Typeblocking: Keyboard Integration with Block Programming in StarLogo Nova

by

Terrance Liang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
May 24, 2019

Certified by . . . .
Eric Klopfer
Director, MIT Scheller Teacher Education Program
Thesis Supervisor
May 24, 2019

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee
May 24, 2019

Typeblocking: Keyboard Integration with Block Programming in StarLogo Nova

by

Terrance Liang

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Block programming acts as an elementary introduction to computer programming for children and teenagers. There are clear benefits to using the graphical representation of code in the form of blocks, but the dragging and dropping mechanism may begin to feel tedious for users who are more experienced with the system. In addition, the large library of blocks may seem overwhelming to novices who have just begun familiarizing themselves with the language. This thesis will cover typeblocking, a feature that integrates the keyboard with block programming and translates user text input to blocks, and how it improves the experience of coding in a block programming environment like StarLogo Nova.

Thesis Supervisor: Eric Klopfer
Title: Director, MIT Scheller Teacher Education Program

Acknowledgments

I would first like to thank my direct supervisor, Daniel Wendel, the lead designer and developer of StarLogo Nova. He has been a great advisor and mentor throughout my Masters program and provided tremendous support in my work on typeblocking.

I must also acknowledge Lisa Stump, a former developer on the StarLogo team, who was an incredible help on the technical side of this project and left behind incredible code for me to use. I must also thank the rest of the StarLogo team for being so enjoyable to work with, making this past year extremely fun.

To my research advisor, Eric Klopfer, and the entire staff at Scheller Teacher Education Program, thank you for providing me such an incredible community to be a part of.

I'm also extremely grateful to all my friends and coworkers - Judy Perry, Emma Anderson, Jeffrey Zhang, Chris Womack, Kevin Fang - who have directly helped me by providing user feedback on my project.

Finally, I would like to thank my family for their everlasting support and encouragement.

Contents

1 Introduction **13**

1.1 StarLogo Nova . . . 14

1.2 Block Programming and its Shortcomings . . . 15

1.3 Typeblocking as a Solution . . . 15

2 Related Work **17**

2.1 StarLogo TNG . . . 17

2.2 App Inventor . . . 19

2.3 Gameblox . . . 20

2.3.1 Text Editors and Computer Programming IDEs . . . 22

3 Definitions **23**

4 Challenges **27**

4.1 Context . . . 27

4.2 Translation . . . 28

4.3 User Interface . . . 29

5 Context Awareness **33**

5.1 Socket Types . . . 33

5.2 Context Object . . . 33

6 Translation Logic **35**

6.1 Base Approach . . . 35

6.2 Drop-downs 36
6.2.1 Argument 36
6.2.2 Within Block Insertion 37
6.3 String Sockets 38
6.4 String Arguments 38
6.5 Nested Blocks 39

7 User Interface **41**

7.1 Workspace Navigation 41
7.1.1 Cursor and Focus Block 41
7.2 Navigation 42
7.2.1 Internal Inputs and Drop-downs 46
7.2.2 Deletion 46
7.3 String to Block Translation 47
7.3.1 Input Element 47
7.3.2 Block Options 47
7.3.3 Sorting Logic 48
7.3.4 Cursor Location 49

8 User Testing and Reception **51**

8.1 Informal User Test Feedback and Changes Made 51
8.2 Formal User Test Plan 52
8.2.1 Observations 54
8.3 Reception 55

9 Future Work **57**

9.1 Shortcuts 57
9.2 Improved Nested Block Logic 57
9.3 Format of Block Options 58
9.4 Redesign Cursor 58
9.5 Incorporate Edit Commands 59

9.6 Integrate with Type Safety 59
9.7 More Extensive User Testing 60

10 Conclusion 61

List of Figures

1-1 StarLogo Nova [11] 14

2-1 StarLogo TNGs Typeblocking 18
2-2 App Inventors Typeblocking [8] 20
2-3 Gameblox Block Editor [4] 21
2-4 Gameblox Text Editor [4] 21
2-5 Sublime Text Auto-complete [1] 22

3-1 Example Widgets (push button, toggle button, data box) 23
3-2 Create-Each-Do is a command block, which takes in three arguments (number of agents, breed of agents, and commands that each agent will execute). 24
3-3 Blocks drawers are on the left of the workspace 25
3-4 Example of each socket type (command, string, and boolean) 25
3-5 In this block stack, the Create-Each-Do block is the focus block, and the cursor is over the number-of-breeds socket 26

4-1 The set-trait block has different drop-down argument values depending on the context, which needs to be aware of what the valid breed is for the focus block and what unique traits it has. 28
4-2 The context needs to be aware of custom variables created earlier in the block stack. 28
4-3 Expected cursor behavior for key presses 30

6-1 Block results for base approach with user input “create” 36

6-2 Typeblocking shows drop-down options instead of block options . . . 36
6-3 Examples of drop-down assignments with pre-filled values 37
6-4 The literal string will be an option for string socket type sockets . . . 38
6-5 String argument parsing 39
6-6 Expected nested block output for "set my size to 5 + random decimal" 39
6-7 Translation logic can currently only handle one level of nesting 40
6-8 Block results for current implementation with user input "create" . . 40

7-1 Example behavior of Up key pressed multiple times in sequence . . . 42
7-2 Example behavior of Down key pressed multiple times in sequence . . 43
7-3 Example behavior of Left key pressed multiple times in sequence . . . 44
7-4 Example behavior of Right key pressed multiple times in sequence . . 45
7-5 The user can enter an arguments' internal input 46
7-6 Prioritize "clock" over "set clock to [ ]" when user input is "cl" 48
7-7 Prioritize "create [ ] [ ](s)" and "create [ ] [ ](s)each do [ ]" before "create [ ] [Hero](s)" and "create [ ] [Hero](s) each do [ ]". Prioritize "create [ ] [ ](s)" before "create [ ] [ ](s) each do [ ]". 49

8-1 User test example that users were asked to code 53

9-1 Examples of various ideas for block option formats 58

Chapter 1

Introduction

Computer Science has been increasing its presence in the work industry over the past few years, with computing jobs as the number one source for new wages. In addition to job opportunities, computer science teaches important concepts in computational thinking, which is beneficial in many industries. As a result, there has been a push to integrate computer science into younger education to allow early exposure to the field [3]. One particular method of introducing it is through block programming environments like Scratch [10], App Inventor[8], and StarLogo Nova[11]. These environments are generally directed towards younger children and teenagers, intended to prepare them for more advanced programming environments. Users can build programs, such as games or simulations, through the use of code blocks, which can be manipulated and connected to one another via mouse-based dragging and dropping. Unlike traditional text editor environments, this provides visual guidance and prevents syntax errors that beginners may make. It helps shift the focus of users towards more central computing tasks like algorithm design and pattern recognition. With use over time, students should be able to naturally recognize syntax and commands as they are familiarizing themselves with the environment [6].

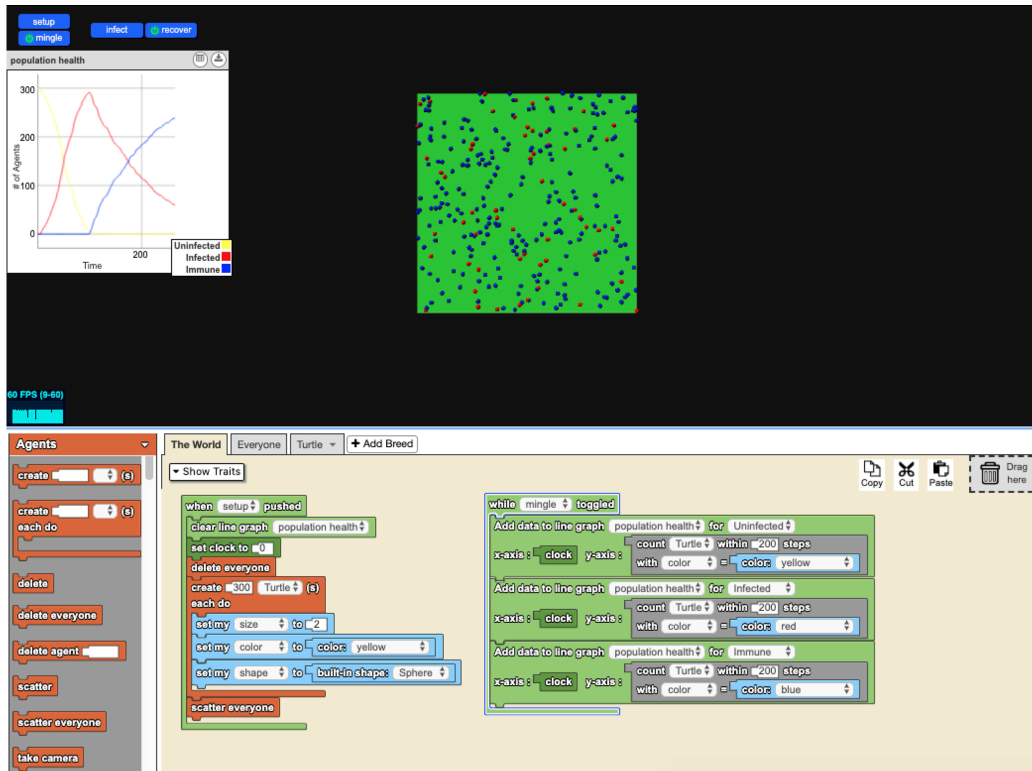


Figure 1-1: StarLogo Nova [11]

1.1 StarLogo Nova

This thesis will be an extension of an existing block programming language, StarLogo Nova, developed at the MIT Scheller Teacher Education Program [11]. StarLogo Nova builds on the legacy of the Logo programming language. As a graphical programming language, it uses puzzle-shaped blocks that indicates to users what pieces of logic can be connected to each other. Thanks to the robust 3D renderer that powers Spaceland, the space where students' programmed agents exist, the programmer can clearly visualize their complex programs. Another great benefit of StarLogo Nova is that it is entirely web based, unlike its predecessor StarLogo TNG. All of these features of StarLogo Nova allow it to provide to students the opportunity to "create games and simulations to study diverse concepts in science and math" [12]. As a block programming environment targeted towards secondary students, StarLogo Nova is a great candidate where typeblocking can be beneficial for both expert and new users.

1.2 Block Programming and its Shortcomings

One of the greatest benefits of block programming is the abstraction that it provides to the complex details of computer science. The details of code are abstracted into blocks for users to connect to one another. This allows younger kids to develop computational thinking and problem solving skills without the need to learn the complex syntax that normally comes with learning new programming languages [9].

Like with most programming languages, there is still a learning curve with block programming. Instead of remembering syntax, users are now tasked with recognizing the many blocks that exist, which may be overwhelming for a novice. To familiarize themselves with the environment and to understand what blocks exist and what they do, new users must learn through practice and constant use of the system.

Abstracting code into blocks may be helpful in visualizing code, but they heavily focus on dragging blocks onto a workspace and connecting them afterwards. For a novice user, this is extremely helpful in mapping out their thought process (much like building something with wooden blocks). However, once users begin to familiarize themselves with the system and the language, they can begin to find the drag-and-drop mechanism to be tedious and repetitive.

1.3 Typeblocking as a Solution

This thesis proposes *typeblocking* as a solution to both of the problems mentioned above. Typeblocking is the integration of the keyboard with the StarLogo Nova system, providing the functionality to take keyboard input to navigate and insert blocks. By being able to translate text input to blocks, it should provide a quick way for users to find and insert the blocks that they are looking for. This can act as a search function for new users who are familiarizing themselves with what blocks exist. For more experienced users, typeblocking should streamline the way for blocks to appear on

their workspace due to the improvements in speed and flow with the use of a keyboard [7].

To ensure that typeblocking improves work flow, we want to minimize the amount of distractions while programming. To accomplish this, we want to avoid having the user constantly switch between the mouse and keyboard and we want to keep their attention directed at the blocks. The user interface must also feel natural so that users can easily learn how to navigate around their workspace, while writing and editing blocks with the keyboard. The translation interface's logic should align with that of a typical user - the most appropriate block options should appear first and the selection mechanism should be obvious. We also want to expand the search algorithm to support auto-completion, with the added logic of being able to take a user's input as arguments or nested blocks and have them populate blocks appropriately.

To encapsulate all these ideas, this thesis aims to design typeblocking to be a system with three core features: context awareness of blocks to understand what blocks are appropriate, robust parsing logic for text to block translation, and a smooth user interface for navigation and user input. With all these components, typeblocking can provide a faster coding experience compared to the drag-and-drop mechanism that block programming traditionally relies on.

Chapter 2

Related Work

Typeblocking's main three components are context awareness, translation logic, and user interface. The designs and features of these components are inspired by existing programs and other groups' interpretations of typeblocking.

2.1 StarLogo TNG

As mentioned, StarLogo Nova succeeds StarLogo TNG, an offline version of the system [13]. Corey McCaffrey, a former graduate student at MIT, implemented typeblocking in StarLogo TNG for his Masters thesis in May of 2006. His thesis goes in depth on the benefits of keyboard usage in graphical programming languages [7]. In his implementation, McCaffrey extracted common techniques from text processing environments to apply typeblocking to StarLogo TNG. McCaffrey's implementation included a type ahead functionality, which returned the first result containing the user's inputted text. It also allowed users to select blocks and type directly into them, where only valid compatible blocks are displayed in the options. This required some form of context awareness - typeblocking had some understanding of the logic in the current block stack and the agents that exist. Beyond context awareness, McCaffrey implemented a cursor to indicate where the user currently is in the workspace and also provided the ability for users to navigate around their workspace with arrow keys [7]. However, in the upgrade from a Java program to a web platform, StarLogo

underwent several changes and most of the implementation was rewritten from the ground up. The biggest changes occurred in the block design and architecture. Previously in TNG, blocks did not have built-in arguments and unique blocks are made for what would've been arguments (if there was a Turtle and an Ant breed, there would exist both a "create Turtle" block and a "create Ant" block). This made the translation logic a little more straightforward as it just involves a sub-string search through block labels. Unfortunately, this logic is incompatible with the new block designs of StarLogo Nova, which does have arguments within blocks to make them more dynamic. As a result, the typeblocking functionality had to be deprecated in StarLogo Nova. This thesis will hope to re-implement and build upon several of McCaffrey's approach and design decisions but suited around StarLogo Nova.

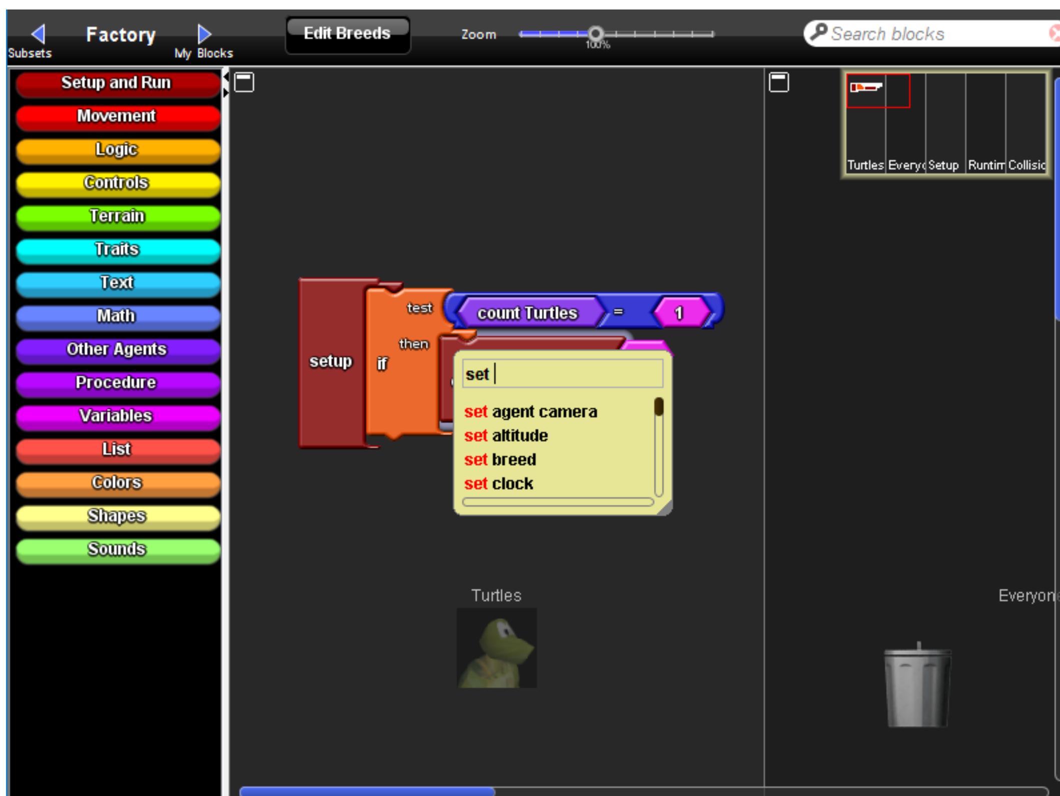


Figure 2-1: StarLogo TNGs Typeblocking

2.2 App Inventor

Another block programming environment that implements typeblocking is MIT's App Inventor. In the App Inventor workspace, the user can type any key and a fixed search box appears on the upper left hand corner, allowing users to filter through all possible options that match the input that the user types in. The user can select a block from the list provided and the block is dropped into the last clicked location on the workspace. If the user clicks a block prior to typing, App Inventor attempts to connect the two blocks. In the case that they are incompatible, it will simply drop the block near the clicked block [14]. App Inventor provides a great first approach to typeblocking, but there are improvements that can be made to enhance the experience. For example, their search box is fixed in the upper left hand corner, which forces the user to constantly gaze back and forth between their blocks and the upper left hand corner of the workspace. This can be distracting and may disrupt a user's flow. Also, their filter function only checks if a block's label contains the user's inputted text, but does not consider arguments nor nested blocks (blocks within blocks). In addition, the user interface does not provide the ability to navigate around the workspace with the keyboard, which forces the user to constantly switch between the keyboard and the mouse, which is arguably slower than just using the mouse. These shortcomings provide great insight on features we hope to include in StarLogo Nova.

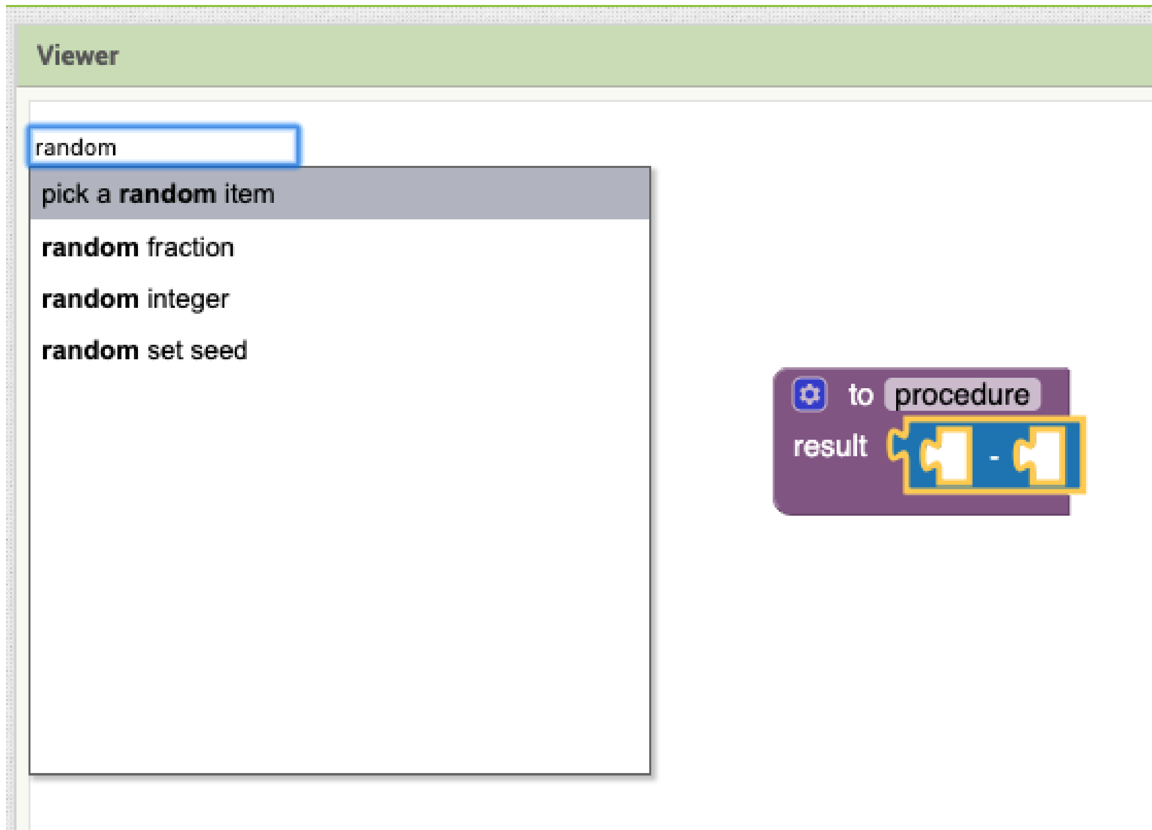


Figure 2-2: App Inventors Typeblocking [8]

2.3 Gameblox

Text to block translation can also be interpreted as generating blocks from code text, written in the syntax of traditional programming languages. One system that interprets it this way is Gameblox, a block programming environment that allows young users to create games [5]. The program provides the function to toggle between "Block Editor" and "Text Editor" mode. In block editing mode, users program with the traditional drag-and-drop block programming environment. In text editing mode, the blocks that are on the page are converted into text in an object oriented programming language. Users may write code (with auto-completion provided) and if the syntax is correct, the code can be visualized as blocks when the user switches back to block editing mode. This system familiarizes users with object oriented programming language syntax, preparing them for more traditional programming languages. Although this is a valid interpretation of text to block translation, it will not be used

in typeblocking because our goal is to enhance the block programming experience, and not to convert users to coding in traditional programming languages.

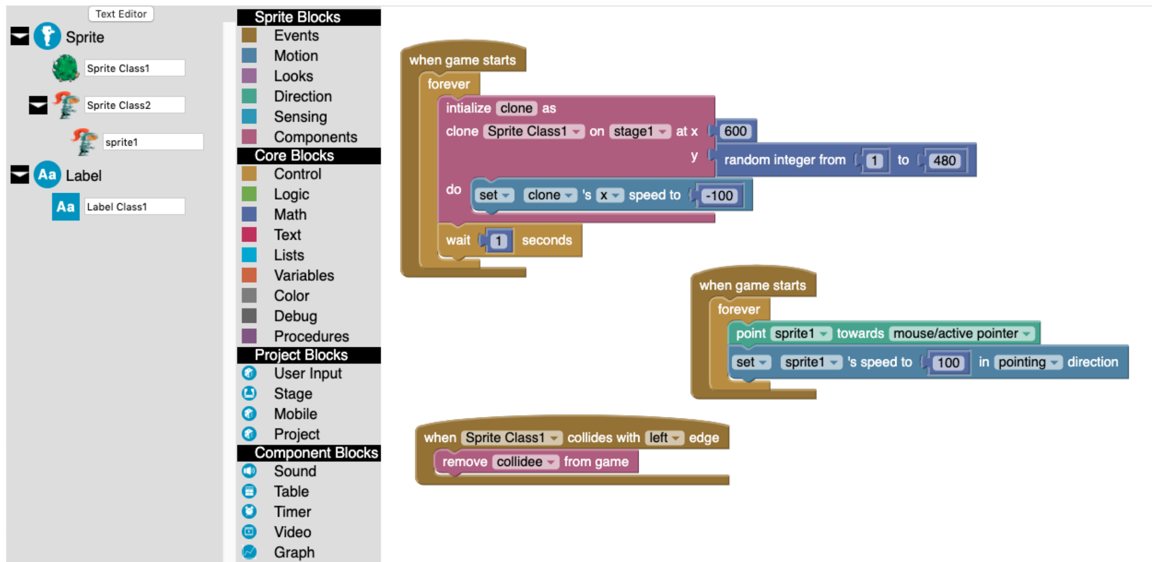


Figure 2-3: Gameblox Block Editor [4]

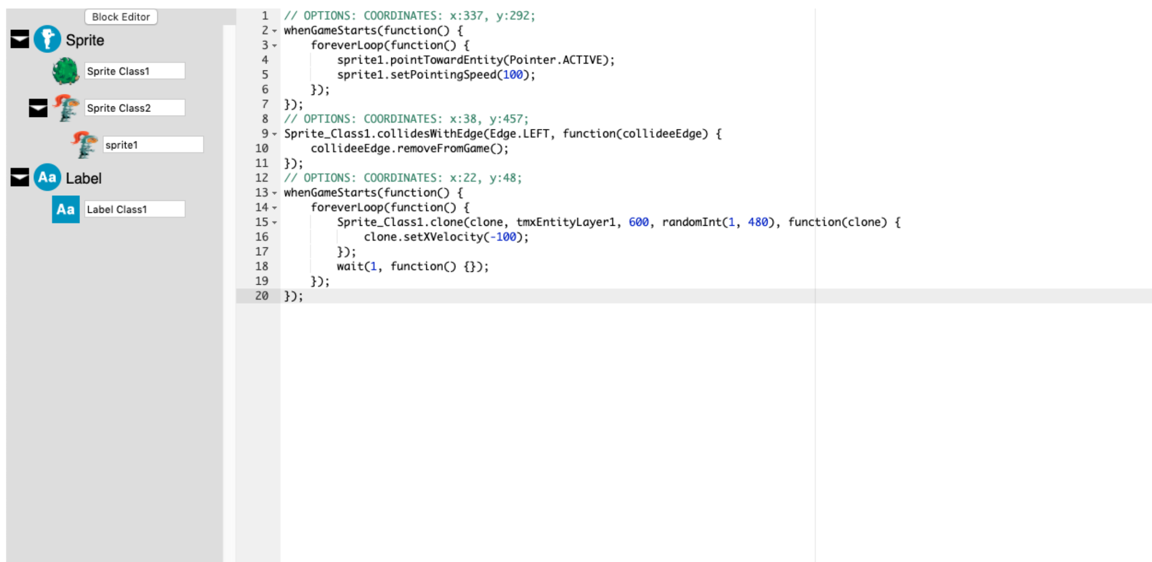


Figure 2-4: Gameblox Text Editor [4]

2.3.1 Text Editors and Computer Programming IDEs

The user interface of typeblocking takes a lot of influence from text editors and computer programming integrated development environments (IDEs). One feature is auto-completing, as seen in IDEs like Sublime Text [1], allowing options to appear that contain parts of the users input. This allows users to execute the command (or block, in our case) without needing to type out the entirety of the phrase. In text editors, a cursor helps indicate to users where they currently are on the page and where text will be inserted next. An important feature that exists in both IDEs and text editors is navigation with the use of arrow keys - moving to the previous/next line with the up/down arrow keys, moving to the previous/next character with the left/right arrow keys, etc. Because most users are familiar with text editors and its paragraph format, we incorporated these design ideas into typeblocking so that users can easily learn how to use it.

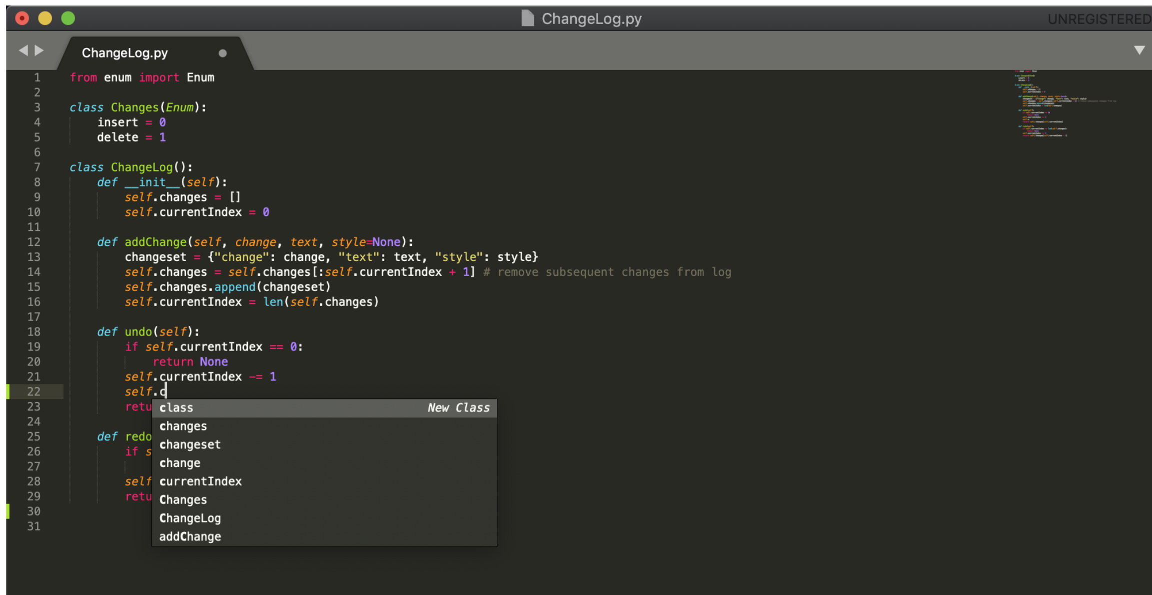


Figure 2-5: Sublime Text Auto-complete [1]

Chapter 3

Definitions

Since this thesis is built on top of StarLogo Nova, this paper will be using a few terms that are specific to the StarLogo Nova environment:

Agents are the user-programmed objects that exist in the virtual space (coined SpaceLand in StarLogo Nova). Agents are the equivalent of turtles in Logo.

Breeds are StarLogo Nova's abstraction for classes of agents. Breeds have traits, or internal variables. These traits may be unique to the breed and are defined by the user. There may also be traits that are common to all breeds, like shape and size.

Widgets are items on the user interface that either take in user input or display information defined by the user. These include buttons, sliders, data boxes, tables, labels, and graphs.

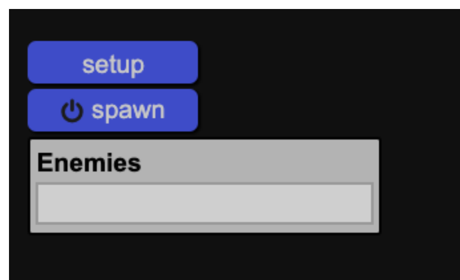


Figure 3-1: Example Widgets (push button, toggle button, data box)

Blocks act as abstractions for code in StarLogo Nova. They are all puzzle shaped, where the shape indicates to users what socket type (defined below) they are. Certain blocks take arguments in the form of strings, drop-downs, or other blocks.

Figure 3-2: Create-Each-Do is a command block, which takes in three arguments (number of agents, breed of agents, and commands that each agent will execute).

The **Workspace** is the space where the code is located and where users can drag and drop their blocks. The workspace contains pages (see top of Figure 3-3), one for each breed, which is where the coder can write breed specific commands and define breed specific traits. Aside from the breed pages, there also exists a page for The World, which is intended for widget handling and setup code, and a page for Everyone, which is intended for commands intended for all breeds.

Drawers are located on the left side of the users workspace and can be treated as libraries that hold all the blocks. Blocks are categorized into drawers depending on what their use case is. Some example drawers include Agents (agent related blocks), Math (mathematical blocks), and Interface (blocks that interact with widgets). Blocks in a drawer are colored to match the color of their drawer.

Sockets are the connection points on blocks. The shape of the socket indicates what blocks it can connect to and corresponds with the socket types of blocks.

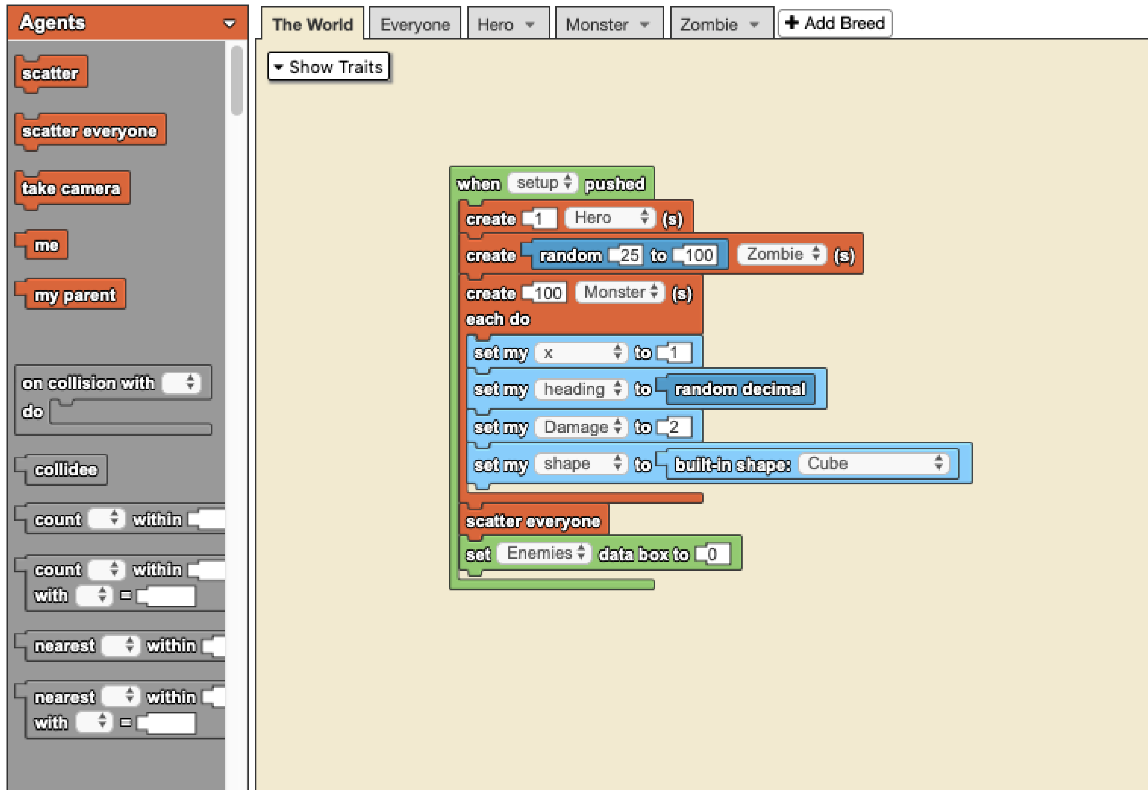


Figure 3-3: Blocks drawers are on the left of the workspace

Socket Types are similar to data types in most other programming languages, but in the context of blocks. As mentioned before, the shape of a block corresponds with its socket type. This defines what blocks it can connect to by forcing the user to match the shape of the block with the shapes of sockets. For StarLogo Nova, the socket types are strings, booleans, and commands.

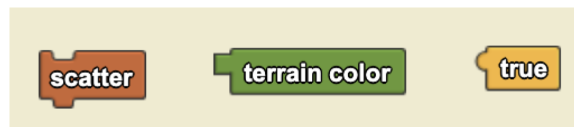


Figure 3-4: Example of each socket type (command, string, and boolean)

The **focus block** is the block that the user is currently working with. This is analogous to that of the current line that the user is on in a paragraph of text. In StarLogo Nova, the block's border becomes blue and white to indicate to the users that it is focused.

The **cursor** is a component introduced with typeblocking. Just like in text editors, the cursor indicates to users where on the focus block that the user will be inserting blocks (or strings) next. Cursors are placed over either the drop-down arguments or the sockets of blocks.

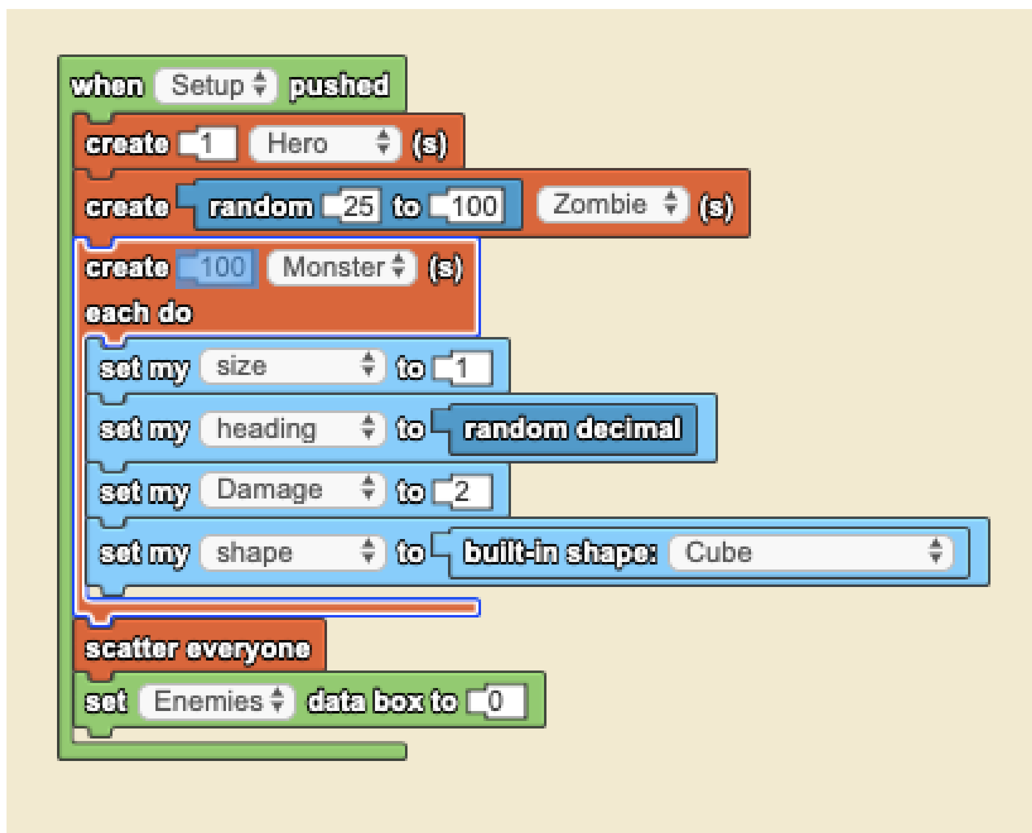


Figure 3-5: In this block stack, the Create-Each-Do block is the focus block, and the cursor is over the number-of-breeds socket

Chapter 4

Challenges

After careful analysis of McCaffrey's work on StarLogo TNG and of StarLogo Nova itself, there are several challenges with implementing typeblocking. Important design decisions need to be made with each component of typeblocking to ensure that the user experience is optimized to be the best it can.

4.1 Context

One reason users prefer computer programming IDEs like Sublime Text [1] over text editors is that IDEs have knowledge of the programming language that the user is writing in. These systems also have some understanding of the code that has been written already, such as what functions exist, what arguments they take, and what global variables exist. All of this information is embedded into the system's auto-completion. This feature speeds up the coding process by only requiring the user to type parts of the code they intend on writing and having the system finish writing the remainder. This is a feature that is desired in typeblocking as well.

However, in order to properly implement this, we must also implement a concept of context with StarLogo Nova. This context needs to understand globally shared information such as globally defined traits, what breeds exist, what unique traits different breeds have (see the example in Figure 4-1), what procedures are defined,

what widgets exist, and more. These are not things that can be pre-computed as these values can always be added, changed, or removed anytime during the programming process. In addition, the context is dependent on the location of the focus block and cursor in the workspace to determine factors such as what kind of blocks can be connected to it (comparing socket types), what the current breed is for the focus block (see Figure 4-1), or what variables were defined earlier in the stack (see Figure 4-2). This must all be taken into account when considering context for typeblocking.

(a) Context using the breed of the current page (Monster)

(b) Context using the breed that is created by the create-each-do block (Hero)

Figure 4-1: The set-trait block has different drop-down argument values depending on the context, which needs to be aware of what the valid breed is for the focus block and what unique traits it has.

(a) Context is aware of the existence of the custom variable

(b) Context is unaware of the custom variable that is later in the block stack

Figure 4-2: The context needs to be aware of custom variables created earlier in the block stack.

4.2 Translation

One of the core functions to typeblocking is to be able to take user inputted text and translate that into blocks. There are many interpretations to how this can be

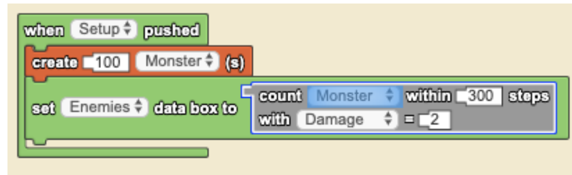
implemented. The first challenge comes with representing blocks as strings and then performing some method of pattern matching to determine which blocks the user was intending on writing. As mentioned previously, this translation logic must be able to provide some form of auto-completion. This would have to include some form of a sub-string search within the block's string representation.

Although a simple sub-string search function can provide a lot of functionality, there can be much more extensions to this translation logic. Since blocks can have strings as arguments or blocks as arguments, it would be beneficial if typeblocking also allows users to pre-populate blocks with argument values before inserting them into the workspace. This would require either a defined grammar or some way of parsing through the block's string representation to determine which values are intended to be arguments. Defining a grammar may be difficult as they are traditionally built static and require the state of the environment to remain constant. Since StarLogo Nova intends on supporting user-created custom blocks, a dynamic grammar would need to be made which is much more complex. Even if a grammar is decided and arguments can be parsed, there must be additional logic to parse the arguments into either another block or simply as an argument value. This level of nesting of blocks can be theoretically limitless, so the logic used should be simple and efficient to account for performance.

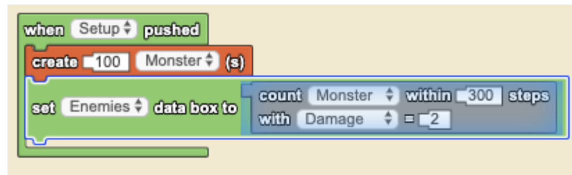
4.3 User Interface

Since StarLogo Nova is a visual programming language, the typeblocking user interface must be very easy to use. Otherwise, it would be counter-productive if users find the experience to be unnatural. One big influence to typeblocking is text editors and programming language IDEs. We want the controls to be similar to what a user would use when they are typing something like an essay. This would mean that users should be able to navigate the workspace with just their arrow keys and can type directly without needing to use the mouse much or any other keys.

However, unlike text editors, block programming languages have both blocks and sockets as objects. Sockets are parts of blocks and the closest analogy to that in text editors is lines and words within a line. Since sockets are not always lined up horizontally in a block, the user should still be able to navigate the workspace paragraph style. That is, if they press down, the element below their cursor should be selected next. This element can either be a block or a socket (Figure 4-3b) depending on the location of the cursor. The situation becomes more complex when there is a child block where the cursor is (Figure 4-3c). There needs to be a method to enter in and exit out of a child block, without the need of unnatural key commands. These are all issues that are crucial to how natural typeblocking will feel when users are navigating around the workplace, so the cursor and its navigation logic needs to be very carefully designed.



(a) Pressing down should move to the socket below the cursor (socket with “Damage” as its value)



(b) There should be a way to enter the child block at the cursor

Figure 4-3: Expected cursor behavior for key presses

In addition to navigation, the block option selection interface needs to be defined as well. The interface should include an input element that takes text and a list of what blocks (and arguments) can be inserted at the current location that matches what the user typed. When users begin typing, they should be able to get a clear indication of what and where they are typing. The options must be shown in a visible and clear manner and sorted in a logical fashion. Both StarLogo TNG and App Inventor

allowed users to navigate through options with the arrow keys and select with the enter key. This makes sense since it resembles that of computer programming IDEs and regular web-page drop-downs. As a result, this logic must also be carefully constructed to not override or be overridden by the navigation logic of the cursor.

The following three chapters explain my solution to these challenges and justifications for why I approached them that way. This project was made through an iterative process with many informal user testing sessions throughout the development period. As a result, many of the design decisions made are based on feedback by co-workers and friends who were using experimental versions of the typeblocking system.

Chapter 5

Context Awareness

In order to properly translate user input into blocks, the system must first know what blocks can be inserted at the location where the cursor is. What the system needs to understand involves several factors: where in the block stack the cursor is at, existing global variables (breeds, widgets, etc.), and what page the block stack is on. All of this information is encapsulated into what we define to be *context*.

5.1 Socket Types

Just like how users are given the visual cue that two blocks can not connect by looking at the shape of its puzzle-piece shape, typeblocking must also understand what blocks can be connected to the socket highlighted by the cursor. The system first considers all possible blocks and compares the socket types of each block (which are all defined in an internal library). If the socket type of a block matches the expected type of the cursor socket (also defined in the library), then it is passed on to the translation logic for pattern matching with the user input.

5.2 Context Object

In StarLogo Nova, the flow of logic and how blocks are connected are represented in the back-end through nodes. These objects form tree structures, where the top node

is usually the highest tier block in a block stack. Nodes include information about what blocks are connected and also what arguments are set for the block they represent. When a block is inserted onto a page, the system fetches a context object. This object includes two pieces of information: the node that represents that block and a breed. Once the block is inserted onto the page, the system walks down from the top node of the block stack to the node defined in the context. As it walks down, it keeps track of any local variables created. This logic also fetches all global variable and procedure names. Using all this information, it determines what drop-down values are valid for the block's arguments.

Typeblocking also needs to know what drop-down arguments exist for a given block. When it explores through the possible blocks that can be inserted, it also considers all the arguments that can be pre-populated to allow users to be able to insert blocks whose arguments are already set. The difference with the original logic is that we need to know what values are valid before we insert the block. To do so, we fetch the context from the focus block (instead of creating one from scratch). However, there are two instances where this context is not entirely correct: when the focus block is a create-each-do block (a block that creates a number of agents of a user-selected breed and takes in commands that those agents will execute on creation) and a breed different from the current page is used (see Figure 4-1b), or when there is no focus block and the user is inserting onto the page itself. For the former, we use the context of the create-each-do block and update the breed with the one defined in its breed argument. If the breed argument is not set, we set the breed of the context to be undefined, which populates drop-down arguments for all possible breeds. This is done so that the users have the freedom to code for any breed and can define the breed later. For the case where there is no focus block, we create a context object with an empty node and the breed of the workspace's current page. With the appropriate context, we can use the existing logic to determine drop-down arguments for blocks.

Chapter 6

Translation Logic

With the appropriate context, the valid options for blocks and arguments have already been significantly narrowed down. What comes next is the need to take account of what the user inputs and match that with the appropriate block.

6.1 Base Approach

Our initial approach to the translation logic involves doing a simple sub-string search, as App Inventor did. This first requires a string representation of the blocks. Visually, to the user, all arguments are replaced with empty brackets to correspond to argument sockets. For example, a create-each-do block looks like "create [ ] [ ](s) each do [ ]". Before this string is searched through, it goes through a little formatting to remove characters users would not normally type - such as brackets or parentheses. If the user input is a sub-string of this formatted block string, it appears as an option for typeblocking. At this current stage, this implementation is not entirely robust, but it does provide a means to auto-completion since any result containing the user input is outputted as an option. The following sections will touch on how this implementation is extended.

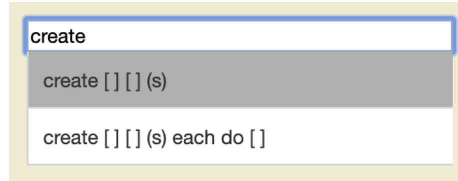


Figure 6-1: Block results for base approach with user input "create"

6.2 Drop-downs

In StarLogo Nova, many blocks have drop-downs for arguments. These drop-downs may either be dynamic and unique to the code that the user is writing, or they are built-in. Typeblocking handles drop-downs in two ways: directly as an argument or as argument assignments within a block insertion.

6.2.1 Argument

All arguments are treated as sockets and may therefore be highlighted by the cursor. When the cursor is over a drop-down argument, the user cannot physically insert a block at that location. As a result, typeblocking does not provide options that will be inserted, but rather will show what options are available in the drop-down itself. Users are still able to use sub-string search to narrow down their options. The reason for incorporating this is so that users do not need to rely on the drop-down web page element to select a value, and can instead continue to use the typeblocking interface. This minimizes any interruptions to the work flow of the user as they're using typeblocking.

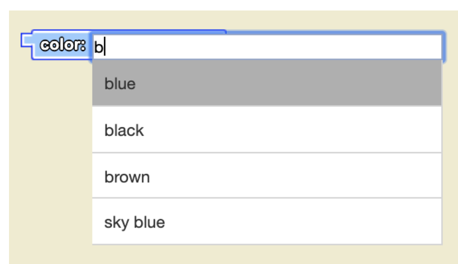
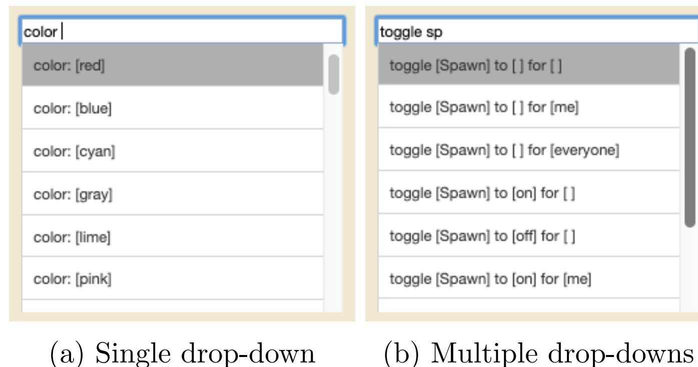


Figure 6-2: Typeblocking shows drop-down options instead of block options

6.2.2 Within Block Insertion

As mentioned in a previous chapter, we want to fetch possible drop-down arguments when we consider blocks to be inserted. If there exists drop-down arguments for a block, the argument values are inserted with brackets at the appropriate position ("create [ ] [Monster](s) each do [ ]"). For blocks with multiple drop-down arguments, we show a linear combination of all the arguments that can be set within the block. However, with enough drop-down arguments, this may result in too many linear combinations of them and may result in an overwhelming number of options. The benefits of having all these options is that novices have the opportunity to explore arguments that they are not familiar with. The downside is that there are too many arguments and it can take time to look through all of them to find the appropriate block. The alternative is to hide all drop-down arguments and only show them if users begin typing them (when "color" is typed, only "color: [ ]' will appear, but when typing "color b", then "color: [blue]", "color: [black]", and "color: [brown]" will appear). After careful consideration, I decided to keep the information density as is, since experienced users can just type more to narrow down their options (type "color red" instead of just "color"). To address the issue of finding the appropriate block from too many options, a lot more emphasis is put on the sorting logic (described in detail in Section 7.2.3) to ensure that the most relevant and appropriate blocks are at the top of the list, so users do not need to search very far and still have the opportunity to explore other options.



(a) Single drop-down (b) Multiple drop-downs

Figure 6-3: Examples of drop-down assignments with pre-filled values

6.3 String Sockets

For sockets whose socket types are strings, we also take the literal user input and provide the option to insert it as a string.

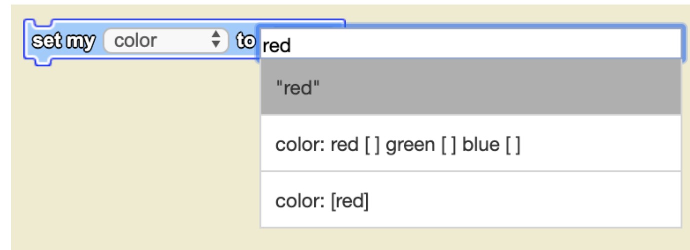


Figure 6-4: The literal string will be an option for string socket type sockets

6.4 String Arguments

Typeblocking attempts to pre-fill string arguments upon block insertion. When the user input returns no valid results (ie. it did not match any sub-string), we try to parse parts of the user input into string arguments. However, this logic becomes more complex than a simple sub-string search as it requires some degree of parsing to differentiate what part of the user input is for the argument and what part is for the block string itself.

The original idea was to create a static grammar, but we want to avoid having to write a grammar that is specific to the current built-in blocks. This is to account for user-created custom blocks, a feature that StarLogo Nova hopes to support in the future. To have a more general solution, we use a formatted version of a block string (with or without drop-down arguments) and apply a running word search to match the words of the user's input with the words in the block string. When a matching word is found, we assume that the user is writing that part of the block. Once we reach an unmatched word, we assume it is part of an argument (if an argument is expected at that location) and will listen until we reach a matching word again to determine where the string argument ends. We also consider the cases where a matching word

may still be part of the argument and will look for matching words further ahead.

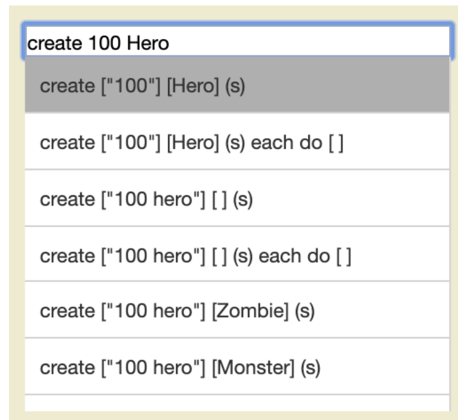


Figure 6-5: String argument parsing

6.5 Nested Blocks

One stretch goal for this logic was to provide support for nested blocks. That is, users are able to write a complete phrase and the system would be able to parse it and fill in block arguments with the relevant blocks. For example, the user can type "set my size to 5 + random decimal" and the system is expected to be able to parse it as "set my [size] to [["5"] + [random decimal]]" to return the results as seen in Figure 6-6.

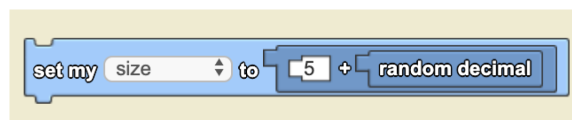
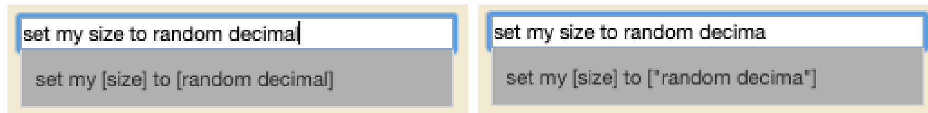


Figure 6-6: Expected nested block output for "set my size to 5 + random decimal"

This logic is currently implemented in typeblocking to an extent. We take advantage of the string argument parsing logic and attempt to match the string argument to an appropriate block. One limit to the current implementation is that it supports only up to one level of nesting. Therefore, the example above does not work as that requires an additional level of parsing (for the values 5 and the random decimal block), but simpler phrases like "set my size to random decimal" will work. The implementation is kept at a single level to keep computation simple. Using any additional recursive

methods from there on would involve much more complicated logic and may begin to hurt performance. Another limitation to this implementation is that the entirety of the nested block must be typed. This is needed for the current logic to determine if the user intended to insert strings or insert blocks.



(a) Entirety of block must be typed for nested blocks

(b) Otherwise, it will be treated as a string

Figure 6-7: Translation logic can currently only handle one level of nesting

Another reason that nested block logic is kept at this simple implementation was mainly due to observations during user testing. With the convenience of typeblocking and auto-completion, it is much easier to type parts of a block phrase to get the desired block and fill in the arguments later, than typing out the entirety of the block phrase. The only desire for nested blocks are when the block phrases are relatively short, which can generally be handled by one level of nesting. Possible improvements to the nested block logic will be discussed in chapter 9 on future work.

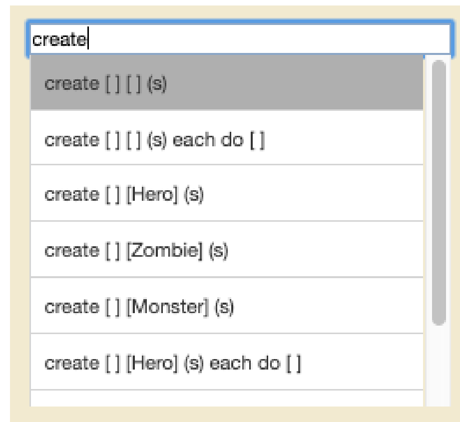


Figure 6-8: Block results for current implementation with user input “create”

Chapter 7

User Interface

The goal of typeblocking is to improve user coding experience by shifting the coding process away from the tedious drag-and-drop mechanism and towards keyboard use. To accomplish this, the system needs to do more than just translate string to blocks - it must also provide the means to navigate the workspace. As a result, the user interface for typeblocking can be sorted into two main components: workspace navigation and string to block translation.

7.1 Workspace Navigation

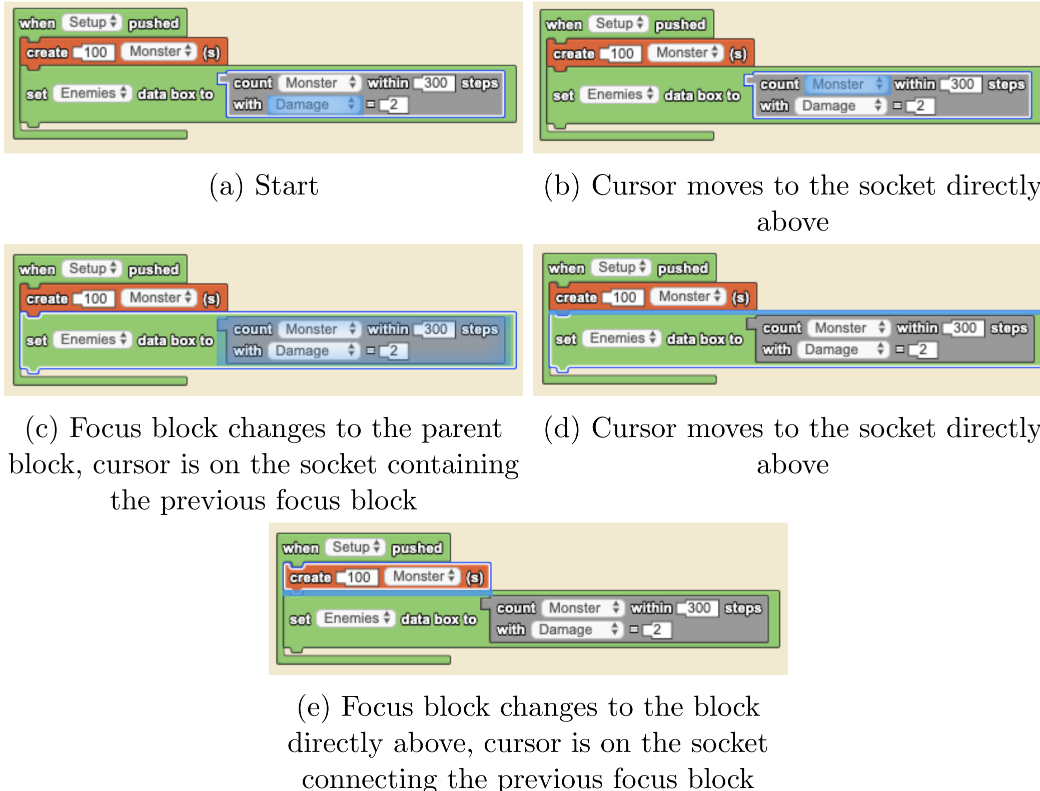
7.1.1 Cursor and Focus Block

As defined previously, the focus block indicates to the user which block they are currently working with. It is updated anytime the user clicks a new block, when a new block is added through typeblocking, or when the user moves the cursor into a new block (explained below). For typeblocking to work, the concept of a cursor needed to be introduced. The cursor appears anytime the focus block is changed. Since the cursor is a new feature introduced to StarLogo Nova, we also provided means to remove it when users who do not want to use typeblocking and find it distracting. The user can reset the focus block (and with it, the cursor), by simply clicking an empty spot in the workspace. Alternatively, they may use the Escape key.

7.2 Navigation

To control the cursor or change the focus block, the user may use the arrow keys. The movement is designed to emulate that of a traditional text editor. However since blocks are much more complex than a line in a text editor, there must exist additional behavior. This is especially crucial for dealing with child blocks (blocks nested within another block as an argument). The behavior of the cursor depends on the direction pressed by the user and where the cursor is currently located:

UP: If there exists a socket directly above the cursor within the same focus block, the cursor will move to that socket. If no such socket exists, the focus block will change to the block directly above it. This block may be a before block (a command block directly on top of it) or the parent block that contains the current focus block. If there is no valid block, then nothing happens. If the focus block changes, the cursor moves to the socket that connects to the previous focus block.



(a) Start

(b) Cursor moves to the socket directly above

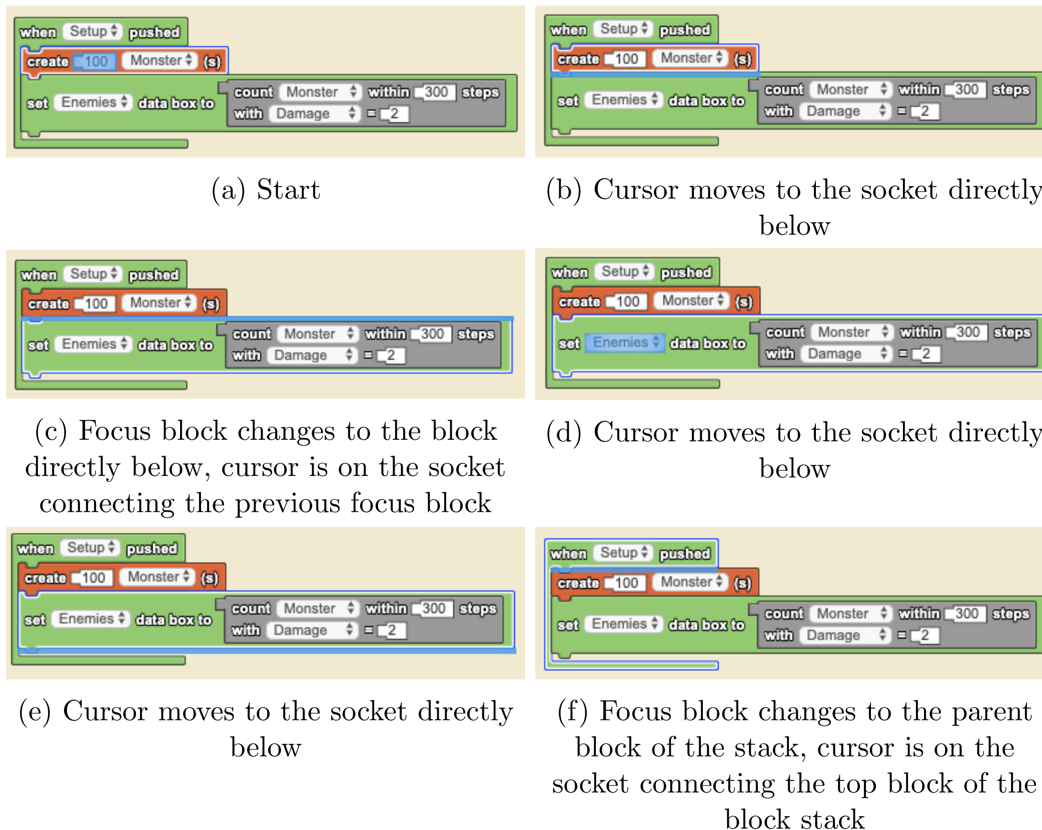
(c) Focus block changes to the parent block, cursor is on the socket containing the previous focus block

(d) Cursor moves to the socket directly above

(e) Focus block changes to the block directly above, cursor is on the socket connecting the previous focus block

Figure 7-1: Example behavior of Up key pressed multiple times in sequence

DOWN: If there exists a socket directly below the cursor within the same block, the cursor will move to that socket. If no such socket exists, the focus block will change to the block directly below it. This is either an after block (a command block directly below it), a child command block, or the parent block that contains the current focus block. If the focus block does not directly have a parent block, it looks for the parent block of the top block of the focus block's block stack (a sequence of command blocks). If there is no valid block, then nothing happens. As with UP, the cursor moves to the socket that connects to the previous focus block (or the socket that connects to the top of the block stack).



(a) Start

(b) Cursor moves to the socket directly below

(c) Focus block changes to the block directly below, cursor is on the socket connecting the previous focus block

(d) Cursor moves to the socket directly below

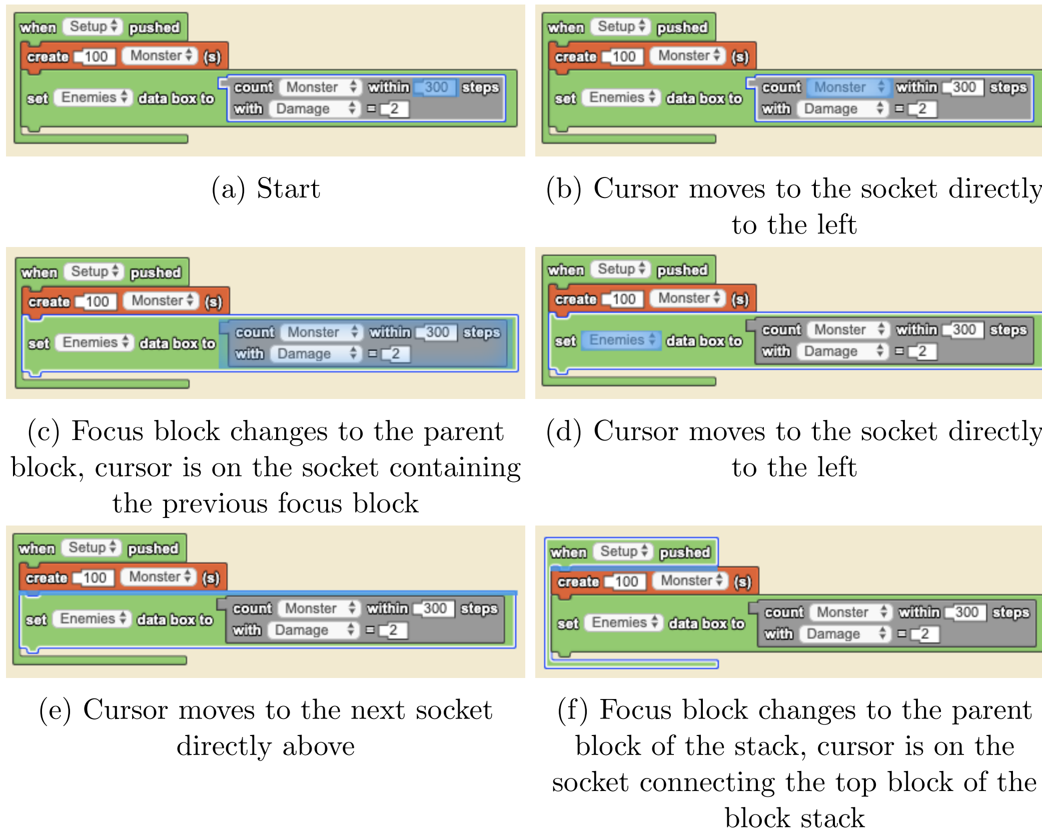
(e) Cursor moves to the socket directly below

(f) Focus block changes to the parent block of the stack, cursor is on the socket connecting the top block of the block stack

Figure 7-2: Example behavior of Down key pressed multiple times in sequence

LEFT: If there exists a socket directly to the left of the cursor within the same block, the cursor will move to that socket. If no such socket exists, it attempts to move to the next socket above it, where the rightmost socket is prioritized (similar to wrapping to the end of the previous line in a text editor). Otherwise, if the current

focus block is an argument of a parent block, the focus block changes to that block and the cursor is set to the socket that the original focus block was connected to.



(a) Start

(b) Cursor moves to the socket directly to the left

(c) Focus block changes to the parent block, cursor is on the socket containing the previous focus block

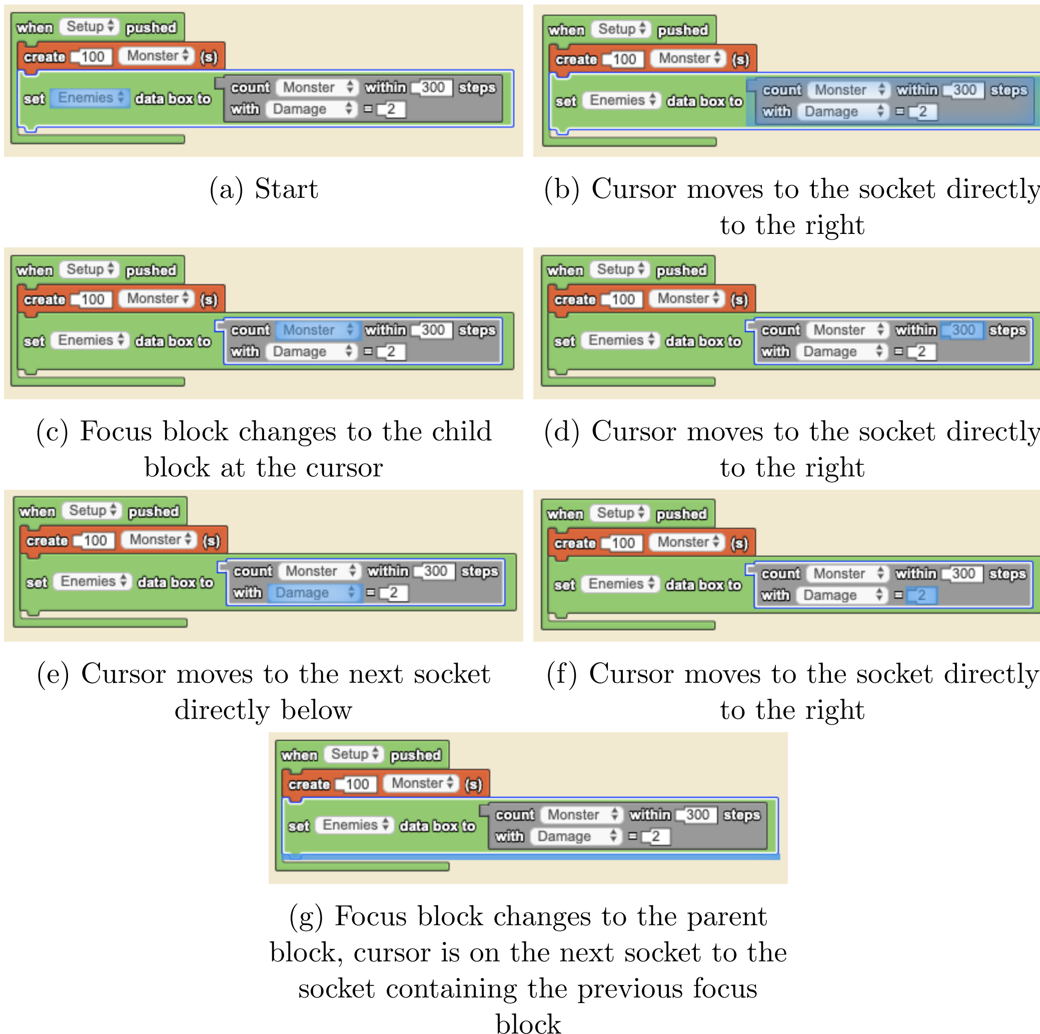
(d) Cursor moves to the socket directly to the left

(e) Cursor moves to the next socket directly above

(f) Focus block changes to the parent block of the stack, cursor is on the socket connecting the top block of the block stack

Figure 7-3: Example behavior of Left key pressed multiple times in sequence

RIGHT: If the current socket has a block as an argument. The focus block changes to the argument block and the cursor is set to the first socket of the child block. Otherwise, if there exists a socket directly to the right of the cursor within the same block, the cursor will move to that socket. If no such socket exists, it attempts to move to the next socket below it, where the leftmost socket is prioritized (similar to wrapping to the beginning of the next line in a text editor). If there are no sockets left, the focus block changes to the parent block of the current focus block and the cursor will attempt to move to the socket to the right of the socket containing the original focus block (this is to avoid a loop of entering and exiting a child block).



(a) Start

(b) Cursor moves to the socket directly to the right

(c) Focus block changes to the child block at the cursor

(d) Cursor moves to the socket directly to the right

(e) Cursor moves to the next socket directly below

(f) Cursor moves to the socket directly to the right

(g) Focus block changes to the parent block, cursor is on the next socket to the socket containing the previous focus block

Figure 7-4: Example behavior of Right key pressed multiple times in sequence

As mentioned before, these keyboard controls are intended to emulate a text editor. Toggling the cursor between sockets of the same block involves very simple logic. Similarly, toggling the focus block up and down is fairly straightforward as well. What was the most tricky was the ability to enter argument blocks and pop back out to the parent block. Originally, the implementation required the user to press the Shift key and Right key to indicate that they want to change the focus to an argument child block, and press the Shift key and Left key to pop back to the parent block. However, through testing, users reported that it was non-intuitive and unnatural. As a result, most of the keyboard logic for navigation is limited to be done with just the arrow keys.

7.2.1 Internal Inputs and Drop-downs

In StarLogo Nova, string sockets are also input elements - if users wanted to fill in arguments with a string, they'd have to type directly into the socket. With the introduction of typeblocking, we still want to maintain the ability to modify these string socket inputs and also drop-down arguments. Since these argument sockets can be hovered by the cursor, the user can use the Enter key to switch the focus of the web page to the internal element (either the input element or the drop-down element) and may modify them directly. When they enter the element, the cursor will disappear. Once the user has finished typing the string in the input or has selected a drop-down, they can press Enter for the cursor to reappear back over the socket. Alternatively, they may also use the Escape key for the cursor to return. This feature was included as a request from a user tester, who pointed out that there are instances where they want to use the built-in web element but still be able to toggle back to typeblocking when desired.

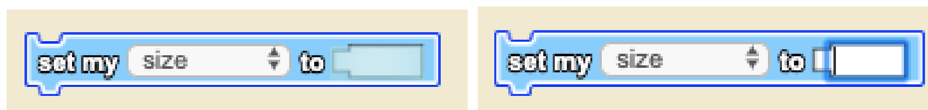


Figure 7-5: The user can enter an arguments' internal input

7.2.2 Deletion

To delete blocks, the user can press the Backspace key (delete key on Apple machines) to delete the focus block. If the focus block has any blocks within it as arguments, those blocks will be deleted as well. However, command blocks in a block stack remain and the before and after block of a deleted block will connect with one another. If, however, the top block in a stack is deleted (regardless of the type of block it is), the entire stack is deleted.

7.3 String to Block Translation

In order to convert text to blocks, there must be means for the user to type text. Inspired by App Inventor's typeblocking, I took a very similar design for how users can input text and what block options look like.

7.3.1 Input Element

When the user types any valid key on the keyboard, an input element appears on the workspace. Valid keys include the alphanumeric characters, special characters, and the space bar. This excludes the use of modifier keys (besides the Shift key). When using App Inventor [8], I noticed that the input element always appears at the top left corner, but the block appears at my last clicked location. This ended up being distracting as my gaze would always look away from my code. As a result, I wanted to implement the input element to also appear at the last clicked location. If the user has not clicked any spot, the user input then defaults to appear on the top left corner of the workspace.

7.3.2 Block Options

As the user types, options begin to populate below the input element. These options can be toggled with the up and down arrow keys and selected with the Enter key. The option may also be selected by clicking it. Examples of what this looks like can be seen in the figures in Chapter 6. The space the options are populated with are of fixed size, but as users toggle through them, the options will scroll up/down when the user reaches near the end of the window. When the input element is present on the workspace, cursor and focus block navigation is turned off to avoid race conditions with the keys. Once a block option has been selected, the appropriate block appears and connects to the socket that the cursor is on. If there is no cursor, the block is dropped directly onto the workspace at the location of the input element. This design is created based on how StarLogo TNG and App Inventor implements typeblocking and resembles drop-downs in web pages. If the user decides to not insert blocks

anymore, they can hide the input element by clicking an empty spot in the workspace or press the Escape key.

7.3.3 Sorting Logic

Because we decided to keep the linear combinations of all drop-down arguments for valid blocks, there may be instances where a user input can result in multiple, even hundreds of, options. To avoid needing users to scroll through all the options, we want to make sure that these options are sorted to be the most optimal. The sorting logic is as follows (in this sequence, the next prioritization is only needed if there is a tie in the current one):

1. Prioritize the blocks by where the user's input appears in the block string (from left to right). This is to emulate auto-completion.

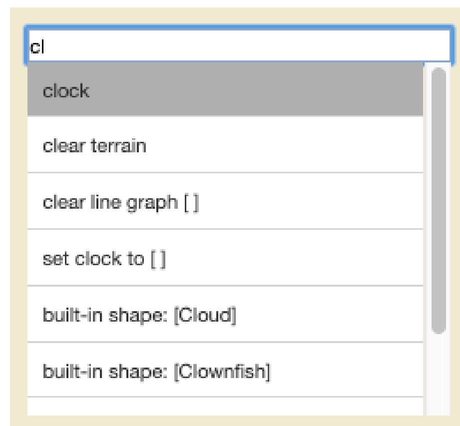


Figure 7-6: Prioritize "clock" over "set clock to [ ]" when user input is "cl"

2. Prioritize the blocks by the amount of filled arguments (descending order) since blocks with empty arguments should appear first (unless the user wrote arguments, which is handled by the previous condition).

3. Prioritize the blocks by the number of arguments overall (descending order). Certain blocks in StarLogo Nova are extensions of others ("create-each-do" block can be seen as extension of the "create" block). Thus, we want these extensions to appear after the base blocks.

Figure 7-7: Prioritize "create [ ] [ ](s)" and "create [ ] [ ](s)each do [ ]" before "create [ ] [Hero](s)" and "create [ ] [Hero](s) each do [ ]".
Prioritize "create [ ] [ ](s)" before "create [ ] [ ](s) each do [ ]".

4. By this step, the importance of these blocks are roughly the same. The following priorities exist mainly to maintain a consistent sorting order:

   - Prioritize blocks by the index of the first empty argument (ascending order).
   - Prioritize blocks by the index of the first argument (ascending order).
   - Finally, prioritize the blocks by length of the string. This differentiation is usually for the same block with the same arguments filled, but with one or two values different. The shorter one should appear first.

7.3.4 Cursor Location

Originally, when a block is inserted into a cursor, the cursor reappears at the original socket, over the newly inserted block. After a few suggestions from user testing, the cursor now appears at the next empty socket in the block. The intention is to follow a typical user's work flow logic. As a user codes, their goal might be to populate all the arguments in a block. By automatically moving to the next socket, they may directly continue inserting argument values, without needing to use the arrow keys (which some users have claimed can sometimes be a little distracting and interfered with their flow). Similarly, when a user inputs a string into a blank argument or selects a

value from an originally blank drop-down, the cursor moves to the next blank socket. If a block is inserted with pre-determined arguments, the cursor now appears over the first empty socket, instead of the top-left socket.

Chapter 8

User Testing and Reception

One of the major concerns with typeblocking was whether or not the user interface was smooth and if the translation logic follows what they would expect from text to block translation. Throughout the implementation process, I would constantly query the opinions of my friends and co-workers in the lab. Their response and feedback provided a lot of insight towards making typeblocking feel as natural as possible. Below are a few pivotal design designs made thanks to feedback from user-testing. After a majority of the implementation for typeblocking was complete, a more formal user testing was conducted on friends and co-workers who were familiar with StarLogo Nova.

8.1 Informal User Test Feedback and Changes Made

Below are a few of the changes made based on user feedback received from informal user testing:

- The design of the cursor has been changed many times during implementation. As explained in a previous section, a user suggested that the cursor should move to the next argument to follow the flow of the user.

- Similarly, controlling how the cursor appears and disappears when a user is in an internal input argument or drop-down also had some discussion around it. Some

users preferred to have a "light" version of the cursor over the socket to remind the user that they are currently focused there. Others found it distracting and preferred that it disappeared and appear back when the user is done editing the field. The latter was kept in the final implementation as it looked cleaner stylistically.

- Navigation for the cursor has also been of much debate. As mentioned before, the original implementation used the Shift key to shift the focus in and out of child blocks. Users have commented that this felt unnatural, so I ended up implementing the logic with just the arrow keys. However, the behavior of how the cursor moves was also debated. Originally, the up and down arrows only controlled the focus block, but users have noted that it seems unnatural if there is a socket in the same block that is above/below the cursor. By changing those keys to handle this, it made the navigation system more closely resemble the paragraph-style of text editors.

- When using type blocking, many users are drawn towards using the tab key to move the cursor over (since that is what users can do when completing online forms). This was not handled originally, and as a result, the focus of the web page shifts to some other place in the page. One idea was to use the tab key to toggle the cursor, but since there can be many interpretations of what the next socket is, it was decided to be kept out instead to avoid having any confusing or misleading behaviors.

- The sorting function for block options has also been tinkered with many times. There are many ways to interpret what the correct logic is and user testing has provided a lot of insight on what the priorities should look like.

8.2 Formal User Test Plan

For both experienced and novice users, they are tasked with coding an example program that consists of two stacks. They are asked to code this program once

without typeblocking (only drag-and-drop) and once with ONLY typeblocking (no drag-and-drop).

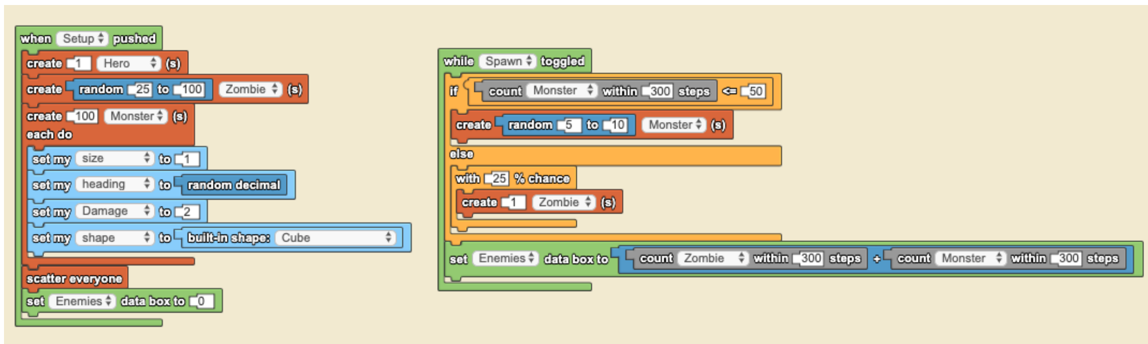


Figure 8-1: User test example that users were asked to code

For experienced users, the goal is to notice patterns and thought processes when they use the drag-and-drop mechanism in StarLogo Nova and how they change when they are using typeblocking. For novice users, the goal is to notice the challenges that new users have when using StarLogo Nova and observe if typeblocking improved that experience in any way.

1. Ask user to spend 5 minutes familiarizing themselves with StarLogo Nova as is, without typeblocking. Explain how it works as a language and clarify any questions they may have. This step is not needed for experienced users as they are already familiar with the system.

2. Ask user to replicate the example with StarLogo Nova as is, without typeblocking. Note thought process and time for completion.

3. Ask user to spend 5 minutes familiarizing themselves with typeblocking. Note concepts or ideas if they have trouble understanding. Explain as needed if they have questions about functionality.

4. Ask user to replicate the example with typeblocking, and to avoid using the mouse as much as possible. If the user needs to use the mouse (other than to close typeblocking/start a new block stack), make sure they explain why they need to do so. Note thought process and time for completion.

8.2.1 Observations

- There was a common strategy among the experienced users when reproducing a block stack. When they were dragging a block from a specific drawer, they would drag out all the blocks that they needed from the same drawer. They still followed the block stack ordering, but they would have the blocks scattered across their workspace and would connect them when they appeared in the ordering.

- When forced to use only typeblocking, the work flow was much more structured. Blocks and arguments were mostly inserted and filled in the order shown in the example they were following.

- Novice users struggled to find the appropriate blocks they needed and may spend several seconds scrolling through the drawers. Experienced users recognize better where to find blocks, however, there are certain uncommon blocks that even experienced users did not realize existed.

- When users made a mistake in the ordering of blocks without typeblocking, they had to resort to using the mouse to manually reorder them as typeblocking does not support that yet. With typeblocking, some users preferred to delete the block and insert the correct one at the right spot.

- Of all the user tests, only two people have attempted to do nested blocks. When asked later what they expect nested blocks to look like, they replied with relatively short examples like 5 + 7 or set my heading to random decimal.

- For all users, but one, there was a clear speedup when using only typeblocking.

- Interestingly, the fastest users with typeblocking have the most experience programming in traditional languages. This is probably because they are already adapted to using a keyboard for code writing, which may indicate a correlation between coding experience and efficiency with typeblocking.

8.3 Reception

Overall, the response to typeblocking has been positive and every user has stated that they can get use to the feature and expect to perform better as if they are more familiar with it. When asked about how natural the user interface felt, they all responded that it was very intuitive and easy to understand.

One experienced user pointed out that new users might struggle with typeblocking if they are not aware of the name of a specific block. The exploration of new blocks can be better done by scrolling through the drawers. They also pointed out that, even though it may be easy to just delete and re-insert blocks, one benefit of dragging and dropping is that reordering can be done fairly easy with it. These two examples prove that there are still clear benefits to the drag-and-drop mechanism of block programming languages. However, it is still clear from these user tests that the addition of typeblocking to StarLogo Nova has improved user experience and productivity.

Chapter 9

Future Work

From user feedback and from reflection of the implementation of typeblocking, there are several ideas that could be implemented to improve typeblocking:

9.1 Shortcuts

For experienced users (and specifically people who have used legacy versions of StarLogo), there is a desire to shorthand or nickname the names of blocks. For example, "crt" for "create", "create do" for "create-each-do", or "dlt" for "delete." This logic can either be hard-coded, or the translation logic can be extended to incorporate some level of natural language processing.

9.2 Improved Nested Block Logic

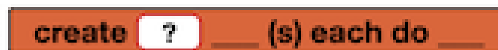
The current logic is not perfect, but is satisfactory for general use cases. A more robust logic would be able to handle any level of nesting and if used correctly with shortcuts, can make creating complex block phrases much faster. However, once multiple-level nesting is supported, there is still the restraint that requires the user to type out the entire block phrase. Future work could explore options that may relax this constraint, such as providing nested auto-completion or a well-defined grammar that can detect incomplete phrases and convert them to blocks. In addition, the

current logic only works for string arguments. Future implementation should be able to handle boolean and command socket types as well.

9.3 Format of Block Options

Block options are currently in the form of strings, which may be hard to visualize without seeing the blocks themselves (another reason type blocking currently benefits experienced users more). If these options were instead the blocks themselves (or smaller versions of them), it can provide a better experience to users. In StarLogo Nova's error console, a condensed drawing of blocks are used to indicate what block has an error. However, this design is hard to use for typeblocking options as it is too small to fit multiple argument values and it does not support nested blocks. Future work can explore proper ways to show block options in a dense but informative fashion.

(a) Initial mock-up of using the actual blocks for the options in typeblocking



(b) StarLogo Nova's error console's design for blocks. Note that it doesn't accurately represent the shape of a create-each-do block

Figure 9-1: Examples of various ideas for block option formats

9.4 Redesign Cursor

The cursor was a needed implementation to signal to users where blocks should be inserted. However, cursors were not necessarily designed as part of StarLogo Nova, but rather designed around it. There are several features that cursors do that seem

unnatural in terms of architecture and behavior. For example, cursors in text editors are seen as a location in between words. Analogously, this would mean that cursors should be in between blocks, but our implementation designs the cursor to be part of the blocks. Exploring other options ended up being outside of the scope of this thesis, but is definitely worth looking into through experimentation and user feedback.

9.5 Incorporate Edit Commands

Sticking with the inspiration from text editors, it would be nice if users are able to perform tasks like reordering (as seen in the user tests) or copy and pasting. Copy and Paste are already functions in StarLogo Nova, and Undo/Redo are expected to be released soon, but there needs to be careful design around incorporating these functions with the keyboard to avoid overriding existing key listeners that are used with typeblocking. In addition, the unique design of the cursor may also brings complications as having a cursor over a socket with a block can indicate that the edit command is either for the child block that the cursor is over or for the focus block itself.

9.6 Integrate with Type Safety

Currently, typeblocking allows users to use blocks that match socket types. These are blocks that can be physically connected when using the drag-and-drop mechanism, but may fail upon compilation (ie. setting the size of an agent to the color blue). A fellow graduate student on the StarLogo Nova team, Michael Belland, has been working on implementing error detection and type safety into the environment [2]. This includes coercing general socket types (like strings) to more specific data types (numbers, colors, strings, etc.) and checking if they are used appropriately. Integrating type safety check into typeblocking can provide additional aid to users as they're coding by hiding blocks that are not type safe to minimize chance of error. However, by hiding these blocks, we may also be hindering the learning process of

novices who may benefit much more by discovering themselves that those blocks are invalid. These trade-offs would need to be considered when attempting to integrate type safety into typeblocking.

9.7 More Extensive User Testing

Due to the lack of available experienced StarLogo Nova users, it was difficult to properly analyze the benefits of typeblocking. In addition, the user test in Chapter 8 took only around 10 minutes to conduct. Since users have stated that more familiarity might improve their performance, having users spend more time using typeblocking might produce more appropriate feedback or results. If given the opportunity, it would be ideal to introduce typeblocking to a class of users familiar with the environment and gauge their response and user experience with it to get a better understanding of what works and what does not work with the current typeblocking interface.

Chapter 10

Conclusion

Block programming has succeeded for many years on its own and has provided a great resource for students to begin learning programming. Typeblocking was created on the idea of increasing productivity by offering a faster means of coding in place of the traditional drag-and-drop mechanisms. The intention was to aide novices in learning the environment and to expedite the coding process for experts who find the dragging and dropping tedious.

My implementation is not perfect, but there are noticeable benefits to its use. The ability to insert blocks directly on the page with just the keyboard does seem faster than dragging and dropping blocks from the drawers. Unfortunately, it does require prior knowledge of what blocks exist. As a result, at its current stage, the benefit of typeblocking seems to be more directed towards experts who are already familiar with the StarLogo Nova environment and less so for novices familiarizing themselves with block programming. However, I do believe my proposal offers a strong improvement to block programming and will be a convenient feature for StarLogo Nova users.

Bibliography

[1] A Sophisticated Text Editor for Code, Markup and Prose. *Sublime Text*, Sublime Text, www.sublimetext.com/.

[2] Belland, Michael D. ”Error Detection and Reporting in StarLogo Nova Block-based Programming Language.” Masters thesis, Massachusetts Institute of Technology, 2019.

[3] Computer Science Education Stats. *CSEd Week*, Code.org, csedweek.org/promote.

[4] *Gameblox*, MIT Scheller Teacher Education Program, gameblox.org/.

[5] Gameblox. *MIT Scheller Teacher Education Program*, 27 Oct. 2018, education.mit.edu/project/gameblox/.

[6] Kelleher, Caitlin, and Randy Pausch. Lowering the Barriers to Programming. *ACM Computing Surveys*, vol. 37, no. 2, 2005, pp. 83137., doi:10.1145/1089733.1089734.

[7] McCaffrey, Corey. ”StarLogo TNG : the convergence of graphical programming and text processing. Masters thesis, Massachusetts Institute of Technology, 2006. doi: 1721.1/36904.

[8] *MIT App Inventor*, Massachusetts Institute of Technology, appinventor.mit.edu/.

[9] Pappano, Laura. Learning to Think Like a Computer. *The New York Times*, The New York Times, 4 Apr. 2017, www.nytimes.com/2017/04/04/education/edlife/teaching-students-computer-code.html.

[10] *Scratch*, MIT Media Lab, scratch.mit.edu/.

[11] *StarLogo Nova*, MIT Scheller Teacher Education Program, www.slnova.org/.

[12] ”StarLogo Nova”. *MIT Scheller Teacher Education Program*, 8 Mar. 2019, education.mit.edu/project/starlogo-nova/.

[13] "StarLogo TNG". *MIT Scheller Teacher Education Program*, 29 Oct. 2018, education.mit.edu/project/starlogo-tng/.

[14] Typeblocking. *Typeblocking — Explore MIT App Inventor*, MIT App Inventor, appinventor.mit.edu/explore/tips/typeblocking.html.