Improving the Usability of App Inventor through Conversion between Blocks and Text

Karishma Chadha

April 25, 2014

Abstract

In blocks programming, users compose programs by combining visual fragments (blocks) shaped like jigsaw-puzzle pieces. The shapes suggest how the blocks fit together, reducing syntactic frustrations experienced by novices when learning textual programming. MIT App Inventor 2 (AI2), a popular online environment for Android app development, democratizes programming through its easy-to-use blocks language. However, while simple blocks programs are easy to read and write, complex ones become overwhelming. Creating and navigating nontrivial blocks programs is tedious, and AI2's current inability to copy blocks between projects inhibits reusing blocks code or sharing blocks code with others.

To address these issues, I have created a new textual language, TAIL, that is isomorphic to AI2's blocks language and provided a means for converting between them. TAIL syntax is designed to provide users with a systematic way to translate the visual information on the blocks into text. I have extended AI2 with a set of code blocks (for expressions, statements, and top-level declarations) in which users can type TAIL code representing AI2 blocks. These code blocks have the same meaning as the larger block assemblies they represent. Programmers can also convert back and forth be-

tween these code blocks and the original AI2 blocks. Language isomorphism guarantees that a round-trip conversion (from text to blocks and back, or blocks to text and back) begins and ends with the identical program.

To implement the TAIL language, I wrote a grammar for the ANTLR parser generator to generate a JavaScript lexer and parser for TAIL. I use actions in the grammar to translate the TAIL parse tree into AI2's XML tree representation for blocks.

This project aims to (1) increase AI2's usability by providing an efficient means for reading, constructing, sharing, and reusing programs, and (2) ease users' transitions from blocks programming to text programming.

Contents

Chapter 1 Introduction **1**

1.1 Blocks Programming Languages 1

1.2 MIT App Inventor 3

1.3 Limitations in App Inventor 6

1.3.1 Writing Complex Programs 6

1.3.2 Reading and Searching Programs 7

1.3.3 Reusing or Sharing Programs 8

1.4 One Solution to Fix them All 9

1.5 Road Map 14

Chapter 2 Related Work **15**

2.1 Scratch 16

2.1.1 Scratch Project Summary 16

2.1.2 ScratchBlocks 17

2.2 Blockly 20

2.3 PicoBlocks 21

2.4 SLASH 24

2.5 Cabana 26

2.6 Previous Work on Blocks & Text Conversion in App Inventor 26
2.6.1 The Java Bridge 27
2.6.2 Rendering Android App Inventor Code Blocks as Pseudo-Python Code 28
2.6.3 Venthon: A First Take at a Creating a New Textual Language for App Inventor 28
2.7 Code to Blocks 33

Chapter 3 TAIL Language Design 35
3.1 Design Principles 35
3.1.1 Language Isomorphism 35
3.2 Easy Transition to TAIL 41
3.2.1 TAIL Syntax 41

Chapter 4 Integrating into App Inventor & Creating the TAIL Language 49
4.1 Architecture 49
4.1.1 Description of AI2 Architecture 49
4.1.2 Extending AI2 with TAIL Code Blocks 50
4.1.3 Blocks to Text Converter 54
4.1.4 Creating TAIL 57
4.2 Implementation of TAIL & Text to Blocks Conversion 60
4.2.1 What is a Parser Generator? 60
4.2.2 Lexing with ANTLR 61
4.2.3 Parsing with ANTLR 62
4.2.4 Tree Conversion with ANTLR 64

Chapter 5 Conclusion and Future Work 67
5.1 Current State 67
5.2 Immediate Future 68
5.2.1 Adding all AI2 Blocks to the TAIL Grammar 68
5.2.2 Creating YAIL Generators for Statements and Declarations 70
5.3 Near Future 71
5.3.1 User Studies 71
5.3.2 Changing Conversion Architecture to Improve Performance 72
5.3.3 Converting Screens, Workspaces, and Entire Projects . 73
5.3.4 Making Text Readable 75
5.3.5 Generalizing TAIL Grammar Rules 76
5.3.6 Adding Language Abbreviations 76
5.3.7 Improving Error Handling 78
5.3.8 Creating Tutorials & Documentation for AI2 Users . . 78
5.4 Far Future 78
5.4.1 Venthon, Venti, and More! 79
5.4.2 TAIL Text Editor 79
5.4.3 Blocks Language for Existing Text Language 79
5.4.4 Interactive Environment to Edit Corresponding Blocks and Text Languages Side-By-Side 80

Chapter 1

Introduction

1.1 Blocks Programming Languages

In blocks programming languages, users compose programs by combining visual program fragments shaped like jigsaw-puzzle pieces (*blocks*). The shapes of the blocks strongly suggest how they should fit together, thereby reducing common syntax errors experienced by novices when learning to program in a textual language. Blocks are usually chosen from a collection (see discussion of *drawers* in Section 1.2) which are often categorized by the type of functionality the blocks provide. These collections are often visually similar in some way. In many programming languages, blocks belonging to different collections are often distinguished by different colors (i.e. all the blocks pertaining to a single collection will be the same color, but a different color from blocks of a different collection). This means that users do not necessarily have to recall names of blocks by memory, but rather can recognize them by color. Blocks also contain other useful visual information which is not nec-

essarily readily available in textual programming languages. For example, blocks expecting other blocks as inputs often have labels indicating what input they are expecting. Even more simply useful visual information is that, at the very least, users will have an idea of how many inputs a block takes based on the shape of the block (i.e. the number of *sockets* the block has; see Section 3.2.1 for an explanation of sockets). For all of these and many more reasons, blocks programming languages are a powerful tool for lowering the barriers to learning computer science. With these programming environments, novices can focus on learning the concepts, thinking, and problem solving skills associated with computer science principles rather than being hindered by the frustrations of syntax errors that differ in each language.

Blocks programming languages have become quite popular in recent years. Many such languages have arisen such as: Scratch [Scra], an online environment to create games and animations; PicoBlocks [Pic], an environment to program mini-computers to use with physical LEGO [Leg] blocks; Blockly [Bloa], a blocks language framework that has been used to create numerous blocks-based microworlds, including the incredibly popular Hour of Code [Hou] microworlds; and MIT App Inventor [Ai1a; Ai2a], an online environment to develop mobile applications for Android devices.

Other blocks languages that are worth mentioning, but not discussed in this paper are StarLogo TNG [Staa], StarLogo Nova [Stab], Tynker [Tyn], and Hopscotch [Hop].

1.2 MIT App Inventor

MIT App Inventor began as a Google project that is now being developed and maintained in the MIT Media Lab as part of the Center for Mobile Learning. It has gained wide popularity in recent years, reaching over 1.5 million users worldwide [Ai1b; Ai2b]. App Inventor aims to democratize programming through its easy to use blocks programming language, and by removing much of the overhead of the Android SDK, so that even those who are not familiar with programming can easily build Android applications.

MIT App Inventor has gone through two major iterations, App Inventor Classic (*AI1*) [Ai1a] and App Inventor 2 (*AI2*) [Ai2a]. The overarching design of both iterations is the same. The tool consists of two parts, the Designer (Figure 1.1) and the Blocks Editor (Figure 1.2).

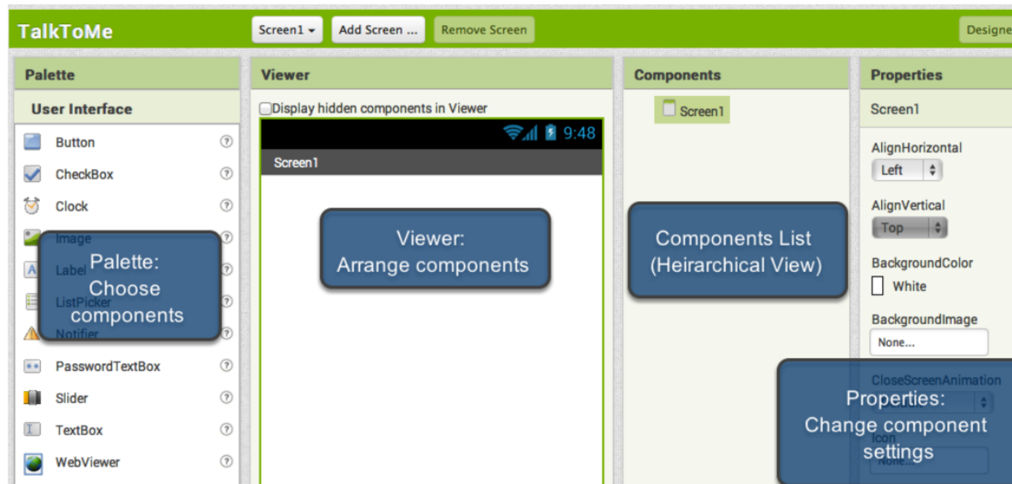


Figure 1.1: The Designer window for AI2

The Designer is responsible for creating the look of the application as well as defining any components the application will use. A component can

be a visual part of the application (such as a button or a text box), or a non-visual part of the application responsible for communicating with the device's hardware (such as a timer, or interfaces interacting with the device's camera or location sensor).

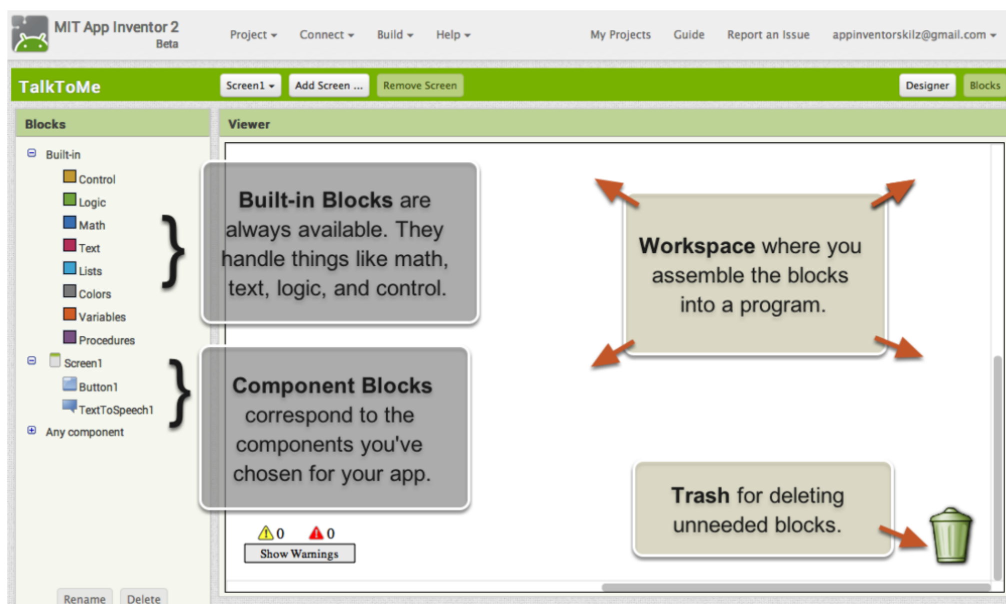


Figure 1.2: The Blocks Editor window for AI2

The Blocks Editor is the environment in which users can program the functionality of the components they have added to their application (using the Designer). The Blocks Editor uses a blocks programming language for users to code their applications. The blocks programming language used in AI1 was based on the MIT Open Blocks framework [Ope], a Java library created as part of the MIT STEP program for defining blocks programming languages. The second iteration of App Inventor (AI2) uses a blocks programming language designed in JavaScript using Blockly, being developed at

Google.

Both iterations of the Blocks Editor (and many other blocks programming environments) use the simple design of *drawers* of blocks, and a "workspace". In App Inventor, there is a list of drawers on the left hand side of the Blocks Editor and a blank area on the right hand side for users to construct their programs. The drawers are organized in categories based on what kind of functionality they provide (`control`, `math`, `lists`, `variables`, etc.).

The blocks are different colors (colored by the colors of their drawer) so that they are easier to visually identify and distinguish from one another. Users drag blocks they want to use from these drawers out into the workspace to compose their programs. Figure 1.3 shows some simple constructions of blocks.

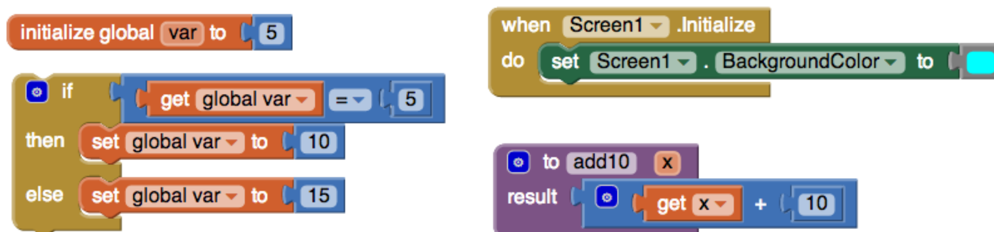


Figure 1.3: Simple Blocks Constructions

App Inventor programs, called *projects* consist of several *screens* defined in the Designer. Each screen has a set of components (also defined in the Designer) and a blocks program (defined in the Blocks Editor).

1.3 Limitations in App Inventor

App Inventor is a low-barrier tool for users of all backgrounds to write Android applications. However, App Inventor has a few key limitations.

1.3.1 Writing Complex Programs

Blocks programing languages are easy to learn, and users are able to create programs quickly in these languages. However, while it is easy and quick to write simple programs, it becomes increasingly tedious and time consuming to construct more complex programs. For some users, it may prove to be easier and faster to type text than to click, drag, and connect blocks together. Complex programs quickly become inefficient to create. Functions as simple as the quadratic formula, half of which is depicted in Figure 1.4, use many blocks (note that Figure 1.4 uses collapsed variable reference blocks; the fully expanded blocks would greatly increase the length of the entire block structure). Making an error in composing the blocks the first time adds even more inefficiency because the users are then required to unplug and replug several blocks to correct the error(s). Even a seasoned blocks programmer is likely to make some of these frustrating errors.

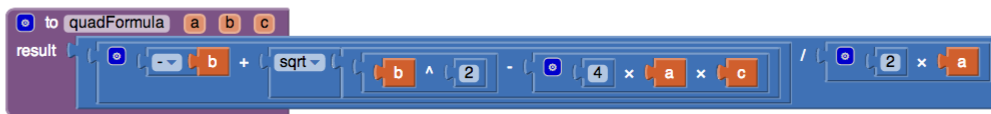
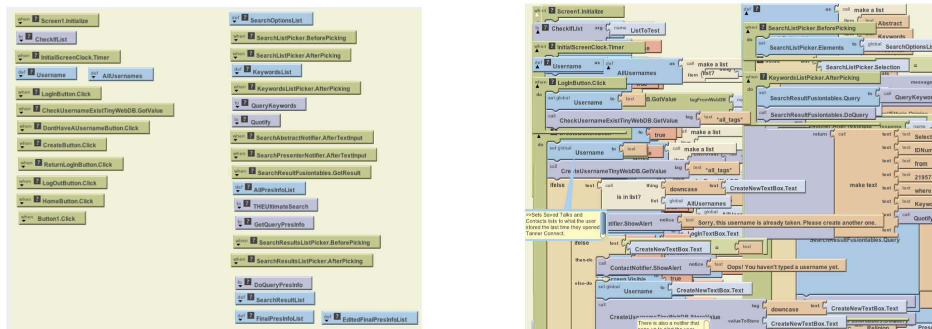


Figure 1.4: Half of the Quadratic Formula as represented in AI2

1.3.2 Reading and Searching Programs

As users get the hang of programming in App Inventor, their applications become larger and more complex. Figure 1.5a and Figure 1.5b depict the workspace for the TannerConnect app [Tan], a mobile application (in AI1) created by two students, Sonali Sastry and Charlene Lee, in an introductory CS class [Cs1] at Wellesley College teaching mobile application development using AI1.



(a) Workspace with Collapsed Blocks (b) Workspace with Expanded Blocks

Figure 1.5: TannerConnect application [Tan] developed by Sonali Sastry and Charlene Lee

It is extremely difficult for someone (even the project owners) to read this program or search it for a particular piece of code. Figure 1.5b shows what the workspace looks like when all of the collapsed blocks in Figure 1.5a are expended using App Inventor's `expand all` feature. App Inventor does offer an organization feature which arranges the blocks so that they are aligned and not overlapping, however even with all of the blocks organized, the program is very difficult to read. Additionally App Inventor does not currently offer any searching features; if someone reading the program needs to find a particular piece of code, he/she must shuffle through the blocks of the entire program

to find it.

The difficulty of reading and searching programs is currently a very large limitation of App Inventor as there are many people who would wish to read one's program. These parties include the developers of the app themselves, App Inventor users who wish to learn from another project (perhaps this project is hosted in the App Inventor gallery, a place where users can post their projects publicly), or in the case where App Inventor is taught in a Computer Science curriculum, instructors who wish to provide feedback and support to their students having trouble developing these programs. Thus, making App Inventor easier to read is a high priority.

1.3.3 Reusing or Sharing Programs

App Inventor does not currently offer a means of sharing projects or replicating/reusing parts of AI2 blocks programs across projects. The only way to achieve either of these tasks is for users to download the projects (in the form of archived files) and send them to the person (or people) they wish to share the projects with. If the receiving user(s) wishes to incorporate parts of one project into another (even just between two of the user's own projects), the only way to accomplish this would to be to view the two projects side by side and reconstruct the part in question in the second project. This is extremely inefficient, and inhibits users from reusing blocks code in multiple projects and sharing blocks code with others.

1.4 One Solution to Fix them All

Textual languages make writing, reading, searching, sharing and reusing programs much easier. In order to address the issues outlined above, I have created a new textual language, TAIL, the Textual App Inventor Language, that is isomorphic to AI2's blocks programming language, and provided a mechanism for converting between arbitrary AI2 block assemblies and TAIL code.

TAIL syntax is designed to provide users with a systematic way to translate the visual information on the blocks into text.

I have extended AI2 with a set of code blocks (Figure 1.6) for users to specify TAIL code for expressions, statements, and top-level declarations, representing original AI2 blocks. These code blocks have the same meaning as the larger block assemblies they represent.

In order to ease the ability to read blocks programs, I provide the ability to convert between AI2 blocks and TAIL code. Users can convert any set of AI2 blocks into TAIL code using the TAIL code blocks I have added. Section 5.3.3 talks about converting entire screens, workspaces, and projects into TAIL, allowing users to view the code pertaining to the app in a more concise, textual form, easy for reading. To facilitate the writing and sharing of AI2 programs, I provide the ability to write App Inventor code in TAIL, using my added TAIL code blocks, and convert the TAIL text in the code blocks into the original AI2 blocks language. This improves upon the current ability to share projects or fragments of projects because users can duplicate code across projects by first converting projects into TAIL, and copy-pasting TAIL code into a separate project to duplicate code across projects. Users

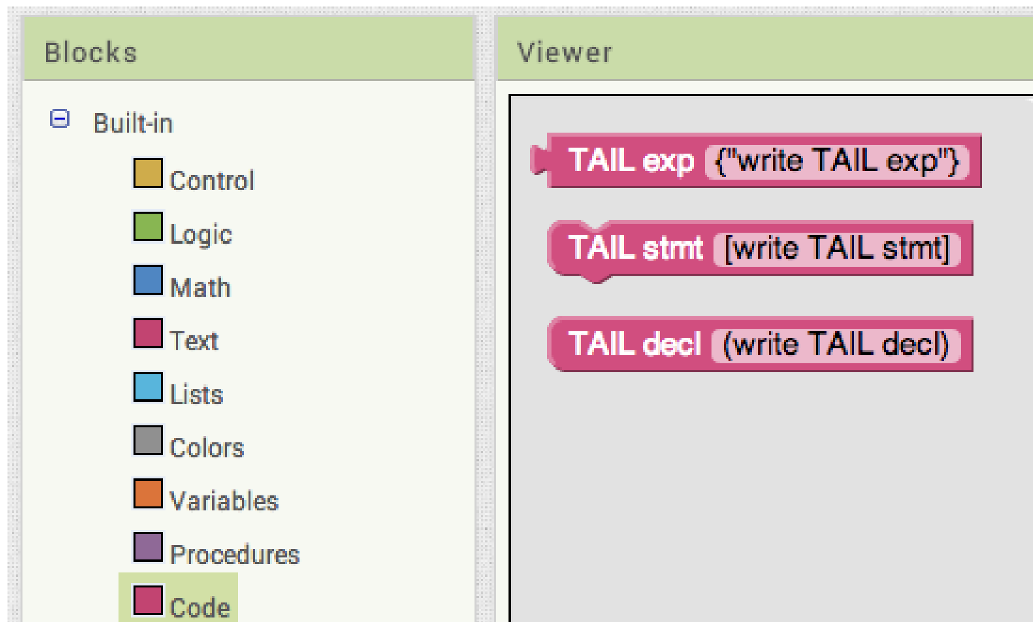


Figure 1.6: TAIL Code Blocks I have added to App Inventor

can convert this TAIL code back into blocks to continue programming with the AI2 blocks if they prefer. This project also provides the ability to write code in TAIL. The TAIL code is semantically equivalent to the AI2 blocks code it represents, thus allowing users to write AI2 programs solely with TAIL if they prefer, or to combine TAIL and AI2 blocks together using the provided TAIL code blocks I added. Language isomorphism guarantees that round trip conversions (from blocks to text and back or text to blocks and back) begin and end with the identical program. Figure 1.7 to Figure 1.11 shows some sample conversion in both directions.

If the user starts out with a set of AI2 blocks he/she wishes to convert to TAIL, the user can click an option in the context menu `Convert to TAIL` (Figure 1.8) which will then transform the set of AI2 blocks rooted at the

Figure 1.7: AI2 Blocks to be Converted into TAIL Code Blocks

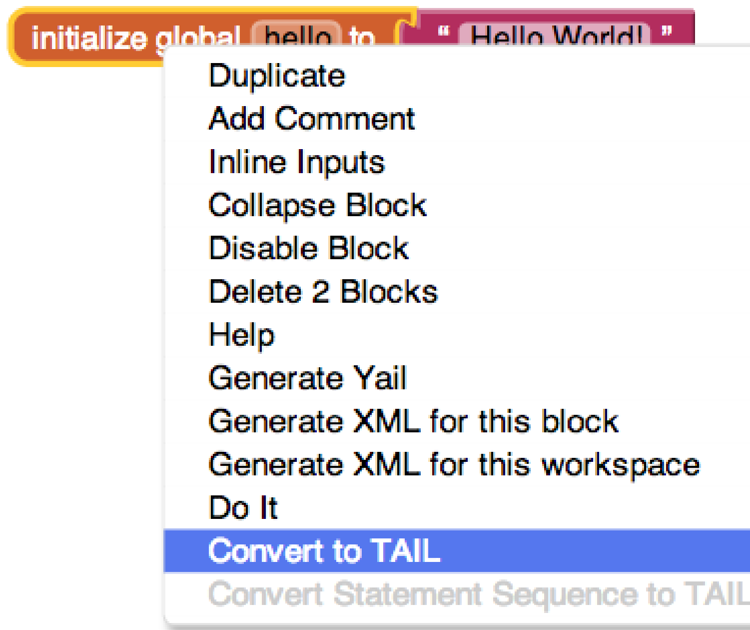


Figure 1.8: Click Context Menu Option for Converting Blocks to TAIL

Figure 1.9: Resulting TAIL Code Block

clicked block, into the TAIL code blocks, replacing the original blocks. The resulting TAIL code block has the same functionality as the AI2 blocks.

Users can choose which blocks to convert, regardless of their position in

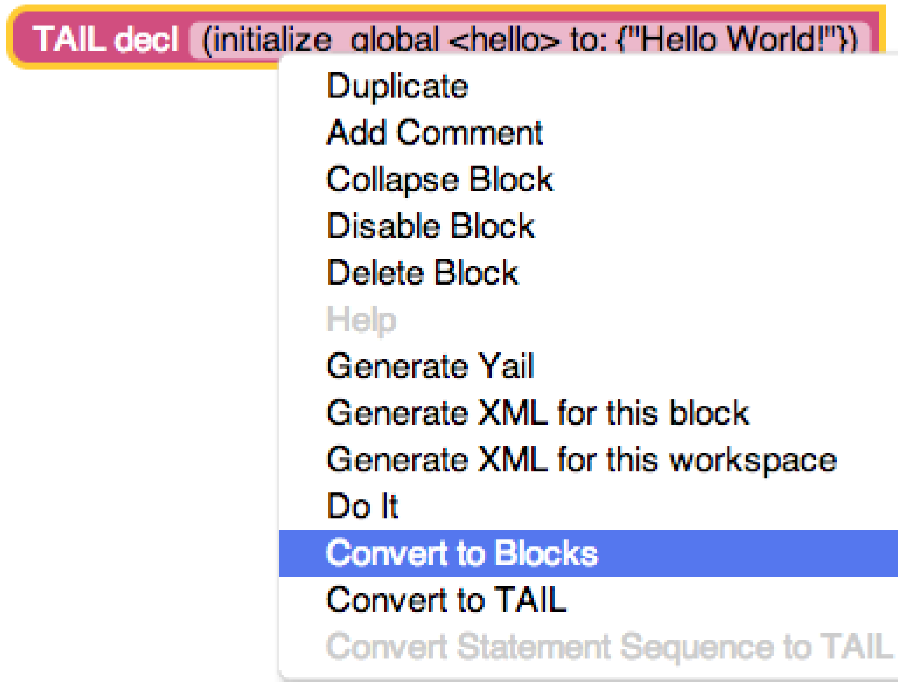


Figure 1.10: Converting TAIL to AI2 Blocks

Figure 1.11: Conversion yields the original AI2 Blocks

the syntax tree. While Figures 1.7 to 1.11 depict the round-trip conversion of a top-level, global variable declaration block, Figures 1.12 to 1.14 depicts three possible conversions of the same set of AI2 blocks. Each figure represents a conversion of the blocks at a different level of the syntax tree for the set of blocks or TAIL text. All of the 6 sets of blocks (2 sets of blocks in each figure) carry the same semantics even though the representations are

different. These figures also show that it is possible to use just AI2 blocks, just a single TAIL text block, or even a combination of AI2 blocks and TAIL text blocks in any given AI2 program.

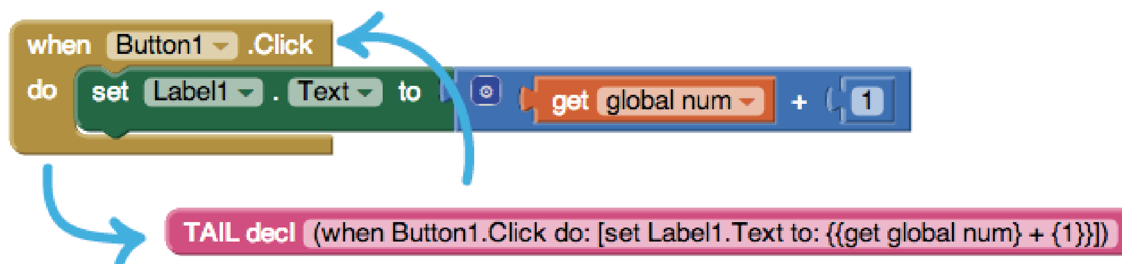


Figure 1.12: Conversion of top level event handler

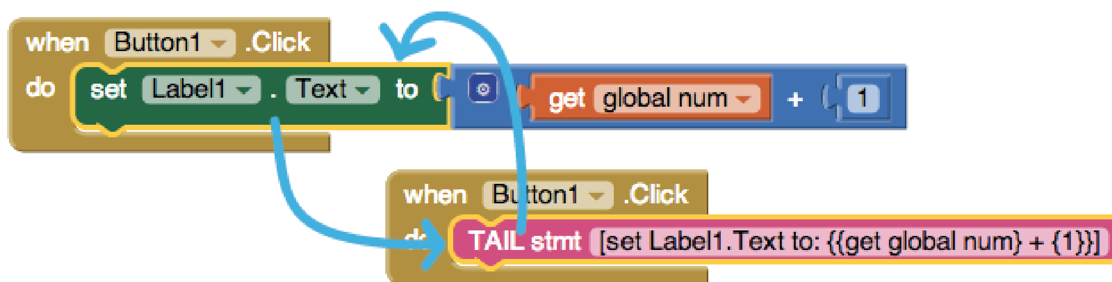


Figure 1.13: Conversion of statement block inside event handler

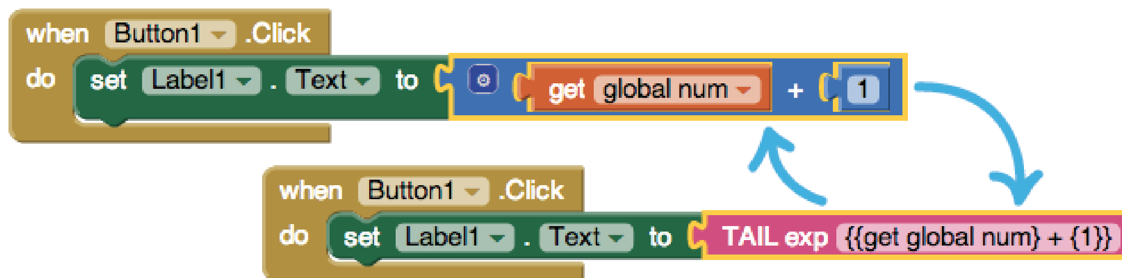


Figure 1.14: Conversion of nested expression block

This project aims to (1) increase AI2's usability by providing an efficient

means for reading, constructing, and sharing programs, and (2) ease users' transitions from blocks programming to text programming.

1.5 Road Map

The rest of this paper is organized as follows:

- Chapter 2 discusses work that is related to this project in the contexts of exploring the relationship between visual and textual programming languages, and the various notions of *converting to text*.
- Chapter 3 gives an overview of the TAIL language design process, from general guiding principles to a detailed discussion of the TAIL syntax.
- Chapter 4 discusses the architecture and implementation details of this project.
- Chapter 5 details the current state of this project, what remains to be done, and ideas for future related projects, to further the work done here.

Chapter 2

Related Work

Research of related projects reveals that there are varying notions of what it means to "convert blocks to text." As I have mentioned above, my aim for this project is to create and provide a means of converting to/from a textual languages that is semantically and syntactically isomorphic to AI2's blocks language. This entails matching TAIL syntax to each and every block in AI2 and designing TAIL in such a way that round trip conversions between the two languages yield the the original code that began the conversion (regardless of whether the conversion begins with blocks or text). I talk more about the design of TAIL in Chapter 3.

In this chapter, I talk about other projects, both related and unrelated to App Inventor that explore the relationship between blocks programming languages (or other visual programming languages) and textual programming languages, and the different notions of converting blocks to text in each of these projects.

2.1 Scratch

Scratch is a popular blocks programming environment for creating animations and games. There have been two unique notations of representing Scratch blocks programs in text form.

2.1.1 Scratch Project Summary

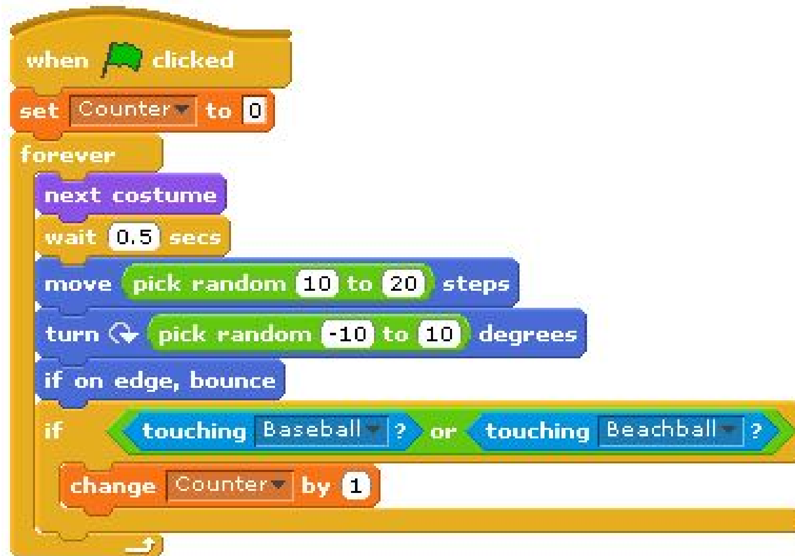
Desktop versions (i.e. the non-web-based versions) of Scratch allow users to *summarize* their programs in a text format. The context menu option allowing this conversion is slightly obscured. Users must hold down the `shift` key and click `File` in the Scratch menu in order to view the option `Write Project Summary`, which creates a text version of the Scratch blocks code in an output file.

As App Inventor projects consist of multiple screens with code associated for each screen, Scratch projects consist of multiple sprites (animated objects) and scripts associated for each sprite in the project.

The `Write Project Summary` option converts the block scripts for each sprite and compiles all of the scripts into one file. Figure 2.2 is the project summary file of the sample blocks in Figure 2.1 [Scrb] depicts the textual project summary of a sample scratch script.

In addition to a text form of all of the scripts in the project, the project summary file also includes other potentially useful information such as "revision history and summary of sprites and sounds used". [Scrb]

The generated text is a read-only summary of the Scratch blocks. It is not executable code, and it cannot be converted back into Scratch blocks.



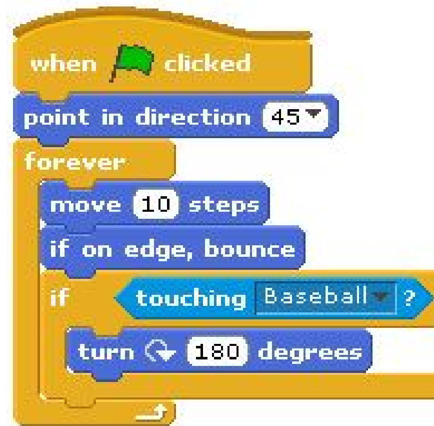


Figure 2.1: Scratch block scripts for different sprites

2.1.2 ScratchBlocks

Users of Scratch, sometimes referred to as "Scratchers", have developed plugins [Scrc; Scrd] that allow users to specify Scratch blocks in an HTML-like syntax for use on the Scratch Wiki, or the Scratch user forums for users

```
Sprite: Bat
  Costumes (2):
    bat1-a (172x244)
    bat1-b (172x244)
  Sounds (0):
  Stacks (1):
    when green flag clicked
        set "Counter" to "0"
        forever
            next costume
            wait 0.5 secs
            move (pick random 10 to 20) steps
            turn  (pick random -10 to 10) degrees
            if on edge, bounce
            if ((touching s[ ]?) or (touching s[ ]?))
                change "Counter" by 1
    end
--------
Sprite: Beachball
  Costumes (1):
    beachball1 (80x80)
  Sounds (0):
  Stacks (2):
    when green flag clicked
    end     point in direction 45
    forever
        move 10 steps
        if on edge, bounce
        if (touching s[Baseball]?)
            turn  180 degrees
```

Figure 2.2: Project summary file corresponding to Figure 2.1

or developers to refer to specific sets of Scratch blocks in their posts on either medium. The first of these plugins, *ScratchBlocks* was later replaced by *ScratchBlocks2*, but for the intents of the discussion here in this section, they provide the same functionality: the ability to translate a piece of text (with a specific syntax) into images of Scratch blocks (or a set of blocks) for use in the Scratch wiki and forums. This is perhaps the most similar but also the most different from the relationship between TAIL and AI2 Blocks. ScratchBlocks is its own language responsible for rendering images of Scratch blocks, whereas TAIL code can stand alone and carries the se-

mantics of the AI2 blocks code it represents. Figure 2.3 shows a sample rendering of a ScratchBlocks script as an image of blocks in Scratch (as it might be displayed on the Scratch Wiki). The documentation for this plugin notes that the plugin "tries to match the code you write as closely as possible, and doesn't check [for] correct syntax" [Scrc]. There are additional tools that users of the wiki and the forums can use that will "take blocks directly from a Scratch project, and turn them into text [to] paste inside a `<scratchblocks>` tag". This plugin, in conjunction with some tools provided by other Scratchers [Scre; Scrf], allows for two kinds of conversions: from blocks inside a scratch project, to the ScratchBlocks script text (which will then render this text as an image on the Scratch Wiki); and just the latter half of the conversion mentioned, converting the ScratchBlocks script text into an image of a set of Scratch blocks.

It is important to note that the ScratchBlocks and ScratchBlocks2 plugins generate text code that is only for use on the Scratch Wiki and Scratch forums (and not in the actual Scratch blocks programming environment). However, there are additional tools which can convert Scratch projects [Scre] or scripts that have been added to the Scratch backpack (a mechanism Scratch uses to share/reuse code between projects) [Scrf] to the ScratchBlocks text for use on the Wiki/forums.

This take on conversions between blocks and text is entirely different from that of the project discussed in this paper. These plugins allow the conversion from Scratch blocks to the text that is responsible for creating an *image* of the original Scratch blocks that the text represents. This entirely different view on blocks to text (to image) conversions puts an entirely different spin

on the relationship between different representations of programs.

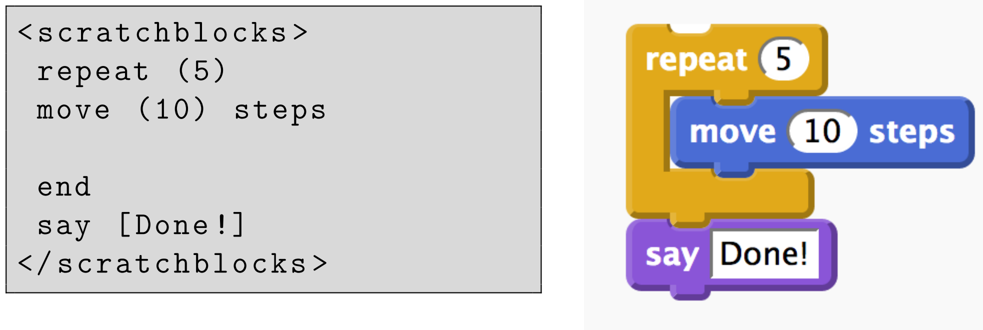


Figure 2.3: ScratchBlocks script rendered as an image of a set of blocks in Scratch

2.2 Blockly

Bockly is a web-based blocks programming environment being developed at Google. Blockly is referred to as a "visual programming editor"; however, the underlying concept of Blockly is the same as the blocks languages used in Scratch and MIT App Inventor 1 and 2. In fact, in the transition from the Java applet to the JavaScript based blocks editor, AI1 transitioned from the MIT OpenBlocks framework to using Blockly as the underlying blocks language framework for AI2. The project wiki for Blockly includes a number of sample applications built in Blockly which demonstrate different possible uses of the environment. One such sample application, *Code* [Blob] allows exporting a Blockly program into JavaScript, Python, Dart, or XML.

It is worth noting that while this particular application of Blockly does allow for conversion of Blockly code to executable code in any one of the four textual programming languages listed above, conversion in the other direction

is not possible. This is in part due to the fact that Python, JavaScript, Dart, and XML respectively each do not have one-to-one correspondences with the blocks language of Blockly. Arbitrary code written in one of these textual languages cannot be translated into Blockly because the textual languages are not isomorphic to Blockly.

Language *isomorphism* is one of the main design principles that guiding the creation of TAIL. The work for this project is motivated by the necessity for App Inventor users to be able to not only convert AI2 blocks into text, but also to be able to write arbitrary code in a textual language (TAIL), and be able to convert this textual code to the visual syntax of AI2 blocks. Isomorphism of the two languages guarantees that any round trip conversions will yield the original code (whether the origins were AI2 blocks or TAIL text).

Because it is possible to translate from any Turing-complete textual programming language to any other, translating from Blockly to the particular languages mentioned here doesn't say much about the specific relationship between blocks and text languages.

2.3 PicoBlocks

PicoBlocks, from The Playful Invention Company [Pic], is another blocks-programming environment based on research from the MIT Media Lab. This software is intended for use with physical devices (mini-computers) called PicoCrickets (Figure 2.4) which can be used alongside LEGO pieces to create interesting mobile, interactive projects. The PicoBlocks programming envi-

Figure 2.4: A PicoCricket

ronment is used to program these PicoCrickets which can be combined with traditional (and non-traditional) lego pieces as indicated above.

Figure 2.5, an image obtained from the PicoBlocks reference manual [Pic], depicts an overview of the PicoBlocks environment and how it is used.

In addition to the blocks language, PicoBlocks includes a built-in textual language for users to define new blocks. While the PicoBlocks *blocks language* also allows users to define their own blocks (see Figure 2.6 and Figure 2.7), blocks with *inputs* cannot be specified in the blocks language, but rather only in the text language. Thus, the blocks language is meant to be used in conjunction with the textual language for more advanced users to be able to create more complex code.

PicoBlocks does not provide a means for conversion between the blocks

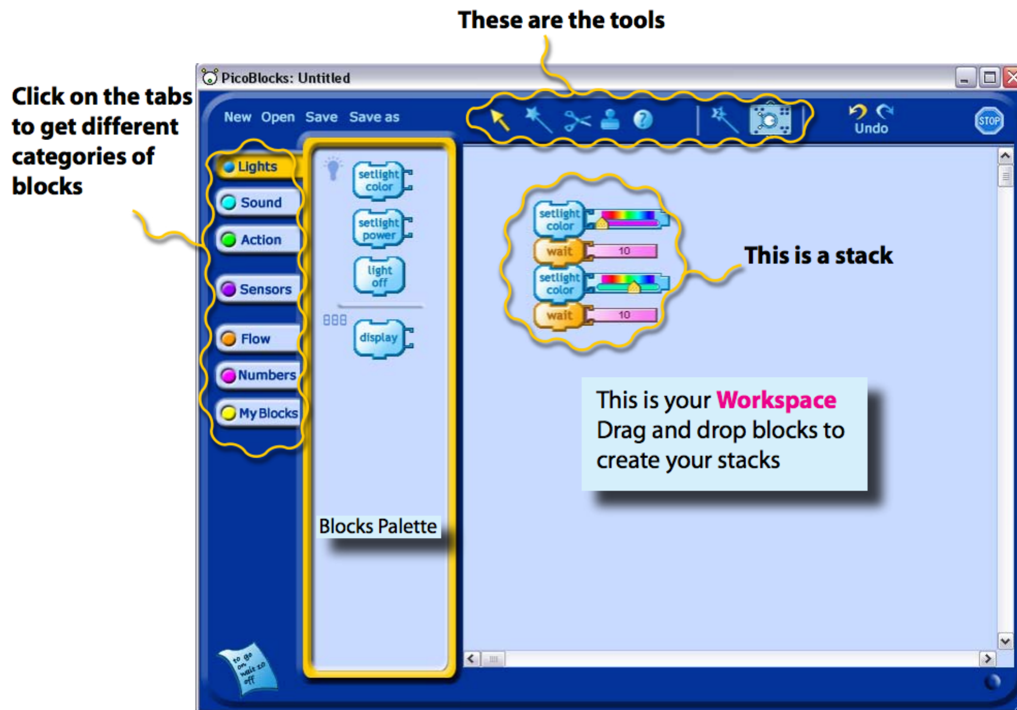


Figure 2.5: The PicoBlocks environment [Pic]

and text, rather, the two are in separate windows, but they are components of the same program. The text language allows users to specify functions and procedures which then appear as single blocks in the blocks window which users can then use with the rest of their blocks program. Thus, the text language is just a place for users to write declarations (e.g. new block definitions, with or without inputs).

Thus this project illustrates yet another way that blocks and text languages can be used in conjunction.

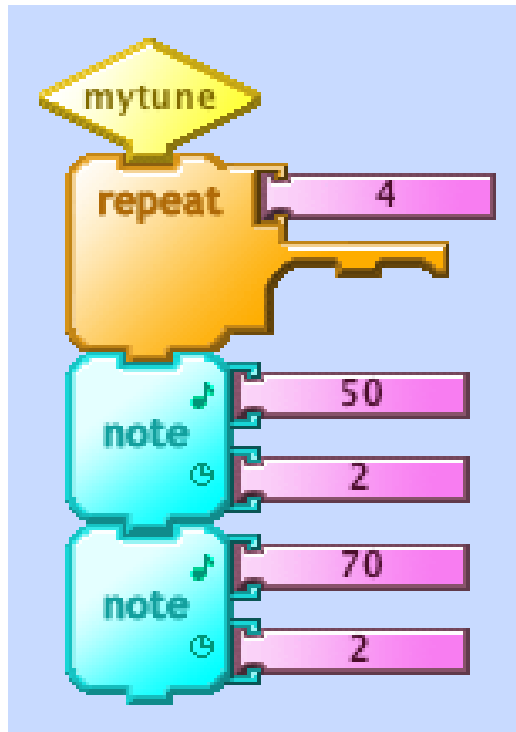


Figure 2.6: A PicoBlocks custom block declaration

Figure 2.7: This block appears in the special `MyBlocks` drawer after creating the block definition in Figure 2.6

2.4 SLASH

SLASH [Beh] is a student research project by Kara Behnke at the University of Colorado Boulder, the ATLAS Institute. A modification of Scratch, SLASH uses blocks-based programming to produce code in the Linden Script-

ing Language (LSL) [Beh], to program behavior in the Second Life virtual world. Behnke notes that "the goal of SLASH is to improve first-time programming experiences for non-[CS]-majors by juxtaposing block-based programming with [textual] programming language syntax" [Beh]. This research project is also motivated by the goal of attempting to ease the transition for novice programmers from blocks programming to textual programming. The work involves the generation of textual code from blocks code, but does not allow the conversion in the other direction. There is no connection between the semantics of the textual language and the semantics of the blocks programming language; instead of translating between two syntax tree representations of the same language, the blocks code in SLASH is used to generate LSL syntax.

This project has the similar goal of attempting to ease the transition from blocks programming to textual programming for novice programmers, by "modifying the Scratch [blocks programming] environment to enable students to learn [textual] language syntax while they program using blocks" [Beh]

This project attempts to fulfill its goals through the unique approach of allowing students to create programs using blocks —therefore making full use of all the advantages of blocks programming environments as mentioned in the previous chapter —however the students are required to "compare and analyze their blocks programs with the generated LSL scripts", thus pushing students towards translating their understanding of program structure and composition (which they've gained from blocks programming), into textual programming.

2.5 Cabana

Cabana, being developed by the Department of Behavior and Logic Inc., is another web-based application for the development of mobile applications across platforms. Like App Inventor, Cabana aims to "make app development easier" [Dic12]. Instead of a blocks programming language, Cabana uses an environment designed like a wiring diagram which links together nodes representing code modules. Cabana comes with pre-existing code modules representing basic program fragments (for loops, if statements, etc.). Additionally, Cabana allows users to specify their own code modules using a built-in feature to specify nodes in JavaScript code. Thus, Cabana allows for experienced programmers to write code more efficiently using JavaScript, whereas novice programmers can still use the built-in wire diagram format easily regardless of prior programming experience. The developers of Cabana argue that this combination of wiring diagram and JavaScript nodes "gives Cabana an advantage over App Inventor" by freeing users from the restriction of a "novice-centered programming environment like App Inventor's block language" [Dic12].

2.6 Previous Work on Blocks & Text Conversion in App Inventor

This section continues the discussion of related work, but outlines projects which are specific to App Inventor.

2.6.1 The Java Bridge

A subset of developers of MIT App Inventor have also created the App Inventor Java Bridge (I will refer to this as the Java Bridge). The Java Bridge is designed to make a transition between using App Inventor for developing Android applications to developing applications in Java using the Android SDK. The Java Bridge allows App Inventor users to incorporate App Inventor components into apps they create in Java with the standard Android SDK tools. This application is a good transition step for App Inventor users who are or have become familiar with Java and wish to eventually transition from App Inventor to the Android SDK for application development.

David Wolber of the App Inventor Developer Team has developed a tool to use alongside the Java Bridge [Bri] which translates existing App Inventor applications into Java applications which make calls to the Java Bridge.

Again, the Java Bridge, and David Wolber's tool, mentioned above, offer a different meaning of the notion of converting blocks to text. This tool offers a way to convert entire App Inventor projects into full Java projects. However, there is not a way to translate between just the blocks program in App Inventor to a Java program, or even more specifically there is no way to translate a subset of the blocks in the App Inventor blocks editor workspace to a piece of Java code representing just that subset of blocks. Additionally, users cannot specify code in Java to translate into the AI2 blocks.

2.6.2 Rendering Android App Inventor Code Blocks as Pseudo-Python Code

Philip Guo, a former student at MIT, proposed a project similar to the efforts of this project [Phia]. He proposed to provide App Inventor users with a tool for converting App Inventor blocks code to a pseudo-Python textual language (a new textual language, with syntax very similar to that of Python). The goals of this additional feature were to improve the readability of App Inventor Blocks code and to give novice programmers some "practice reading code in textual form [to] help them transition to more advanced programming courses" [Phia].

As Guo notes in his feature proposal, "the rendered pseudo-Python code would be read-only," thus this feature does not include the additional capabilities to specify a piece of code in this pseudo-Python syntax and have it render as blocks or be able to convert it to blocks. Thus this feature solves only one of the three limitations in App Inventor mentioned in Section 1.3.

Guo includes a prototype [Phib] of this feature in his proposal that converts a given set of AI1 blocks using their underlying YAIL code (see Section 4.1.1 for description of YAIL) into read-only pseudo-Python code.

2.6.3 Venthon: A First Take at a Creating a New Textual Language for App Inventor

Inspired by the proposal mentioned by Philip Guo's work described in the previous section, Erin Davis and I designed Venthon (App Inventor + Python), a new textual language for App Inventor with Python-esque syntax. Venthon

was designed to take Guo's work a step further and allow users to convert blocks and text in either direction (specifying code in AI blocks and converting to Venthon or specifying code in Venthon and converting to AI blocks).

Venthon Language Design

As mentioned above, Venthon has Python-esque syntax. This syntax, inspired by Philip Guo's pseudo-Python syntax, was chosen because it has fewer syntactic markers than other popular programming languages usually introduced to novice programmers (e.g. Java, C/C++). Additionally, Venthon strips down to the bare minimum syntactic features of Python to keep Venthon as simple as possible because it is designed to have the exact expressiveness of the App Inventor blocks language while increasing its ease of use. Thus, unlike the Java Bridge, Venthon does not add any functionality that is not already available in the App Inventor blocks language. Additionally, unlike the Java Bridge, or the Android SDK, users of App Inventor do not need to learn additional concepts (e.g. creating applications in Java, compiling programs, additional features offered in the Android SDK). Users need only learn how to convert each App Inventor block into Venthon in order to translate between the blocks and text languages. The advantage in this is that Venthon would be integrated into App Inventor, so users can continue using the App Inventor Designer (described in Section 1.2), but they would have the additional ability to choose whether to specify their code in the App Inventor blocks language or Venthon depending on their preferences.

Figure 2.8 and Figure 2.9 depict an example of what blocks from AI1 would look like in Venthon.

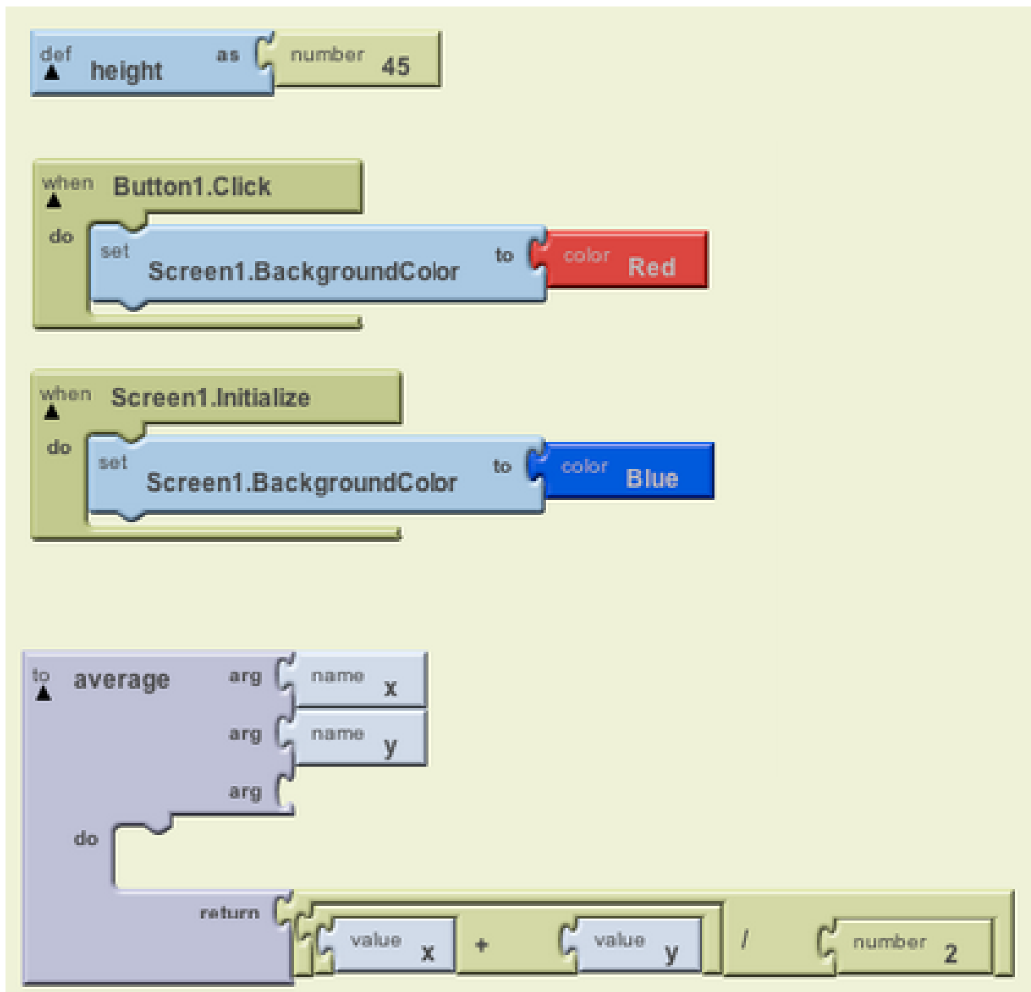


Figure 2.8: Sample AI1 Blocks to be Rendered in Venthon

As shown in Figure 2.8 and Figure 2.9, top level declarations in AI1 such as global variable declarations or event handlers look very much the same as top-level declarations (e.g. global variable declarations and procedure declarations) in Python. As in Python, a colon followed by indented lines of code denote a sequence of statements, where the indentation level of the particular line of code indicates which scope it is a part of.

```
height = 45

when Button1.Click():
  Screen1.BackgroundColor = color.red

when Screen1.Initialize():
  Screen1.BackgroundColor = color.blue

fun average(x, y)=
  (x+y)/2
```

Figure 2.9: Venthon Representation of Sample AI1 Blocks

As can be noted, some of the aspects of Venthon syntax stray quite a bit from traditional Python syntax. The function declaration at the bottom of the page looks closer to the syntax of a functional language such as ML. The keyword fun, representing a function declaration, was chosen to differentiate between the AI1 function declaration block and the AI1 procedure declaration block. This distinction arises from the fact that the App Inventor blocks language does not have a "return" statement block, but instead uses other methods to have a block return an expression as its output. This is a small example of what kinds of things need to be taken into consideration when designing a language that is isomorphic to the App Inventor blocks language. Chapter 3 discusses in detail the design of TAIL.

Venthon was designed as a first take at a textual language for App Inventor, with the idea (or rather future goal) in mind that there could be multiple syntaxes for a textual language for App Inventor, and ideally each of these syntaxes would be isomorphic to each other, thus allowing the App Inventor user to pick and choose which language he/she prefers to use.

Erin implemented the AI1 blocks to Venthon conversion, while I implemented the Venthon parser and the Venthon to AI1 blocks conversion.

The implementation of Venthon is very similar to that of TAIL (discussed in further detail in Section 4.2). I specified Venthon syntax in a grammar using ANTLR [Ant], a parser generator which takes a language grammar as input and produces a lexer and parser for the language defined by the input grammar. Options in the ANTLR grammar file allow the user to specify aspects of the generated parser and lexer. One of the most important options to specify in the grammar is the target language in which to generate the lexer and parser for the language being created.

The Venthon parser generated by ANTLR was configured to produce an intermediate JSON representation of the information from the parsed Venthon program. This intermediate representation, called JAIL, (JSON App Inventor Language) was to be used as a stepping stone between Venthon and the AI1 blocks language.

Though considerable progress was made on the implementation of Venthon, the transition from AI1 to AI2 required that the Venthon parser and lexer be converted from Java to JavaScript, just as the rest of the code in App Inventor had made the same transition. Additionally while the original implementation converts a Venthon program into the intermediate JAIL notation, the AI2 architecture better lends itself to converting the textual language program into the underlying XML representation of AI2 blocks (discussed in depth in Chapter 4). Finally, the design of the Venthon syntax and the Venthon implementation remain incomplete as the language needs to be redesigned to better adhere to AI2 and there are remaining design

decisions for some very specific details of the language.

Thus, while continuing the Venthon implementation, I made the decision to put Venthon development on hold and instead design a language with a more systematic syntax which would be easily predictable given any set of AI2 blocks. This language, TAIL, is the primary focus of this paper, and is discussed in detail over the next few chapters.

2.7 Code to Blocks

MIT students Chazz Sims and Jimmy Hernandez worked on a project, Code to Blocks (C2B) [SH], that is perhaps most similar (with the exception of Venthon) to the work I have done on TAIL, and is the most recent work previous work of this kind in App Inventor.

This project is motivated by many of the same issues that motivate the project discussed in the rest of this paper. The Code to Blocks work by Sims and Hernandez along with the work I have done on Venthon, and TAIL all address the limitations in App Inventor discussed in Section 1.3.

C2B translates between AI2 blocks and Python. Sims and Hernandez were successful in converting between blocks and text in both the blocks to text and text to blocks directions. However, as noted in the C2B write-up, "the scope of Python syntax without an Android Block representation is greater than initially expected" [SH]. Python and the AI2 blocks language are not isomorphic, meaning that there is not a one-to-one correspondence in both directions: between Python and Ai2 blocks as well as between AI2 blocks and Python. While it may be the case that every block in the AI2

blocks language can be expressed as a piece of Python code, and therefore any valid AI2 program can be translated into valid Python, it is not the case that any valid Python program can be translated into a valid AI2 program. In fact, the C2B write-up provides a list of Python constructs that have no equivalent representation in the AI2 blocks programming language. These constructs include, but are not limited to the following: lambda functions, dictionaries, list comprehensions, class definitions, etc..

The work on C2B reinforces the importance of language isomorphism when considering the conversion from the chosen textual representation to the AI2 blocks. The language isomorphism principle and its importance are discussed in more detail in Section 3.1.1.

Chapter 3

TAIL Language Design

3.1 Design Principles

The design of the TAIL language was guided by a few motivating principles, discussed in detail in the following sub-sections. Following these discussions is a description of the TAIL syntax, and how it corresponds to the AI2 blocks language.

3.1.1 Language Isomorphism

As mentioned in Section 1.4, I created TAIL in an attempt to combat the limitations in App Inventor (Section 1.3). With the creation of TAIL, I hope to make App Inventor more readable and searchable, ease user's ability to share App Inventor code across projects and to easily recreate segments of code in multiple projects, and to allow App Inventor users to write code in a more traditional way (i.e. through typing text instead of clicking and dragging visual components with a mouse). The textual language can adhere to

those users who are perhaps more experienced programmers but are looking to create Android apps through an easy-to-use interface, as well as to those users who are ready to transition from blocks programming languages to the more traditional textual programming languages.

In order to address the issue of making App Inventor programs more readable and searchable through a textual language, App Inventor users must have a means of converting their existing AI2 projects (or AI2 projects they create in the future) into the textual language. This one-way conversion is enough to solve the problem of reading and searching AI2 projects. Thus, for this specific problem, having read-only text (such as the work by Philip Guo discussed in Section 2.6.2) would suffice.

Another limitation of App Inventor, mentioned in Section 1.3, is the inefficiency and tediousness of creating complex applications in a blocks programming language. While App Inventor makes mobile application development simpler by removing the overhead of the Android SDK and by providing an environment where a lack of programming experience is not a hindrance, users (experienced programmers and novices alike) will quickly become accustomed to the concept of creating programs by connecting visual program fragments together. As the user becomes more accustomed to blocks programming, the user also begins to create more and more complicated applications, and consequently, these applications become more tedious to create. Blocks programming languages have the unfortunate downside that they become increasingly frustrating to use the more one understands how to use them, because users become quickly attuned to the concept of combining blocks to make programs, the user wants to be able to perform these actions

quickly. However, for the majority of computer users, they are more attuned to and efficient at typing than they are at clicking and dragging objects with a mouse or trackpad. The repetitive motion of click-and-drag is much slower than typing (for the efficient typist). For experienced programmers, programming with a mouse is extremely frustrating whereas typing code is the norm.

Thus, providing App Inventor users with a textual alternative to the blocks programming language provided would increase efficiency of many of the users who are already familiar with programming, as well as users who are ready to transition beyond blocks programming. Allowing users to convert from the textual programming language to the blocks programming languages allows the users to personalize their programming experience by combining the two forms of programming as suits their needs. Thus, for users who are beginning the transition from blocks programming to textual programming don't need to give up programming in the AI2 blocks language in order to attempt programming in TAIL. These users can attempt writing code in TAIL and convert it to blocks to test their understanding.

Finally, combining the conversion in both directions (blocks to text and text to blocks) allows for the solution to the third limitation in App Inventor: sharing code between projects.

Currently, App Inventor users would have to go through something similar to the following scenario in order to replicate code between multiple projects.

App Inventor's Current Sharing Scenario:

> User A has an app (user-A-app) and User B is interested in this app and how User A programmed it. User B talks to User

A and looks at the code in user-A-app and wants to recreate something similar in one of User B's own projects (user-B-app). User A agrees to let User B use part of the code from user-A-app. Therefore, User A downloads its project (in the form of a `.aia` file—denoting an App Inventor project file) and sends it to User B (through email or some other file transfer method). User B then uploads User A's project into User B's own App Inventor account. User B opens user-B-app in a separate browser window. User B views its copy of user-A-app side-by-side with user-B-app. *User B then meticulously reconstructs part of the code from user-A-app in user-B-app, by dragging, dropping, and clicking blocks together.*

Providing App Inventor users with the capabilities to convert from blocks to text and also from text to blocks allows for an easier method of sharing entire AI2 programs or parts of programs across different projects. The new sharing scenario would be much simpler:

> User A converts user-A-app to TAIL, copies and pastes this TAIL code in an email to User B. User B then copy-pastes this TAIL code (or part of this TAIL code) into user-B-app and converts to blocks.

Additional Details of the Conversion

Above, I talk about "converting" from blocks to text and "converting" from text to blocks. On a larger scale the term conversion refers to the translation

of an entire AI2 project (including information from the Blocks Editor as well as the Designer, both mentioned in Section 1.2) into TAIL text. As noted in Chapter 5, I have not currently implemented this, but Section 5.3.3 discusses in detail what this large scale conversion will entail. On a smaller scale, the term conversion also applies to translating parts of programs such as individual blocks or sets of blocks. As mentioned in Section 1.4, the App Inventor user is allowed to choose which block or set of blocks to convert into TAIL (or vice-versa), regardless of where this block or set of blocks lies in the syntax tree.

It is also important to note that though I talk specifically about how conversions allow users to combat the current limitations in App Inventor, conversions are not necessary for the user's mobile app to function correctly. Specifically I want to point out that TAIL code need not be converted to AI2 blocks before running the app as the piece of TAIL code carries the exact semantics of the AI2 code it represents.

All of what is mentioned above is possible largely due to the fact that TAIL and AI2 blocks are isomorphic languages. Formally this means that there is a bijection between the languages. Informally, this directly implies that round trip conversions between blocks and text (i.e. from TAIL to AI2 blocks to TAIL, or from AI2 blocks to TAIL to AI2 blocks) will yield the original code (whether it is TAIL code or AI2 blocks code) that began the conversion cycle. Language isomorphism is the major design principle that guides all of the decisions regarding details of the TAIL syntax.

Thus, there are a few important benefits of language isomorphism.

Users cannot express functionality in TAIL that they cannot express in

AI2 blocks, and they cannot express functionality in AI2 that they cannot express in TAIL. This allows users to program AI2 code solely in TAIL. Thus, users can write a TAIL program to specify an Android app. Users do not need to convert their TAIL programs back into AI2 blocks in order to create a working app. Thus the issue found in the Code to Blocks project (Section 2.7) of users being able to express things in the Python language that hold no meaning with regards to programming an App Inventor app, is eliminated with an isomorphic text language.

The round-trip conversion allows users (especially novice programmers) to help transition from the blocks programming language to textual programming languages because users can write TAIL text and revert back to blocks to see the notation they are more familiar and comfortable with. Users can also convert a set of blocks to text to check what the TAIL for a given set of blocks would look like.

Finally, language isomorphism allows for creating programs that combine both the text and blocks syntaxes. Instead of not being able to translate a program into text until the program is complete, with a language that is isomorphic to the AI2 blocks language, users can translate individual or sets of fragments of their programs into TAIL. Each AI2 block can be translated into TAIL individually or sets of blocks can be translated into TAIL as a whole. Thus, users can choose which parts of their programs they want to express in TAIL and which parts they want to express in AI2 blocks. As an example, users can choose to convert certain sub-blocks to TAIL for notational brevity (e.g. the quadratic formula). In other situations, users can choose to keep the blocks notation where they prefer it (e.g. to have a

visual reminder of the number of arguments that a block takes).

3.2 Easy Transition to TAIL

In addition to adhering to the language isomorphism principle, all TAIL syntax decisions were guided by the aim to make the transition from the AI2 blocks language to the TAIL text language as easy as possible. Thus, the syntax of TAIL is designed to allow users to easily predict the equivalent TAIL text for a given set of AI2 blocks. The details of the syntax are presented in the following section.

3.2.1 TAIL Syntax

The TAIL syntax is relatively systematic (given the visual information on the equivalent AI2 blocks), and is guided by some simple rules:

1. ***TAIL expressions are represented using curly braces.***
2. ***TAIL statements are represented using square brackets.***
3. ***TAIL top level declarations are denoted by parens.***
4. ***Given an AI2 block, the corresponding TAIL code will have the same title as the AI2 block where any spaces in the title of the block are replaced with underscores.***
5. ***Labels on AI2 blocks are followed by a colon in TAIL.***

These rules are described in more detail below.

Please note that some of these TAIL examples use the symbol `{$}`. This symbol is not valid TAIL. Instead, this symbol is a placeholder, for the purpose of these examples, for a valid TAIL expression which should replace the `{$}`. Without a valid TAIL expression in place of `{$}`, the entire TAIL fragment is invalid (see discussion in Section 5.3.3 about representing incomplete AI2 blocks in TAIL). Additionally, note that I have not created a place holder for empty statement sequences. This is because the empty statement sequence is a valid statement sequence in AI2 and consequently TAIL.

Expressions

Expression blocks (see sample expression blocks in Figure 3.1) are blocks with `plugs` (shown in Figure 3.2a), which indicate that the block outputs a value. Any AI2 block can have any number of `sockets` (shown in Figure 3.2b and Figure 3.2c). A `socket` on a block indicates that this block expects a value as input. The shapes of the `plugs` and `sockets` indicate that the two blocks should fit together. Note that there are some sockets that expect a certain type of input, but this type is not represented with a different `plug` shape, so blocks with `plugs` of the incorrect type will bounce away from the `socket` when trying to connect the two, to indicate that these two blocks cannot be connected together.

TAIL expressions are represented using curly braces. (Figure 3.3)

Note that the left curly brace `{` is visually similar to the plug on AI2 expression blocks.

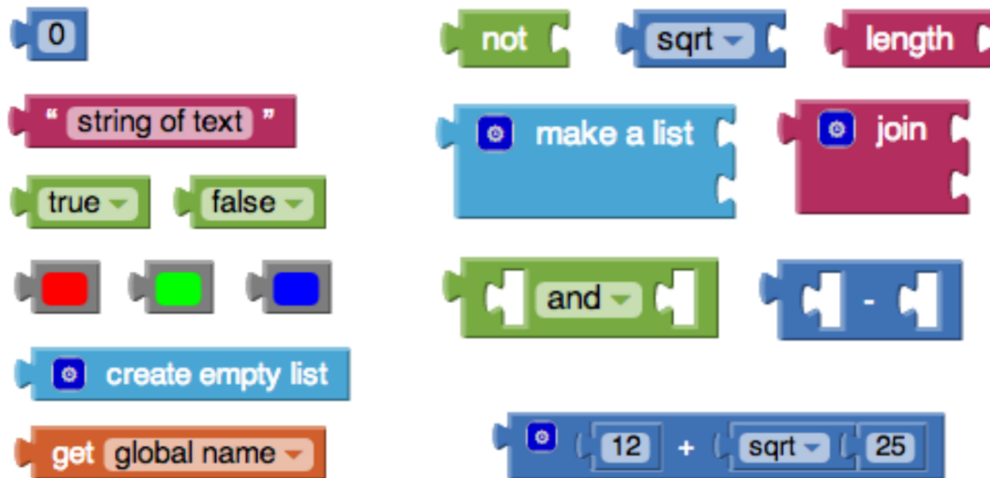


Figure 3.1: AI2 Expression Blocks

(a) A Plug (b) A Socket (c) An Inline Socket

Figure 3.2: Plugs and Sockets in AI2 Blocks

Statements

Statement blocks (see sample statement blocks in Figure 3.4) are blocks that compose vertically with each other to form a sequence of statements. The vertical composition of the blocks is indicated by `nubs` and `notches` (Figure 3.5) on the top and bottom of the blocks. `Nubs` and `notches` are referred to as `previous connector` and `next connector` in AI2 jargon.

TAIL statements are represented using square brackets. (Figure 3.6)

```
{0}          {not {$}}   {sqrt {$}}  {length {$}}

{"string of text"}  {list {$} {$}}  {join {$} {$}}

{true} {false}      {{$} and {$}} {{$} - {$}}

{color red} {color green} {color blue}

{list}

{get global name}       {{12} + {sqrt {25}}}
```

Figure 3.3: TAIL Expressions Corresponding to AI2 Expression Blocks in Figure 3.1

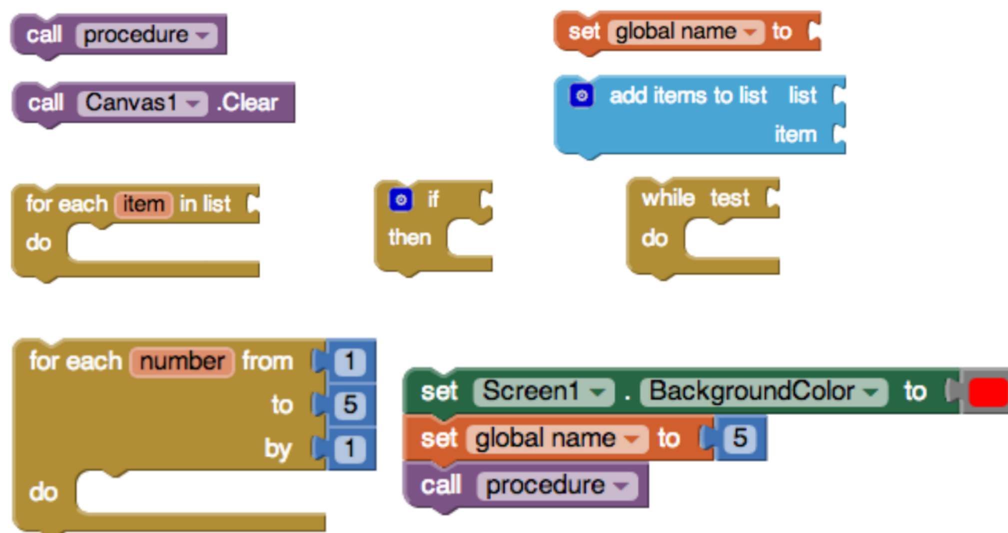
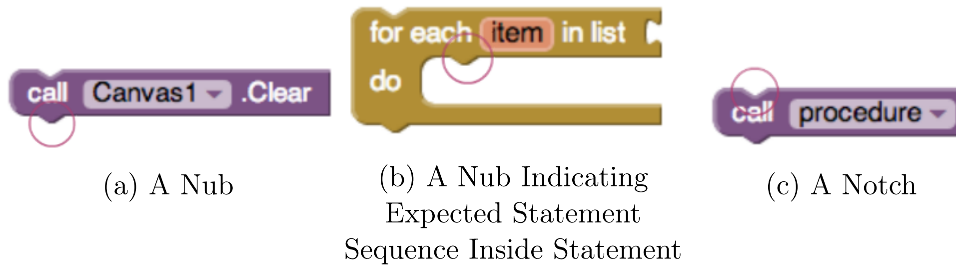


Figure 3.4: AI2 Statement Blocks

Declarations

Top level declarations blocks (Figure 3.7) such as global variable declarations, event handlers, and procedure or function declarations, do not have any



(a) A Nub

(b) A Nub Indicating Expected Statement Sequence Inside Statement

(c) A Notch

Figure 3.5: Nubs and Notches in AI2 Blocks

```
[call procedure]          [set global name to: {$}]

[call Canvas1.Clear]      [add_items_to_list
                            list: {$} item: {$}]

           [if {$} then:]  [while test: {$} do:]

[for_each <item> in list: {$} do:]

      [set Screen1.BackgroundColor to: {color red}]
      [set global name to: {5}]
      [call procedure]

[for_each <number> from: {1} to: {5} by: {1} do:]
```

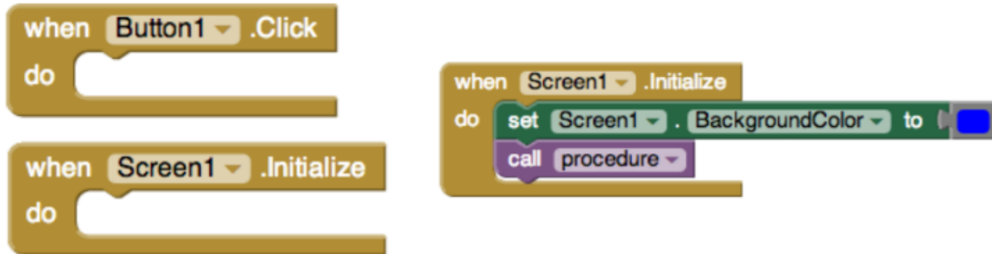
Figure 3.6: TAIL Statements Corresponding to AI2 Statement Blocks in Figure 3.4

external connectors which allow them to be composed with other blocks. Thus, these blocks cannot be placed inside of any other block, precisely because these are top-level blocks (they belong at the top of the syntax tree).

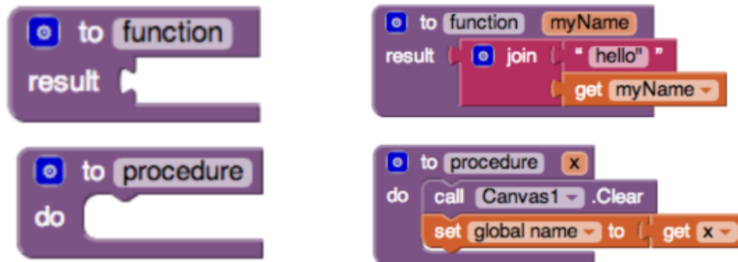
TAIL top level declarations are denoted by parens. (Figure 3.8)

Finally, these last two rules are a bit more general.

Event Handlers



Procedures & Functions



Global Variables

Figure 3.7: AI2 Top Level Declaration Blocks

Given an AI2 block, the corresponding TAIL code will have the same title as the AI2 block where any spaces in the title of the block are replaced with underscores.

An example of a title on a block is `initialize global` on the global variable declaration block, or `for each` on the for loop block. These titles

```
(when Button1.Click do: )

              (when Screen1.Initialize
                do: [set Screen1.BackgroundColor to:
                          {color blue}]
                    [call procedure])

(when Screen1.Initialize do: )



(to <function> result: {$})
                  (to <function> <myName>
                    result: {join
                              {"hello"}
                              {get myName}})
(to <procedure> do: )
                  (to <procedure> <x>
                    do: [call Canvas1.Clear]
                        [set global name
                          to: {get x}])



(initialize_global <name> to: {$})
                  (initialize_global <name> to: {3})
```

Figure 3.8: TAIL Top Level Declarations (TLDs) Corresponding to AI2 TLD Blocks in Figure 3.7

are converted to `initialize_global` and `for_each` respectively in the TAIL counterparts of these two blocks.

Titles of AI2 blocks are not to be confused with labels which appear before `sockets`, `plugs`, or `nubs` (discussed above).

Labels on AI2 blocks are followed by a colon in TAIL.

I will conclude this section by noting that part of the reason that development on Venthon (see Section 2.6.3) was put on hold was that there were more design details to consider in Venthon because the syntax was not as systematic (cannot be described as concisely in a few rules) as the TAIL syntax, which systematically follows from visual information on the AI2 blocks. Thus, even though Venthon uses fewer syntactic markers, the syntactic structure of Venthon programs is more different from the syntactic structure of AI2 blocks.

Chapter 4

Integrating into App Inventor & Creating the TAIL Language

In Chapter 3 I discuss TAIL language design (i.e. why TAIL looks the way it looks). In this chapter I outline how I integrated TAIL into App Inventor. This chapter discusses a wide variety of topics from general architecture of my project, to design details, to my specific implementation.

4.1 Architecture

4.1.1 Description of AI2 Architecture

The AI2 blocks language and TAIL are just two different representations of the same abstract syntax tree for representing AI2 programs. In addition to these two representations, there are two more underlying representations that must be mentioned. AI2 blocks have an underlying XML tree

representation where each AI2 block is an XML block element denoted by `<block>...</block>`. Underlying the AI2 blocks is another representation, YAIL (Young Android Intermediate Language) which carries the semantics of the blocks and is the stepping stone towards generating executable code (representing the AI2 program) to run on the Android device. These four representations (an example depicted in Figures 4.1 to 4.4, AI2 blocks, XML, YAIL, and TAIL are all varying representations of the same syntax tree.

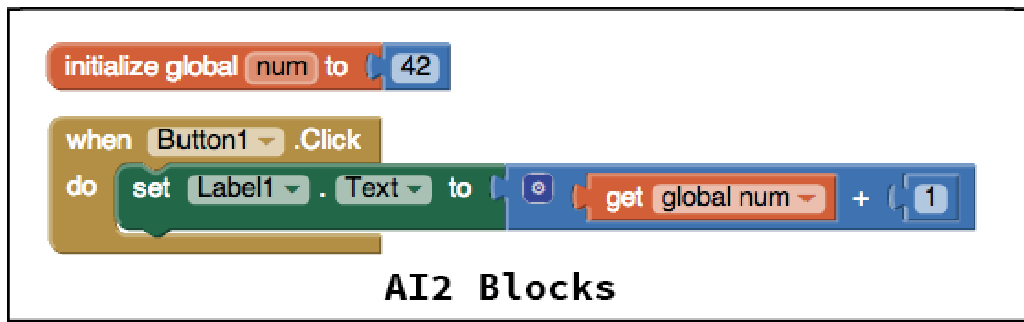


Figure 4.1: AI2 Blocks representation of syntax tree

4.1.2 Extending AI2 with TAIL Code Blocks

As noted in Section 1.4, I not only create a new language, TAIL, but I also provide a means for AI2 users to convert from AI2 blocks to TAIL code and from TAIL code to AI2 blocks. I have accomplished this by adding a new set of blocks (*code blocks*) to AI2 (Figure 1.6).

There are three code blocks, one for each of the three different types program fragments in AI2 (discussed in Section 3.2.1): expression blocks, statement blocks, and top level declaration blocks.

These new code blocks allow AI2 users to specify TAIL code within the

```
<block type="global_declaration" inline="false" x="80" y="3">
    <title name="NAME">num</title>
    <value name="VALUE">
      <block type="math_number">
        <title name="NUM">42</title>
      </block>
    </value>
</block>
<block type="component_event" x="81" y="43">
  <mutation component_type="Button" instance_name="Button1"
     event_name="Click"></mutation>
  <title name="COMPONENT_SELECTOR">Button1</title>
  <statement name="DO">
    <block type="component_set_get" inline="false">
      <mutation component_type="Label" set_or_get="set"
         property_name="Text" is_generic="false"
         instance_name="Label1"></mutation>
      <title name="COMPONENT_SELECTOR">Label1</title>
      <title name="PROP">Text</title>
      <value name="VALUE">
        <block type="math_add" inline="true">
          <mutation items="2"></mutation>
          <value name="NUM0">
            <block type="lexical_variable_get">
              <title name="VAR">global num</title>
            </block>
          </value>
          <value name="NUM1">
            <block type="math_number">
              <title name="NUM">1</title>
            </block>
          </value>
        </block>
      </value>
    </block>
  </statement>
</block>
```

XML

Figure 4.2: XML representation of syntax tree

editable labels on the blocks.

The resulting edited code block is valid if it contains valid TAIL code for

```
(def g$num 42)

(define-event Button1 Click()(set-this-form)
  (set-and-coerce-property! 'Label1 'Text
    (call-yail-primitive +
      (*list-for-runtime*
        (get-var g$num) 1 )
      '(number number ) "+")
   'text))
```

YAIL

Figure 4.3: YAIL representation of syntax tree (determines the code that is executed on the Android device)

```
(initialize_global <num> to: {42})

(when Button1.Click do:
    [set Label1.Text to:
        {{get global num} + 1}])
```

TAIL

Figure 4.4: TAIL representation of syntax tree

the type of block it is. This means that the TAIL expression code block must contain a valid TAIL expression, the TAIL statement code block must contain a valid TAIL statement, but if the TAIL expression code block contains a

valid TAIL *statement*, the TAIL expression code block is still invalid). If the edited code block is valid, then this code block carries the semantics of the AI2 block(s) it represents and can be used just as the corresponding AI2 block(s) would be used when running the mobile app on an Android device. Thus users need not convert back to blocks in order to run the app. Note that since the valid TAIL code block carries the semantics of the AI2 blocks it represents, the TAIL code blocks are shaped like the types of AI2 program fragments they represent. The TAIL expression code block is itself an expression block because it has a `plug`. The TAIL statement code block is itself a statement block because it has nubs and notches. The TAIL top level declaration code block is itself a top level declaration block because it does not have any external connectors such as `plugs`, `sockets`, nubs, or notches. See Figure 4.5 for a valid TAIL expression code block.

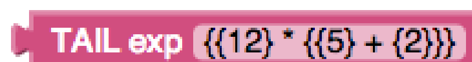


Figure 4.5: A valid TAIL expression code block

If the resulting edited code block contains invalid TAIL code, meaning that the user either specified incorrect TAIL or the wrong type of TAIL fragment for the specific code block (i.e. the user entered a valid TAIL *expression* inside a TAIL *statement* code block), the code block will display an error icon (with a corresponding error message), and the code block will carry the semantics of an AI2 error. See Figure 4.6 for an invalid TAIL code block.

In addition to the ability to specify TAIL code inside these code blocks

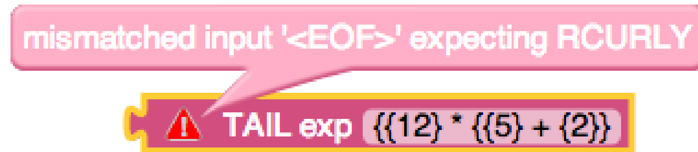


Figure 4.6: An invalid TAIL expression code block

and use a TAIL code block as is, users can convert any *valid* TAIL code block to the AI2 block or set of blocks it represents.

4.1.3 Blocks to Text Converter

Section 3.1.1 talks in depth about a two way conversion between the AI2 blocks and TAIL text. The blocks to text conversion proves to be much simpler than the text to blocks conversion. In order to convert AI2 blocks to the TAIL code blocks with the appropriate TAIL text, I created a blocks to text converter. This converter is specified in a JavaScript file inside the AI2 code base.

As mentioned in Section 4.1.1, AI2 blocks have an underlying XML representation which is based on their high-level descriptors in the JavaScript based blocks editor portion of AI2 code. AI2 provides useful functions for translating a given block or set of blocks to its XML DOM and translating a given XML DOM for a block or set of blocks to the block/set of blocks the XML DOM represents. As briefly noted in Section 2.6.3, I use this XML representation of the blocks as the stepping stone for the blocks and text conversions. Thus, instead of having to convert the SVG graphics of the block to TAIL code, I can just use AI2's built-in XML DOM conversion function

to translate a given set of blocks into its underlying XML, and then translate this XML to TAIL code.

The blocks to text converter first translates a given set of blocks to its XML DOM. Recall that an XML DOM is actually just a tree, as is any given program (all programs can be viewed in terms of their abstract syntax tree). The blocks to text converter walks down the XML Tree starting at the root element. In AI2's underlying XML representation, the root element will always be a block element denoted with a block XML tag (`<block>...</block>`). As the converter walks down the XML tree, it accumulates a string, converting each part of the XML tree to a substring representing the part of the TAIL abstract syntax tree that is equivalent to this part of the XML tree. By the time the converter has walked down to the end of the tree, the entire XML tree has been converted into the corresponding TAIL code.

After accumulating the entire string with the TAIL code, the blocks to text converter creates an XML element representing a TAIL code block (discussed in Section 4.1.2) with this TAIL code inside its edited label. The resulting XML element can then be converted into the visual TAIL code block (containing the new, valid TAIL code) using the built-in AI2 XML to blocks conversion function.

An App Inventor user can convert a set of AI2 blocks using the context menu on any given block in the AI2 Blocks Editor workspace. The context menu for any AI2 block will have a `Convert to TAIL` option which when clicked will replace the given block any blocks nested inside of it into a corresponding TAIL code block. It is important to note that the `Convert to TAIL` context menu option will be un-clickable if the AI2 block has any empty

`sockets` as empty `sockets` are errors in AI2 programs (not in the workspace itself, but rather in a valid program). In my current implementation, any errors must be corrected before being able to convert AI2 blocks to TAIL code blocks. However, it is important to allow users to convert between the notations as they are programming. This means that I must implement incomplete blocks (i.e. any blocks with empty sockets) in TAIL. Adding incomplete blocks to TAIL will allow users to convert between blocks and text as they are in the midst of programming. For example, if a user starts off programming in the AI2 blocks language, and in the midst of plugging blocks together, the user realizes he/she needs to create a long arithmetic expression that he/she would prefer to write in text, the user should be able to convert the entire construction to TAIL and type the rest in TAIL (note that the user can also plug in a TAIL expression code block and type the TAIL for the arithmetic expression inside this code block).

Allowing incomplete blocks to be represented in TAIL is also necessary for converting entire workspaces and projects into TAIL. This is discussed in Section 5.3.3.

Interesting Design Detail

Any given AI2 block has the option to convert to TAIL. This option converts the given AI2 block into a corresponding TAIL code block. The TAIL code blocks I have added to the AI2 blocks language are themselves AI2 blocks; thus, TAIL code blocks can also be converted into TAIL. App Inventor users can have valid, nested TAIL code blocks (Figure 4.7) which are semantically equivalent to the AI2 blocks represented by the TAIL code block that is most

deeply nested. The blocks in Figure 4.7, Figure 4.8, and Figure 4.9 are all semantically equivalent.

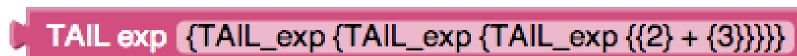


Figure 4.7: A deeply nested TAIL code block

Figure 4.8: TAIL code block at the deepest level of nesting from Figure 4.7

Figure 4.9: AI2 block represented by TAIL code block in Figure 4.8

4.1.4 Creating TAIL

Conversion from text to blocks proves to be much more complicated than the conversion in the opposite direction. The following section offers a high-level description of the three major steps involved in creating TAIL and converting pieces of TAIL code into blocks.

Step 1: Lexing

When a computer reads a programming language, it first breaks the text of the program into lexical tokens. The mechanism responsible for performing

this function is called a *lexer*. A lexer takes as its input a string of text of the entire program. The lexer then breaks up the text of the program into tokens, the smallest, meaningful units of the programming language. These lexical tokens often include (but are not limited to) identifiers, strings, whitespace, syntactic markers (such as parens, semi colons, brackets, etc.), numbers, and keywords of the language. The lexer will use rules specified in the lexical grammar (a specification of the lexical syntax) to break up the tokens. The TAIL lexer is described in detail in Section 4.2.2.

Step 2: Parsing

The next step the computer takes in reading a programming language is to take the tokens generated by the lexer arrange the tokens in a tree structure representing the syntactic structure of the program. The mechanism responsible for this function is called a *parser*. The parser works together with the lexer. A parser takes as its input the tokens that the lexer emits, and outputs an abstract syntax tree representing the syntactic structure of the program. Like the lexer, parsers also use rules from the grammar (specification of syntactic structure) of the language being parsed to determine how tokens can compose together to form a valid program.

Above I mention that one possible lexer token is whitespace. In most programming languages, whitespace tokens are thrown out when the tokens are being fed to the parser, because in many programming languages, whitespace is not an important token in determining the syntactic structure of a program. In languages like Python, however, whitespace is very important because it is the token that is used to indicate block structures and scope

within a Python program. This was a non-trivial component of the parser for Venthon (which, like Python, also uses whitespace to denote block structure/scope of a Venthon program), however, whitespace tokens can be thrown out in the context of parsing TAIL.

The TAIL parser is described in detail in Section 4.2.3.

Step 3: Tree Conversion

The final step to take in the context of converting the fully parsed, valid TAIL program into AI2 blocks is to convert the abstract syntax tree generated by the parser into the blocks. Section 4.1.3 talks about the underlying XML representation of the blocks. AI2 has built-in functions to convert an XML tree (with the valid syntax for the XML representation) to the AI2 blocks it represents.

I translate the abstract syntax tree generated by the TAIL Parser (discussed in Section 4.2.3) to the underlying XML representation of the AI2 blocks the parsed TAIL code represents.

To put the tree conversion in the context of the TAIL code blocks (described in Section 4.1.2), each TAIL code block is running the TAIL lexer and parser and parsing (which includes lexing) the text inside the editable text box on the TAIL code block. If the text inside the editable text box is invalid TAIL code, a red error icon appears on the TAIL code block. When clicked on, the red error icon reveals the first parse error for the invalid TAIL text. When all errors have been corrected and the TAIL code block has correct TAIL code (in the appropriate TAIL code block), the user can click on the context menu of the block which has two new options: the `Convert to`

`TAIL` option, mentioned in Section 4.1.3, and a `Convert to Blocks` option. The `Convert to Blocks` option performs the tree conversion.

The tree conversion is described in more detail in Section 4.2.4.

4.2 Implementation of TAIL & Text to Blocks Conversion

In order to implement TAIL and the text to blocks conversion, I used ANTLR, Another Tool for Language Recognition [Ant], a parser generator (described in Section 4.2.1). I then integrate the output of ANLTR into the AI2 code base (more specifically the TAIL code blocks I added), to allow App Inventor users to specify AI2 code using TAIL text and optionally convert the TAIL text into AI2 blocks.

4.2.1 What is a Parser Generator?

ANTLR, Another Tool for Language Recognition, developed by Terrence Parr, is a parser generator. Parser generators, like ANTLR, take, as input, a grammar specification for the language the user is trying to create. The parser generator then produces a lexer and parser for the language specified based on its grammar. The input grammar, written in ANTLR's own grammar syntax, consists of lexer and parser rules which indicate what a valid program for the language being specified should look like. The lexer and grammar rules are described in detail in Section 4.2.2 and Section 4.2.3 respectively.

ANTLR grammars have a number of different options that users can set to obtain different results. One of the most important options the user can specify is the target language in which ANTLR should generate the parser and lexer. The target language for the TAIL grammar is JavaScript just as the rest of the AI2 blocks editor code is in JavaScript.

As an important note, the most recent versions of ANTLR (ANTLR v.3.4-v.3.5 and ANTLR 4) do not have a working JavaScript target, so I used ANTLR v.3.3 in this project, the most recent version of ANTLR that has a working JavaScript target.

4.2.2 Lexing with ANTLR

I use ANTLR to generate the TAIL lexer by specifying lexer rules for my TAIL grammar. There are three different kinds of lexer rules. There are simple lexer rules (Figure 4.10) that specify tokens that are string literals (examples of such tokens are syntactic markers or keywords in the language).

```
tokens {
        LCURLY = '{';
        RCURLY = '}';

        ADD = '+';

        MULTIPLY = '*';
}
```

Figure 4.10: Sample lexer rules matching string literals

There are complex lexer rules (Figure 4.11) that use a regular expression syntax to match text. Each instance of these two kinds of lexer rules is a specification of a different lexical token.

```
NUMBER : (DIGIT* DOT DIGIT+ | DIGIT+ (DOT)?);
```

Figure 4.11: Lexer rule matching numbers in TAIL

Finally, there is a third kind of lexer rule called a *fragment*, which is a helper rule which can only be used inside other lexer rules and does not specify its own token. Figure 4.14 is an example of a fragment.

```
fragment
DIGIT : ('0' .. '9');
```

Figure 4.12: A lexer fragment rule

4.2.3 Parsing with ANTLR

ANTLR grammar files consist of both lexer and parser rules. Parser rules are similar to lexer rules. Instead of specifying tokens, each parser rule specifies part of the syntactic structure of a TAIL program. The portion of the grammar consisting solely of parser rules is a context-free grammar. The ANTLR generated TAIL parser will follow the grammar rules, starting at a top level rule (whatever is calling the parser can specify which top level parser rule to begin with), and following sub-rules, until each the sub-rule reaches all terminal rules. A terminal rule is a parser rule which matches

only lexer rules, thus ending the rule tree. Figures 4.13 to 4.16 depicts some sample parser rules.

```
expression_block
        : LCURLY expression RCURLY
        ;
```

Figure 4.13: The top level parser rule for TAIL expressions

```
expression
        : math_expr
        | atom
        ;
```

Figure 4.14: Following sub rules from Figure 4.13

```
math_expr
        :  mutable_arith_expr
        | non_mutable_arith_expr
        | special_math_expr
        | unary_math_expr
        | math_trig_expr
        ;

mutable_arith_expr

        : a=expression_block
        (ADD  | MULTIPLY )
        b=expression_block
        ;
```

Figure 4.15: Following sub rules from Figure **??**

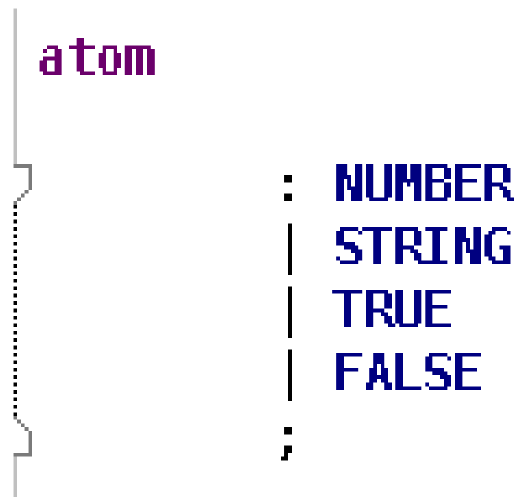


Figure 4.16: The terminal parser rule

4.2.4 Tree Conversion with ANTLR

Once the input TAIL text has been parsed (and if the parser has determined that it is valid TAIL code), the output abstract syntax tree is converted into the underlying XML representation of the AI2 blocks corresponding to the TAIL text. I accomplish this tree conversion simultaneously as the parser generates the abstract syntax tree using ANTLR *actions*. ANLTR grammars have the additional feature of allowing the user to specify bits of executable code (in the target language for the grammar, JavaScript in this case) throughout the different grammar rules. These bits of code, called actions, are executed as the parser matches the tokens in the token stream provided by the lexer to the parser rules in the grammar. The ANTLR actions can be interspersed through out a rule to be executed at different points of the rule matching.

Figure 4.17 depicts the ANTLR parser rule responsible for matching the

`atom` rule from Figure 4.16 with actions included.

```
atom returns [var elt]
@init{
  $elt = document.createElement("block");

  var title = document.createElement("title");
}
  : NUMBER {
    $elt.setAttribute("type","math_number");

    title.setAttribute("name","NUM");
    title.innerHTML = $NUMBER.text;
    $elt.appendChild(title);
  }
  | STRING {
    $elt.setAttribute("type","text");

    title.setAttribute("name","TEXT");
    var text = $STRING.text;
    title.innerHTML = text.substring(1,text.length-1);
    $elt.appendChild(title);
  }
  | TRUE {
    $elt.setAttribute("type","logic_boolean");

    title.setAttribute("name","BOOL");
    title.innerHTML = "TRUE";
    $elt.appendChild(title);
  }
  | FALSE {
    $elt.setAttribute("type","logic_boolean");

    title.setAttribute("name","BOOL");
    title.innerHTML = "FALSE";
    $elt.appendChild(title);
  }
  ;
```

Figure 4.17: The terminal parser rule `atom` with ANTLR actions embedded

Chapter 5

Conclusion and Future Work

5.1 Current State

In its current state, the TAIL language has support for several (but not all) of the different program fragments in AI2. All three types of program fragments (expressions, statements, and top level declarations) have been implemented in AI2. Round trip conversions work for all of the blocks that can currently be specified in TAIL (i.e. there are rules for these blocks in the TAIL grammar). There are still outstanding blocks that need to be added to the TAIL grammar and the blocks to text converter. Additionally, YAIL generators (which specify the semantics of the blocks) currently only exist for the TAIL expression code block and not TAIL statement or declaration blocks.

5.2 Immediate Future

There is quite a bit of work that remains to be done before TAIL can be integrated into a future official release of AI2. Part of this work is discussed in this section; this is work that can be completed in the short term future (a few weeks). The rest of this work is discussed in the following section; this is work that can be completed in the slightly longer term (1-3 months)).

5.2.1 Adding all AI2 Blocks to the TAIL Grammar

Although many of the AI2 blocks can be expressed in TAIL, not all AI2 blocks have been added to the TAIL language. Currently most individual AI2 blocks are accounted for in their own parser grammar rules. There are a few exceptions (e.g. arithmetic expressions, event handlers, component methods) in which a single parser grammar rule accounts for many AI2 blocks. For example all AI2 event handlers can be expressed in TAIL using only a single parser grammar rule. All event handler blocks have a syntactic structure which can be abstracted into one high level rule. Other than the few exceptions in which one rule can express multiple AI2 blocks, most AI2 blocks must be accounted for in their own specific parser grammar rules. Examples of such blocks are depicted in Figure 5.1.

It is necessary to add all of the AI2 blocks to the TAIL grammar. This will currently require adding an individual rule for the all of the blocks that TAIL does not currently support.

The process of adding an AI2 block to the TAIL grammar is straightforward because the TAIL grammar is designed to make the translation from

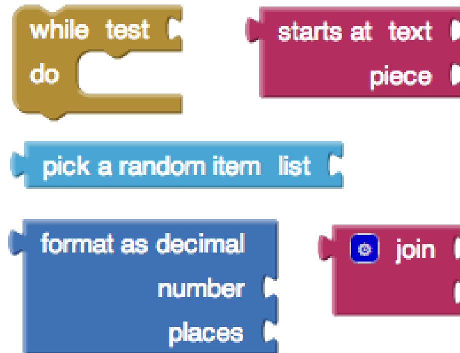


Figure 5.1: Examples of blocks that require individual grammar rules

AI2 blocks to TAIL syntax as easy as possible using the simple language design rules outlined in Section 3.2.1. However, though this process is straightforward, it is very tedious because the developer must not only create a new TAIL parser rule pertaining to the block, but also add the ANTLR actions to translate the TAIL text into the AI2 blocks' underlying XML representation (as described in Section 4.2.3 and Section 4.2.4). Section 5.3.5 discusses the possibility of creating some sort of abstraction to simplify the process of adding an AI2 block to the TAIL grammar.

In addition to adding the rest of the AI2 blocks to the TAIL grammar, I want to allow for the conversion of on-block comments. AI2 allows users to add comments to individual blocks (via the context menu for the block). These comments appear as an icon next to the block's title with a `?` on it (Figure 5.2).

When this comment icon is clicked, there is a speech bubble (similar to that for block errors), with an editable text area in which the user can type some comments about the block. These comments can easily be incorporated

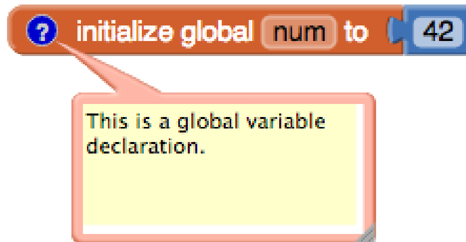


Figure 5.2: Block with comment

into the TAIL language so that they are not lost during blocks to text conversions. Figure 5.3 gives an example of what the block in Figure 5.2 might look like in TAIL, with the comments preserved.

```
(?| This is a global variable declaration. |?
   initialize_global <num> to: {42})
```

Figure 5.3: Example of how comments may be represented in TAIL

5.2.2 Creating YAIL Generators for Statements and Declarations

TAIL code blocks carry the semantics of the AI2 blocks they represent. However, the semantics of TAIL code blocks has so far only been implemented for the TAIL expression code block. Support for the TAIL statement and declaration code blocks has yet to be added. The implementation of the semantics (i.e. the YAIL generators) for these remaining code blocks is straightforward and is similar to the implementation of the semantics for the TAIL expression code block.

5.3 Near Future

5.3.1 User Studies

As soon as a large enough subset of the most popular AI2 blocks have been added to TAIL, it is very important that I do at least some preliminary user testing for different user groups (novice programmers unfamiliar with blocks programming languages, novice programers with block programming experience, and more experienced programmers who use AI2).

Preliminary user testing could prove to be very useful in testing details of the TAIL language before it becomes available to the App Inventor community. Additionally, it is important and will prove to be very useful to test whether AI2 users are inclined to start using TAIL to program in AI2. If there is an inclination to use TAIL, at what stage they are most likely to find the TAIL code capabilities useful (if at all). Is the TAIL syntax intuitive, systematic, and predictable as I hope? Is AI2 easier or more difficult to use with the addition of the text language?

In addition to questions about the language itself, it is also important to test whether the interface of having additional code blocks in AI2 is useful or if it is clunky and not likely to be used. However, this is something that preliminary user testing will not provide, but will require measuring over time. Thus, it might be useful to have AI2 record when people use the new text features to see how popular they are.

5.3.2 Changing Conversion Architecture to Improve Performance

The current implementation of the TAIL code blocks runs the TAIL parser constantly, in a loop, when the user connects his/her AI2 project to the Android device for testing. Additionally, the set of AI2 blocks corresponding to the TAIL code in the TAIL code block are also generated (invisibly), unbeknownst to the AI2 user. It is these invisible blocks that are then executed in when the app is being run on the Android device. This is horribly inefficient because the parser will execute hundreds of lines of code in order to parse the TAIL code, and the more complicated the TAIL code, the more parser code will be executed. This performance issue can easily be improved by changing the architecture of the implementation slightly. Instead of constantly generating the AI2 blocks for a given TAIL code block when the app is running on the Android device, the corresponding AI2 blocks can be *invisibly present at all times*, linked to the TAIL code block. App Inventor allows invisible disabled blocks to be on the workspace. These blocks are skipped at runtime, and they are invisible to the user.

When a user opts to convert a TAIL code block to AI2 blocks, the hidden AI2 blocks are simply made visible, and the visible TAIL code block is now hidden from the user. When the user needs to execute a TAIL code block, instead of having to run the parser and invisibly generate the set of AI2 blocks to execute in place of the TAIL code block, the YAIL generator for the TAIL code block can simply run the YAIL generator for the hidden AI2 block that already exists. The invisible AI2 blocks should be updated if the TAIL code inside the code block has changed. Thus, it is also important to

discern when the user has finished typing, because it is impractical to try and generate the corresponding hidden AI2 block(s) for every character by character change in the TAIL text. Additional improvement can be made by tracking which sub-blocks of the hidden AI2 blocks are affected by the change in the TAIL code, and only converting those sub-blocks that are related to the most recent change.

The blocks to text conversion can also be improved. Using a process similar to the one described above, linking pre-generated invisible TAIL code blocks to existing AI2 blocks, does not add much efficiency, because the blocks to text conversion process is already very efficient. However, when converting from blocks to text, it would be more efficient to remember the set of blocks, so that if no changes have been made to the blocks, the blocks to text converter need not run again, instead, a cached copy of the text can be returned.

Making this change will greatly improve the functionality and performance of the implementation of the conversion between blocks and text.

Additionally, the improvement mentioned for the text to blocks conversion fixes a current issue with my implementation.

5.3.3 Converting Screens, Workspaces, and Entire Projects

In order to complete the conversion process, users need to be able to convert entire workspaces, screens, and projects into TAIL and back into the identical workspace, screen, or project respectively. Achieving this goal requires many considerations, discussed below.

Converting Screens

When considering the conversion of entire screens, it is important to consider how individual screens and the code pertaining to the specific screens should be represented in TAIL.

Converting Workspaces

Converting workspaces provides some interesting design challenges. In Section 4.1.3, I mention that only complete and valid code fragments can be converted from blocks to text. Blocks with empty sockets are errors (invalid) in AI2, and thus cannot be converted to TAIL in the current implementation, because they do not form a valid program. However, in order to convert workspaces, I need to add a TAIL representation for the user's current workspace and I need to allow this workspace representation to include incomplete blocks such that round trip conversions from blocks to TAIL to blocks will not lose information about the incomplete blocks users had on their workspaces. Additionally it is important to consider the other kinds of information that a workspace provides. A concrete example of such other kinds of information is the position of the blocks on the workspace. If App Inventor users organize their workspaces by positioning blocks in certain fashions or collapsing certain blocks, converting the workspace to TAIL would lose the block positioning and collapsing information. This behavior is undesirable, thus TAIL code should include a representation for information related to block positioning, which blocks are expanded or collapsed, which blocks are enabled or disabled, which blocks have inline inputs or external inputs (especially if the block does not have its default input display type),

as well as any other important information that may be lost in the conversion process.

Converting Projects

Converting entire projects is not only limited to the conversion of the AI2 blocks and information from the AI2 blocks editor. This conversion also needs to keep track of information from the AI2 designer such as which components have been added to which screens, their arrangement on the screen, and all of the information associated with these components (if the user has edited the default initialization of the fields associated with any components). Thus it is also important to consider how components and associated information should be translated into TAIL code.

5.3.4 Making Text Readable

It is important to place careful consideration on how the text language should be presented to the user. Currently TAIL code blocks only allow a single line of text, but TAIL code can quickly become very long. It would be more ideal to allow the user to write in multiple lines of text. Accomplishing this requires some manipulation of the visual component of the AI2 blocks. The AI2 blocks are DOM elements. The TAIL code blocks, in particular, as well as other original AI2 blocks have text boxes inside them (HTML input tag with type attribute set to "text"). The text box only allows a single line of text, of unrestricted length. This text box can be changed to a text area (HTML textarea tag), which would allow multiple lines of text. Making this change requires extending the underlying Blockly framework to allow a

text area field for use within the TAIL code blocks. Additionally, it will be necessary to pretty print (format) the TAIL text that is the result of a blocks to text conversion.

Section 5.4.2 discusses the idea of creating a text editor for TAIL for a larger, more long-term project.

5.3.5 Generalizing TAIL Grammar Rules

Because each AI2 block has its own block type, and each AI2 block's underlying XML is structured a bit differently, trying to conform the TAIL representations of these AI2 blocks to general parser rules is not ideal. However, it is strongly worth considering whether it would be feasible and useful to create a table of unique identifiers for AI2 blocks (whether this identifier is the block's title or some other information on the block such as number of arguments etc.) and the components and general structural requirements of the underlying XML representation of the block (block type attribute, mutations, etc.).

5.3.6 Adding Language Abbreviations

Though TAIL syntax is designed to allow for systematic conversions from blocks to text, in comparison to Venthon with its fewer syntactic markers, TAIL could be considered verbose and clunky. For this reason, it might be helpful to add abbreviations to the language. A sample abbreviation might include removing the curly brace wrapping from around variable references or literal expressions such as numbers, strings, and booleans. See Figure 5.4 for examples of what the abbreviations could look like.

```
Original TAIL                Abbreviation
{{12} + {{5} * {3}}}         {12 + 5 * 3}
{42 * {get x}}               {42 * x}
```

Figure 5.4: Sample TAIL abbreviations

Adding abbreviations to the language requires careful thought, as adding abbreviations may require adding other mechanisms to the language to make sure that the language is not ambiguous. Using the abbreviation in the first example in Figure 5.4 would require TAIL to handle operator precedence, as the removal of the curly braces removes grouping as well.

When thinking about abbreviations and other syntactic sugar, it is important to consider how these might affect the language isomorphism. Consider the following scenario. An abbreviated arithmetic expression with a variable reference (e.g. `{2 * x + 1}`) is converted from TAIL into blocks. The resulting blocks are converted back to TAIL, yielding the full form of the variable reference (e.g. `{{{2} * {get x}} + {1}}`. Though all three of these representations (the abbreviated variable reference, the AI2 blocks, and the expanded form of the variable reference) are all semantically equivalent, the round trip conversion did not yield the original TAIL text that began the conversion.

This particular problem could perhaps be solved by adding operator precedence and by equating abbreviated variable references in TAIL to collapsed variable reference blocks in the AI2 blocks language, but it is easy to imagine that such solutions might not always be possible.

5.3.7 Improving Error Handling

The TAIL code blocks produce an error icon whenever there is an error in the TAIL code inside the editable label of the block. When clicked, the error icon displays an error message generated directly from ANTLR. In the case of invalid component, component event, component property, or component field names, I have specified my own error messages.

It is important to make sure that error messages (whether the ones I have specified or the ones generated by ANTLR) are actually helpful to the user. It is best to test this with user studies.

5.3.8 Creating Tutorials & Documentation for AI2 Users

Before this work can be integrated into an official AI2 release, it is necessary to create tutorials and pages of documentation for both AI2 users and developers detailing all the specifics of TAIL as well as how to use the additional AI2 code blocks. Of course, all of the code and functionality needs to be very well tested and will go through iterations of code reviews as well.

5.4 Far Future

The following sections describe interesting long-term projects that are related to this work.

5.4.1 Venthon, Venti, and More!

Section 2.6.3 describes work I did with a fellow student on Venthon, a textual programming language for App Inventor with Python-esque syntax. This work remains incomplete, but would be interesting to finish. It would be great to offer multiple syntaxes for users to choose from. Venti, (App Inventor + Java) is another possibility. Java, in addition to Python, is another programming language that is often taught to novices. Venti might prove to be a helpful stepping stone between App Inventor and the Android SDK for the users who wish to use some of the features of the Android SDK that are not available in App Inventor.

5.4.2 TAIL Text Editor

With textual programming languages, it is important to have a good text editor. It may prove to be helpful to AI2 users to have a TAIL Text Editor window in addition to the Designer and the Blocks Editor. The text editor might provide useful features like debugging, syntax highlighting, and paren/bracket matching to name a few.

5.4.3 Blocks Language for Existing Text Language

Now that I have explored the relationship between blocks programming languages and textual programming languages by creating a textual language for an existing blocks language, it might be interesting to approach the relationship the other way around. What would a blocks language for an existing textual language such as Python look like? It could be helpful if there were

an isomorphism between Python and a new blocks language, and such a blocks language could be used to introduce novices to programming concepts in an introductory Computer Science class, with the plans to transition to the isomorphic textual language a bit later in the course.

5.4.4 Interactive Environment to Edit Corresponding Blocks and Text Languages Side-By-Side

This project goes hand-in-hand with the project idea outlined in the previous section. It would be interesting to have an environment in which two isomorphic languages (a blocks language and a text language) can be edited side by side. If the edits on one side were reflected in the code on the opposite side in real time, users of such an environment could potentially benefit from understanding how exactly the two languages relate. It could also help novice programmers transition from blocks languages to text languages more easily.

Bibliography

[Ai1a] MIT Center for Mobile Learning, App Inventor Classic home page, http://explore.appinventor.mit.edu/classic, accessed Feb. 24, 2014.

[Ai1b] AI1 usage statistics from http://manhole.mit-appinventor-experimental.appspot.com/, accessed Feb. 21, 2014.

[Ai2a] MIT Center for Mobile Learning, App Inventor 2 home page, http://appinventor.mit.edu, accessed Feb. 24, 2014.

[Ai2b] AI2 usage statistics from http://appinventor.mit.edu/ai2stats/, accessed Feb. 22, 2014.

[Ant] Terrence Parr, ANTLR 3 website, http://www.antlr3.org/, accessed Apr 24, 2014.

[Beh] Kara A. Behnke. *SLASH: Scratch-based visual programming in Second Life for introductory computer science education.*

[Bloa] Neil Fraser, Blockly website, https://code.google.com/p/blockly, accessed Feb. 24, 2014.

[Blob] Blockly Code Demo, https://blockly-demo.appspot.com/static/apps/code/index.html?lang=en, accessed Apr 24, 2014.

[Bri] Java Bridge website, java.appinventor.org, accessed Apr 24, 2014.

[Cs1] CS117 Inventing Mobile and Apps, Wellesley College introductory computer science course. http://cs.wellesley.edu/~cs117, accessed Apr, 24, 2012.

[Dic12] Paul E. Dickson. "Teaching mobile computing using Cabana". In: *Journal of Computing Sciences in Colleges* 27.6 (2012), pp. 128–134.

[Hop] Hopscotch website, https://www.gethopscotch.com/, accessed Feb. 24, 2014.

[Hou] code.org, Hour of Code website, http://code.org/learn, accessed Feb. 24, 2014.

[Leg] The Lego Group, LEGO.com Home, http://www.lego.com/en-us/, accessed Apr. 24, 2014.

[Ope] OpenBlocks home page, MIT Scheller Teacher Education Program, http://education.mit.edu/openblocks, accessed Feb 24, 2014.

[Phia] Philip Guo, Proposal to render Android App Inventor visual code blocks as pseudo-Python code, http://people.csail.mit.edu/pgbovine/android_to_python/, accessed Apr 24, 2014.

[Phib] Philip Guo, Online demo of Android App Inventor yail to pseudo-Python renderer, http://people.csail.mit.edu/pgbovine/android_to_python/yail_to_python_demo.html, accessed Apr 24, 2014.

[Pic] The Playful Invention Company, PicoCricket Reference Guide, version 1.2a, http://www.picocricket.com/pdfs/Reference_Guide_V1_2a.pdf, accessed Mar. 22, 2012.

[Scra] Scratch project, MIT Lifelong Kindergarten Group, http://scratch.mit.edu/, accessed Feb 24, 2014.

[Scrb] Marina Myburgh, Printing the scripts for a Scratch program, http://itisgr8.blogspot.com/2012/01/printing-scripts-for-scratch-program.html, accessed Apr 24, 2014.

[Scrc] Scratch Wiki, Block Plugin, http://wiki.scratch.mit.edu/wiki/Block_Plugin_(2.0), accessed Apr 24, 2014.

[Scrd] Scratch Wiki, Block Plugin Demo, http://blob8108.github.io/scratchblocks2/, accessed Apr 24, 2014.

[Scre] Scratch Wiki, ScratchBlocks generator, http://blob8108.github.io/scratchblocks2/generator, accessed Apr 24, 2014.

[Scrf] Scratch Discussion Forums, Convert your scripts to [scratchblocks] on Scratch 2.0, http://scratch.mit.edu/discuss/topic/14413/, accessed Apr 24, 2014.

[SH] Chazz Sims and Jimmy Hernandez. *Code to Block Interface Component Extension.* Course 6.s063, Final project, https://github.com/JDub20/CodeBlocks?source=c, accessed Apr 24, 2014.

[Staa] StarLogo TNG project, MIT Scheller Teacher Education Program, http://education.mit.edu/projects/starlogo-tng, accessed Feb 24, 2014.

[Stab] StarLogo Nova project, MIT Scheller Teacher Education Program, http://education.mit.edu/projects/starlogo-nova, accessed Apr 24, 2014.

[Tan] Charlene Lee and Sonali Sastry, Tanner Connect App, Wellesley College Computer Science Department introductory CS course, Fall 2011. https://sites.google.com/site/cs117charleneandsonali/, accessed Apr 24, 2014.

[Tyn] Tynker website, http://www.tynker.com/, accessed Feb. 24, 2014.

TAIL Grammar

```
grammar TAIL;
options {
  language = JavaScript;
  backtrack = true;
}
tokens {
  LSQUARE = '[';
  RSQUARE = ']';
  LCURLY = '{';
  RCURLY = '}';
  LPAREN = '(';
  RPAREN = ')';
  LANGLE = '<';
  RANGLE = '>';
  DOT = '.';
  COMMA = ',';

  //Known Keywords
  TRUE = 'true';
  FALSE = 'false';
  WHEN = 'when';
  IF = 'if';
  THEN = 'then:';
  ELSE = 'else:';
  ELSE_IF = 'else_if:';
  FOR_EACH = 'for_each';
  DO = 'do:';
  RESULT = 'result:';
```

```
TO = 'to';
LABEL_TO = 'to:';
CALL = 'call';
INIT_GLOBAL_VAR = 'initialize_global';
INIT_LOCAL_VAR = 'initialize_local';
GET = 'get';
SET = 'set';
GLOBAL = 'global';
IN = 'in:';

//operators
NOT = 'not';
AND = 'and';
OR = 'or';
LEQ = '<=';
GEQ = '>=';
LOGIC_EQ = 'equals';
LOGIC_NOT_EQ = 'not_equals';
EQ = '=';
NOT_EQ = '!=';

ADD = '+';
SUBTRACT = '-';
MULTIPLY = '*';
DIVIDE = '/';
POWER = '^';

//Unary Ops (neg will be the same as subtract)
SQRT = 'sqrt';
ABS = 'abs';
LOG = 'log';
E_EXP = 'e^';
ROUND = 'round';
CEILING = 'ceiling';
FLOOR = 'floor';

//Trig Ops
SIN='sin';
COS='cos';
TAN='tan';
```

```
  ASIN='asin';
  ACOS='acos';
  ATAN='atan';

  //Colors
  COLOR = 'color';
  MAKE_COLOR = 'make_color';
  BLACK = 'black';
  BLUE = 'blue';
  WHITE = 'white';
  MAGENTA = 'magenta';
  RED = 'red';
  LIGHT_GRAY = 'light_gray';
  PINK = 'pink';
  GRAY = 'gray';
  ORANGE = 'orange';
  DARK_GRAY = 'dark_gray';
  YELLOW = 'yellow';
  GREEN = 'green';
  CYAN = 'cyan';

  //lists
  MAKE_LIST = 'make_a_list';
  LIST = 'list';

  //Generic Component Block Stuff
  OF_COMPONENT = 'of_component:';
  FOR_COMPONENT = 'for_component:';
  COMPONENT = 'component';

  //TAIL Block Stuff
  TAIL_EXP = 'TAIL_exp';
  TAIL_STMT = 'TAIL_stmt';
}

@lexer::members{
  var errors = [];
 TAILLexer.prototype.emitErrorMessage = function(error) {
    //var hdr = getErrorHeader(e);
    //var msg = getErrorMessage(e, tokenNames);
```

```
    errors.push(error);
  }
 TAILLexer.prototype.getErrors = function() {
    return errors;
  }
}

@members{
  var errors = [];
 TAILParser.prototype.emitErrorMessage = function(error)
   {
    //var hdr = getErrorHeader(e);
    //var msg = getErrorMessage(e, tokenNames);
    errors.push(error);
  };
 TAILParser.prototype.getErrors = function() {
    return errors;
  };
 TAILParser.prototype.recoverFromMismatchedToken =
   function(input, ttype, follow){
  throw new
   org.antlr.runtime.MismatchedTokenException(ttype,
   input);
 }

 TAILException = function(msg) {
  TAILException.superclass.constructor.call(this, msg);
  this.message = msg;
 };
 org.antlr.lang.extend(TAILException, Error, {
  name: "org.antlr.runtime.TAILException"
 });

 TAILParser.prototype.isValidComponentName =
   function(componentName){
  var componentInstance =
   Blockly.ComponentInstances[componentName];
  //from appinventor/blocklyeditor/src/component.js
  return (typeof componentInstance == "object" &&
   componentInstance.uid != null);
```

```
};
TAILParser.prototype.isValidComponentFieldName =
  function(fields, componentType, fieldName){
 //I am using "field" as a general name for event,
  property or method
 //fields should be of the form "events", "properties",
  or "methods"
 var componentInfo =
  Blockly.ComponentTypes[componentType].componentInfo;
   var componentFields = componentInfo[fields];
   for (var i = 0; i<componentFields.length; i++){
     if(componentFields[i].name === fieldName){
       return true;
     }
   }
   return false;
};
}

@rulecatch{
  catch (re){
    throw re;
  }
}


/*----------------------------------------------------------
 * PARSER RULES
 *----------------------------------------------------------*/

// program
//  : (eventHandler | procdef | funcdef |
   globalvar_decl)*;

// eventHandler
//  : LPAREN WHEN dotted_name (var_decl)? KEYWORD
   (statement_block)* RPAREN;

// procdef
```

```
//   : LPAREN TO IDENTIFIER DO (statement_block)* RPAREN;
   //should a valid program allow empty procdefs and
   event handlers?

// funcdef
//   : LPAREN TO IDENTIFIER RESULT expression_block
   RPAREN;

// globalvar_decl
//   : LPAREN INIT_VAR var_decl 'to:' expression_block;

// var_decl
//   : LANGLE IDENTIFIER RANGLE;

// workspace returns [var xml]
// @init{
//   $xml = document.createElement("xml");
// }
//   : (statement_block
   {$xml.appendChild(statement_block.elt); } |
   expression_block
   {$xml.appendChild($expression_block.elt);})*

top_level_block returns [var elt]
  : LPAREN top_level RPAREN {$elt = $top_level.elt;}
  ;

top_level returns [var elt]
  : global_var_decl {$elt = $global_var_decl.elt;}
  | procedure_decl {$elt = $procedure_decl.elt;}
  | function_decl {$elt = $function_decl.elt;}
  | event_handler {$elt = $event_handler.elt;}
  ;

global_var_decl returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","global_declaration");
  $elt.setAttribute("inline","false");
```

```
  var title = document.createElement("title");
  title.setAttribute("name","NAME");
  var value = document.createElement("value");
  value.setAttribute("name","VALUE");
}
  : INIT_GLOBAL_VAR LANGLE IDENTIFIER RANGLE LABEL_TO
     expression_block
  {
    title.innerHTML = $IDENTIFIER.text;
    value.appendChild($expression_block.elt);
    $elt.appendChild(title);
    $elt.appendChild(value);
  }
  ;

procedure_decl returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","procedures_defnoreturn");

  var hasMutations = false;
  var mutation = document.createElement("mutation");
  var argsCount = 0;

  var name = document.createElement("title");
  name.setAttribute("name","NAME");

  var var_title_arr = [];
}
  : TO LANGLE proc_name=IDENTIFIER RANGLE
     {name.innerHTML = $proc_name.text;}
   (LANGLE arg_name=IDENTIFIER RANGLE {
     hasMutations = true;
     var arg = document.createElement("arg");
     arg.setAttribute("name",$arg_name.text);
     mutation.appendChild(arg);
     var var_title = document.createElement("title");
     var_title.setAttribute("name","VAR"+argsCount);
     var_title.innerHTML = $arg_name.text;
     var_title_arr.push(var_title);
```

```
      argsCount++;
     })*
    DO suite {
     if(hasMutations){
       $elt.appendChild(mutation);
     }
     $elt.appendChild(name);
     for(var i=0; i<var_title_arr.length; i++){
       $elt.appendChild(var_title_arr[i]);
     }
     var seq = $suite.elt;
     seq.setAttribute("name","STACK");
     $elt.appendChild(seq);
    }
   ;

function_decl returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","procedures_defreturn");

  var hasMutations = false;
  var mutation = document.createElement("mutation");
  var argsCount = 0;

  var name = document.createElement("title");
  name.setAttribute("name","NAME");

  var var_title_arr = [];

  var value = document.createElement("value");
  value.setAttribute("name","RETURN");
}
  : TO LANGLE func_name=IDENTIFIER RANGLE
      {name.innerHTML = $func_name.text;}
    (LANGLE arg_name=IDENTIFIER RANGLE {
      hasMutations = true;
      var arg = document.createElement("arg");
      arg.setAttribute("name",$arg_name.text);
      mutation.appendChild(arg);
```

```
      var var_title = document.createElement("title");
      var_title.setAttribute("name","VAR"+argsCount);
      var_title.innerHTML = $arg_name.text;
      var_title_arr.push(var_title);
      argsCount++;
     })*
    RESULT expression_block {
     if(hasMutations){
       $elt.appendChild(mutation);
     }
     $elt.appendChild(name);
     for(var i=0; i<var_title_arr.length; i++){
       $elt.appendChild(var_title_arr[i]);
     }
     value.appendChild($expression_block.elt);
     $elt.appendChild(value);
    }
   ;

event_handler returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","component_event");
  var mutation = document.createElement("mutation");
  //mutation.setAttribute("component_type",)
  //mutation attributes will be set inside the body of
     the rule
  //dotted names allow spaces...which we don't want
     allowed...
  var title = document.createElement("title");
  title.setAttribute("name","COMPONENT_SELECTOR");
}
  : WHEN component=IDENTIFIER DOT event=IDENTIFIER
  {
    var componentName = $component.text;
    var eventName = $event.text;
    var componentInstance =
       Blockly.ComponentInstances[componentName];
    var componentType;
    if (this.isValidComponentName(componentName)){
```

```
    componentType =
       Blockly.Component.instanceNameToTypeName(componentName);
    mutation.setAttribute("component_type",
       componentType);
    mutation.setAttribute("instance_name",
       componentName);
    title.innerHTML = componentName;
  } else {
    throw new TAILException("Invalid component name: "
       + componentName);
    //this.emitErrorMessage("Invalid component name: "
       + componentName);
    //the parser will continue even after this error
       because syntactically this is still correct...
  }
  if(this.isValidComponentFieldName("events",
     componentType, eventName)){
    mutation.setAttribute("event_name", eventName);
  }else{
    throw new TAILException("Invalid event name: " +
       eventName);
    //this.emitErrorMessage("Invalid event name: " +
       eventName);
  } //no need for else case, we've already added an
     error to the errors array above

}
 (LANGLE arg=IDENTIFIER RANGLE)* //apparently we don't
    have to put these in the DOM....
 DO suite
 {
  var statements = $suite.elt;
  statements.setAttribute("name","DO");

  $elt.appendChild(mutation);
  $elt.appendChild(title);
  $elt.appendChild(statements);
 }
;
```

```
expression_start returns [var xml]
@init{
  $xml = document.createElement("block");
}
  : expression_block
     {$xml.appendChild($expression_block.elt);}
  ;

expression_block returns [var elt]
  : LCURLY expression RCURLY {$elt = $expression.elt;}
  ;

suite returns [var elt]
@init{
  $elt = document.createElement("statement"); //TODO
     figure out what goes here???!!!??
  //the name attribute of the set of statements will be
     set by whatever is calling this rule.
  var count = 0;
  var prevStatementBlock;
  var currentStatementBlock;
  var stmt_arr = [];
}
  : (statement_block
    {
     if (count === 0){ // this is the very first
       statement
      prevStatementBlock = $statement_block.elt;
      $elt.appendChild(prevStatementBlock);
     }else{ //all of the rest of the statement blocks
      var next = document.createElement("next");
      var currentStmt = $statement_block.elt;
      next.appendChild(currentStmt);
      prevStatementBlock.appendChild(next);
      prevStatementBlock = currentStmt;
     }
     count++;
    })*

  //This code used to be inside the parens above....
```

```
  // {
  //  if (count === 0){ // this is the very first
     statement
  //    prevStatementBlock = $statement_block.elt;
  //    $elt.appendChild(prevStatementBlock);
  //  }else{ //all of the rest of the statement blocks
  //    var next = document.createElement("next");
  //    var currentStmt = $statement_block.elt;
  //    next.appendChild(currentStmt);
  //    prevStatementBlock.appendChild(next);
  //    prevStatementBlock = currentStmt;
  //  }
  //  count++;
  // }
  //TODO: stuff needs to go here...actiony things...
  // {
  //  for (var i = 0; i<stmt_arr.length; i++){
  //    currentStatementBlock = stmt_arr[i];
  //    if(i===0){
  //      $elt.appendChild(currentStatementBlock);
  //      prevStatementBlock = currentStatementBlock;
  //    } else{
  //      var next = document.createElement("next");
  //      next.appendChild(currentStatementBlock);
  //
     prevStatementBlock.appendChild(currentStatementBlock);
  //      prevStatementBlock = currentStatementBlock;
  //    }
  //  }
  // }
  ; //think about requiring newlines

  // {//stmt_arr.push($statement_block.elt);}

statement_block returns [var elt]
  : LSQUARE statement RSQUARE
  {$elt = $statement.elt;}
  ;

statement returns [var elt]
```

```
  : if_stmt {$elt = $if_stmt.elt;}
  | variable_set_stmt {$elt = $variable_set_stmt.elt;}
  | component_stmt {$elt = $component_stmt.elt;}
  | tail_stmt {$elt = $tail_stmt.elt;}
  ;

if_stmt returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("inline","false");
  $elt.setAttribute("type","controls_if");

  var mutation = document.createElement("mutation");
  var mutations = false;
  var else_if_count = 0;
  var else_count = 0;
}
  : IF e1=expression_block {
    var val = document.createElement("value");
    val.setAttribute("name","IF0");
    val.appendChild($e1.elt);
    $elt.appendChild(val);
  }
  THEN a=suite{
    var then_stmts = $a.elt;
    then_stmts.setAttribute("name", "DO0");
    //then_stmts.appendChild($a.elt);
    $elt.appendChild(then_stmts);
  }
  ((ELSE_IF {mutations = true; else_if_count++;}
     e2=expression_block{
    var value = document.createElement("value");
    value.setAttribute("name","IF"+else_if_count);
    value.appendChild($e2.elt);
    $elt.appendChild(value);
  } THEN b=suite {
    var else_if_stmts = $b.elt;
    else_if_stmts.setAttribute("name","DO"+else_if_count);
    // else_if_stmts.appendChild($b.elt);
    $elt.appendChild(else_if_stmts);
```

```
  })* (ELSE c=suite {
    mutations = true;
    else_count++;

    var else_stmts = $c.elt;
    else_stmts.setAttribute("name","ELSE");
    // else_stmts.appendChild($c.elt);
    $elt.appendChild(else_stmts);
  })?)?
  {
    if(mutations){
      if (else_if_count !== 0){
        mutation.setAttribute("elseif",else_if_count);
      }
      if (else_count !== 0){
        mutation.setAttribute("else",else_count);
      }
      $elt.insertBefore(mutation,
         $elt.firstElementChild);
    }
  }
  ;

variable_set_stmt returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","lexical_variable_set");
  $elt.setAttribute("inline","false");

  var title = document.createElement("title");
  title.setAttribute("name","VAR");

  var var_name = "";

  var value = document.createElement("value");
  value.setAttribute("name","VALUE");
}
  : SET (GLOBAL {var_name += "global ";})? IDENTIFIER
     {var_name += $IDENTIFIER.text;} LABEL_TO
     expression_block {
```

```
    title.innerHTML = var_name;
    $elt.appendChild(title);

    value.appendChild($expression_block.elt);
    $elt.appendChild(value);
  }
  ;

component_stmt returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("inline","false");
  var mutation = document.createElement("mutation");
  var isGeneric = false;
}
  : SET component=IDENTIFIER DOT property=IDENTIFIER
     (OF_COMPONENT of_comp=expression_block {isGeneric =
     true;})? LABEL_TO to=expression_block {

    $elt.setAttribute("type","component_set_get");
    mutation.setAttribute("set_or_get", "set");

    var componentName = $component.text;
    var propName = $property.text;

    var componentType;
    if(isGeneric){
      if(!Blockly.ComponentTypes.haveType(componentName)){
        throw new TAILException("Invalid Generic
           Component Name: " + componentName);
      } else{
        componentType = componentName;
      }
    } else{
      if(!this.isValidComponentName(componentName)){
        throw new TAILException("Invalid Component Name:
           " + componentName);
      } else{
        componentType =
           Blockly.Component.instanceNameToTypeName(componentName);
```

```
    }
  }

  mutation.setAttribute("component_type",componentType);
  mutation.setAttribute("is_generic", isGeneric);

  if(!this.isValidComponentFieldName("properties",
     componentType, propName)){
    throw new TAILException("Invalid Component
       Property Name: " + propName);
  } else {
    mutation.setAttribute("property_name",propName);
    if(!isGeneric){
      mutation.setAttribute("instance_name",componentName);
      var title = document.createElement("title");
      title.setAttribute("name","COMPONENT_SELECTOR");
      title.innerHTML = componentName;
      $elt.appendChild(title);
    }
    var title = document.createElement("title");
    title.setAttribute("name","PROP");
    title.innerHTML = propName;
    $elt.appendChild(title);

    if(isGeneric){
      var componentVal =
         document.createElement("value");
      componentVal.setAttribute("name","COMPONENT");
      componentVal.appendChild($of_comp.elt);
      $elt.appendChild(componentVal);
    }

    var value = document.createElement("value");
    value.setAttribute("name","VALUE");
    value.appendChild($to.elt);
    $elt.appendChild(value);
  }
  $elt.insertBefore(mutation, $elt.firstElementChild);
}
| {var valArr = []; var argCount=0;}
```

```
CALL component=IDENTIFIER DOT method=IDENTIFIER
(FOR_COMPONENT for_comp=expression_block
   {isGeneric=true;})?
(LABEL arg=expression_block {
  var val = document.createElement("value");
  val.setAttribute("name","ARG"+argCount);
  val.appendChild($arg.elt);
  valArr.push(val);
})*{
  $elt.setAttribute("type","component_method");

  var componentName = $component.text;
  var methodName = $method.text;

  var componentType;
  if(isGeneric){
    if(!Blockly.ComponentTypes.haveType(componentName)){
      throw new TAILException("Invalid Generic
         Component Name: " + componentName);
    } else{
      componentType = componentName;
    }
  } else{
    if(!this.isValidComponentName(componentName)){
      throw new TAILException("Invalid Component Name:
         " + componentName);
    } else{
      componentType =
         Blockly.Component.instanceNameToTypeName(componentName);
    }
  }
  mutation.setAttribute("component_type",componentType);
  mutation.setAttribute("is_generic", isGeneric);
  if(!this.isValidComponentFieldName("methods",
     componentType, methodName)){
    throw new TAILException("Invalid Component Method
       Name: " + methodName);
  } else {
    mutation.setAttribute("method_name",methodName);
    if(!isGeneric){
```

```
        mutation.setAttribute("instance_name",componentName);
        $elt.appendChild(mutation);

        var title = document.createElement("title");
        title.setAttribute("name","COMPONENT_SELECTOR");
        title.innerHTML = componentName;
        $elt.appendChild(title);
      } else{
        $elt.appendChild(mutation);
        var value = document.createElement("value");
        value.setAttribute("name","COMPONENT");
        value.appendChild($for_comp.elt);
        $elt.appendChild(value);
      }
    }
    for (var i = 0; i<valArr.length; i++){
        $elt.appendChild(valArr[i]);
    }
  }
  ;


tail_stmt returns [var elt]
@init{
  console.log("In the TAIL STMT Rule...");
  $elt = document.createElement("block");
  $elt.setAttribute("type","code_write_tail_stmt");

  var title = document.createElement("title");
  title.setAttribute("name","CODE");
}
  : TAIL_STMT statement_block
  {
    title.innerHTML =
       $text.substring($TAIL_STMT.text.length,$text.length).trim();
    $elt.appendChild(title);
  }
  ;

expression returns [var elt]
```

```
  : if_expr {$elt = $if_expr.elt;}
  //| do_result_expr {$elt = $do_result_expr.elt;}
  | logic_expr {$elt = $logic_expr.elt;}
  | not_expr {$elt = $not_expr.elt;}
  | compare_eq_expr {$elt = $compare_eq_expr.elt;}
  | compare_math_eq_expr {$elt =
     $compare_math_eq_expr.elt;}
  | compare_math_expr {$elt = $compare_math_expr.elt;}
  | math_expr {$elt = $math_expr.elt;}
  | init_local_expr {$elt = $init_local_expr.elt;}
  | component_expr {$elt = $component_expr.elt;}
  | variable_ref_expr {$elt = $variable_ref_expr.elt;}
  | color_expr {$elt = $color_expr.elt;}
  | list_expr {$elt = $list_expr.elt;}
  | tail_expr {$elt = $tail_expr.elt;}
  | atom {$elt = $atom.elt;}
  ;

// assign_stmt
//  : SET IDENTIFIER ('to:')? expression_block;

// if_stmt
//  : IF expression_block THEN (statement_block)*
   (ELSE_IF (statement_block)*)* ELSE (statement_block)*;



/****************** Expressions ****************/

if_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","controls_choose");
  $elt.setAttribute("inline","false");
}
  : IF a=expression_block THEN b=expression_block ELSE
     c=expression_block
  {
    var testVal = document.createElement("value");
    testVal.setAttribute("name","TEST");
```

```
    testVal.appendChild($a.elt);

    var thenVal = document.createElement("value");
    thenVal.setAttribute("name","THENRETURN");
    thenVal.appendChild($b.elt);

    var elseVal = document.createElement("value");
    elseVal.setAttribute("name","ELSERETURN");
    elseVal.appendChild($c.elt);

    $elt.appendChild(testVal);
    $elt.appendChild(thenVal);
    $elt.appendChild(elseVal);
  }
  ;

// do_result_expr returns [var elt]
// @init{
//   $elt = document.createElement("block");
//   Attribute type = document.createAttribute("type");
//   type.value = "controls_do_then_return";
//   Attribute inline =
   document.createAttribute("inline");
//   inline.value = "false";

//   $elt.setAttribute(type);
//   $elt.setAttribute(inline);
// }
//   : 'do' (statement_block)* RESULT expression_block;
   //TODO finish this when I add statements

logic_expr returns [var elt]
@init{
  $elt = document.createElement("block");

  $elt.setAttribute("type","logic_operation");
  $elt.setAttribute("inline","true");

  var operation = "";
}
```

```
  : a = expression_block
   (AND {operation="AND";}| OR {operation="OR";})
   b = expression_block
   {
    var title = document.createElement("title");
    title.setAttribute("name","OP");
    title.innerHTML = operation;

    var valA = document.createElement("value");
    valA.setAttribute("name","A");
    valA.appendChild($a.elt);

    var valB = document.createElement("value");
    valB.setAttribute("name","B");
    valB.appendChild($b.elt);

    $elt.appendChild(title);
    $elt.appendChild(valA);
    $elt.appendChild(valB);
   }
   ;

not_expr returns [var elt]
@init{
  $elt = document.createElement("block");

  $elt.setAttribute("type","logic_negate");
  $elt.setAttribute("inline","false");
}
  : NOT expression_block
  {
    var value = document.createElement("value");
    value.setAttribute("name","BOOL");
    value.appendChild($expression_block.elt);

    $elt.appendChild(value);
  }
  ;

compare_eq_expr returns [var elt]
```

```
@init{
  $elt = document.createElement("block");

  $elt.setAttribute("type","logic_compare");
  $elt.setAttribute("inline","true");

  var operation = "";
}
  : a=expression_block (LOGIC_EQ {operation = "EQ";} |
     LOGIC_NOT_EQ {operation = "NEQ";})
     b=expression_block
  {
    var title = document.createElement("title");
   title.setAttribute("name","OP");
   title.innerHTML = operation;

   var valA = document.createElement("value");
   valA.setAttribute("name","A");
   valA.appendChild($a.elt);

   var valB = document.createElement("value");
   valB.setAttribute("name","B");
   valB.appendChild($b.elt);

   $elt.appendChild(title);
   $elt.appendChild(valA);
   $elt.appendChild(valB);
  }
  ;

compare_math_eq_expr returns [var elt] //exactly the
   same as compare_eq_expr except for commented line
   below and the operators used
@init{
  $elt = document.createElement("block");

  $elt.setAttribute("type","math_compare");
  //this is the only difference between this rule and
     the rule above
  $elt.setAttribute("inline","true");
```

```
  var operation = "";
}
  : a=expression_block (EQ {operation = "EQ";} | NOT_EQ
     {operation = "NEQ";}) b=expression_block
  {
    var title = document.createElement("title");
   title.setAttribute("name","OP");
   title.innerHTML = operation;

   var valA = document.createElement("value");
   valA.setAttribute("name","A");
   valA.appendChild($a.elt);

   var valB = document.createElement("value");
   valB.setAttribute("name","B");
   valB.appendChild($b.elt);

   $elt.appendChild(title);
   $elt.appendChild(valA);
   $elt.appendChild(valB);
  }
  ;

compare_math_expr returns [var elt]
@init{
  $elt = document.createElement("block");

  $elt.setAttribute("type","math_compare");
  $elt.setAttribute("inline","true");

  var operation = "";
}
  : a=expression_block (LANGLE {operation = "LT";}|
     RANGLE {operation = "GT";}| LEQ {operation =
     "LTE";}| GEQ {operation = "GTE";})
     b=expression_block
  //EQ and NOT_EQ are not listed here, because although
     they're meant for math blocks, you can plug-in non
     math blocks in their sockets
```

```
  {
    var title = document.createElement("title");
   title.setAttribute("name","OP");
   title.innerHTML = operation;

   var valA = document.createElement("value");
   valA.setAttribute("name","A");
   valA.appendChild($a.elt);

   var valB = document.createElement("value");
   valB.setAttribute("name","B");
   valB.appendChild($b.elt);

   $elt.appendChild(title);
   $elt.appendChild(valA);
   $elt.appendChild(valB);
  }
  ;

math_expr returns [var elt]
  : mutable_arith_expr {$elt = $mutable_arith_expr.elt;}
  //for add and multiply which allow mutations in
     Blockly (TAIL will NOT be allowing mutations)
  | non_mutable_arith_expr {$elt =
     $non_mutable_arith_expr.elt;}
  | special_math_expr {$elt = $special_math_expr.elt;}
  //this is for modulo_of, remainder_of, and quotient_of
  | unary_math_expr {$elt = $unary_math_expr.elt;}
  | math_trig_expr {$elt = $math_trig_expr.elt;}
  ;

mutable_arith_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  //type will get a value inside the rule
  $elt.setAttribute("inline","true");

  //initalizing these in advance to be used later
  var mutation = document.createElement("mutation");
  var itemCount = 0;
```

```
    var valArr = [];
    //var value;

    var addValue = function(element){
      var value = document.createElement("value");
      value.setAttribute("name", "NUM" + itemCount);
      value.appendChild(element);
      valArr.push(value);
      itemCount++;
    }
  }
    : a=expression_block
     {
       // value = document.createElement("value");
       // value.setAttribute("name", "NUM" + itemCount);
       // value.appendChild($a.elt);
       // valArr.push(value);
       // itemCount++;
       addValue($a.elt);
     }
     (   (ADD b=expression_block
          {
            // value = document.createElement("value");
            // value.setAttribute("name", "NUM" + itemCount);
            // value.appendChild($b.elt);
            // valArr.push(value);
            // itemCount++;
            addValue($b.elt);
          }
      )+
      {$elt.setAttribute("type","math_add");}
     | (MULTIPLY c=expression_block
        {
         // value = document.createElement("value");
         // value.setAttribute("name", "NUM" + itemCount);
         // value.appendChild($c.elt);
         // valArr.push(value);
         // itemCount++;
         addValue($c.elt);
        }
```

```
    )+
    {$elt.setAttribute("type","math_multiply");}
   )
   {
    mutation.setAttribute("items",itemCount);
    $elt.appendChild(mutation);

    for (var i = 0; i<valArr.length; i++){
      $elt.appendChild(valArr[i]);
    }
   }
  ;

non_mutable_arith_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  //type will get a value inside the rule
  $elt.setAttribute("inline","true");
}
  : a=expression_block
  (
    SUBTRACT {
      $elt.setAttribute("type","math_subtract");
    }
    | DIVIDE {
      $elt.setAttribute("type","math_division");
    }
    | POWER {
      $elt.setAttribute("type","math_power");
    }
  )
  b=expression_block
  {
    var valA = document.createElement("value");
   valA.setAttribute("name","A");
   valA.appendChild($a.elt);

   var valB = document.createElement("value");
   valB.setAttribute("name","B");
   valB.appendChild($b.elt);
```

```
    $elt.appendChild(valA);
    $elt.appendChild(valB);
   }
   ;
special_math_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","math_divide");
  $elt.setAttribute("inline","true");

  var operation = "";
}
  :('modulo_of' {operation="MODULO";}
    | 'remainder_of' {operation="REMAINDER";}
    | 'quotient_of' {operation="QUOTIENT";})
  a=expression_block DIVIDE b=expression_block
  {
    var title = document.createElement("title");
    title.setAttribute("name","OP");
    title.innerHTML = operation;

    var dividend = document.createElement("value");
    dividend.setAttribute("name","DIVIDEND");
    dividend.appendChild($a.elt);

    var divisor = document.createElement("value");
    divisor.setAttribute("name","DIVISOR");
    divisor.appendChild($b.elt);

    $elt.appendChild(title);
    $elt.appendChild(dividend);
    $elt.appendChild(divisor);
  }
  ;

unary_math_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","math_single");
```

```
  $elt.setAttribute("inline","false");

  var operation = "";

}
  : op=(SQRT {operation = "ROOT";}
    | ABS {operation = "ABS";}
    | SUBTRACT {operation = "NEG";}
    | LOG {operation = "LN";}
    | E_EXP {operation = "EXP";}
    | ROUND {operation = "ROUND";}
    | CEILING {operation = "CEILING";}
    | FLOOR {operation = "FLOOR";})
  expression_block
  {
    var title = document.createElement("title");
    title.setAttribute("name","OP");
    title.innerHTML = operation;

    var value = document.createElement("value");
    value.setAttribute("name","NUM");
    value.appendChild($expression_block.elt);

    $elt.appendChild(title);
    $elt.appendChild(value);
  }
  ;

math_trig_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","math_trig");
  $elt.setAttribute("inline","false");

  var operation = "";
}
  : (SIN {operation="SIN";}
    |COS {operation="COS";}
    |TAN {operation="TAN";}
    |ASIN {operation="ASIN";}
```

```
    |ACOS {operation="ACOS";}
    |ATAN {operation="ATAN";})
  expression_block
  {
    var title = document.createElement("title");
    title.setAttribute("name","OP");
    title.innerHTML = operation;

    var value = document.createElement("value");
    value.setAttribute("name","NUM");
    value.appendChild($expression_block.elt);

    $elt.appendChild(title);
    $elt.appendChild(value);
  }
  ;

init_local_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","local_declaration_expression");
  var mutation = document.createElement("mutation");
  var localName;
  var titleArr = [];
  var title;
  var count = 0;
  var valArr = [];
  var value;
}
  : INIT_LOCAL_VAR (LANGLE IDENTIFIER RANGLE LABEL_TO
     a=expression_block
    {
      localName = document.createElement("localname");
      localName.setAttribute("name",$IDENTIFIER.text);
      mutation.appendChild(localName);

      title = document.createElement("title");
      title.setAttribute("name","VAR" + count);
      title.innerHTML = $IDENTIFIER.text;
      titleArr.push(title);
```

```
      value = document.createElement("value");
      value.setAttribute("name", "DECL"+count);
      value.appendChild($a.elt);
      valArr.push(value);
      count++;
    })+
   {
    $elt.appendChild(mutation);
    titleArr.forEach(function(title){
      $elt.appendChild(title);
    });
    valArr.forEach(function(value){
      $elt.appendChild(value);
    });
   }
   IN b=expression_block
   {
    var returnVal = document.createElement("value");
    returnVal.setAttribute("name","RETURN");
    returnVal.appendChild($b.elt);
    $elt.appendChild(returnVal);
   }
  ;

variable_ref_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","lexical_variable_get");

  var variable = "";
}

  : (GET)? (GLOBAL {variable += "global ";})?
     IDENTIFIER //the space after global is very
     important
  {
    variable += $IDENTIFIER.text;
    var title = document.createElement("title");
    title.setAttribute("name","VAR");
```

```
    title.innerHTML = variable;

    $elt.appendChild(title);
  }
  ;

color_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  var title = document.createElement("title");
  title.setAttribute("name","COLOR");

  var type = "color_";
}
  : COLOR ( (BLACK {title.innerHTML="#000000"; type +=
     $BLACK.text;}
        | BLUE {title.innerHTML="#0000ff"; type +=
       $BLUE.text;}
        | WHITE {title.innerHTML="#ffffff"; type +=
       $WHITE.text;}
        | MAGENTA {title.innerHTML="#ff00ff"; type +=
       $MAGENTA.text;}
        | RED {title.innerHTML="#ff0000"; type +=
       $RED.text;}
        | LIGHT_GRAY {title.innerHTML="#cccccc"; type +=
       $LIGHT_GRAY.text;}
        | PINK {title.innerHTML="#ffafaf"; type +=
       $PINK.text;}
        | GRAY {title.innerHTML="#888888"; type +=
       $GRAY.text;}
        | ORANGE {title.innerHTML="#ffc800"; type +=
       $ORANGE.text;}
        | DARK_GRAY {title.innerHTML="#444444"; type +=
       $DARK_GRAY.text;}
        | YELLOW {title.innerHTML="#ffff00"; type +=
       $YELLOW.text;}
        | GREEN {title.innerHTML="#00ff00"; type +=
       $GREEN.text;}
        | CYAN {title.innerHTML="#00ffff"; type +=
       $CYAN.text;}
```

```
        )
       {
         $elt.setAttribute("type",type);
         $elt.appendChild(title);
       }
     | HEX //custom_color
      {
       title.innerHTML = $HEX.text;
        $elt.setAttribute("type","color_black");
        //because there's no other default color type
        $elt.appendChild(title);
       }
      )
   | MAKE_COLOR expression_block
   {
     $elt.setAttribute("type","color_make_color");
     $elt.setAttribute("inline","false");
     var value = document.createElement("value");
     value.setAttribute("name","COLORLIST");
     value.appendChild($expression_block.elt);
     $elt.appendChild(value);
   }
   ;

list_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","lists_create_with");

  var mutation = document.createElement("mutation");
  var item_count = 0;

  var val_block_arr = [];
  var val_block;
}
  : (LIST | MAKE_LIST)
   (options {greedy=true;}: expression_block
     {
      item_count++;
      val_block = document.createElement("value");
```

```
      val_block.setAttribute("name", ("ADD" +
     (item_count-1)));
      val_block.appendChild($expression_block.elt);
      val_block_arr.push(val_block);
     }
   )*
  {
    mutation.setAttribute("items",item_count);
    $elt.appendChild(mutation);
    val_block_arr.forEach(function (block) {
      $elt.appendChild(block);
    });
  }
  ;

   /*(options {k=2;}: COMMA b=expression_block
     {
      item_count++;
      val_block = document.createElement("block");
      val_block.setAttribute("name", "ADD" +
     (item_count-1));
      val_block.appendChild($b.elt);
      val_block_arr.add(val_block);
     }
   )?*/


component_expr returns [var elt]
@init{
  $elt = document.createElement("block");
  var mutation = document.createElement("mutation");
  var isComponentSetGet = false;
  var isGeneric = false;
  var value;
}
  : COMPONENT component=IDENTIFIER {
    $elt.setAttribute("type","component_component_block");
    var componentName = $component.text;
    var componentType;
    if(!this.isValidComponentName(componentName)){
```

```
    throw new TAILException("Invalid Component Name: "
       + componentName);
  } else{
    componentType =
       Blockly.Component.instanceNameToTypeName(componentName);
  }

  mutation.setAttribute("component_type",componentType);
  mutation.setAttribute("instance_name",
     componentName);
  $elt.appendChild(mutation);

  var title = document.createElement("title");
  title.setAttribute("name","COMPONENT_SELECTOR");
  title.innerHTML = componentName;
  $elt.appendChild(title);
}

| component=IDENTIFIER DOT property=IDENTIFIER
   {isComponentSetGet = true;} (OF_COMPONENT
   expression_block {isGeneric = true;})?
{
  var componentName = $component.text;
  var propName = $property.text;

  //figure out component type based on whether this is
     a generic component block or not
  var componentType;
  if(isGeneric){
    $elt.setAttribute("inline","false"); //an extra
       thing to put in the DOM only in the case where
       isGeneric
    value = document.createElement("value");
    value.setAttribute("name","COMPONENT");
    value.appendChild($expression_block.elt);

    if(!Blockly.ComponentTypes.haveType(componentName)){
      throw new TAILException("Invalid Generic
         Component Name: " + componentName);
    } else{
```

```
    componentType = componentName;
  }
} else{
  if(!this.isValidComponentName(componentName)){
    throw new TAILException("Invalid Component Name:
       " + componentName);
  } else{
    componentType =
       Blockly.Component.instanceNameToTypeName(componentName);
  }
}
mutation.setAttribute("component_type",componentType);

if(!isGeneric){
  mutation.setAttribute("instance_name",componentName);

  var compSelectorTitle =
     document.createElement("title");
  compSelectorTitle.setAttribute("name","COMPONENT_SELECTOR");
  compSelectorTitle.innerHTML = componentName;
  $elt.appendChild(compSelectorTitle);
}
if(isComponentSetGet){
  $elt.setAttribute("type","component_set_get");
  mutation.setAttribute("set_or_get","get");
  if(!this.isValidComponentFieldName("properties",
     componentType, propName)){
    throw new TAILException("Invalid Component
       Property Name: " + propName);
  }else{
    mutation.setAttribute("property_name",propName);
    var title = document.createElement("title");
    title.setAttribute("name","PROP");
    title.innerHTML = propName;
    $elt.appendChild(title);
  }
    mutation.setAttribute("is_generic",isGeneric);
}else{
  $elt.setAttribute("type","component_component_block");
}
```

```
    $elt.insertBefore(mutation, $elt.firstElementChild);
    if(isGeneric){
      $elt.appendChild(value);
    }
  }
  ;

tail_expr returns [var elt] //DOES THIS WORK??
@init{
  $elt = document.createElement("block");
  $elt.setAttribute("type","code_write_tail_exp");

  var title = document.createElement("title");
  title.setAttribute("name","CODE");
}
  : TAIL_EXP expression_block
  {
    title.innerHTML =
       $text.substring($TAIL_EXP.text.length,$text.length).trim();
    $elt.appendChild(title);
  }
  ;

atom returns [var elt]
@init{
  $elt = document.createElement("block");

  var title = document.createElement("title");
}
  : NUMBER {
    $elt.setAttribute("type","math_number");

    title.setAttribute("name","NUM");
    title.innerHTML = $NUMBER.text;
    $elt.appendChild(title);
  }
  | STRING {
    $elt.setAttribute("type","text");
```

```
    title.setAttribute("name","TEXT");
    var text = $STRING.text;
    title.innerHTML = text.substring(1,text.length-1);
    $elt.appendChild(title);
  }
  | TRUE {
    $elt.setAttribute("type","logic_boolean");

    title.setAttribute("name","BOOL");
    title.innerHTML = "TRUE";
    $elt.appendChild(title);
  }
  | FALSE {
    $elt.setAttribute("type","logic_boolean");

    title.setAttribute("name","BOOL");
    title.innerHTML = "FALSE";
    $elt.appendChild(title);
  }
  ;

dotted_name
  : IDENTIFIER '.' IDENTIFIER;


/*---------------------------------------------------------
* LEXER RULES
*----------------------------------------------------------*/

fragment
ALPHA : ('a' .. 'z' | 'A' .. 'Z');

fragment
DIGIT : ('0' .. '9');

fragment
ALPHA_NUM
  : ALPHA | DIGIT
  ;
```

```
fragment
ESC
  : '\\' .
  ;

NUMBER : (DIGIT* DOT DIGIT+ | DIGIT+ (DOT)?);

//TODO: CHECK AI2 rules for Identifiers
IDENTIFIER : (ALPHA | '_')
        (ALPHA | '_' | DIGIT)*; //identifiers cannot
     start with numbers

LABEL : (ALPHA | '_')+ ':';

KEYWORD : (ALPHA | '_')* ':';

STRING
  : ('\'' (ESC | ~('\\' | '\n' | '\''))* '\'')
  | ('"' (ESC | ~('\\' | '\n' | '"'))* '"')
  ;

HEX
  : '#' ALPHA_NUM ALPHA_NUM ALPHA_NUM ALPHA_NUM
     ALPHA_NUM ALPHA_NUM
  ;

WS :  ( ' '
    | '\t'
    | '\r'
    | '\n'
    ) {$channel=HIDDEN;}
  ;
```

TAIL Code Blocks

```
/**
 * Visual Blocks Language
 *
 * Copyright 2012 Google Inc.
 * http://code.google.com/p/blockly/
 *
 * Licensed under the Apache License, Version 2.0 (the "
   License");
 * you may not use this file except in compliance with
   the License.
 * You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in
   writing, software
 * distributed under the License is distributed on an "AS
    IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
   express or implied.
 * See the License for the specific language governing
   permissions and
 * limitations under the License.
 */

/**
 * @fileoverview Blocks for textual languages in App
   Inventor
 * @author kchadha@wellesley.edu (Karishma Chadha)
```

```
 */

if (!Blockly.Language) Blockly.Language = {};

if (!Blockly.Language.code) Blockly.Language.code = {};

Blockly.Language.code.parser = function(input){
  var cstream = new org.antlr.runtime.ANTLRStringStream(
     input);
  var lexer = new TAILLexer(cstream);
  var tstream = new org.antlr.runtime.CommonTokenStream(
     lexer);
  var parser = new TAILParser(tstream);
  return parser;
};

Blockly.Language.code_write_tail_exp = {
  category: Blockly.LANG_CATEGORY_CODE,
  init: function(){
    this.setColour(Blockly.CODE_CATEGORY_HUE);
    this.appendDummyInput().appendTitle('TAIL exp').
   appendTitle(
      new Blockly.FieldCodeBlockInput('{"write TAIL exp
   "}'), 'CODE');
    this.setOutput(true, null);
    this.setTooltip(Blockly.
     LANG_CODE_WRITE_TAIL_EXP_TOOLTIP);
    this.errors = [{name:"checkANTLRErrors"}];
    this.appendCollapsedInput().appendTitle('TAIL', '
     COLLAPSED_TEXT');
  },
  onchange: Blockly.WarningHandler.checkErrors,
  typeblock: [{ translatedName: Blockly.
   LANG_CODE_WRITE_TAIL_EXP }],
  prepareCollapsedText: function(){
    var textToDisplay = this.getTitleValue('CODE');
    if (textToDisplay.length > 8) //8 is a length of 5
     plus 3 dots
      textToDisplay = textToDisplay.substring(0, 5) +
     '...';
```

```
    this.getTitle_('COLLAPSED_TEXT').setText(
     textToDisplay, 'COLLAPSED_TEXT');
  },
  codeCustomContextMenu: function(options){
    var myBlock = this;
    var convertToBlocksOption = {text: "Convert to Blocks
   "};
   if(!this.errorIcon && !this.disabled){
      convertToBlocksOption.enabled = true;
    } else{
      convertToBlocksOption.enabled = false;
    }
    convertToBlocksOption.callback = function(){
      Blockly.Language.code_write_tail_exp.
   generateAIExpressionBlock(myBlock,true);
      myBlock.dispose(true,false);
    };
    options.push(convertToBlocksOption);
  }
};

/**
* This function generates the new blocks equivalent to
   the tail expression block.
* @param {!Sting} tailText The TAIL text that needs to be
    parsed. This is not always
* the same as the current tyext in the block (see
   generator for this block).
* @param {!Blockly.Block} myBlock The TAIL expression
   block to convert into AI blocks.
* @param {!Boolean} keepConnections If true, the new
   blocks will replace the old blocks
* and keep the TAIL blocks connections (if it is
   connected to anything). If false, the
* connections will not be kept (this is for use in the
   generator, where we essentially
* want to generate invisible blocks, but instead we
   generate blocks elsewhere in the
* workspace and dispose of them when we're done using
   them).
```

```
* @return {!Blockly.Block} Returns the root block of the
   generated set of blocks.
*/
Blockly.Language.code_write_tail_exp.
   generateAIExpressionBlock = function(myBlock,
   keepConnections){
  try{
    var parser = Blockly.Language.code.parser(myBlock.
       getTitleValue('CODE'));
    var newBlockDom = parser.expression_block();
   //console.log("New Block Dom: \n" + Blockly.Xml.
     domToPrettyText(newBlockDom));
   var newBlock = Blockly.Xml.domToBlock_(myBlock.
     workspace, newBlockDom);

   if(keepConnections){Blockly.BlocklyEditor.
     repositionNewlyGeneratedBlock(myBlock,newBlock);}
   return newBlock;
  } catch (error){
    if (!(error instanceof org.antlr.runtime.
       RecognitionException)){
      throw error;
    }
  }
};

 //if the tail block is connected to another block,
    replace the connections for the newly generated
    blocks
  // if(keepConnections && myBlock.outputConnection.
   targetConnection){
  //  var otherBlockConnection = myBlock.outputConnection
   .targetConnection;
  //  var newConnection = newBlock.outputConnection;
  //  myBlock.unplug(true,true);
  //  otherBlockConnection.connect(newConnection);
  // } else { //else move new generated blocks to
   location of myBlock

  //  // Move the duplicate next to the old block.
```

```
  //  var xy = myBlock.getRelativeToSurfaceXY();
  //  // if (Blockly.RTL) {
  //  //   xy.x -= Blockly.SNAP_RADIUS;
  //  // } else {
  //  //   xy.x += Blockly.SNAP_RADIUS;
  //  // }
  //  // xy.y += Blockly.SNAP_RADIUS * 2;
  //  newBlock.moveBy(xy.x, xy.y);
  // }

Blockly.Language.code_write_tail_stmt = {
  category: Blockly.LANG_CATEGORY_CODE,
  init: function(){
    this.setColour(Blockly.CODE_CATEGORY_HUE);
    this.appendDummyInput().appendTitle('TAIL stmt').
   appendTitle(
      new Blockly.FieldCodeBlockInput('[write TAIL stmt
   ]'), 'CODE');
    this.setPreviousStatement(true);
    this.setNextStatement(true);
    //this.setTooltip(Blockly.
     LANG_CODE_WRITE_TAIL_EXP_TOOLTIP);
    this.errors = [{name:"checkANTLRErrors"}];
    this.appendCollapsedInput().appendTitle('TAIL', '
     COLLAPSED_TEXT');
  },
  onchange: Blockly.WarningHandler.checkErrors,
  typeblock: [{ translatedName: Blockly.
   LANG_CODE_WRITE_TAIL_STMT }],
  prepareCollapsedText: function(){
    var textToDisplay = this.getTitleValue('CODE');
    if (textToDisplay.length > 8) //8 is a length of 5
     plus 3 dots
      textToDisplay = textToDisplay.substring(0, 5) +
     '...';
    this.getTitle_('COLLAPSED_TEXT').setText(
     textToDisplay, 'COLLAPSED_TEXT');
  },
  // codeStatementSequenceCustomContextMenu: function(
   options){
```

```
  //  var myBlock = this;
  //  var convertToBlocksOption = {text: "Convert
   Statement Sequence to Blocks"};
  //  if(!this.errorIcon && !this.disabled){
  //    convertToBlocksOption.enabled = true;
  //  } else{
  //    convertToBlocksOption.enabled = false;
  //  }
  //  convertToBlocksOption.callback = function(){
  //    Blockly.Language.code_write_tail_stmt.
   generateAIStatementBlock(myBlock,false,true);
  //    myBlock.dispose(true,false);
  //  };
  //  options.push(convertToBlocksOption);
  // },
  codeCustomContextMenu: function(options){
    var myBlock = this;
    var convertToBlocksOption = {text: "Convert to Blocks
   "};
   if(!this.errorIcon && !this.disabled){
      convertToBlocksOption.enabled = true;
    } else{
      convertToBlocksOption.enabled = false;
    }
    convertToBlocksOption.callback = function(){
      Blockly.Language.code_write_tail_stmt.
   generateAIStatementBlock(myBlock,true);
      myBlock.dispose(false,false); //we don't want to
   use the healstack option here
    };
    options.push(convertToBlocksOption);
  }
};

/**
* This function generates the new blocks equivalent to
  the tail expression block.
* @param {!Sting} tailText The TAIL text that needs to be
    parsed. This is not always
```

```
* the same as the current text in the block (see
   generator for this block).
* @param {!Blockly.Block} myBlock The TAIL expression
   block to convert into AI blocks.
* @param {!Boolean} keepConnections If true, the new
   blocks will replace the old blocks
* and keep the TAIL blocks connections (if it is
   connected to anything). If false, the
* connections will not be kept (this is for use in the
   generator, where we essentially
* want to generate invisible blocks, but instead we
   generate blocks elsewhere in the
* workspace and dispose of them when we're done using
   them).
* @return {!Blockly.Block} Returns the root block of the
   generated set of blocks.
*/

Blockly.Language.code_write_tail_stmt.
   generateAIStatementBlock = function(myBlock,
   keepConnections){
  try{
    var parser = Blockly.Language.code.parser(myBlock.
       getTitleValue('CODE'));
    var newBlockDom = parser.statement_block();
    var newBlock = Blockly.Xml.domToBlock_(myBlock.
       workspace, newBlockDom);

    if(keepConnections){
      Blockly.BlocklyEditor.repositionNewlyGeneratedBlock
         (myBlock,newBlock);
    }
    return newBlock;
  } catch (error){
    if (!(error instanceof org.antlr.runtime.
       RecognitionException)){
      throw error;
    }
  }
};
```

```
// var reposition = true; //variable which indicates
   whether the newly generated block(s) need to be
   repositioned or not

  // //if the tail block is connected to another block,
     replace the connections for the newly generated
     blocks
 // if(keepConnections){
 //   if(myBlock.previousConnection.targetConnection){
 //     //no need to reposition if the block is connected
     to a previous block
 //   //(this if statement will handle putting the block
    in the correct place)
 //     reposition = false;
 //     var previousBlockConnection = myBlock.
    previousConnection.targetConnection;
 //     var newConnection = newBlock.previousConnection;
 //     myBlock.unplug(false,false);
 //     previousBlockConnection.connect(newConnection);
 //   }
 //   if(myBlock.nextConnection.targetConnection){
 //     //if this block only has a next block but not a
    previous block, we do need to reposition
 //   //if this block did have a previous block, we don't
     need to reposition, but the above if
 //   //statement will take care of that, so we don't
    need to change the value of reposition here
 //     var nextBlockConnection = myBlock.nextConnection.
    targetConnection;
 //     var newConnection = newBlock.nextConnection;
 //     //disconnecting child of myBlock so that we can
    connect it to the newly generated block
 //     var nextBlock = myBlock.nextConnection.
    targetBlock();
 //     nextBlock.unplug(false,false);
 //     newConnection.connect(nextBlockConnection);
 //   }
 // }
 // if(reposition){
```

```
//    var xy = myBlock.getRelativeToSurfaceXY();
//   newBlock.moveBy(xy.x, xy.y);
// }
// // if(!newBlock.previousConnection.targetConnection)
    { //else move new generated blocks to location of
    myBlock
  // //  //this doesn't need to happen for a tail stmt
     block that has a previous block connected,
  // //  //but it does need to happen for a block that
     doesn't, but does have a next block,
  // //  //so we should just do this in all cases to be
     safe....

// //  // Move the duplicate next to the old block.
// //  //console.log("WHAT IS HAPPENING????!!!!!!");
// //  var xy = myBlock.getRelativeToSurfaceXY();
// //  newBlock.moveBy(xy.x, xy.y);
  // // }

Blockly.Language.code_write_tail_decl = {
  category: Blockly.LANG_CATEGORY_CODE,
  init: function(){
    this.setColour(Blockly.CODE_CATEGORY_HUE);
    this.appendDummyInput().appendTitle('TAIL decl').
   appendTitle(
      new Blockly.FieldCodeBlockInput('(write TAIL decl)
   '), 'CODE');
    //this.setTooltip(Blockly.
     LANG_CODE_WRITE_TAIL_EXP_TOOLTIP);
    this.errors = [{name:"checkANTLRErrors"}];
    this.appendCollapsedInput().appendTitle('TAIL', '
     COLLAPSED_TEXT');
  },
  onchange: Blockly.WarningHandler.checkErrors,
  typeblock: [{ translatedName: Blockly.
   LANG_CODE_WRITE_TAIL_DECL }],
  prepareCollapsedText: function(){
    var textToDisplay = this.getTitleValue('CODE');
    if (textToDisplay.length > 8) //8 is a length of 5
     plus 3 dots
```

```
      textToDisplay = textToDisplay.substring(0, 5) +
     '...';
    this.getTitle_('COLLAPSED_TEXT').setText(
     textToDisplay, 'COLLAPSED_TEXT');
  },
  // codeStatementSequenceCustomContextMenu: function(
   options){
  //  var myBlock = this;
  //  var convertToBlocksOption = {text: "Convert
   Statement Sequence to Blocks"};
  //  if(!this.errorIcon && !this.disabled){
  //    convertToBlocksOption.enabled = true;
  //  } else{
  //    convertToBlocksOption.enabled = false;
  //  }
  //  convertToBlocksOption.callback = function(){
  //    Blockly.Language.code_write_tail_stmt.
   generateAIStatementBlock(myBlock,false,true);
  //    myBlock.dispose(true,false);
  //  };
  //  options.push(convertToBlocksOption);
  // },
  codeCustomContextMenu: function(options){
    var myBlock = this;
    var convertToBlocksOption = {text: "Convert to Blocks
   "};
   if(!this.errorIcon && !this.disabled){
      convertToBlocksOption.enabled = true;
    } else{
      convertToBlocksOption.enabled = false;
    }
    convertToBlocksOption.callback = function(){
      Blockly.Language.code_write_tail_stmt.
   generateAITopLevelBlock(myBlock,true);
      myBlock.dispose(false,false); //we don't want to
   use the healstack option here
    };
    options.push(convertToBlocksOption);
  }
};
```

```
Blockly.Language.code_write_tail_stmt.
    generateAITopLevelBlock = function(myBlock,
    keepConnections){
  try{
    var parser = Blockly.Language.code.parser(myBlock.
        getTitleValue('CODE'));
    var newBlockDom = parser.top_level_block();
    console.log("DOM: " + Blockly.Xml.domToPrettyText(
        newBlockDom));
    var newBlock = Blockly.Xml.domToBlock_(myBlock.
        workspace, newBlockDom);

    if(keepConnections){
      Blockly.BlocklyEditor.repositionNewlyGeneratedBlock
          (myBlock,newBlock);
    }
    return newBlock;
  } catch(error){
    if (!(error instanceof org.antlr.runtime.
        RecognitionException)){
      throw error;
    }
  }
};
```

YAIL Code Generator

```
Blockly.Yail = Blockly.Generator.get('Yail');
var oldTailText;
var expCode;
Blockly.Yail.code_write_tail_exp = function() {
 var tailText = this.getTitleValue('CODE');
 if(!oldTailText){
  oldTailText = tailText;
 }else{
  if(tailText === oldTailText){
    return expCode;
  }
  else{
    oldTailText = tailText;
  }
 }
 var canParse = this.getTitle_('CODE').textDoneChanging;
 if(!canParse){ return expCode; }
 newBlocks = Blockly.Language.code_write_tail_exp.
   generateAIExpressionBlock(this, false);
 var code = Blockly.Yail.blockToCode(newBlocks);
 newBlocks.dispose(true, false);
 expCode = code;
 return code;
};
```

Blocks to Text Converter

```
//Blocks -> Text Converter

Blockly.BlocksToTextConverter = {};

Blockly.BlocksToTextConverter.expressionBlocks = ["
   controls_choose","logic_operation","logic_negate","
   logic_compare","math_compare",
"math_add", "math_multiply", "math_subtract", "
   math_division", "math_power", "math_divide", "
   math_single", "math_abs", "math_neg",
"math_round", "math_ceiling", "math_floor", "math_trig",
   "math_cos", "math_tan", "local_declaration_expression
   ", "lexical_variable_get", "color_black",
"color_blue", "color_white", "color_magenta", "color_red
   ", "color_light_gray", "color_pink", "color_gray", "
   color_orange",
"color_dark_gray", "color_yellow", "color_green", "
   color_cyan", "color_make_color", "lists_create_with",
   "math_number", "text",
"logic_boolean", "logic_false", "component_set_get", "
   code_write_tail_exp"];

Blockly.BlocksToTextConverter.statementBlocks = ["
   controls_if","lexical_variable_set","
   code_write_tail_stmt", "local_declaration_statement",
   "component_set_get"];

Blockly.BlocksToTextConverter.topLevelBlocks = ["
   global_declaration","procedures_defnoreturn","
```

```
   procedures_defreturn","component_event", , "
   procedures_callreturn", ];

Blockly.BlocksToTextConverter.tailText;

Blockly.BlocksToTextConverter.type = "";

Blockly.BlocksToTextConverter.
   getImmediateChildrenByTagName = function(element,
   tagName){
  var elementsWithTag = [];
  var current = element.firstElementChild;
  while(!!current){
    if(current.nodeName.toLowerCase() === tagName.
       toLowerCase()){
      elementsWithTag.push(current);
    }
    current = current.nextElementSibling;
  }
  return elementsWithTag;
}

Blockly.BlocksToTextConverter.checkEmptySockets =
   function(block){
  if(Blockly.WarningHandler.checkEmptySockets.call(block)
     ){
    return true;
  } else{ //check decendants of block
    var children = block.getChildren();
    for (var i = 0; i<children.length; i++){
      if(Blockly.WarningHandler.checkEmptySockets.call(
         children[i])){
        return true;
      }
    }
    return false;
  }
}
```

```
Blockly.BlocksToTextConverter.blockToTAIL = function(
   block){
  var root = Blockly.Xml.blockToDom_(block);
  Blockly.BlocksToTextConverter.tailText = "";
  //translate root node
  Blockly.BlocksToTextConverter.translateBlock(root);

  var tailBlockDom = document.createElement("block");
  tailBlockDom.setAttribute("type", Blockly.
     BlocksToTextConverter.type);
  var title = document.createElement("title");
  title.setAttribute("name","CODE");
  title.innerHTML = Blockly.BlocksToTextConverter.
     tailText;
  tailBlockDom.appendChild(title);

  var tailBlock = Blockly.Xml.domToBlock_(block.workspace
     ,tailBlockDom);
  return tailBlock;
};

Blockly.BlocksToTextConverter.translateBlock = function(
   element){
  var tagName = element.nodeName.toLowerCase();
  if(tagName === "block"){
    var type = element.getAttribute("type");
    if(type === "component_set_get"){
      var mutations = Blockly.BlocksToTextConverter.
         getImmediateChildrenByTagName(element, "mutation
         ");
      var mutation;
      if(mutations.length !== 0){
        mutation = mutations[0];
      }
      if(mutation.getAttribute("set_or_get") === "get"){
        Blockly.BlocksToTextConverter.type = "
           code_write_tail_exp";
        Blockly.BlocksToTextConverter.
           translateExpressionBlock(element);
      } else{
```

```
        Blockly.BlocksToTextConverter.type = "
            code_write_tail_stmt";
        Blockly.BlocksToTextConverter.
            translateStatementBlock(element);
      }

    }
    else{
      if(Blockly.BlocksToTextConverter.expressionBlocks.
         indexOf(type) !== -1){
      //element is an expression block
      Blockly.BlocksToTextConverter.type = "
       code_write_tail_exp";
      Blockly.BlocksToTextConverter.
       translateExpressionBlock(element);
     } else if(Blockly.BlocksToTextConverter.
       statementBlocks.indexOf(type) !== -1){
       //element is a statement block
       Blockly.BlocksToTextConverter.type = "
         code_write_tail_stmt";
       Blockly.BlocksToTextConverter.
         translateStatementBlock(element);
      } else if(Blockly.BlocksToTextConverter.
         topLevelBlocks.indexOf(type) !== -1){
       //element is a top level block
       Blockly.BlocksToTextConverter.type = "
         code_write_tail_decl";
       Blockly.BlocksToTextConverter.
         translateTopLevelBlock(element);
      } else{
      //do nothing
      console.log("I'm getting not getting an expression,
        statement, or top level block.")
     }
    }
  }
}

/**************Expression Blocks**************/
```

```
Blockly.BlocksToTextConverter.translateExpressionBlock =
   function(element){
  var elementType = element.getAttribute("type");
  //expression block
  Blockly.BlocksToTextConverter.tailText += '{';
  Blockly.BlocksToTextConverter["translate_" +
     elementType].call(this, element);
  Blockly.BlocksToTextConverter.tailText += '}';
}

Blockly.BlocksToTextConverter.translate_controls_choose =
    function(element){
  var children = element.children;
  var ifBlock = children.namedItem("TEST").
     firstElementChild;
  var thenBlock = children.namedItem("THENRETURN").
     firstElementChild;
  var elseBlock = children.namedItem("ELSERETURN").
     firstElementChild;

  Blockly.BlocksToTextConverter.tailText += 'if ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     ifBlock);
  Blockly.BlocksToTextConverter.tailText += ' then: ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     thenBlock);
  Blockly.BlocksToTextConverter.tailText += ' else: ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     elseBlock);
}

Blockly.BlocksToTextConverter.translate_logic_operation =
    function(element){
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;
  var op = children.namedItem("OP").innerHTML.toLowerCase
     ();
```

```
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' ' + op + '
     ' ;
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}

Blockly.BlocksToTextConverter.translate_logic_negate =
   function(element){
  var children = element.children;
  var child = children.namedItem("BOOL");

  Blockly.BlocksToTextConverter.tailText += "not";

  Blockly.BlocksToTextConverter.translateExpressionBlock(
     child);
}

Blockly.BlocksToTextConverter.translate_logic_compare =
   function(element){
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;
  var op = children.namedItem("OP").innerHTML;

  if(op === "EQ"){
    op = "equals";
  } else{
    op = "not_equals";
  }

  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' ' + op + '
     ' ;
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}
```

```
Blockly.BlocksToTextConverter.translate_math_compare =
   function(element){
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;
  var op = children.namedItem("OP").innerHTML;

  if(op === "EQ"){
    op = '=';
  } else if(op === "NEQ"){
    op = '!=';
  } else if(op === "LT"){
    op = '<';
  } else if(op === "GT"){
    op = '>';
  } else if(op === "LTE"){
    op = '<=';
  } else {
    op = '>='
  }

  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' ' + op + '
     ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}

Blockly.BlocksToTextConverter.translate_math_add =
   function(element){
  var children = element.children;

  var mutationBlock;
  for(var i=0; i<children.length; i++){
    var child = children.item(i);
    if(child.nodeName.toLowerCase() === "mutation"){
      mutationBlock = child;
    }
  }
```

```
  var numItems = (!!mutationBlock) ? parseInt(
     mutationBlock.getAttribute("items")) : 0;

  var value = children.namedItem("NUM0").
     firstElementChild;
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     value);

  for (var i = 1; i<numItems; i++){
    var value = children.namedItem("NUM"+i).
       firstElementChild;

    Blockly.BlocksToTextConverter.tailText += ' + ';
    Blockly.BlocksToTextConverter.
       translateExpressionBlock(value);
  }
}

Blockly.BlocksToTextConverter.translate_math_multiply =
   function (element) {
  var children = element.children;

  var mutationBlock;
  for(var i=0; i<children.length; i++){
    var child = children.item(i);
    if(child.nodeName.toLowerCase() === "mutation"){
      mutationBlock = child;
    }
  }

  var numItems = (!!mutationBlock) ? parseInt(
     mutationBlock.getAttribute("items")) : 0;

  var value = children.namedItem("NUM0").
     firstElementChild;
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     value);

  for (var i = 1; i<numItems; i++){
```

```
    var value = children.namedItem("NUM"+i).
       firstElementChild;

    Blockly.BlocksToTextConverter.tailText += ' * ';
    Blockly.BlocksToTextConverter.
       translateExpressionBlock(value);
  }
}

Blockly.BlocksToTextConverter.translate_math_subtract =
   function (element) {
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;

  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' - ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}

Blockly.BlocksToTextConverter.translate_math_division =
   function (element) {
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;

  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' / ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}

Blockly.BlocksToTextConverter.translate_math_power =
   function (element) {
  var children = element.children;
  var aBlock = children.namedItem("A").firstElementChild;
  var bBlock = children.namedItem("B").firstElementChild;
```

```
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     aBlock);
  Blockly.BlocksToTextConverter.tailText += ' ^ ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     bBlock);
}

Blockly.BlocksToTextConverter.translate_math_divide =
   function(element){
  var children = element.children;
  var dividend = children.namedItem("DIVIDEND").
     firstElementChild;
  var divisor = children.namedItem("DIVISOR").
     firstElementChild;

  var op = children.namedItem("OP").innerHTML.toLowerCase
     () + '_of';

  Blockly.BlocksToTextConverter.tailText += op + ' ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     dividend);
  Blockly.BlocksToTextConverter.tailText += ' / ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     divisor);
}

Blockly.BlocksToTextConverter.translate_math_single =
   function(element){
  var children = element.children;
  var expr = children.namedItem("NUM").firstElementChild;

  var op = children.namedItem("OP").innerHTML.toLowerCase
     ();
  if (op === "root") {
    op = 'sqrt';
  } else if(op === "neg"){
    op = '-';
  } else if(op === "ln"){
    op = 'log';
```

```
  } else if(op === "exp"){
    op = 'e^';
  } //no else case, op should remain the same for abs,
     round, ceiling, and floor

  Blockly.BlocksToTextConverter.tailText += op + ' ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     expr);
}

Blockly.BlocksToTextConverter.translate_math_abs =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_single(
     element);
}

Blockly.BlocksToTextConverter.translate_math_neg =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_single(
     element);
}

Blockly.BlocksToTextConverter.translate_math_round =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_single(
     element);
}

Blockly.BlocksToTextConverter.translate_math_ceiling =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_single(
     element);
}

Blockly.BlocksToTextConverter.translate_math_floor =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_single(
     element);
}
```

```
Blockly.BlocksToTextConverter.translate_math_trig =
   function(element){
  var children = element.children;
  var expr = children.namedItem("NUM").firstElementChild;

  var op = children.namedItem("OP").innerHTML.toLowerCase
     ();

  Blockly.BlocksToTextConverter.tailText += op + ' ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     expr);
}

Blockly.BlocksToTextConverter.translate_math_cos =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_trig(
     element);
}

Blockly.BlocksToTextConverter.translate_math_tan =
   function(element){
  Blockly.BlocksToTextConverter.translate_math_trig(
     element);
}

Blockly.BlocksToTextConverter.
   translate_local_declaration_expression = function(
   element){
  Blockly.BlocksToTextConverter.tailText += '
     initialize_local';
  var children = element.children;

  var mutationBlock;
  for(var i=0; i<children.length; i++){
    var child = children.item(i);
    if(child.nodeName.toLowerCase() === "mutation"){
      mutationBlock = child;
    }
  }
```

```
  var numVars = (!!mutationBlock) ? mutationBlock.
     childElementCount : 0;

  for(var i=0; i<numVars; i++){
    Blockly.BlocksToTextConverter.tailText += ' <';
    var varName = children.namedItem("VAR"+i).innerHTML;
    var exprBlock = children.namedItem("DECL"+i).
       firstElementChild;
    Blockly.BlocksToTextConverter.tailText += varName +
       '> to: ';
    Blockly.BlocksToTextConverter.
       translateExpressionBlock(exprBlock);
  }

  Blockly.BlocksToTextConverter.tailText += ' in: ';
  var returnExprBlock = children.namedItem("RETURN").
     firstElementChild;
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     returnExprBlock);
}

Blockly.BlocksToTextConverter.
   translate_lexical_variable_get = function(element){
  Blockly.BlocksToTextConverter.tailText += 'get ';
  var title = element.firstElementChild;
  Blockly.BlocksToTextConverter.tailText += title.
     innerHTML;
}

Blockly.BlocksToTextConverter.translate_color_black =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#000000"){
    Blockly.BlocksToTextConverter.tailText += 'color
       black';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
```

```
}

Blockly.BlocksToTextConverter.translate_color_blue =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#0000ff"){
    Blockly.BlocksToTextConverter.tailText += 'color blue
       ';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_white =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ffffff"){
    Blockly.BlocksToTextConverter.tailText += 'color
       white';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_magenta =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ff00ff"){
    Blockly.BlocksToTextConverter.tailText += 'color
       magenta';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}
```

```
Blockly.BlocksToTextConverter.translate_color_red =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ff0000"){
    Blockly.BlocksToTextConverter.tailText += 'color red
       ';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_light_gray
   = function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#cccccc"){
    Blockly.BlocksToTextConverter.tailText += 'color
       light_gray';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_pink =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ffafaf"){
    Blockly.BlocksToTextConverter.tailText += 'color pink
       ';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}
```

```
Blockly.BlocksToTextConverter.translate_color_gray =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#888888"){
    Blockly.BlocksToTextConverter.tailText += 'color gray
       ';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_orange =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ffc800"){
    Blockly.BlocksToTextConverter.tailText += 'color
       orange';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_dark_gray =
    function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#444444"){
    Blockly.BlocksToTextConverter.tailText += 'color
       dark_gray';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}
```

```
Blockly.BlocksToTextConverter.translate_color_yellow =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#ffff00"){
    Blockly.BlocksToTextConverter.tailText += 'color
       yellow';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_green =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#00ff00"){
    Blockly.BlocksToTextConverter.tailText += 'color
       green';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}

Blockly.BlocksToTextConverter.translate_color_cyan =
   function(element){
  var hexValue = element.firstElementChild.innerHTML.
     toLowerCase();
  if (hexValue === "#00ffff"){
    Blockly.BlocksToTextConverter.tailText += 'color cyan
       ';
  } else{
    Blockly.BlocksToTextConverter.tailText += 'color ' +
       hexValue;
  }
}
```

```
Blockly.BlocksToTextConverter.translate_color_make_color
    = function(element){
  var children = element.children;
  var expr = children.namedItem("COLORLIST").
     firstElementChild;

  Blockly.BlocksToTextConverter.tailText += 'make_color
     ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     expr);
}

Blockly.BlocksToTextConverter.translate_lists_create_with
    = function(element){
  Blockly.BlocksToTextConverter.tailText += 'list';
  var children = element.children;
  var mutation = parseInt(element.firstElementChild.
     getAttribute("items"));
  for(var i=0; i<mutation; i++){
    var valBlock = children.namedItem("ADD" + i);
    if(valBlock){
      var block = valBlock.firstElementChild;
      Blockly.BlocksToTextConverter.tailText += ' ';
      Blockly.BlocksToTextConverter.
         translateExpressionBlock(block);
    }
  }
}

Blockly.BlocksToTextConverter.translate_math_number =
   function(element){
  Blockly.BlocksToTextConverter.tailText += element.
     firstElementChild.innerHTML;
}

Blockly.BlocksToTextConverter.translate_text = function(
   element){
  Blockly.BlocksToTextConverter.tailText += '"' + element
     .firstElementChild.innerHTML + '"';
}
```

```
Blockly.BlocksToTextConverter.translate_logic_boolean =
   function(element){
  Blockly.BlocksToTextConverter.tailText += element.
     firstElementChild.innerHTML.toLowerCase();
}

Blockly.BlocksToTextConverter.translate_logic_false =
   function(element){
  Blockly.BlocksToTextConverter.translate_logic_boolean(
     element);
}

Blockly.BlocksToTextConverter.
   translate_code_write_tail_exp = function(element){
  Blockly.BlocksToTextConverter.tailText += 'TAIL_exp';

  var child = element.firstElementChild;
  var expression_text = child.innerHTML;

  Blockly.BlocksToTextConverter.tailText += ' ' +
     expression_text;
}

/****************Statement Blocks*****************/

Blockly.BlocksToTextConverter.translateStatementBlock =
   function(element){
  var elementType = element.getAttribute("type");
  //expression block
  Blockly.BlocksToTextConverter.tailText += '[';
  Blockly.BlocksToTextConverter["translate_" +
     elementType].call(this, element);
  Blockly.BlocksToTextConverter.tailText += ']';
}

Blockly.BlocksToTextConverter.translateStatementSequence
   = function(element){
//  var elementType = element.getAttribute("type");
```

```
  if(!!element){ //if element is not null (make sure it's
      not the empty statement)
    //var children = element.children;
   var curBlock = element.firstElementChild; //this
     should be the first statement block
   Blockly.BlocksToTextConverter.translateStatementBlock(
     curBlock);
   var next_arr = Blockly.BlocksToTextConverter.
     getImmediateChildrenByTagName(curBlock,"next");
   if(next_arr.length === 1){
    Blockly.BlocksToTextConverter.
     translateStatementSequence(next_arr[0]);
   }
  }
}
// Blockly.BlocksToTextConverter.
   translate_controls_choose = function(element){
//  var children = element.children;
//  var ifBlock = children.namedItem("TEST").
   firstElementChild;
//  var thenBlock = children.namedItem("THENRETURN").
   firstElementChild;
//  var elseBlock = children.namedItem("ELSERETURN").
   firstElementChild;

//  Blockly.BlocksToTextConverter.tailText += 'if ';
//  Blockly.BlocksToTextConverter.
   translateExpressionBlock(ifBlock);
//  Blockly.BlocksToTextConverter.tailText += ' then: ';
//  Blockly.BlocksToTextConverter.
   translateExpressionBlock(thenBlock);
//  Blockly.BlocksToTextConverter.tailText += ' else: ';
//  Blockly.BlocksToTextConverter.
   translateExpressionBlock(elseBlock);
// }

Blockly.BlocksToTextConverter.translate_controls_if =
   function(element){
  var children = element.children;
```

```
var mutations = Blockly.BlocksToTextConverter.
   getImmediateChildrenByTagName(element,"mutation");

var ifBlock = children.namedItem("IF0").
   firstElementChild;
var thenSuite = children.namedItem("DO0");

Blockly.BlocksToTextConverter.tailText += 'if ';
Blockly.BlocksToTextConverter.translateExpressionBlock(
   ifBlock);
Blockly.BlocksToTextConverter.tailText += ' then: ';
Blockly.BlocksToTextConverter.
   translateStatementSequence(thenSuite);

//if this block is in a seq of statements which have
   mutation sub-elements,
//then mutations.length will be > 1 ------ ignore this
   comment, bug fixed by Karishma 03.21.14
if (mutations.length === 1){
  var elseIfCount = parseInt(mutations[0].getAttribute
     ("elseif"));
  var elseCount = parseInt(mutations[0].getAttribute("
     else"));
  for(var i=1; i<=elseIfCount; i++){
    var elseIfBlock = children.namedItem("IF"+i).
       firstElementChild;
    var thenBlocks = children.namedItem("DO"+i);

    Blockly.BlocksToTextConverter.tailText += ' else_if
       : ';
    Blockly.BlocksToTextConverter.
       translateExpressionBlock(elseIfBlock);
    Blockly.BlocksToTextConverter.tailText += ' then:
       ';
    Blockly.BlocksToTextConverter.
       translateStatementSequence(thenBlocks);
  }

  for(var i=1; i<=elseCount; i++){
    var elseBlocks = children.namedItem("ELSE");
```

```
      Blockly.BlocksToTextConverter.tailText += ' else:
         ';
      Blockly.BlocksToTextConverter.
         translateStatementSequence(elseBlocks);
    }
  } else{
    console.log("Blocks To Text Converter -
       translate_controls_if: something is wrong with
       mutations");
  }
}

Blockly.BlocksToTextConverter.
   translate_lexical_variable_set = function(element){
  var children = element.children;
  var varName = children.namedItem("VAR").innerHTML;
  var value = children.namedItem("VALUE").
     firstElementChild;

  Blockly.BlocksToTextConverter.tailText += 'set ' +
     varName + ' to: ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     value);
}

Blockly.BlocksToTextConverter.
   translate_code_write_tail_stmt = function(element){
  Blockly.BlocksToTextConverter.tailText += 'TAIL_stmt';

  var child = element.firstElementChild;
  var stmt_text = child.innerHTML;

  Blockly.BlocksToTextConverter.tailText += ' ' +
     stmt_text;
}


/************Top Level Blocks*****************/
```

```
Blockly.BlocksToTextConverter.translateTopLevelBlock =
   function(element){
  var elementType = element.getAttribute("type");
  //expression block
  Blockly.BlocksToTextConverter.tailText += '(';
  Blockly.BlocksToTextConverter["translate_" +
     elementType].call(this, element);
  Blockly.BlocksToTextConverter.tailText += ')';
}

Blockly.BlocksToTextConverter.
   translate_global_declaration = function(element){
  var children = element.children;
  var varName = children.namedItem("NAME").innerHTML;
  var value = children.namedItem("VALUE").
     firstElementChild;

  Blockly.BlocksToTextConverter.tailText += '
     initialize_global <' + varName + '> to: ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     value);
}

Blockly.BlocksToTextConverter.
   translate_procedures_defnoreturn = function(element){
  var children = element.children;
  var procName = children.namedItem("NAME").innerHTML;

  Blockly.BlocksToTextConverter.tailText += 'to <' +
     procName + '> ';

  var numParams = 0;
  var mutations = Blockly.BlocksToTextConverter.
     getImmediateChildrenByTagName(element,"mutation");
  if(mutations.length === 1){
    numParams = mutations[0].childElementCount;
  }else{
    console.log("Blocks To Text Converter -
       translate_procedures_defnoreturn: something is
```

```
        wrong with mutations");
  }

  for(var i=0; i<numParams; i++){
    var argName = children.namedItem("VAR"+i).innerHTML;
    Blockly.BlocksToTextConverter.tailText += '<' +
        argName + '> ';
  }

  var suite = children.namedItem("STACK");
  Blockly.BlocksToTextConverter.tailText += 'do: ';
  Blockly.BlocksToTextConverter.
     translateStatementSequence(suite);
}

Blockly.BlocksToTextConverter.
   translate_procedures_defreturn = function(element){
  var children = element.children;
  var procName = children.namedItem("NAME").innerHTML;

  Blockly.BlocksToTextConverter.tailText += 'to <' +
     procName + '> ';

  var numParams = 0;
  var mutations = Blockly.BlocksToTextConverter.
     getImmediateChildrenByTagName(element,"mutation");
  if(mutations.length === 1){
    numParams = mutations[0].childElementCount;
  }else{
    console.log("Blocks To Text Converter -
        translate_procedures_defnoreturn: something is
        wrong with mutations");
  }

  for(var i=0; i<numParams; i++){
    var argName = children.namedItem("VAR"+i).innerHTML;
    Blockly.BlocksToTextConverter.tailText += '<' +
        argName + '> ';
  }
```

```
  var retVal = children.namedItem("RETURN").
     firstElementChild;
  Blockly.BlocksToTextConverter.tailText += 'result: ';
  Blockly.BlocksToTextConverter.translateExpressionBlock(
     retVal);
}

Blockly.BlocksToTextConverter.translate_component_event =
    function(element){
  var children = element.children;
  var componentName = children.namedItem("
     COMPONENT_SELECTOR").innerHTML;
  var mutations = Blockly.BlocksToTextConverter.
     getImmediateChildrenByTagName(element,"mutation");
  if(mutations.length !== 1){
    console.log("BlocksToTextCovereter -
       translate_component_event: Something is wrong with
        this block.")
  }
  var eventName = mutations[0].getAttribute("event_name")
     ;

  Blockly.BlocksToTextConverter.tailText += 'when ' +
     componentName + '.' + eventName + ' do: ';

  var suite = children.namedItem("DO");

  Blockly.BlocksToTextConverter.
     translateStatementSequence(suite);
}

//Hacky....fix later!!
Blockly.BlocksToTextConverter.translate_component_set_get
    = function(element){
  var children = element.children;
  var componentName = children.namedItem("
     COMPONENT_SELECTOR").innerHTML;
  var propName = children.namedItem("PROP").innerHTML;
```

```
  var mutations = Blockly.BlocksToTextConverter.
     getImmediateChildrenByTagName(element,"mutation");
  if(mutations.length !== 1){
    console.log("BlocksToTextCovereter -
       translate_component_event: Something is wrong with
        this block.")
  }
  var setGet = mutations[0].getAttribute("set_or_get");

  if(setGet === "set"){
    var valBlock = children.namedItem("VALUE").
       firstElementChild;
    Blockly.BlocksToTextConverter.tailText += 'set ' +
       componentName + '.' + propName + ' to: ';
    Blockly.BlocksToTextConverter.
       translateExpressionBlock(valBlock);
  } else{
    Blockly.BlocksToTextConverter.tailText +=
       componentName + '.' + propName;
  }
}
```