

Error Detection and Reporting in StarLogo Nova Block-based Programming Language

by

Michael D Belland

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© 2019 Michael D Belland. All rights reserved.

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document
in whole or in part in any medium now known or hereafter created.

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

Certified by.....
Eric Klopfer
Director of MIT Scheller Teacher Education Program
Thesis Supervisor

Accepted by
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Error Detection and Reporting in StarLogo Nova Block-based Programming Language

by

Michael D Belland

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 Electrical Engineering and Computer Science

Abstract

StarLogo Nova is a program where users can create 3D games and simulations by connecting drag-and-drop blocks to create programs. Although the structure of the blocks prevents a class of syntax errors common with typed languages, there is no system in place to catch semantic errors like type errors that most compilers detect. Further, errors that occur during the runtime of the program fail silently, providing the user with no feedback about how to prevent the error and sometimes causing unexpected behavior. In this thesis, the author implements a semantic analysis pass to the StarLogo Nova compiler, as well as other debugging-related features, to improve compile time and runtime error reporting and to give users meaningful feedback about their programs.

Thesis Supervisor: Eric Klopfer

Title: Director of MIT Scheller Teacher Education Program

Acknowledgments

I'm a remarkably sentimental person, so I have a lot of people to thank. Even this list is abridged, but I feel like everyone here directly contributed to my thesis and deserves, at the very least, a shoutout.

I have had the privilege to work with the StarLogo Nova team throughout my career as an undergraduate and graduate student at MIT. Every member of the team has been easy to work with and remarkably smart! But more than that, I'm grateful I can call these amazing individuals of the StarLogo team my friends: Andy Guatemala, Dorothy Ren, Shi-Ke Xue, David Wong, Lena Abdalla, Usman Ayyaz, Phoebe Tse, Ami Suzuki, Terrance Liang, Malcolm Wetzstein, Sophie Russo, and Megan Chao. I'm so lucky to have met all of you.

I owe an especially warm thanks to Daniel Wendel, the StarLogo Nova team lead. Since the time he welcomed me onto the StarLogo team near the end of the Spring 2014 semester, he has been incredibly supportive and flexible through easy and difficult times alike. He also has the uncanny ability to suggest clever approaches to problems I've been working on after I've been working far too long at more convoluted solutions.

A thanks to Eric Klopfer, for cultivating a lab environment where people can be themselves – where people can debate whether sushi is a sandwich or a burrito, go up to Vermont to get specialty ginger ale for a lab-wide taste test, or play board games like Resistance with large, chaotic, and fun groups on Fun Fridays. And thank you to Lisa Stump, a coding wizard who was always willing to offer advice, feedback, and help. The lab misses your brilliant personality greatly, and wishes you the best at your new job!

Finally, I'm deeply grateful for the continued and unconditional support my family has given me through this whole process. Mom, Dad, Matt, Brent – I'll never be able to come up with the right words, but your support means more to me than you'll ever know.

Contents

1	The StarLogo Nova Environment	13
1.1	Background and Problem Statement	13
1.2	Components of StarLogo Nova	14
1.2.1	WebLand	15
1.2.2	Workspace	15
1.2.3	Error Reporting Console	19
1.3	Error Checking	19
1.3.1	In StarLogo Nova	20
1.3.2	In other programming languages	21
2	StarLogo Nova’s Compiler	25
2.1	Program Flow	25
2.2	Yielding Code Pipeline	28
2.3	Blocks as Statements and Expressions	30
3	Catching Compile Time Errors	33
3.1	Contexts	34
3.2	Type Checking	36
3.2.1	AST type annotations	37
3.2.2	type_check()	38
3.3	UtilEvalData and User Input	38
4	Catching Runtime Errors	41

4.1	Inserting the Runtime Check	41
4.1.1	Runtime Errors for the Division Block	44
4.2	Runtime Debugging Blocks	45
4.2.1	Breakpoints	46
4.2.2	For Advanced Data Types	46
4.3	Suspending Code	47
5	Future Work	49
5.1	Front end Internal Name Management	49
5.2	Typed Variable Blocks	50
5.3	Type Checking Code with Side Effects	51
5.4	Extending Code Suspension to Create Do	52
5.5	Integration with Incremental Compile	53
6	Applications and Closing Thoughts	55
6.1	StarLogo Nova as a language workbench	55
6.2	Applications for other block-based programming languages	56
6.3	Conclusion	56
A	Creating a New Block with Type-Checking	57

List of Figures

1-1	The StarLogo Nova User Interface	14
1-2	Various blocks by type and connectors	18
1-3	Erroneous code in other block-based programming languages.	22
2-1	A block stack and a visual representation of its AST	28
3-1	Example errors caught at compile time	33
3-2	Knowing the in-context agent is useful for compile time error checking.	35
3-3	A simple code injection attack on StarLogo Nova	39
4-1	Before and after internal representation of a collection of <code>ASTNodes</code> being processed by the semantic analysis compiler pass	43
4-2	Some new runtime debugging blocks	45
4-3	An example type checking procedure for a struct	46
5-1	A block stack with a side effect (left) and the type checked block stack	51
5-2	A more complicated example of type checking a function with side effects	52

Listings

2.1	A high-level view of the StarLogo Nova compiler	26
2.2	The CalcProduct block's yielding and non-yielding code	29

Chapter 1

The StarLogo Nova Environment

1.1 Background and Problem Statement

StarLogo Nova is a block-based programming language created at The Education Arcade and the Scheller Teacher Education Program (STEP) at MIT [8]. Users can connect program components like puzzle pieces, allowing them to quickly create 3D games or simulations with hundreds or thousands of agents. These puzzle-piece-like blocks allow users to “drag and drop” program components together, meaning users need not memorize the function names or the syntax of a traditional text-based programming language like Python or C.

Block-based programming languages, like Scratch, Blockly, and StarLogo Nova, are designed to be intuitive for novice users. StarLogo Nova in particular is targeted towards an audience of upper elementary to high school students and their teachers [8][9], who are not expected to have had any training with traditional programming languages before using the application. Thus, it is crucial that the application be easy to use and understand for users.

However, a lack of an error checking system has run counter to this goal. Users can create programs that do not behave as expected not because of an incorrect algorithm, but as the result of invalid code. This invalid code can cause errors that cause the application to behave unexpectedly – for example, when a user attempts to turn an object red by inputting “red” as a string instead of as a special color block, the object

would appear white without any warning to the user about why it did so, because the code failed the color assignment silently. Errors like this detract from the user experience. To address these issues, this thesis implements a number of automated checks to catch errors at different stages of the process of making a StarLogo Nova program.

1.2 Components of StarLogo Nova

For the purposes of this thesis, StarLogo Nova’s project interface has three major components: WebLand, the Workspace, and the Error Reporting Console. In Figure 1-1, these components are labeled 1, 2, and 3, respectively.

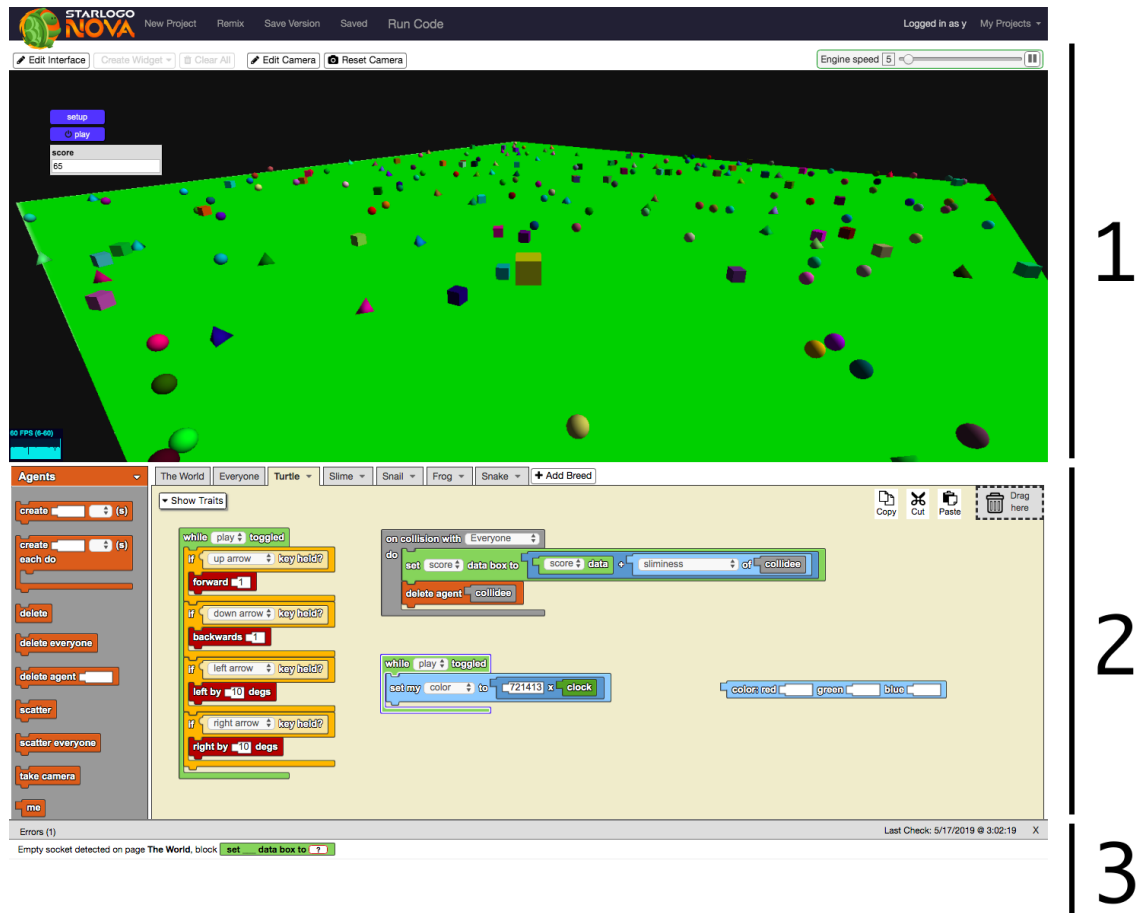


Figure 1-1: The StarLogo Nova User Interface

1.2.1 WebLand

WebLand is a 3D space where the user's program is executed. Objects in StarLogo Nova, called agents, appear here, as well as a large square green terrain and data containers called widgets. WebLand can support thousands of agents in real time, with even better performance depending on a user's computer specifications.

Widgets

Widgets appear on top of the WebLand display as 2D buttons or data. They either display data or accept user input. Blocks related to widgets appear in the Interface drawer of the Workspace. (Drawers will be discussed further in section 1.2.2).

Widgets that display data include Labels, which present read-only text to the user; Data Boxes, which present user-editable text to user; Sliders, which present numerical data that can be changed with a continuous slider; and Line Graphs and Tables, which present associated data to the user in a graphical form. All of these have values that can read or set by blocks in the code. They can also be hidden or shown by the code.

The other two widgets, which accept user input without displaying data, are Push Buttons and Toggle Buttons. These widgets are only used to run code when they are pressed. A Push Button will run code in its block stack once every time it is clicked by the user, while a Toggle Button will run code in its block stack continuously after it is clicked by the user until it is clicked again. A computer power icon (a circle with a line through its top) will appear green if the Toggle Button is on, or a dim black if the Toggle Button is off.

1.2.2 Workspace

The workspace is where users create programs to control what happens in WebLand. It has a large yellow canvas with tabs (or pages) corresponding to breeds, and drawers filled with blocks that users drag and drop onto the canvas to create their programs.

Breeds

Breeds in StarLogo Nova are roughly equivalent to classes in other programming languages, while the individual agents that appear in WebLand are most akin to instances of a breed. Every created Agent must be created as an instance of a given Breed, and certain blocks can reference agents by breed. Each breed has traits, or instance variables; procedures, or methods; and stacks of code created by the user that each instance of the Breed will execute when the StarLogo program is ran. The code that each breed runs will appear in the large yellow area that makes up most of the workspace when the corresponding tab with the breed's name is clicked, called the canvas.

There is a special “The World” breed that cannot be created or deleted by the user. It is unique in that one and only one copy of it exists at all times; in some sense, it can be considered both a breed and an agent, following a singleton pattern. It cannot interact with other objects, but it does have its own traits. It is often used as a sort of `main()` function to set up other agents, as it is the only agent that exists before a program is first run.

There is a very limited concept of inheritance in StarLogo: there is an “Everyone” breed that can be used to reference every kind of breed at once or to give traits or procedures to every breed. Breeds cannot inherit traits from other breeds besides the Everyone breed. There is also no concept of static (class) variables, although a determined programmer could use a trait of The World breed as a sort of global variable to work around that problem with good management. Similarly, static functions could be replicated by procedures in the Everyone breed by referencing the “static variables” held by The World. That said, these workarounds go beyond what the typical user of StarLogo Nova is expected to learn, and a variety of interesting simulations can be created without leveraging the full power of a inheritance-based object oriented programming language.

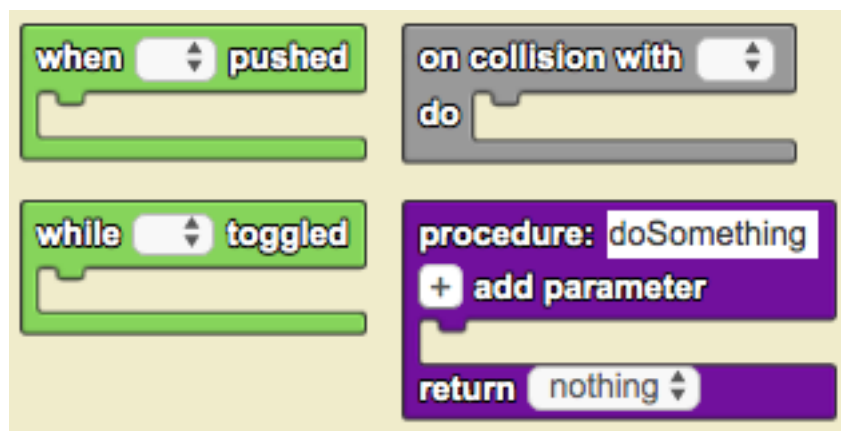
Drawers

Drawers are the gray containers of blocks that appear to the left of the canvas. The dropdown at the top of the drawer can be used to change drawers, or the user can scroll in the drawer area to seamlessly move between drawers. Each drawer contains blocks that are related to specific actions, and each blocks is color-coded corresponding to the drawer in which it can be found.

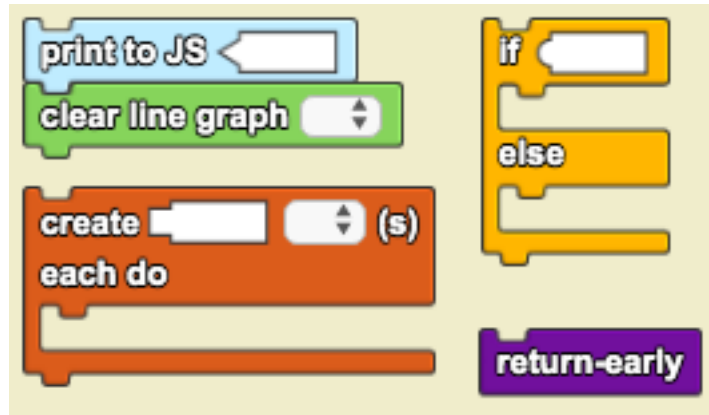
The Drawers in StarLogo Nova are Agents, Detection (used for collisions between agents), Environment (used for the The World agent and the terrain), Interface (widgets), Keyboard, Lists, Logic, Math, Movement, Procedures, Sound, Traits, Variables (used for block stack variables), and Debugger.

Blocks

Blocks are quite literally the building blocks of code in StarLogo Nova. They define both program components and connections between those components in different ways. There are three major types of blocks: top blocks, command blocks, and argument blocks.



(a) Top Blocks



(b) Command Blocks



(c) Argument Blocks

Figure 1-2: Various blocks by type and connectors

Top blocks are entry points that mark the start of code to be executed. They lack puzzle-piece-like connectors of other blocks on their top and left sides, which reflects how they are not dependent on other code to execute before they do. Currently, there are four top blocks, all of which are present in Figure 1-2a – **When [X] Pushed**, **When [X] Toggled**, **On Collision With [X]**, and **Procedure** declaration. Typically, users create a push button widget to set up their environment and a toggle button widget to handle an agent’s actions while the program is run.

Code that does not begin with a top block is treated not unlike commented-out code, so it can be used for figuring out how to place blocks to perform an algorithm before putting them in an executable stack.

Command Blocks are equivalent to statements in a written programming language. They contain empty areas called sockets in which text or argument blocks (or occasionally a dropdown argument) can be placed, and they contain rectangular connectors that top blocks and most other command blocks connect to with their bottom connectors. The **Delete** block and procedure **Return Early** block do not

have bottom connectors, and thus end a code stack; but a code stack can end even if its bottommost block has a rectangular connector that doesn't connect to another block. These are the only kinds of connectors that do not need to be connected to another block for successful program execution.

Argument blocks, on the other hand, are most like expressions in a textual programming language. As expressed by their lack of a top connector, their information by itself is not useful to the program, but with their left connector, they can be inserted into an appropriate socket in another block to produce various useful code.

Note that different kinds of argument blocks can have triangular, square, or circular connectors, which constrain what sockets they can fit in. This serves as a first, preventative step against errors in a user program, because some semantically meaningless combinations cannot be connected to make an invalid program in the first place.

1.2.3 Error Reporting Console

The error reporting console displays errors to the user. For each error, it displays the block that that threw the error, as well as the argument that caused the problem if applicable. A user can then click on the line of the error to automatically move to the corresponding Breed tab with the block in question selected by the interface. At the beginning of this thesis, the error reporting console's functionality was mostly unimplemented, and it would only appear when code compilation did not begin because of an empty socket.

1.3 Error Checking

This section details the current state of error detection in StarLogo Nova and other block-based programming languages, to serve as both a motivation for the thesis and as a perspective from which the progress the thesis contributes to the field can be measured.

1.3.1 In StarLogo Nova

By nature of being a block-based programming language, StarLogo Nova prevents a number of potential user errors by construction. The blocks themselves, by being a proxy for written code, prevent syntactical errors such as a missed semicolon or incorrectly capitalized variable. They also promote stylistic consistency, as there is no need to decide how to indent code, to format function declarations, or to write multiple lines of code on the same line. The structure of the code is apparent just by looking at it, making it easier for others to understand it and help with the debugging process.

Block Connectors help prevent a number semantic errors. It is impossible to insert a procedure **Return Early** block as an **If** statement’s boolean argument, because the former only has a top connector while the latter requires a block with only a left connector. Similarly, it is impossible to insert a list as an argument to an **If** statement because the list’s left connector is triangular, while the **If** statement requires an argument with a circular left connector. Other arguments prevent semantic errors by having their arguments be pre-populated dropdowns; for example, the **Create Do** block has a dropdown argument that can be any breed created by the user (but not the Everyone or The World breeds).

However, this error prevention was not comprehensive, as the number of connectors does not cover all the possible range of data types that a user could create. Thus, a user can attempt to set the size of an object to the agent’s parent (rather than the size of the agent’s parent) or attempt to set the color of an object to the string “red” (rather than the color red). Prior to this thesis there were no protections from these errors, and the code was compiled and would fail silently and continue to run whenever such an error was encountered,

One potential solution would be to explicitly type all expressions and arguments, treating the socket connectors on blocks like function signatures in Java or TypeScript. However, such a direct approach would require more blocks and connectors to do what StarLogo Nova already does, and would require different connectors for

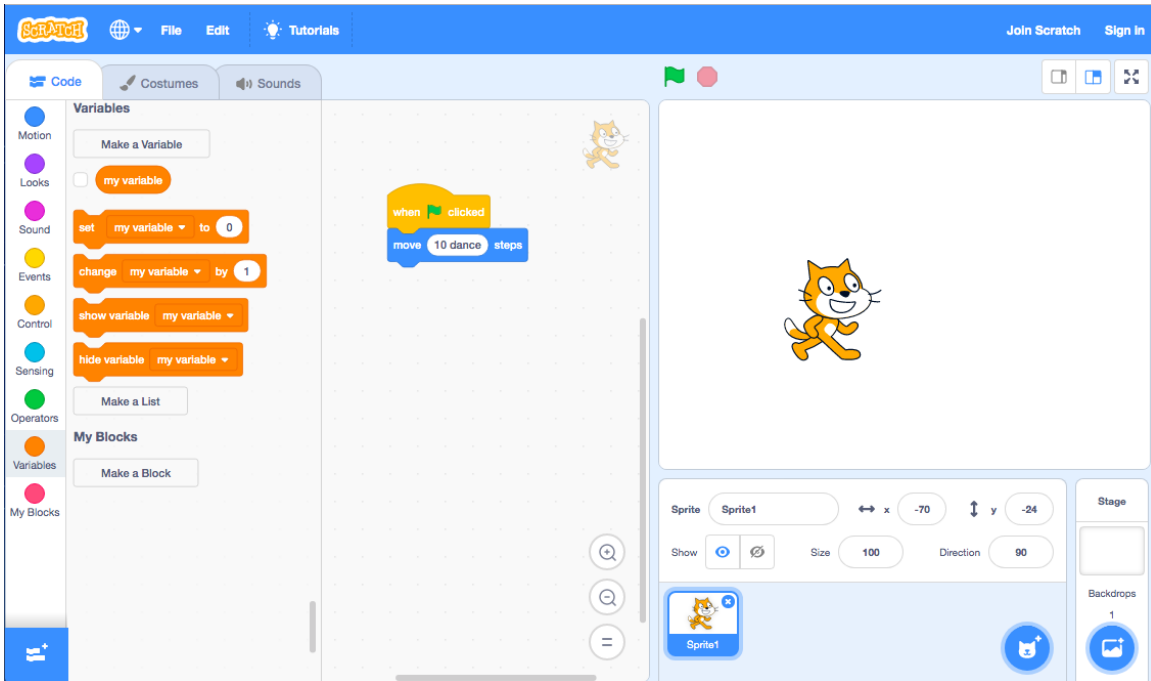
different kinds of lists, necessitating that users declare the types of each of their arguments without using them more flexibly like arrays. In fact, since there are an arbitrarily large number of such lists that a user can make, while there are a limited amount of visually distinguishable connectors, this approach would not scale. StarLogo Nova’s primary audience lies with students and teachers who may not have significant programming experience, so increasing the barrier to entry by requiring the user to learn more connectors and increasing the number of similar blocks that vary only by their connectors is counter-productive. The increased flexibility by not enforcing types with explicit declarations allows the language to be more user friendly, like Python or JavaScript.

Instead, by using a method that fosters productive confusion [2] by allowing the user to make mistakes and then providing direct feedback explaining them, StarLogo Nova aims to allow the user to learn about types as they start to branch off example projects and begin their own experiments with the program, and help them learn the programming fundamentals of type-safe and good code through actually writing code.

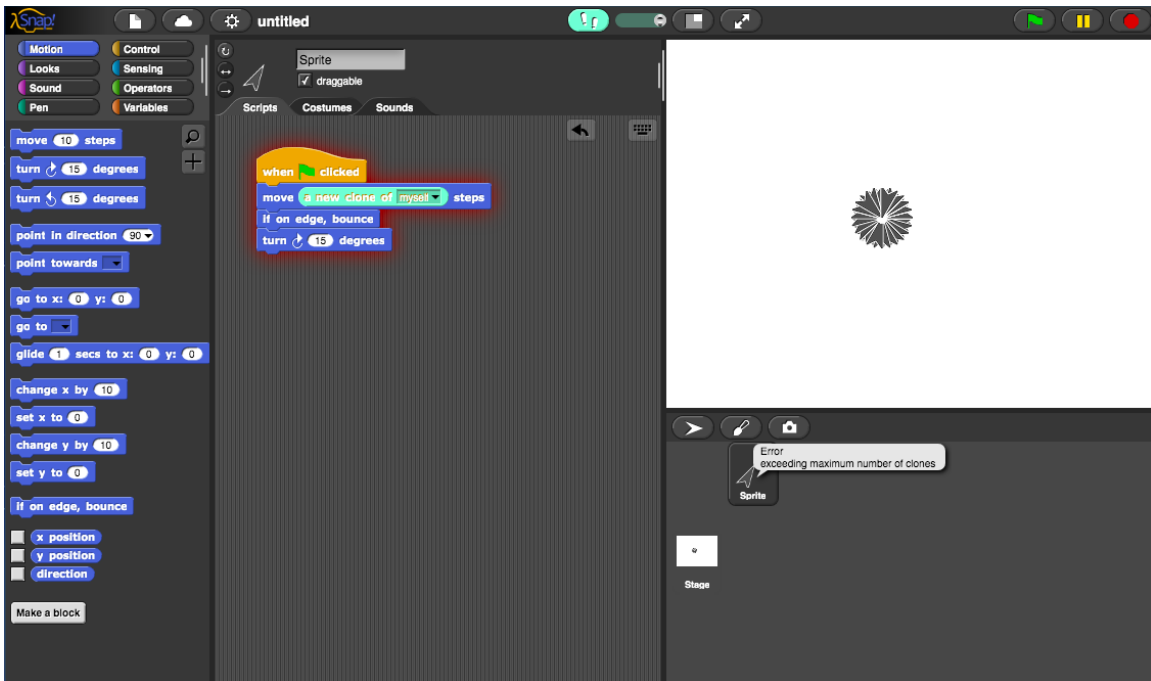
The front end does its best to warn users of potential errors before they even attempt to compile code – out-of-scope variables will appear as “**Invalid**,” and empty sockets will send errors to the error reporting console prior to code compilation. However, the front end cannot catch every error (it is unable to detect if potentially invalid code runs correctly based on undeclared invariants, and it cannot detect errors that occur during program runtime), and it ideally needs to work with the backend compiler to address every issue that users could encounter when writing or executing code.

1.3.2 In other programming languages

How do other block-based programming languages handle error handling and feedback? And can these approaches be applied to StarLogo Nova?



(a) MIT Lifelong Kindergarten Group's Scratch



(b) Snap!, a version of Scratch extended with first class data structures and lists

Figure 1-3: Erroneous code in other block-based programming languages.

(Note: this section is not intended to be a comprehensive analysis of error reporting and type safety in other programming languages, as that is an analysis best

performed by the developers of each system. Instead, it serves as a brief survey of various block-based programming languages and their approaches in the area.)

Scratch, developed by MIT’s Lifelong Kindergarten Group [3], is perhaps the most well-known of all block-based programming languages. It, like StarLogo Nova, relies heavily on front end protections against bad code. In Figure 1-3a, code that attempts to move the cat sprite (or agent, in StarLogo Nova terminology) forward “10 dance” steps will fail silently, but later code in the stack will still be executed. The front end will prevent users from typing in non-numeric characters, but copy-and-pasted characters do not undergo the same check. This is not specifically to say that copy-and-pasted inputs should be vetted before insertion, but instead that the scope of user errors may extend outside of what the developer expects, and that there is a need for backend protection in case a user accidentally makes bad code through unforeseen methods.

Similarly, Snap!, an extension of Scratch developed at UC Berkeley [6], can be told to move a sprite “a new clone of myself” steps. Here in Figure 1-3b, the code similarly fails silently, and while the sprite does not move forward, it does still create a clone of itself. Activating the code stack multiple times creates multiple sprites, and eventually the sprite will report an error about too many copies of the sprite appearing in the game at once – an error unrelated to the actual error (as it could come about via valid means of duplicating a sprite), but a good example of the error reporting system nevertheless.

Indeed, the lack of support for features like type checking and error reporting has spurred various research in the field of block-based programming languages. One such example is a team that is developing Kogi, a tool to make block-based programming environments from context-free grammars, using Blockly as the front end [10]. They identify a formalism of block syntax and a mechanism to identify error locations to users as the two major challenges for providing language workbench support for block based languages (that is, for providing support for tools that would allow one to make domain-specific languages with block-based platforms like StarLogo Nova; further discussion can be seen in Chapter 6). Although they ultimately decide to

create a block-based language from a context free grammar, this thesis approaches the problem from the other direction: given an existing block-based programming language where certain inputs do depend on context, how can it be extended to improve its block syntax or debugging capabilities?

Chapter 2 will define the role of the compiler and engine in StarLogo Nova's execution pipeline, Chapter 3 and 4 will discuss the implementation of tools and the different kinds of errors that these tools catch to catch, and Chapter 5 discusses future work that would further improve the error reporting abilities of StarLogo Nova's backend. Chapter 6 ends the thesis with reflections on the project and discussion about the results of the improved error checking system for both StarLogo Nova and other block-based programming languages.

Chapter 2

StarLogo Nova's Compiler

Compilers take code and translate that code into a more machine-understandable language. They mark the first point at which code is processed, and thus are used to catch errors (called compile time errors) even before a program is run. With its new error checker, StarLogo Nova similarly catches these compile time errors during compilation. It also determines where errors that cannot be determined at compile time can occur, and inserts code to check for these errors (called runtime errors) during compile time so that they can be caught during program execution. This circumvents the regular JavaScript behavior of failing silently and continuing code execution, which is helpful when not intending to show end users unexpected error messages, but which is not useful for program developers for languages like StarLogo Nova who wish to fix such erroneous code. Since the compiler is where most error handling is done and inserted, it is thus important to understand how compilation works to understand how the new error checking code works.

2.1 Program Flow

The compiler will begin working on a user's project when the user first loads a StarLogo Nova project or when the user presses the Run Code button at the top of the page. But before it is run, a front end check for empty block sockets is run, and if any are found, the code is not compiled and the Error Reporting Console will show

the offending blocks and sockets. If no empty sockets are found, the compiler's (or backend app's) `compileAll()` function is called.

The app's `compileAll()` function provides an excellent high-level view of the compiler, as shown in Listing 2.1:

```
public compileAll(): void {
  this.reset(false);
  for (let bp of BlocksReader.getTopBlocks()) {
    Compiler.createTopBlock(bp);
  }
  Compiler.compileAll();
}
```

Listing 2.1: A high-level view of the StarLogo Nova compiler

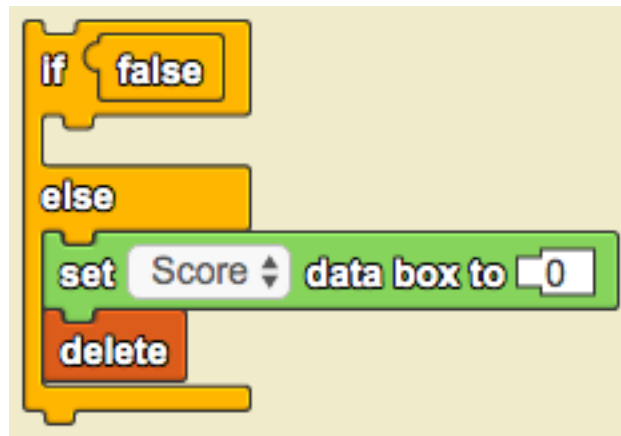
The command `this.reset()` clears the engine's saved internal state. The for loop finds the top block of every block stack on the page, and `createTopBlock()` creates a data structure called an Abstract Syntax Tree (AST) for each stack that begins with a valid top block (as defined in section 1.2.2.3, Blocks). Then the compiler's `Compiler.compileAll()` function uses the created AST to create threads for each block stack's code, which the engine uses to execute the user's program.

StarLogo Nova's AST is composed of a nested set of `ASTNode` objects, where each `ASTNode` corresponds to a particular block in the user's block program. Following recommended programming practice for ASTs in object-oriented programming languages like JavaScript [4], StarLogo Nova uses subclasses of the base `ASTNode` class for each block.

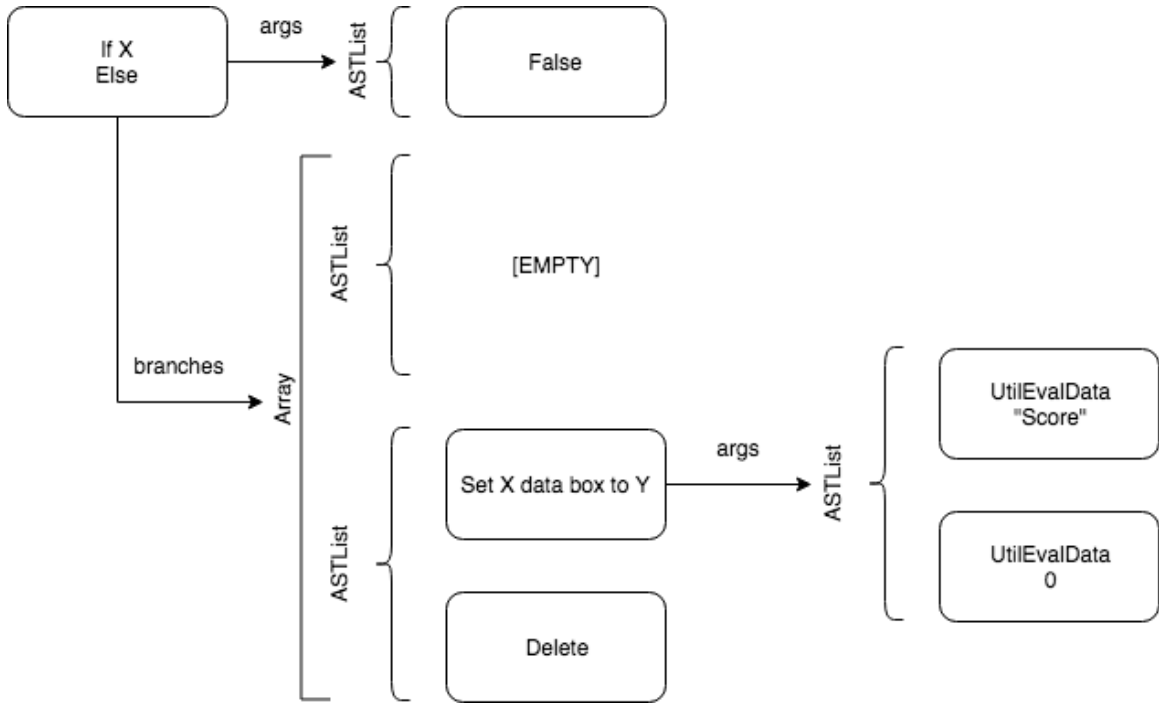
Note that despite the existence of a subclass of `ASTNode` for each block, it is not the case that these subclasses are automatically generated from the front end's implementation of drag-and-drop blocks. (That is to say, front-end error checking mechanisms, such as socket types, do not get transmitted to the compiler, which must perform error checking separately.) In theory, other types of blocks besides the scriptblocks that StarLogo Nova uses could be used with the program if they followed

the backend's API. The compiler is independent of the front end, and a file called `BlockTable.ts` is dedicated towards converting received blocks into the appropriate `ASTNode` for the backend.

When each AST is made, internal sockets of a block are represented in a field called `args` as a list of `ASTNodes` called an `ASTList`, while a series of blocks at the same indentation level (branches) are represented as a list of `ASTLists` (with each branch being its own `ASTList`). This list is only defined at the beginning of such a stack of blocks – blocks with branches, or internal bottom connectors like the **If Else** block or any top block. So, `ASTNodes` in the tree only contain references to blocks they directly contain. A visual representation for the AST for a contrived example of blocks is shown in Figure 2-1.



(a) An example block stack



(b) A cartoon of the AST for the block stack.

Figure 2-1: A block stack and a visual representation of its AST

2.2 Yielding Code Pipeline

The reader may wonder why the compiler would go through the trouble of navigating the block structure to make an AST if it subsequently just navigates the AST to make the threads that the compiler runs. Although the code modularity and architecture afforded by the polymorphisms of `ASTNode` may be convenient, is it worth constructing the AST in the first place? This architectural decision was likely chosen partly because it is an extendable design, but the choice serves to enable support of a particular feature of the compiler.

In particular, StarLogo Nova supports yielding code. Normally, the engine executes threads for each block stack sequentially, because there is no certain way of knowing if interleaving the threads nondeterministically would introduce race conditions for a given set of blocks, but the user can add points where code execution will yield with the `yield` block. This will pause execution of the current thread for the current `Engine.tick()` and resume the thread, with state variables intact, on the

next `Engine.tick()` call. (`Engine.tick()` is discussed in further detail in Chapter 4.) These threads are created with JavaScript generators, rather than usual functions, and sacrifice some speed for ensuring a consistent state through multiple yields.

This means that the code written for yielding code and non-yielding code is rather different! Non-yielding code can take advantage of the fact that it will run without interruption to avoid saving intermediate state, improving performance – especially in a simulation with thousands of agents. Yielding code must save its intermediate results in variables before performing calculations so that code can yield whenever necessary, requiring a setup step and an execution step. Since the same block can either not use yielding code or use yielding code depending on the other blocks in its stack, it is helpful to create the AST so that determining whether a code stack needs to yield or not is simply a recursive `ASTNode` function call away. The code cannot be compiled with just one sequential pass.

Perhaps the best illustration of this is an example from the code, such as that in Listing 2.2. Below is a code fragment from `CalcProduct.ts`, which corresponds to the multiplication block. If it is in a code stack that contains a yield block, it will compile as its `to_js_setup()` code followed by its `to_js_final()` code; if it is not, it will simply run its `to_js_no_yield()` code. It is imperative for the developers to ensure that all `to_js_setup()` and `to_js_final()` code calls are consistently written.

```
public to_js_no_yield(): string {
    return `
    (${this.args.data[0].to_js_no_yield()}*
    ${this.args.data[1].to_js_no_yield()})
    `;
}
```

```
public to_js_setup(): string {
    return `
    ${this.args.data[0].to_js_setup()};
    ${this.args.data[1].to_js_setup()};
    `;
}
```

```

    let __wl_${this.id}_a = ${this.args.data[0].to_js_final()};
    let __wl_${this.id}_b = ${this.args.data[1].to_js_final()};
    ‘;
}

public to_js_final(): string {
    return ‘
    __wl_${this.id}_a * __wl_${this.id}_b
    ‘;
}

```

Listing 2.2: The CalcProduct block’s yielding and non-yielding code

The `to_js_setup()` function sets up both of its arguments, and then saves them to variables so that they will be saved by the javascript generator running the code when it yields. The `to_js_final()` function then takes the saved variables and calculates the product. The `to_js_no_yield()` function is similar to the `to_js_final()` function in this case, but it can be run directly without any need for `to_js_setup()`.

2.3 Blocks as Statements and Expressions

As the template literal return values Listing 2.2’s code sample suggest, block stack code is compiled by passing a template literal to a JavaScript Function or Generator object. The embedded expressions of the top block call the embedded expressions of each of its arguments, and so on, until the final code thread is fully expanded.

Interestingly, this means that that blocks may return template literals that are statements or template literals that are expressions. For instance, the `to_js_setup()` function in Listing 2.2 returns four statements that are punctuated with semicolons, while the `to_js_final()` and `to_js_no_yield()` functions return an expression – the two arguments of the `CalcProduct` block evaluated and multiplied together. Other

blocks may have `to_js_final()` or `to_js_no_yield()` functions that return statements, but `to_js_setup()` will never return an expression. This construction complicates error checking, because a type checking statement or statements cannot be inserted when an expression is expected without causing syntax errors in the compiled code.

Chapter 3

Catching Compile Time Errors

StarLogo Nova converts block stacks into ASTs, and then uses those ASTs in order to generate javascript that executes the commands each block stack of the user's program. However, the ASTs can be further modified before they are used to create code – and this is the approach used by multi-pass compilers for compiling code. In particular, in this thesis a new function called `Compiler.semanticAnalysis()` is added to the code after the block stack ASTs are made and before they are compiled to perform compile time checks that the backend previous lacked. This will be referred to as the semantic analysis pass of the compiler in the paper.

The semantic analysis pass can catch and report on a number of different errors before code execution even begins; a selection of which can be found in Figure 3-1.

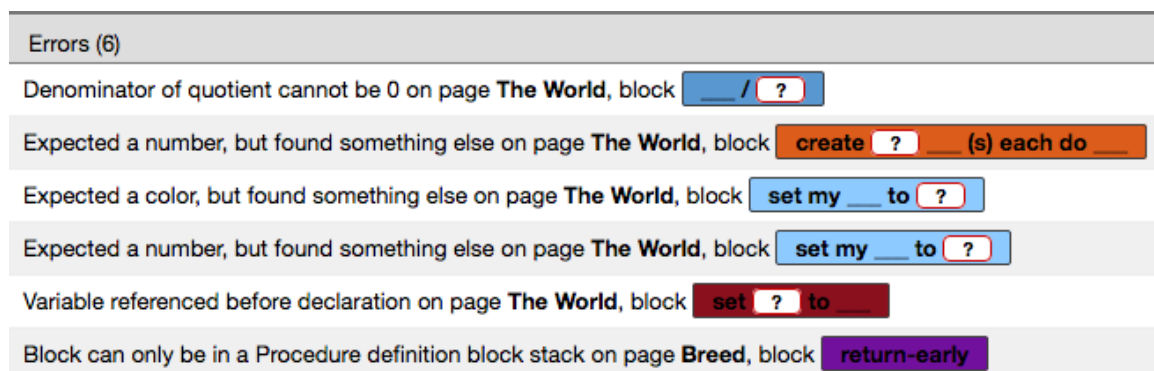


Figure 3-1: Example errors caught at compile time

The semantic analysis pass, like the compiler, gets each top block's `ASTNode` and calls a helper function called `typeCheckNode()` to process its internal and child nodes recursively. It maintains a record of state called a `Context` that is passed to each call, which is discussed further in Chapter 3.1. It also calls each node's `type_check()` function to perform compile time error checking if possible (Chapter 3.2). Chapter 3.3 takes a brief detour and discusses compile time protections against code injection, before the last major function of `type_check()` and `typeCheckNode()`, the ability to insert runtime checks into the code, is discussed in Chapter 4.1.

3.1 Contexts

In isolation, all blocks produce correct code given that their preconditions are met. Errors occur when users attempt to interact with uninitialized program state (such as an undeclared variable or a procedure that belongs to a different block stack) or when blocks receive arguments that do not match their function signature. Thus, each `ASTNode` needs to know relevant information about where it is and what functions it can and cannot use. A data structure called a `Context` was created to represent this information. The first part of `typeCheckNode()` uses the `Context` to catch and throw errors that occur because of incorrect program state, like accessing an uninitialized variable or using a return-early procedure call in a non-procedure stack.

To more concretely explain the problem `Contexts` address, consider Figure 3-2. `World_num` is a trait of the The World breed, `breed1_score` is a trait of Breed 1, and `breed2_score` is a trait of Breed 2, while `doThing` is a procedure either on Breed 1's page or on the Everyone page. The block stack is a part of the The World breed's page.

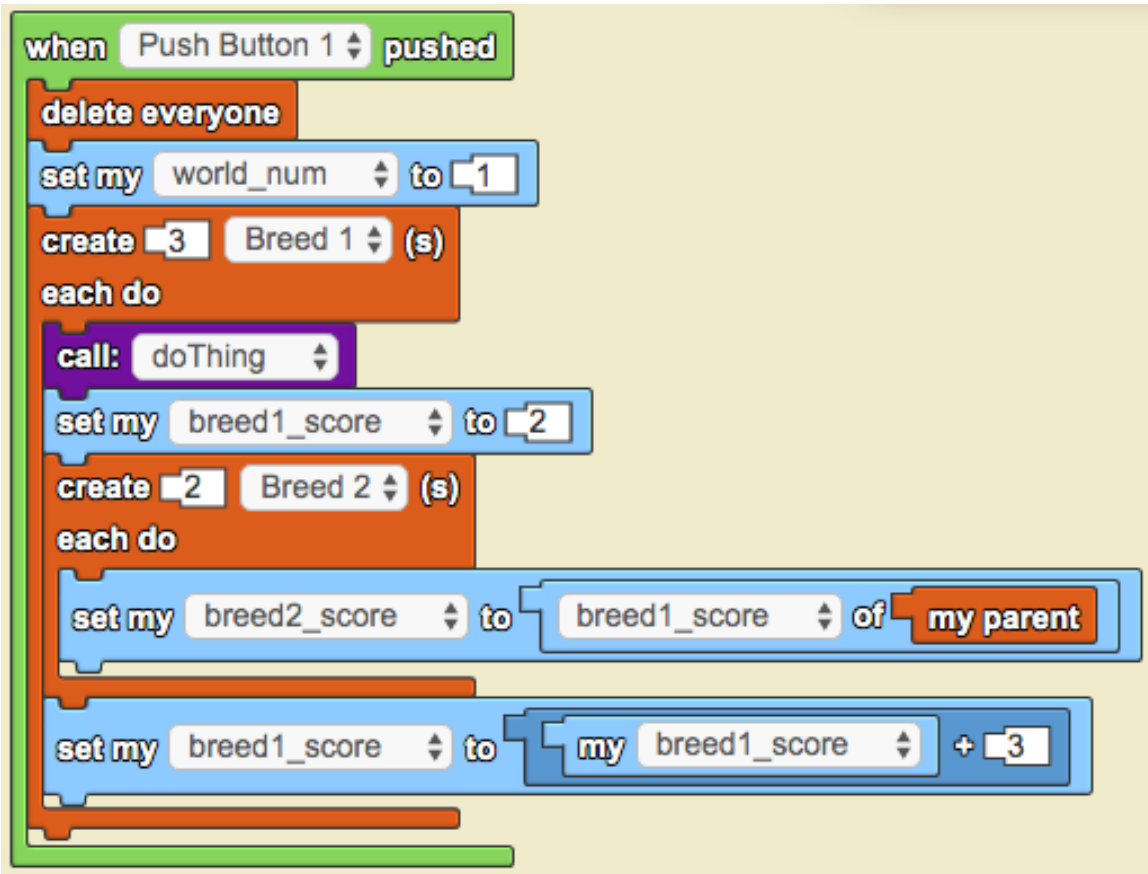


Figure 3-2: Knowing the in-context agent is useful for compile time error checking.

The block stack in Figure 3-2 is semantically correct. The **Delete Everyone** and **Set my [world_num] to [1]** blocks contextually belong to The World, but the **Create Do** block after that changes the references and procedures inside of it to that of the breed it creates; in this case, Breed 1. The **Create Do** block inside this **Create Do** block contextually refers to Breed 2 instead. Thus, for example, the **Set my [breed1_score] to [2]** block cannot set either world_num or breed2_score, because it is a direct descendant of the **Create Do** block for Breed 1.

Depending on whether the doThing procedure is defined on the Everyone breed page or the Breed1 breed page, it either can or cannot be referred to in Breed 2's **Create Do** block.

The **Context** keeps a record of the currently in-scope agent, collidEE (if any), variable list, and top block name. The in-scope agent handles references to procedures

or traits (and the occasional reference to the **Me** agent block), while the collidee handles references to a collidee’s procedures and traits. Variables, which can be defined in a block stack and only used in that particular block stack, are handled by the variable list. The top block name is used in a single scenario – preventing the procedure **Return Early** block from appearing in block stack with a collision or widget top block – but its use can be easily extended if additional types of blocks that required specific top blocks were to be created in the future.

The **Context** could potentially use an additional field for the agent that generated the current in-scope agent in the processed block stack, if applicable. This would enable references to the **My Parent** agent could be resolved in compile time in a limited number of cases. The feature was not implemented because it is anticipated that most references to the **My Parent** agent would either be resolved by using “The World” instead or would occur on individual Breed canvases outside of the contrived nested create-do pattern shown in Figure 3-2. In other words, such an addition to the **Context** addresses an issue that most users do not face, and would not catch a majority of the cases that the users face. Though the feature was not added because the additional storage and maintenance overhead were deemed to outweigh the benefit, this could prove to be a useful addition to Contexts if the situation changes in the future. Instead, the validity of properties related to the **My Parent** block are passed to runtime to be processed then.

3.2 Type Checking

The base AST class accepts an ordered list of **ASTNodes** (that is, an **ASTList**) for its internal arguments and a list of **ASTLists** for its branches. Even when each block’s class extends **ASTNode**, they do not further type their arguments. New type annotations and a new **type_check()** function were added to **ASTNode** and implemented in each block’s subclass so that each block’s type requirements are checked in compile time, or so that a runtime check can be inserted into the AST if the check cannot be completed in compile time.

3.2.1 AST type annotations

The base `ASTNode` class now contains boolean type annotation functions such as `is_number()` or `is_breed()` that return `true` when their return value matches a particular type. For many blocks, these function merely return a fixed truthy value – e.g. the **My Parent** block will always return an `Agent` (even the singleton `The World` trait reports itself as its parent), and the **Pi** block will always return a number. Idiomatically, the intention of the block is important for these functions: for instance, the division and remainder blocks have an `is_number()` function that always returns `true`, even though the underlying code would return an `NaN` value or a technically-numeric-but-not-really `Infinity` value if their second arguments were 0. These division blocks type check their arguments, so they defer to their `type_check()` functions to determine if an argument is invalid. Although the `is_number()` return value could be set to be `true` if and only if the arguments were valid, implementing it like that could potentially cause the error checking system to report two errors: one for the division block itself, and one for whatever argument in the division block is invalid. The approach of following block intention prevents any blocks that the division block is nested in from reporting a redundant type error to the user, which is helpful as multiple error reports for the same block are likely to confuse the user into thinking there are multiple errors.

`ASTNodes` are also annotated with a `can_change_type()` function. This function returns `true` if the output of the block is not constrained to a particular type. A random integer block will return `false` for `can_change_type()`, even though its output can change, while a data box widget value will return `true`, even if the code only ever sets it to a numeric value. This function is used by the type checker to determine if compile time are sufficient: if a nested argument is marked as `can_change_type()` by a `type_check()` function, it means that the type could change during runtime, and that the compile error check must be passed.

3.2.2 `type_check()`

The `type_check()` uses the AST annotations are discussion in section 3.2.1 to determine when and where to throw compile errors. A typical `type_check()` function will check its arguments one at a time, and throw a compile time error if the argument fails both the `can_change_type()` function and the corresponding boolean type checking function (such as `is_number()`). Sometimes additional checks are run – for instance, if the denominator of the division block cannot change type, it is evaluated and a compile time error is thrown if its divisor is 0.

Perhaps the most curious and powerful feature of `type_check()` is that it can perform limited type coercion. Consider a user who attempts to set an agent’s color to the string “red.” The `type_check()` function for the **Set my Trait** block runs an `is_color()` check on its second argument, which happens to be a `UtilEvalData` block (where `UtilEvalData` is inserted as a wrapper for all non-block arguments as a part of the `createTopBlocks` for loop before the `semanticAnalysis()` call). When `UtilEvalData` calls its `is_color()` function, it will convert strings such as “red” to the appropriate internal representation for the color, because the block that contains it has effectively said what role the `UtilEvalData` plays in the user’s code. A non-fatal error can be thrown during this type coercion to guide the user towards the more idiomatic way of expressing colors with a dedicated color block. (Ideally, non-fatal errors will appear as a separate kind of Warning as a better way of communicating to the user that they are non-fatal.)

A number of new convenience functions, such as `type_check_arg_is_color()`, were also created to type check individual arguments and throw compile errors to the Error Reporting Console, so that developers can focus on specifying the arguments and their required types at a higher level for `type_check()`.

3.3 `UtilEvalData` and User Input

StarLogo Nova processes user input by enclosing it in double quotes and passing it to its enclosing block for processing. However, it previously did not sanitize user inputs,

so special characters like backslash (\) would not appear. Similarly, unintended characters such as the newline character (\n) would appear. This was fixed by escaping those special characters with backslashes – backslashes first, then double quotes, so that the escape character for the double quotes would not itself be escaped afterward.

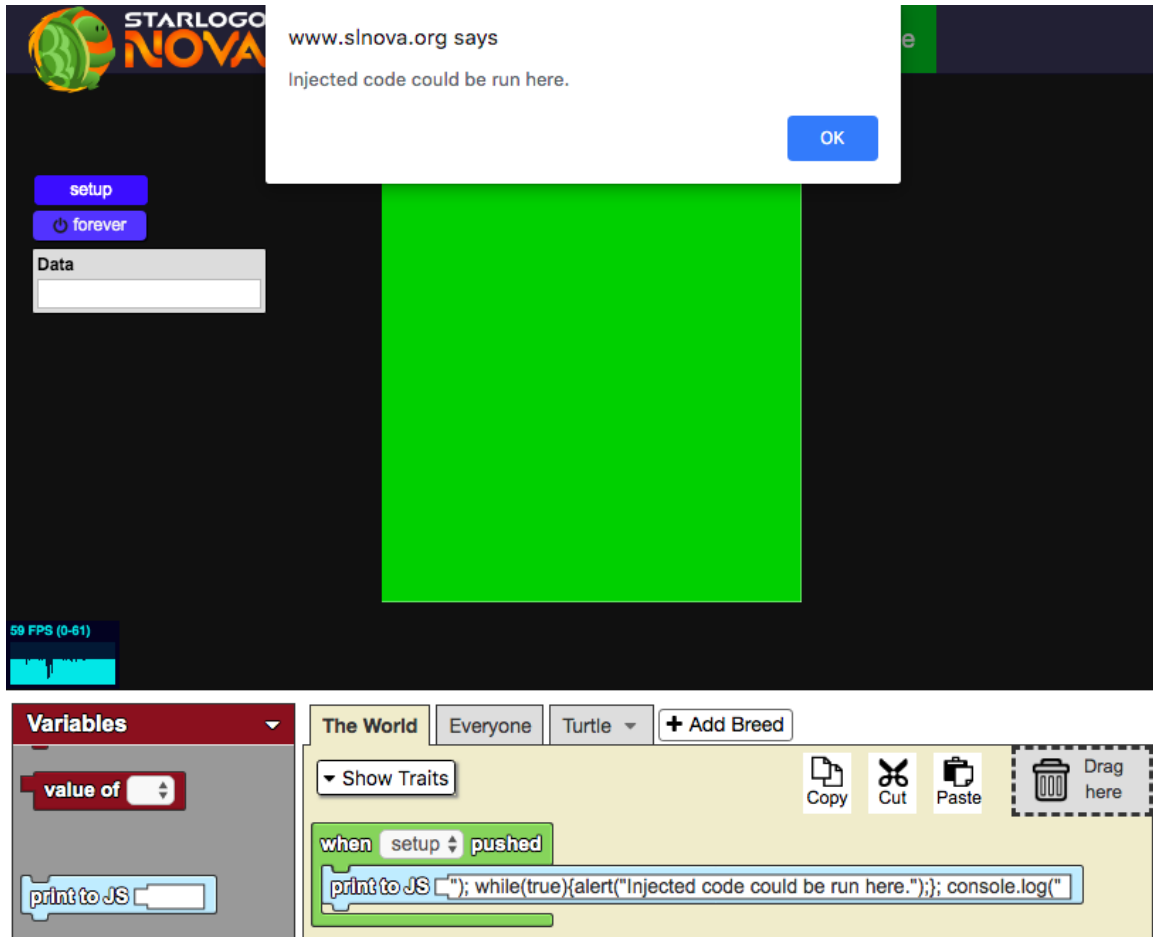


Figure 3-3: A simple code injection attack on StarLogo Nova

Sanitizing the inputted code also prevents a class of code injection attacks against users, as with the proper input, a malicious user could have the engine execute arbitrary javascript, as shown in Figure 3-3. If the creator of the malicious code shared their code with other players in Presentation Mode, where blocks do not appear so as to wholly commit the focus to WebLand, they could execute malicious javascript in the players' browsers without the users being able to see how or why malicious code was being executed on their machines.

Chapter 4

Catching Runtime Errors

Not every potential error in the code can be caught during the compilation step, so the runtime Engine also contributes to error checking in the system. Section 4.1 outlines how the Engine encounters and evaluates runtime checking code when it executes a user's code, and Section 4.2 discusses how the same tools that the Engine uses to type check code have been implemented as blocks so that users can run their own runtime error checks. (These newly created blocks like those in Figure 4-2 are able to fit the system's code execution paradigm and can be easily used by either the compiler or by any user curious enough to try out more advanced debugging features.) Section 4.3 takes a step back to look at the specifics of execution beyond running user code, and discusses the execution of the Engine itself and its thread management in the face of errors.

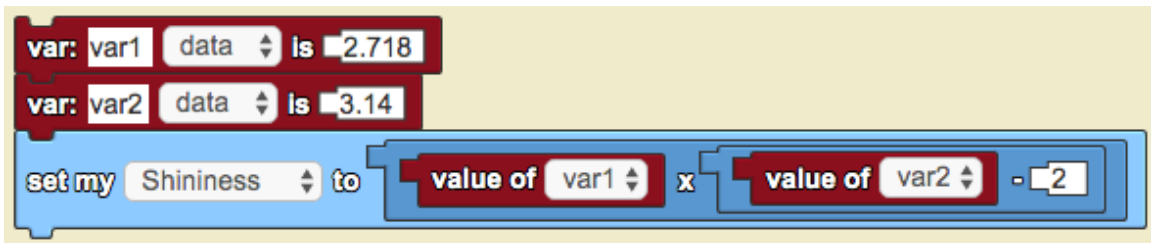
4.1 Inserting the Runtime Check

Although `type_check()` as discussed in Chapter 3 is useful for catching a number of errors at compile time, it cannot catch every error. StarLogo Nova has a number of blocks that can return untyped values (called "data") that can change during program execution. For instance, a block that sets an agent's size to the value of a widget may work at compile time if the widget's value is numeric, but there is nothing stopping the user from changing the widget value to a non-numeric string during program

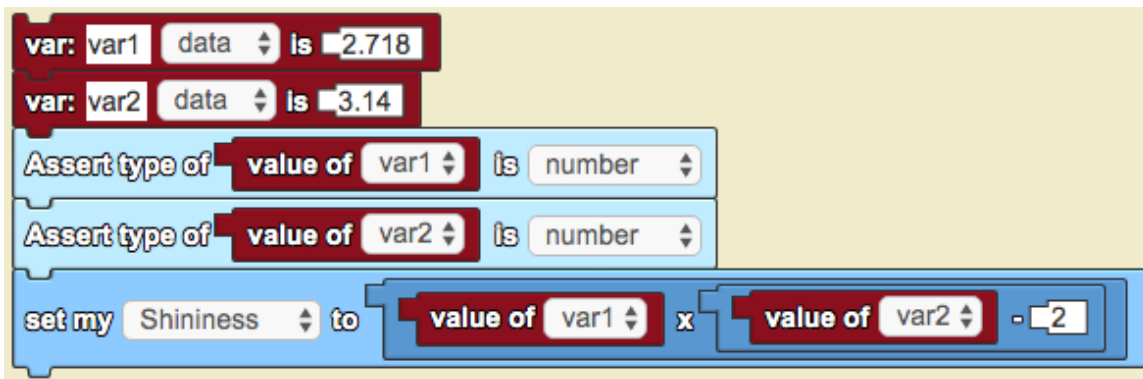
execution. This would cause what is called a runtime error. The compiler determines what blocks can change type during program execution and inserts type checking code into the AST so that these errors will be caught if and when they come up.

Most of the work managing runtime errors happens before the Engine begins executing. The `type_check()` function in section 3.2.2 not only throws compile errors, but also adds runtime error checks to the AST when compile time checking is impossible (when `can_change_type()` returns `true`). However, care must be taken as runtime code checking insertion requires error checking statements (where statements are as defined in section 2.3) that can exit the containing thread, which cannot be inserted into code as statements (because they cannot fit grammatically) or as first-class JavaScript functions (because such functions can end their own execution, but not that of their containing thread).

Figure 4-1a displays potential user code, while Figure 4-1b represents code that has been modified to include type checking blocks. The compiler should convert its internal AST for code like that in Figure 4-1a into an internal AST that would be equivalent to that of Figure 4-1b after the `semanticAnalysis()` pass is complete. As the type checking code itself is a collection of statements, it can be represented as a command block; and as a command block, it cannot fit where argument blocks would go. Thus, while the **Value of [var1]** block needs to look to insert its type checking code before its grandparent, the **Value of [var2]** block needs to look to its great-grandparent.



(a) Before type check



(b) Code passed from the compile step

Figure 4-1: Before and after internal representation of a collection of `ASTNodes` being processed by the semantic analysis compiler pass

The natural approach to this problem is to insert the runtime checking command block immediately prior to whatever command block contains it. In Figure 4-1, this would correspond to inserting the runtime checking `Assert` statements before the `Set my [Shininess] to [X]` command block that contains the variable `Value of [X]` blocks. Invalid blocks do not know how nested they are (so `Value of [var1]` doesn't know it's nested two blocks deep, and `Value of [var2]` doesn't know it's nested three blocks deep), and keeping and passing references to the last command block throughout the navigation of the AST would introduce some tricky overhead into the runtime check insertion algorithm.

The solution was instead to propagate the error upwards by returning the errors from the `type_check()` function. This function returns an `Array` of `RuntimeCheck Packet` structs, which contain a `nodeToCheck` field as the expression to evaluate (in Figure 4-1, those would be the `Value of [X]` variable blocks), a type which specifies what type the `nodeToCheck` should evaluate to (“number” in the figure), and a `reportingBlockID` and `reportingBlockSocket` that allow the error to be reported at the right place (the first socket of the multiplication and subtraction blocks in the figure). (There is a small class of user-created blocks that can clash with this implementation when `nodeToCheck` has a side effect, which is discussed further in Section 5.3.)

This error list is returned, propagated up the tree, and combined with error lists generated from its parent nodes until it reaches an `ASTList`. This takes advantage of the architectural design of the blocks: A top block always contains an `ASTList` branch for the stack of command blocks beneath it, top blocks do not contain sockets that require type checking to propagate upwards (as they are typed by a dropdown, effectively an `enum`), and branches are always navigated to after arguments in separate loops for each block. This all means that there will always be an `ASTList` navigation call ancestor somewhere in the call stack for the list of `RuntimeCheckNodes` to propagate up to, and that the two loops that navigate sockets and that navigate branches can have code that appropriately passes the `RuntimeCheckNodes` or that converts them into `ASTNodes` for runtime checking.

Once the `ASTList` has been reached, the list of `RuntimeCheckNodes` is stored in a map with its key being the index of block it came from. Then, once the `ASTList` has been fully processed (and not before, so as to not modify a list while it is navigated through with a `for` loop), the map's key set is walked through backwards, with the `RuntimeCheckNodes` popped and inserted into the index indicated by the corresponding key. By inserting the `RuntimeCheckNodes` in this order, there is no need to worry about shifting the index where the next `RuntimeCheckNode` should be inserted – it will always be inserted at the index of the correct block, appearing right before it and shifting all of the later, already-processed nodes to an incremented index, but not affecting the indices of the `ASTNodes` that come before it.

4.1.1 Runtime Errors for the Division Block

The one exception to this method of type checking is the division block. Unlike other blocks, which need runtime checks if their values can change types, the remainder and quotient blocks need to perform a runtime check so long as its denominator can change at all. Changing the implementation of the `can_change_type()` function would cause runtime checks to be inserted in other parts of the code where they aren't needed, or would require a new set of similar block annotations, which could make the annotation process more confusing for developers.

Ultimately, this runtime check for the division block takes advantage of the yielding pipeline outlined in section 2.2, putting the denominator 0 check in the setup function and requiring the setup/final pathway be taken. (The check for numerical arguments, if applicable, is still inserted into the AST as described in the previous section.) Though this solution is not ideal for most blocks – generators and setup functions create more overhead for slower code – the solution’s negative effects are minimized by being localized to one block, while the positive results include a decreased architectural overhead for the codebase.

4.2 Runtime Debugging Blocks

Since the runtime checks required the development of `ASTNode` classes that naturally fit the paradigm of adding blocks to a program, they could be easily developed into blocks for users to leverage, as well. New blocks developed from these new `ASTNode` subclasses were added to the Debugger drawer of the StarLogo Nova workspace, and a selection of these blocks are shown in Figure 4-2. Most are fairly self-explanatory: **Assert** will throw a runtime error when its condition is not met, while the blocks with round boolean connectors return truthy values depending on whether their conditions are met. **Trait** is a universal trait block that bypasses the front end’s protections against using out-of-context traits, and as of such is simultaneously a useful debugging tool and also a block that is best kept away from most users’ workspaces.

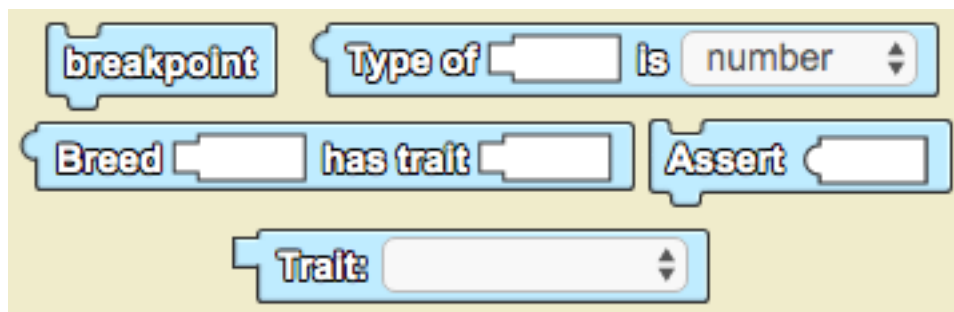


Figure 4-2: Some new runtime debugging blocks

4.2.1 Breakpoints

The breakpoint block will pause code execution in the virtual machine running the thread of the block stack it lies in. This enables advanced users and developers of StarLogo Nova to find their compiled code and step through it with their web browser's debugging tools. It also provides developers the chance to debug new blocks more easily and thoroughly, which should smooth development of new blocks in the future.

4.2.2 For Advanced Data Types

The ambitious user can use these debugging blocks to perform type checking on data structures such as structs or binary tree nodes, which are more advanced than the type checking the compiler itself looks for. Consider a struct of student data representing a student's name, student ID, and a breed (perhaps the students could be magicians, scholars, or knights, for instance).

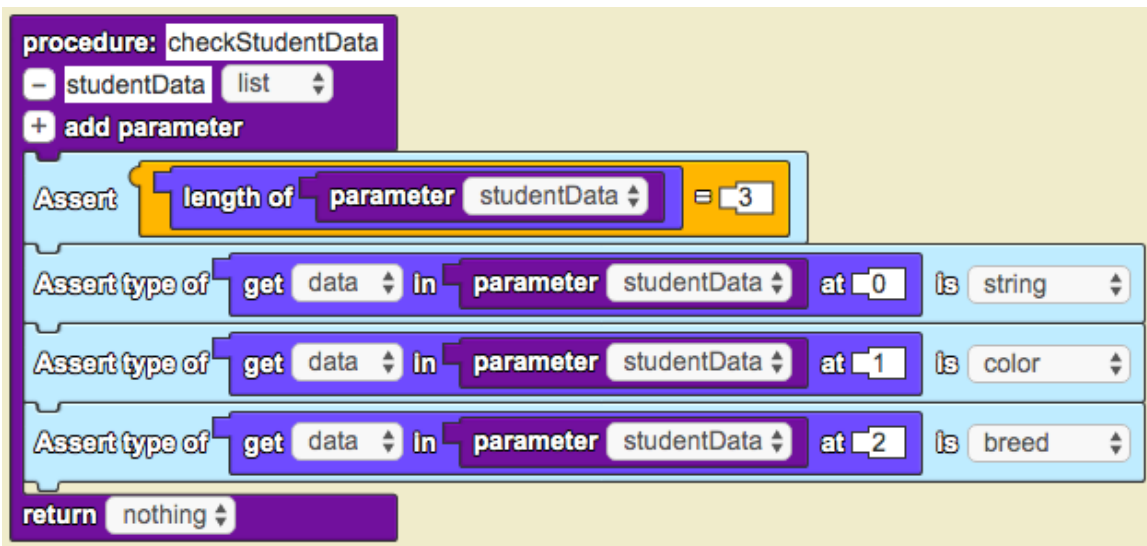


Figure 4-3: An example type checking procedure for a struct

An advanced project creator could create checks like this for creating and modifying a data structure and hide them with block macros called custom blocks, and then share projects with these custom blocks with other users of StarLogo Nova so

that they can enjoy creating programs that leverage the full power of advanced type protection without having to learn and use the debugging blocks to create it for themselves.

4.3 Suspending Code

Program execution in WebLand is managed by a `Viewport.animate()` function that calls `Engine.tick()` a number of times per second depending on the system's current Frames per Second (FPS). `Engine.tick()` goes through all compiled threads and executes them, providing updated states for each agent in WebLand so they can be drawn and animated. These threads must be marked as invalid when they have a compile error or runtime error, and then not run so as to avoid running erroneous code while still providing a best effort to run the user's remaining valid code.

When code fails during compilation, the `ASTNode` corresponding to the top block of the block stack gets marked so that the thread it generates sets a `hasFailed` field to `true`. Code that fails during code runtime sets the thread's `hasFailed` field to `true` and prematurely ends the thread's execution after it has sent a runtime error. When this `hasFailed` field is `true`, it acts as a flag for the engine to skip execution of the thread for the current step and to continue executing any other threads.

Right now, there is no way to unset the `hasFailed` flag other than resetting the compiler by recompiling the user's code. This behavior is desired for the current method of compilation because as there is no way to change the code in the compiler, the flagged code has an error and should not run.

It is also possible to suspend the entirety of WebLand upon an error so that no code stacks execute until a command is run in the web development console to unset a particular engine `isSuspended` flag. With a way to unset this flag via the UI, this would give users a chance to modify the thread that threw the error and restart program execution before an unknown number of `Engine.tick()` cycles without the failed code stack (between when the code stack failed and the user attempted to pause the program) brought the user's current simulation to a potentially unanticipated

intermediate state. When the **Engine** state is unsuspended, it will begin execution on the thread where it had last failed.

Like the **hasFailed** thread flag, the **isSuspended** engine flag looks forward to a future system that can re-compile during program execution rather than only before program execution. Further discussion about such a compilation method can be found in Chapter 5.5. Incremental Compile for StarLogo Nova could be the subject of a thesis by itself, and is outside the scope of this thesis.

Chapter 5

Future Work

This chapter documents issues with or future challenges for the development of type checking. Section 5.1 outlines a communication problem between the front end and the compiler, sections 5.2 and 5.3 outline problems or omissions from the type checking algorithm that were addressed over the course of the thesis that did not have fully completed implementations at the end, section 5.4 outlines an Engine issue that may affect code suspension during runtime, and section 5.5 discusses integration of type checking with an upcoming milestone feature of StarLogo Nova, incremental compile.

5.1 Front end Internal Name Management

The Workspace for StarLogo Nova uses automatically generated names to manage variables, widgets, breeds, and traits. So, for instance, a variable named “Score” may be referred to by the front end as “`node29`,” derived from the ID of the block that initially defined the variable in the block stack. Moving a Variable **Set [Score] to [input]** block from one block stack to another will cause the variable name “Score” to appear as Invalid, even if the new block stack has its own variable named “Score,” because internally the “Score” variable in the new block stack will have a different internal name (in this example, something different than “`node29`”).

However, this complicates the backend compiler’s ability to process user inputs. The backend cannot catch if duplicate variable names are used in the same block

stack, even though such a protection is built in to the backend; but worse, this means that the backend cannot attempt type coercion where it would be reasonable to do so. So, for instance, a user cannot calculate set an agent’s size to the string “Score,” because the compiler only recognizes that variable under a different name. They must use the Variable **Value of [VarName]** block to have the front end convert the variable name they know (“Score”) into the name that the front end uses internally (e.g. “node29”). While explicitly using the **Value of [VarName]** block is of course good form for getting a variable’s value anyway, it means that type coercion of the likes done with colors (changing strings like “red” to the color red when a color is expected) is impossible for the compiler unless the front end changes how it manages variable/trait/procedure names or unless the front end sends the missing user defined names to the compiler through other means.

5.2 Typed Variable Blocks

Variables can be declared as either lists, agents, or “data” (a generic type for any value). They do not support more specific types, such as numbers or colors.

The user can run **Assert** checks on variables after declaration to guarantee that they are a certain type, and the compiler will insert runtime **CheckTypeOf** blocks anywhere where such checks are necessary. So there is no need to have explicitly typed variables. However, explicitly typed variables would eliminate the need to insert runtime checks when using the variable’s value, as attempting to mistype the variable would cause a compilation error or a runtime error at an earlier point in the program (when the variable receives the bad value, rather than when the variable is used with the bad value); thus, the more explicit typing would slightly improve runtime performance. Although they were not implemented because more immediately impactful features were prioritized, typed variable blocks remains a natural extension to the work in the thesis.

5.3 Type Checking Code with Side Effects

Section 4.1 describes a method of type checking by inserting a copy of a block into an **Assert** statement before the command block the block needing type checking is inserted in. Though this method works for all pure functions, users can create Procedure blocks that are impure and have side effects. For instance, perhaps the user-created procedure could increment a Score widget in WebLand by 1 and return the incremented value. Then, the function would be called twice – once by the assert statement, and once by the code the user made – which could cause unexpected behavior in the user’s program.

A wide swatch of the errors caused by calling the Procedure block twice can be avoided by explicitly typing the procedure’s output, so as to avoid needing to insert a runtime check in the first place, but this is not necessarily sufficient to avoid all such errors. A workaround is to assign the procedure’s value to a variable, and then use the variable’s value in place of the procedure block – as the variable declaration will not need type checked, the procedure will only be called once, and when the variable itself is retrieved and type checked by the procedure’s original containing block, that block will type check the evaluated value stored in the variable, not calling the procedure again. An example is shown in Figure 5-1.

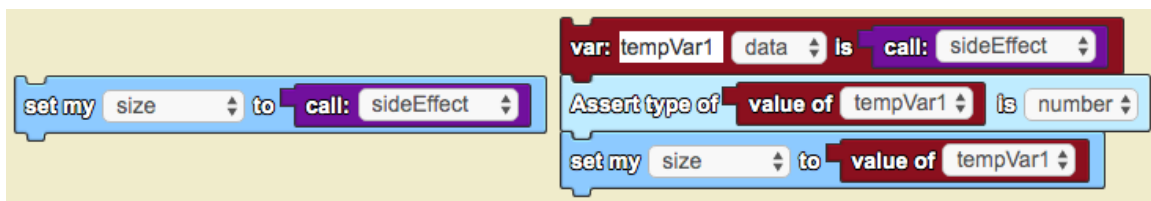


Figure 5-1: A block stack with a side effect (left) and the type checked block stack

This gets trickier when procedures with typed inputs get nested procedures (without typed outputs) as inputs – the most nested procedure calls would have to be stored in a variable first, and then less nested ones, and so on, so that each variable stores a procedure call that does not contain procedure calls. An example is shown in Figure 5-2.

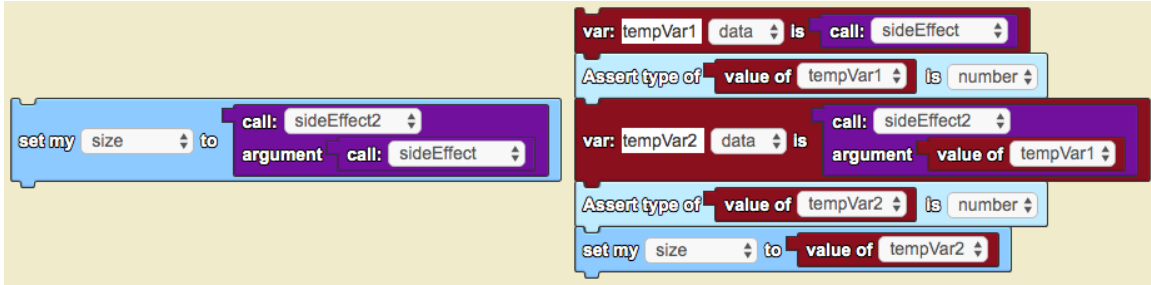


Figure 5-2: A more complicated example of type checking a function with side effects

Replacing nested procedure calls with variables before the containing procedure – that is, adding `tempVar1` before `tempVar2` in Figure 5-2 – could be handled programmatically by waiting until after an `ASTNode`'s `args` had finished processing before replacing the `nodeToCheck` with the `tempVar` variable. (That is, for example, waiting for the `args` check for the `sideEffect2` procedure call to end before replacing the call with `tempVar2`.) In fact, this approach would be very similar to the approach that is already used for inserting `Assert` statements by returning values until a branches loop is reached. Unfortunately, the problem was discovered late into development and insufficient time was available to implement the solution.

5.4 Extending Code Suspension to Create Do

The Execution engine currently handles code suspension in `Engine.ts` with its `tick()` function. When suspended, `tick()` will prematurely end the currently running `for` loop that runs each thread's `step()`, ending execution for the current step. It will end execution before the `for` loop in subsequent steps, so the threads that are run here will remain suspended until the suspension is undone.

However, the `AgentCreateDo` `ASTNode` creates threads from the code its block contains and spawns those threads during runtime, and does not add those threads to the list of threads that `tick()` manages. Thus, even if the `Engine` is suspended in the middle of an `AgentCreateDo` call, the engine will continue to execute all `AgentCreateDo` threads until they finish execution. If suspension is forced for `AgentCreateDo`, the engine will never generate the remaining `AgentCreateDo` threads, and

execution would not save the intermediate state it should have kept track of (how many `AgentCreateDo` threads are left to create).

A potential solution would be to change `AgentCreateDo`'s implementation so that the threads it generates and their execution are managed by `Engine.ts` and its `tick()` function. However, that would require that `tick()` could manage the thread list updating in real time, and that `tick()` could manage temporarily leaving one thread's execution to manage another thread's execution. It lies outside the scope of this thesis.

5.5 Integration with Incremental Compile

In the future, StarLogo Nova plans to implement a feature called “Incremental Compile,” where updated code would be compiled and appropriately modify already-compiled code, saving the effort and time involved with compiling all code again for minor changes. The feature would also allow users to change their code without having to reset their current simulation in WebLand, allowing them to fix runtime errors without losing their game state. In particular, incremental compilers for other applications like Rust take advantage of cached intermediate results and modularity to minimize the amount of code needed to be recompiled [11]; StarLogo Nova's would certainly behave similarly.

Features like runtime code insertion make incremental compile more difficult. A simple action like removing a multiplication block that's an argument to another block requires not only the multiplication block and its nested arguments to be removed, but also potentially one or more `CheckTypeOf` blocks that were inserted by the compiler during the `SemanticAnalysis()` phase of compilation to be removed.

Further, although code suspension was implemented with incremental compile in mind to achieve the goal of fixing errors without resetting program state, it currently works by remembering the index and agent of the suspended thread – references that are not guaranteed to be constant when threads can be added or removed after the initial compilation step. These features may have to be handled differently to ensure they work properly with incremental compile. However, it is hoped that the general

code architecture in place should prove helpful nevertheless when incremental compile comes, despite any tweaks the existing error checking code may need to work well with incremental compile.

Chapter 6

Applications and Closing Thoughts

6.1 StarLogo Nova as a language workbench

The work in this thesis gives StarLogo Nova debugging capabilities, which combined with an easy way to create blocks by typing them (called typeblocking [5]) and StarLogo Nova’s support for custom block macros, means that the platform can be used to develop sets of blocks designed for a specific purposes (called Domain Specific Languages or DSLs). With these features, StarLogo Nova can function as a language workbench tool [1] to develop these DSLs with functionalities close to those of modern IDEs.

In more practical terms, a set of biology blocks could be created for biology classrooms, a set of higher level actions (such as “draw a circle on the terrain”) can be created for programming classrooms, a set of simulation blocks can be created for managing chart widgets, and so forth. When the work of this thesis, error checking and safety, is combined with typeblocking, these custom blocks can easily be assembled and trusted to handle hard domain-specific problems with high level tools that can be used by the appropriate audience, even if that audience is not primarily composed of professional programmers.

6.2 Applications for other block-based programming languages

The implementation outlined in this paper cannot be directly translated to other block-based programming languages, which may have different backend code processing pipelines. However, it does serve to illustrate a practical approach that can be taken towards adding backend error protection and reporting to a block-based programming language – a feature that block-based programming languages do not generally focus on. If a greater effort is put into developing features that IDEs focus on that block-based programming languages do not, then block-based programming languages can move beyond their traditional audience of inexperienced programmers into a tool that can be used for more professional purposes.

6.3 Conclusion

As a block-based programming language, StarLogo Nova presents its users with a powerful and easy-to-learn drag-and-drop programming interface. Part of the power of this language – the lack of explicit typings for a variety of different data types – comes at the expense of a number of errors that the user can make when creating their program. As not all of these errors can be caught before compilation, it becomes necessary for the compiler and runtime engine to also participate in providing feedback about errors to users.

The work in this thesis applies the idea of a semantic analysis pass in traditional text-based programming languages to the StarLogo Nova compiler, while extending StarLogo Nova’s capability to report errors during runtime while maintaining its ability to continue running despite errors. Ideally, this gives users a system that is both as flexible as it was before and more robust against and communicative about errors it encounters. With these improvements, users will be able to focus their debugging efforts on higher level problems such as algorithmic problems rather than type errors, which will hopefully inspire even more complex and creative projects.

Appendix A

Creating a New Block with Type-Checking

This appendix is intended to document the process by which developers of StarLogo Nova can add new blocks that interact properly with the error checking system. Although it is designed to minimally impact development, the error checking system will throw compile time errors for blocks that do not interface with it, preventing the containing block stack's thread from being run, complicating block testing.

A vast majority of the block creation process is the same. The front end script-blocks are added in `Editor/lib/BlockLookup.js`, and the Map in `ts/Compilation/BlockTable.ts` that connects the scriptblocks' name field to the appropriate `ASTNode` subclass is updated with a new `String` to `ASTNode` generator pair. Unless the block opts to take advantage of the yielding pathway for type checking like the division block, the block itself can be implemented in the same way as before, too. The blocks need only be correctly annotated with their return types and with runtime checking information.

By default, type annotation functions like `is_number()` and `can_change_type()` return `false` by default. Depending on the complexity of the block, these can either be set to true for the appropriate values, or they can call their arguments' type annotation functions to determine their values based on their inputs at compile time. Note that it's recommended to return true for a type annotation function when a

block should always return a specific type of value when it returns a valid input – thus, the `CalcQuotient` block would return a constant `true`, as division is always expected to return a number, even though division technically returns a non-numeric NaN value with non-numeric inputs. The division block handles error checking for its arguments by taking advantage of its `type_check()` function.

For the `type_check()` function, it's recommended that each argument is type checked with the appropriate convenience function, such as `type_check_arg_is_num()`, if possible. These convenience functions handle compile time errors and return a `RuntimeCheckPacket` to handle runtime errors. All the non-null `RuntimeCheckPackets` should be combined in a list of `RuntimeCheckPackets` as the function's return value. For other compile time checks that do not depend on types or that depend on comparing multiple argument values at once, it is recommended to first check the `can_change_type()` of the corresponding argument(s) – when this function returns `true`, the argument it was called on requires runtime checking, so the compile time check can be skipped and the appropriate `RuntimeCheckPacket` can be appended to the return value of `type_check()`. Otherwise, an appropriate `sendCompileError()` call should be sent and the `State.killCurrentThread()` flag should be set to signal a compile error if the block's precondition is not met. The convenience functions in `ASTNode.ts` are great examples that can be referenced when creating non-standard backend error checks.

A block only needs to worry about reporting its direct descendant sockets in its `type_check()` function – as the semantic analysis pass navigates the AST itself, it will see if it needs to add more runtime checks for blocks inside those direct descendant sockets when it navigates inside those nested blocks. The compiler, not `type_check()`, handles passing these `RuntimeCheckPackets` to the appropriate `ASTList`. This means that the `type_check()` function need not worry about passing `RuntimeCheckPackets` from its block's arguments to its own return value. As a corollary, this means that any block that does not need error checking itself (such as the `PrintToJSConsole` block) can use the base `ASTNode`'s class default `type_check()` function, which simply returns a `new Array<RuntimeCheckPacket>()`.

Bibliography

- [1] Martin Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? *martinfowler.com*, 2005. <https://www.martinfowler.com/articles/languageWorkbench.html>. Accessed May 22, 2019.
- [2] Arthur Graesser and Patrick Chipman. Exploring Relationships Between Affect and Learning with AutoTutor. *The 13th International Conference on Artificial Intelligence in Education*, 2017.
- [3] MIT Lifelong Kindergarten Group. Scratch. <https://scratch.mit.edu/>. Accessed May 20, 2019.
- [4] Joel Jones. Abstract Syntax Tree Implementation Idioms. *The 10th Conference on Pattern Languages of Programs*, 2003. <https://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf>. Accessed May 21, 2019.
- [5] Terrance Liang. Typeblocking: Keyboard Integration with Block Programming in StarLogo Nova. Master’s thesis, Massachusetts Institute of Technology, Department of Electrical Engineering And Computer Science, June 2019.
- [6] Jens Monig and Brian Harvey. Snap! <https://snap.berkeley.edu/about.html>. Accessed May 20, 2019.
- [7] Jin Pan. Performance Engineering of the StarLogo Nova Execution Engine. Master’s project, Massachusetts Institute of Technology, Department of Electrical Engineering And Computer Science, August 2016.
- [8] MIT Scheller Teacher Education Program. StarLogo Nova. <https://education.mit.edu/project/starlogo-nova/>. Accessed May 16, 2019.
- [9] Cynthia Solomon. Logo, papert and constructionist learning. *Logothings*, 2008. Archived version of logothings.wikispaces.com/. Accessed May 23, 2018.
- [10] Mauricio Verano Merino and Tijds van der Storm. Language Workbench Support for Block-Based DSLs. In *BLOCKS+ Proceedings*, 2018. [Preprint]. http://cs.wellesley.edu/blocksplus/2018/pdfs/blocksplus18_p10_merino.pdf. Accessed May 20, 2019.
- [11] Michael Woerister. Incremental Compilation. *Rust Blog*, 2016. <https://blog.rust-lang.org/2016/09/08/incremental.html>. Accessed May 21, 2019.