
    element.appendChild = function (child) {
        if (child) {
            element.appendChildReal(child); // Only append child if not null
        }
    }
    return element;
}

// catching grammar rule errors
@rulecatch{
    catch (re){
        throw re;
    }
program: (decl_block)* EOF;

decl_block: LSQR decl RSQR;

decl:
  | global_decl
  | procedure_do
  | procedure_result
  | event_handler
  |

global_decl: INITIALIZE GLOBAL ID TO expr_block;

procedure_do: TO proc_name=ID (arg_name=ID)* DO suite;

procedure_result: TO proc_name=ID (arg_name=ID)* RESULT expr_block;

// only supports event handlers without built-in parameters
event_handler: WHEN COMPONENT_PROPERTY DO suite;

expr_block: (LPAREN RPAREN)
  | atom
  | expr;

suite: (stat_block)*
  | LCURLY RCURLY; // empty statement sequence

stat_block: LCURLY stat RCURLY;

stat: control_stat
  | call_procedure_stat
  | var_stat;

// STATEMENT BLOCKS
/* Feb 28, 2020: keep only if_stat for study 1 */
control_stat: if_stat;

if_stat: IF e1=expr_block THEN s1=suite ((ELSE_IF e2=expr_block THEN s2=suite)*
  → (ELSE s3=suite)?)?;

// 03/23/20: only allows procedure calls w/o arguments
// 04/20/20: made 'call' optional

if_stat: IF e1=expr_block THEN s1=suite ((ELSE_IF e2=expr_block THEN s2=suite)*
  → (ELSE s3=suite)?)?;

var_stat: setter;

setter: SET ((GLOBAL? ID) | (COMPONENTPROPERTY)) TO expr_block;

// EXPR BLOCKS

expr: var_expr
  | (LPAREN
    | control_expr
    | logic_expr
    | not_expr
    | compare_eq_expr
    | compare_math_expr
    | math_expr
    | str_expr
    | call_procedure_expr
    | atom
   RPAREN)

control_expr: if_expr | do_expr;

if_expr: IF c=expr_block THEN e1=expr_block ELSE e2=expr_block;

do_expr: DO suite RESULT expr_block;

logic_expr: a=expr_block (AND | OR) b=expr_block;

not_expr: NOT expr_block;

compare_eq_expr: a=expr_block (LOGIC_EQ | LOGIC_NOT_EQ) b=expr_block;
compare_math_expr: a=expr_block (EQ | NEQ | GT | GE | LT | LE) b=expr_block;

// only contains blocks that have no long sequences of helper words
math_expr: mutable_op
| immutable_op
| min_max
| unary_op;

mutable_op: a=expr_block ((PLUS b=expr_block)+ | (MUL c=expr_block)+);

// not allowing a + b * c + d
immutable_op: (a=expr_block (MINUS | DIV | POW) b=expr_block)
| (a=expr_block NEG_NUM); // special hack handling a -b

min_max: (MIN | MAX) a=expr_block (b=expr_block)+;

// only contains blocks without long sequences of helper words
unary_op: (ABS | NEG | LOG | EULER | ROUND | CEILING | FLOOR) expr_block;

str_expr: str_join | str_length;

str_length: LENGTH expr_block;

str_join: JOIN a=expr_block (b=expr_block)+;

var_expr: getter;

getter: ID | (LPAREN GET (GLOBAL)? ID RPAREN);

// 04/20/20: made `call' optional; only supports zero-argument calls
call_procedure_expr: CALL? (ID | (component=ID DOT event=ID));

// OTHER ELEMENTS
atom: NUMBER | NEG_NUM | STRING | TRUE | FALSE | COMPONENT_PROPERTY;

/* ================================
* ==========LEXER RULES==========
* ================================ */

WS : ( ' ' )
| 't' |
| 'r' |
| 'n' |

) -> skip;

// === TOKENS ===

// SYMBOLS
LCURLY: '{';
RCURLY: '}';
LPAREN: '(';
RPAREN: ')';
LSQR: '[';
RSQR: ']';
DOT: '.';

// KEYWORDS
TRUE: 'true';
FALSE: 'false';
WHEN: 'when';
IF: 'if';
THEN: 'then';
ELSE: 'else';
ELSE_IF: 'else if';
DO: 'do';
RESULT: 'result';
TO: 'to';
CALL: 'call';
GET: 'get';
SET: 'set';
GLOBAL: 'global';
IN: 'in';
INITIALIZE: 'initialize';

// OPERATORS
NOT: 'not';
AND: 'and';
OR: 'or';
LT: '<';
GT: '>';
LE: '<=';
GE: '>=';
LOGIC_EQ: 'equals';
LOGIC_NOT_EQ: 'not equals';
EQ: '=';
NEQ: '!=';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
POW: '^';

// Unary Ops
ABS: 'absolute';
NEG: 'neg';
LOG: 'log';
EULER: 'e^';
ROUND: 'round';
CEILING: 'ceiling';
FLOOR: 'floor';
JOIN: 'join';
LENGTH: 'length';

// Math Ops
MIN: 'min';
MAX: 'max';

// INTEGER
fragment DIGIT : ('0'..'9');
fragment HEX_DIGIT : (DIGIT | 'a'..'f' | 'A'..'F');
NUMBER : ((DIGIT* DOT DIGIT+) | (DIGIT+ (DOT)?) | ('0x' (HEX_DIGIT)+));
NEG_NUM : MINUS NUMBER;

// CHARACTERS
fragment ALPHA : ('a'..'z' | 'A'..'Z');
fragment ESC : '\\';

// Identifiers
COMPONENT_PROPERTY: ID DOT ID;
ID : (ALPHA | '_') (ALPHA | '_' | DIGIT)*;

// STRING
STRING: ('\'' (ESC | ~(\'' | '\n' | '\\'))* '\''
       | ('"' (ESC | ~(""" | '\n' | '"'))* '"');
D Implementation of TAM in JavaScript

This is the Implementation for the TAM algorithm in JavaScript. The code is extracted from a script written for the Online Syntax User Study. The script includes other code related to the study, which is not displayed here.

```javascript
class Cell {
  constructor(row, col) {
    this.row = row;
    this.col = col;
    this.value = 0;
    this.prev = null; //Cell
    return this;
  }
  toString() {
    return this.value;
  }
}

function CalcTable() {
  var table = CreateTable();
  var rows = table.length;
  var cols = table[0].length;
  for (var r = 1; r < rows; r++) {
    for (var c = 1; c < cols; c++) {
      CalcCell(table, r, c);
    }
  }
  return table;
}

function CreateTable() {
  var cols = solutionTokens.length + 1;
  var rows = valueTokens.length + 1;
  var table = new Array(rows);
  for (var i = 0; i < table.length; i++) {
    table[i] = new Array(cols);
  }
  for (var r = 0; r < rows; r++) {
    for (var c = 0; c < cols; c++) {
```
for (var c = 0; c < cols; c++) {
  var entry = new Cell(r, c);
  if (r === 0 && c === 0) {
    entry.value = 0;
  }
  else if (r === 0) {
    entry.prev = table[0][c - 1];
    entry.value = entry.prev.value - 2;
  }
  else if (c === 0) {
    entry.prev = table[r - 1][0];
    entry.value = entry.prev.value - 2;
  }
  table[r][c] = entry;
}
return table;

function CalcCell(table, row, col) {
  var cell = table[row][col];
  var cellAbove = table[cell.row - 1][cell.col];
  var cellLeft = table[cell.row][cell.col - 1];
  var cellDiag = table[cell.row - 1][cell.col - 1];
  var scoreFromAbove = cellAbove.value - 2;
  var scoreFromLeft = cellLeft.value - 2;
  var scoreFromDiag = cellDiag.value;
  // Daleiden's implementation compares the token types;
  // My implementation compares the *textual content* of the tokens
  if (solutionTokens[cell.col - 1].text == valueTokens[cell.row - 1].text) {
    scoreFromDiag += 1;
  }
  else
    scoreFromDiag -= 1;
  if (scoreFromAbove >= scoreFromLeft) {
    if (scoreFromDiag >= scoreFromAbove) {
      cell.value = scoreFromDiag;
      cell.prev = cellDiag;
    }
    else {
      cell.value = scoreFromAbove;
      cell.prev = cellAbove;
    }
  }
  else {
    cell.value = scoreFromAbove;
    cell.prev = cellAbove;
  }
}
if (scoreFromDiag >= scoreFromLeft) {
    cell.value = scoreFromDiag;
    cell.prev = cellDiag;
}
else {
    cell.value = scoreFromLeft;
    cell.prev = cellLeft;
}
return false;
}

Cell.prototype.constructor = Cell;

function Backtrace(table) {
    var solutionTokensAligned = [];
    var valueTokensAligned = [];
    var current = table[table.length-1][table[0].length-1];
    while (current.prev !== null) {
        if ((current.row - current.prev.row) == 1) {
            valueTokensAligned.unshift(valueTokens[current.row - 1]);
        } else {
            valueTokensAligned.unshift(null);
        }
        if ((current.col - current.prev.col) == 1) {
            solutionTokensAligned.unshift(solutionTokens[current.col - 1]);
        } else {
            solutionTokensAligned.unshift(null);
        }
        current = current.prev;
    }
    var n = solutionTokensAligned.length;
    var result = new Array(2);
    result[0] = solutionTokensAligned.map((t) => {
        if (t !== null)
            return {
                "text": t.text, "type": t.type;
            return null;});
    result[1] = valueTokensAligned.map((t) => {
        if (t !== null)
            return {
                "text": t.text, "type": t.type;
            return null;});
    return result;
}