

# A Compiler in Snap!: Compiling a Block-Based Language

*Oscar Chan*

Electrical Engineering and Computer Sciences  
University of California, Berkeley

Technical Report No. UCB/EECS-2022-157

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-157.html>

May 20, 2022



Copyright © 2022, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

---

# A Compiler in Snap!/: Compiling a Block-Based Language

by Oscar Chan

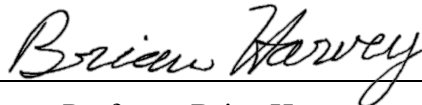
---

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences,  
University of California at Berkeley, in partial satisfaction of the requirements for the  
degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

### Committee:



---

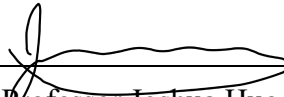
Professor Brian Harvey  
Research Advisor

May 13, 2022

---

(Date)

\* \* \* \* \*



---

Professor Joshua Hug  
Second Reader

20-May-2022

---

(Date)

# A Compiler in Snap!: Compiling a Block-Based Language

Oscar Chan

University of California, Berkeley

ochan2@berkeley.edu

## ABSTRACT

This paper presents a compiler framework for the Snap! blocks programming language. Snap! is a web browser language using JavaScript to run the interface and the original interpretation of the language, therefore it is sensible to use JavaScript as the destination language as well. The compiler reads each Snap! block sent to it within a warp block and translates it to the equivalent JavaScript function that powers the block and powers the Sprites on the Snap! graphical interface. Control looping blocks like the *forever*, *for*, and *repeat* return control to the Snap! graphical interface by using JavaScript Generators pausing themselves in the middle to redraw the graphics and then come back to the block and continues running the compiled program. When the compiled code is executed on several tasks, it was able to attain a speedup that is significantly noticeable for a sizeable program compared to the interpreter and the interpreter-version of warp, but for simple programs, the interpreter warp performed better or as good, and more work needs to be done to support more Snap! blocks.

## Author Keywords

Snap!; JavaScript; compilers; Computer Science Education; block-based programming

## CCS Concepts

•Human-centered computing → Human computer interaction (HCI); •Applied computing → Education; •Software and its engineering → Software notation and tools; Compilers;

## MOTIVATION

Snap! is a graphical blocks-based language inspired by MIT's Scratch. It is built as an educational programming language designed to help create an introduction to mathematical and computational ideas. It is built using a web interface and JavaScript so that it can be run inside a browser without any extra setup needed unless the student prefers downloading Snap! and using it offline. Even then, it is as simple as opening the folder inside a browser after download. As a result, this avoids the problems of installation and syntax errors that tend to discourage and frustrate new programmers from continuing in the field. Snap! is used in various places including UC Berkeley's CS 10 The Beauty and Joy of Computing (a "CS 0" non-majors course), The Beauty and Joy of Computing Massive Open Online Course (MOOC) on edX, and a recommended language in the relatively new Advanced Placement (AP) Computer Science Principles course. With the growth of the language into prospective programmers, it is a good time to introduce new features that allows them to experience more of the programming world, one of which is compiled languages.

There are several advantages to having languages compiled down to as much of the host language as possible (normally would be Assembly or Bytecode on most computers), including allowing faster execution of a program compared to interpreted programs and going directly to the host language. Another advantage to going directly to the host language is cutting away the need for a translation layer in the interpreter for packaged stand-alone executions of Snap! projects, using the Snap! tool<sup>1</sup> since the need to see the source code for users is no more and therefore can cut down the packaged project size by a bit and hopefully let the program run faster and more smoothly.

Since the host is a browser for the reasons mentioned earlier in the Abstract, the decision to compile to JavaScript makes sense. Compiling JavaScript is not a new concept, as other languages like CoffeeScript and TypeScript do just that. A Scratch compiler to JavaScript already exists called LeopardJS, showing that such a concept can exist for Snap!. Since it is written in TypeScript, a form of JavaScript, it becomes more hopeful to be able to write a compiler in the browser using JavaScript.

Because Snap!'s audience is mostly in an educational and beginner-friendly setting, there is not as much incentive to optimize its performance. As a result, not as much extensive research is done in Snap! as there is in other popular programming languages.

Many Snap! users commonly lament about Snap!'s speed. Often the slowdown is bad enough to the point users may fall back on using Python or other languages to complete their projects.

Increasing the performance of Snap! using compilation can benefit in several ways. It can, of course, fix the major problem of some users having concerns about the Snap! performance. Therefore, it could open the possibility of Snap! returning as a programming option for many students and make them less frustrated to use the language, which was the original intention of block-based languages. There also exists a good amount of seasoned programmers who don't want to use traditional typing-based languages and use a block-based language for many of their implementations. Having that back-to-home feeling for them when they use a tool like Snap! would be very beneficial to them as well.

During the need finding stage, it revealed a potential feature for a compiled Snap! program having the ability to, later on, add custom JavaScript code alongside the JavaScript compiled Snap! code. While this is not the scope of our current implementation, having something that creates JavaScript code from

---

<sup>1</sup><http://snapp.citilab.eu/>

Snap! can open doors to a possible additional implementation to add JavaScript code. As a result, it would allow Snap! programmers to use the increased support of JavaScript to create other special implementations and also take advantage of the JavaScript libraries that exist today.

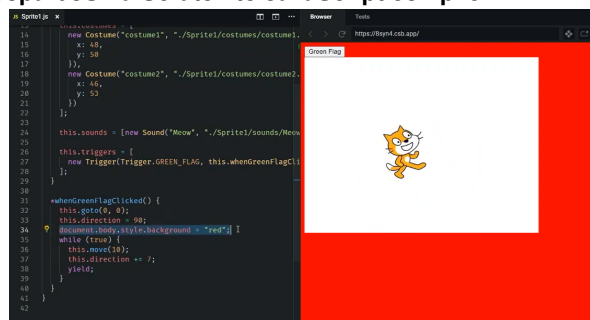
Currently, there aren't any known tools in Snap! that do what this project has set out to do. The closest thing to this project's functionality is the warp block in Snap!, which gives the Snap! block code more time to run the code before returning partial control to the Snap! interface. While using the block does provide some speedup, the warp block does not compile the code nor speed up the interpretation process enough to satisfy user needs.

It is expected that the outcome of the compiler will help speed up computationally heavy tasks while still allowing users to take advantage of the simplicity and coding abstraction of a drag-and-drop block environment.

## RELATED WORK

There has not been any large interest in the programming community in compiling block-based languages. A compiler has been made for Scratch, but that's about it. Although there has been more work related to the compiling to JavaScript as the destination language like with CoffeeScript and TypeScript. This section will focus on two other block-based language compiling where inspiration is taken from.

### LeopardJS - a Scratch to JavaScript compiler



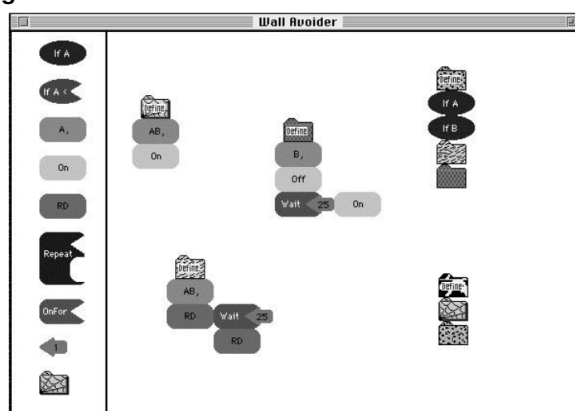
**Figure 1: LeopardJS program running compiled Scratch code with the cat forever running in circles with JavaScript code used turn background red**

Scratch, the language Snap! is heavily influenced<sup>2</sup>, already has a third-party compiler. The LeopardJS tool<sup>3</sup> for Scratch was created to compile Scratch code on a separate website, and the results were promising. Scratch code was able to run faster, but the process of going to a third-party website still made the task of compiling the code an extra step. There was still no way to compile Scratch code in the Scratch user interface where there was less risk of troubleshooting or errors on another website, let alone be self-contained. If users wanted to edit the code, they would first have to add what they want in regular Scratch and then upload a new Scratch file to LeopardJS. This is problematic given the original purpose of Scratch, which is primarily targeting young audiences and the

new programming community, who wouldn't know how to use the resulting JavaScript code immediately and especially when there are bugs in the program that needs to be fixed, which would require going back to the block editor to fix it.

One of the inspirations from this program is the compiler structure. Using string concatenation to create JavaScript code, Leopard can create code that mimics its block-based version as much as possible. Since the tree-like structure of Scratch already makes it its syntax tree, all the code needs to do is to traverse down that tree and generate the required JavaScript code snippets to create the entire program. The resulting code is then placed into a function for the Leopard editor to have access to run it. But because there are no blocks to click on to run the Scratch blocks anymore, the only way to run the code is by Events and clicking the Green Flag.

### LogoBlocks



**Figure 2: Logoblocks sample movement program with wall detector**

LogoBlocks<sup>4</sup> is an old block-based language that was used to create a program onto a computerized Lego brick. It was used in inspiration to create the Lego Mindstorm<sup>5</sup> robot programming language and a graphical user interface, very much similar to Snap! and Scratch. However, because the destination machine is a small computerized Lego brick (called a "Brick"), it has to be compiled to another language. This language has to be small and direct to the machine enough it doesn't take too much space to include an interpreter. LogoBlocks as a result follows the usual path of a compiler to create a spanning tree to parse the block language into a different language called BrickLogo. Afterward, it compiles down to portable code (or p-code) that can be loaded onto the Brick, which is then read and executes the commands it is given. Like Snap!, the purpose of this language was educational and recreational for children and new programmers. However, LogoBlocks required the use of a downloaded and installed tool to then perform the compilation and then have it load into the Brick unit. Another one of the bottlenecks that the program and compiler faced at the time was the ability to have parameterized functions and return statements, which, according to the author, added a whole new level of complexity.

<sup>2</sup>[https://en.scratch-wiki.info/wiki/Snap!\\_\(programming\\_language\)](https://en.scratch-wiki.info/wiki/Snap!_(programming_language))

<sup>3</sup><https://leopardjs.com/>

<sup>4</sup><https://andrewbegel.com/mit/begel-aup.pdf>

<sup>5</sup><http://www.cs.uml.edu/~fredm/papers/magical-machines.pdf>



**Figure 3: Turbo Mode and Warp block**

While there is no intention on having Snap! having to go as low-level as portable assembly code, inspiration can be taken from the steps needed that allowed LogoBlocks to go from a block-based language to a destination language. Additionally, the compiler can use the Abstract Syntax Tree inherently inside Snap! to start creating the JavaScript program.

### ALTERNATIVE TECHNIQUES

Before having a compiler, there were two ways to speed up the Snap! interface: Turbo Mode and the warp block (Fig. 3), both of which do the same thing in different ways. The similarity in both is that they run the Snap! blocks program at the cost of not giving back control to the main interface as often, which is a user interface concern. The only difference is that Turbo Mode applies this optimization indiscriminately (or to all blocks ran) while warp applies this optimization only to the blocks nested inside the block.

For now, these are two great ways to help make the program run faster when live graphics are not a huge issue to the user, but because many of the interface controls lie in the browser and shares the same JavaScript thread with the interpreter, the entire interface starts to lag. The graphics also update less often, but this is the desired behavior by the specification.

We wanted to look for other ways to speed up Snap! to mitigate having problems with the interface interaction. The belief in this project is that running the Snap! code directly in JavaScript fast enough so that the interface can refresh less often.

### MOTIVATING TASKS

There are many different ways users can utilize Snap!. Various environments and project ideas are made easier due to Snap!'s visual design, so we need to ensure that tasks required to complete all kinds of projects are made as easy as possible for users. In Snap!, this means fixing runtime issues with common project types and operations like mathematical calculations, physics simulations, and data science queries/visualization.

We create these tasks based on potential creations Snap! programmers, students, or otherwise can potentially do. A majority of the programmers are students or explorers who have little to no programming experience or are someone who had programming experience in another language for some time but wanted to not deal with the grammatical constraints of a keyboard-based programming language. Our goal with these tasks is to find ways we could mitigate limiting someone's creativity once they get the skills to create sizable projects, which would cause them to have no choice but to learn a whole new language to get their projects to a usable state. Even introducing a compiler can sometimes limit one's creativity. For example if some blocks are not supported or appropriate for

a compiled context, then the block may have to be rewritten specifically for a compiler or that block must be switched to an interpretation context only.

In cases where the user needs to do a lot of mathematical operations, for example when multiplying matrices, Snap! can see significant slowdown even in cases with smaller matrices. Because matrix multiplication is expected to work on very large orders of magnitude, this slowdown is not tolerable. Modern uses for matrix multiplication involve large matrices and repeated multiplication, neither of which are made any easier for users when runtimes are too slow. If they spend too long waiting for the mathematical operations to terminate, they may get discouraged because their mistakes are amplified by the large amounts of waiting. Other than optimizing their matrix and vector multiplication algorithms, there is not currently much to do to speed up the code without compiling it. Even with these optimizations, which we do not necessarily expect Snap! users to have good enough knowledge about to implement, the Snap! code still may not run quickly enough. Using the warp block lags the interface which as a result lags the Snap! graphics that uses these heavy mathematical operations.

In other cases where students may want to run basic physics simulations, there is also a considerable amount of mathematical operations to be done, but in these cases, there is another factor affecting runtime. The need to make graphical updates further slows computation time in Snap!, so being able to run mathematical physics simulations and see the results updated over time can be another challenging task in Snap!. Snap! is ideologically a great language for this because it has the built-in stage for image rendering, but the combination of mathematical operations and visual updates can slow down Snap! significantly. If a user wanted to simulate a planetary body orbiting around a stellar body, generally considered a basic, common, and educational physics simulation, Snap!'s current functionality can't run quickly enough to combine the calculation and the visuals. Even with the aid of the warp block and/or Turbo Mode in Snap!, simple simulations like this are too slow.

Lastly, in cases where students want to use data science tools, Snap! can slow down when working with larger datasets and more complicated table queries. There are a lot of memory reads and writes involved in common table query methods, and even with the aid of using JavaScript functions to execute atomic operations, Snap! slows down on common queries. There are some ways to reduce runtime by optimizing query order, but even for seasoned data scientists, optimizing query order is a difficult problem, so for Snap! users, the assumption cannot be that they will be able to optimize their queries. Snap! needs to have reasonable runtimes for queries of any kind to accommodate Snap! user needs. If a Snap! user has a dataset they found and they want to be able to answer some questions about it using Snap!, they need to be able to get results for their queries quickly so that they can learn as they code. Users using Snap! to conduct data science are likely doing so at an beginner level, so we cannot assume that they will be able to fully understand the blocks they are using and avoid common



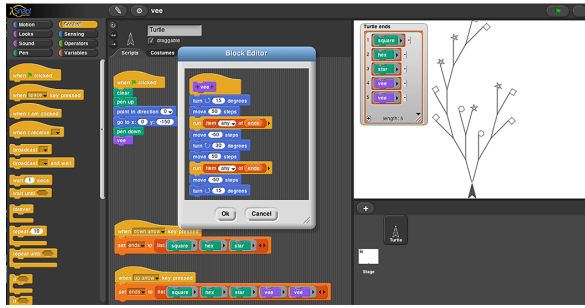


Figure 4: Snap! User Interface with an Example Program<sup>6</sup>

pitfalls. Snap! needs to be able to give immediate feedback when students work with data science tools.

In all these cases, the common issue is that Snap! needs to be able to do operations more efficiently so that users can get feedback on their code more quickly. Users learning to code should not be expected to tolerate large runtimes when they are in the educational process. Instant feedback is important as they learn how to use the tools. These tasks need to be able to be done faster in Snap!.

## DESIGN OF THE TOOL

### Snap! Interface Flow

This section discusses the original technical implementation of Snap! in a general context. Knowing how the flow works helps us identify where the compiler would be best placed.

#### User Interface and Development Environment

The user interface of the Snap! (Fig. 4) includes several blocks that programmers can drag and drop onto the Scripts canvas to create a program. Programmers can make programs for each of the Sprites to control them. The Sprites perform actions based on the block code onto the canvas on the top right. Snap! also can have a series of blocks that make up a custom blocks, or in other words: a function.

#### Snap! Interface Control Flow

The Snap! interface is implemented in a way to return control to the graphical interface after the interpretation of a connected set of blocks, each step of a control loop (using a “doYield” flag), and a timeout defaulting to 500 ms starting from the last start or continuation of the program. There is a while loop that goes through each connected block one at a time until any one of the conditions mentioned is met to return control to the interface.<sup>7</sup> The world is generated onto the browser starting from the `snap.html` file, which the browser looks at to generate onto the browser’s web interface. It will call several functions from the `morphic.js` Web-GUI interface, inspired by Squeak, to generate the Snap! interface and the world loop to trigger the next interpretation step of the code.

The Snap! blocks are connected one after another and the lexical analyzer of the program reflects the blocks as a series of linked blocks one after another, like a linked list. It is then nested in a series of Contexts, which can be seen as frames for

<sup>7</sup><https://github.com/ochan1/Snap/blob/6513c9c/src/threads.js#L671>

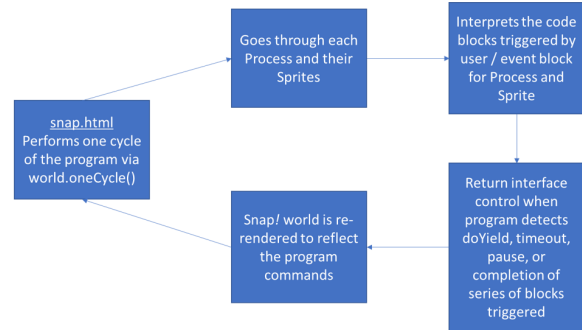


Figure 5: Flow chart of a general overview of Snap! web interface control flow

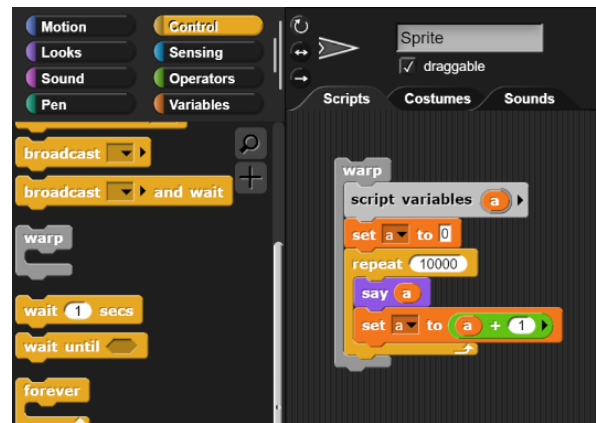


Figure 6: Warp Block in use to compile a series of blocks

each block, for the interpreter loop to go through and process each block, popping the completed Contexts when done or retaining them as a new variable environment for loops or functions.

### Snap! Compiler

This section explains some of the implementations of the compiler. The compiler structure code for some of the sections is listed in Appendix A.

#### User Interaction

Given the purpose and limitations of the compiler, the compiler implementation is abstract away within the warp block. Part of the reason is the tight-nit creation of the interpreter with the Snap! interface, which made it difficult to redesign all of Snap! to have a compiler. In other words, the warp block is being redesigned. It makes sense since the compiler does ignore as many of the “doYield” flags just like what “warp” does. Because the warp block accepts a nested series of blocks that acts as an Abstract Syntax Tree, we can know what program to compile. The code is compiled the first time the block is reached, then it is saved to be reused again. When changes are detected within the blocks, including the inputs, a flag will be set to all the warp blocks that are ancestors at the location being changed to tell the warp block to recompile the block program.

### *Compiler Design and Interface*

The Snap! compiler currently works by having the block integrated into the interpreter. The warp block has a separate function for its JavaScript logic to power the compiler. The function then accepts the series of blocks and a continuing generator. At the first run, the continuing generator parameter is expected to be null, to show that this is the first run of the block and requires starting the compiled code either from scratch or from a previous compilation stored for now in the top block of the block series inside the warp block.

When the series of blocks needs to be compiled if it has never been compiled or needs to be recompiled based on the flag set, the starting block, which is the topmost code block in a connected stack of blocks inside the warp, is used to start the compilation. As mentioned before, Snap! blocks contain the blocks that come after it as next blocks and within it like for parameters as children. The compiler goes through each block and then performs syntax and semantic analysis based on the block name it is currently at. It detects the block used by name, similar to a token reader, and then generates a snippet of code for itself and its children recursively.

### *Just-in-Time (JIT) Compiler within the Interpreter*

The compiling behavior in this project is that the compilation of warp blocks only happens when during the interpretation of the main program a warp block is hit. It is then that the Snap! interpreter will switch to running the inside blocks in a compilation context. If the warp block is marked and needed to be recompiled, it will then generate the JavaScript code needed to run the series of blocks and their powered functions.

Custom Blocks are compiled on a just-in-time basis. During compilation of a series of Snap! blocks, once a Custom Block is found to require compiling, the JavaScript code to represent the block is then also made. More details about its compilation can be found in the "Custom Blocks" section.

### *Reporter Blocks*

Reporter blocks are blocks that return a value to the parent context. When a reporter block is returned onto the global frame, the series of blocks will show a value as a text bubble next to it. Since there is no expectation for more blocks to be connected, the value is returned immediately, and the program is stopped. The interpreter will then get the compiled program return value and then return it to the outer context. If there is no return value, it is by default null. The interpreter will then not report anything for null.

The compiler simply just "returns" the received value, but it also has the additional responsibility to clean up the contexts made inside the function, especially when it exists somewhere in the center of the entire code. As a result, the warp block's context is specially marked to, in a way, sandbox the Contexts so that when popping it doesn't accidentally access the Contexts that belong to the parent interpreter. Custom Block contexts are marked to signify to the reporter block to stop popping once it sees that marker, to signify all the contexts made inside the functions are cleaned up.

### *Control Loops*

Control loops are the trickier blocks. In the interpreter of Snap!, each step of a loop returns control to the interface to allow the user to interact with the interface. This is a problem for compiled programs that required it to return control to the interface at each step since it can't push a new Context for the next step of the program. For certain loops, it is possible to wait for them to end like the "for" and "repeat" loops, even though it may be possible that users will have to wait for a long time. It's even more problematic for the "forever" loop, however, since it never ends and can relinquish control and instead freezes the entire interface.

One idea was to make a new function for the loop, but it is a problem if there was a branch fork where one of them contained a loop, and then merge back together in the end, essentially sharing the same tail of the branching.

We take into inspiration how Leopard implements control loops. The solution was to have a method to pause the program and return control to the interpreter to run the interface. Fortunately, there was a way to do this in JavaScript using Generators. The compiled JavaScript code is then made as a Generator Function using the "function\*" syntax. At the end of each loop step, a JavaScript yield is added to pause the program and give back control to the interpreter and interface for a bit. This is where the third parameter comes in to accept a continuing generator function. The structure of the interpreter allows inputs to be pushed into the current Process so they can be passed in again to the same function. In this case, we push the string name, a block, and the returned generator along with a Context with the "compiler" block again to continue the generator upon next time the interpreter loop is executed. To make sure control is given back to the interface, a "doYield" is also pushed, following the same structure as the other loops in the program.

### *Custom Blocks*

Custom Blocks are equivalent to functions in the Snap! programming language. They are defined in two ways: a block definition and the block instance itself. The common implementation that relates the same body, should it be modified, to the same custom block is the block definition. As a result, the function body is compiled as if it was a separate program, with a new Context at the start and then popped at the end of the program to simulate an environment created and destroyed. A special function is then compiled to for now search linearly through an entire list of all the custom blocks in existence in the current Sprite and the global environment, and then takes the compiled function defined inside the block's definition to run on. Parameters are then parsed to fit the rules of JavaScript parameter names (since Snap! programmers can use any string they want to name their parameters) and avoid duplicates to use as function parameter names in the JavaScript function. Their values are then passed to the original variable names in the inside context of the block during runtime.

### *Optimization - "yielding" to the Interface Less Often*

In the original Snap! interpreter, what happens is that on each step of a loop, every time certain timing blocks are called, and on every Custom Block call, the interpreter yields control back



to the interface to potentially draw on the graphics screen and introduce control back to the interpreter. On the other hand, *warp* and Turbo Mode skip the yields often, but at the cost of lagging the interface.

In the compiled portion of Snap!, whenever the runtime of the JavaScript code uses a generator `yield` to pause the code for a bit, the expectation there is to return control of the JavaScript thread to the interface. However, "yielding" each time produces the same timed-speed as the interpreter, which was a problem.

As a result, taking inspiration from the behavior of *warp* of ignoring yields sometimes, we only yield back to the interface after a certain number of times or when the timeout is reached (as mentioned before, the default is 500 ms), whichever comes first. Currently, this number is set at 300 yields skipped. For a sizable program, the interface doesn't lag and outperforms the interpreter runtime and interface experience using a *warp* block. However, for small programs, the compiled code still lags the interface and the interpreter outperforms the compiled version.

#### *Limitations of the Compiler*

At its current stage and as can be seen during the usability testing, it can only run certain basic blocks and setups.

Cloning is not possible as there's a structure different than if the Sprite selected is used. By default, most blocks come with their function names attached and only special blocks like control and reporter blocks need something special to retain both interface control and value returning in the compiled state (since if we called the reporter block functions, they only accept a block and not a compiled input).

The main categories of blocks tested in this project are Motion, Control, Operators, and Variables. However, not all the blocks in those categories are tested and are not guaranteed to work. Over time, each block will be tested and made sure it can work with the compiler. As of now, only *async* and *timed* blocks, essentially those that require the use of "doYield" themselves that are not Control blocks, would not work in the compiler since their code requires interaction with the interface itself (especially with the "doYield") and require a complete rewrite of those functions or temporarily returning control to the interpreter for those blocks.

## COMPILER CODE RESULTS

### **Example Compiled Code**

The compiler creates syntactically correct JavaScript code, but not stylish code since this is meant only for JavaScript to see it and not the end-user.

Some examples of Snap! code and their respective compiled JavaScript code are listed in Appendix B.

### **Compiled Code Performance**

Three programs (one simple and two large, based on the program size and complexity) are running on the original Snap! interpreter and also in the compiling *warp* block. The tasks are as follows: A simple task is a simple linear loop program, one large task is a matrix multiplication algorithm, and another

large task is matrix squaring (via looping on matrix multiply of  $A := AA$ , for some matrix *A* with the result saved to matrix variable *A*).

#### *Benchmark Results on the three tasks*

Since the compiler in the *warp* block is a JIT compiler, we time the results for a first run that involves compiling and then running, and also an additional run after that to get accurate timing to see if compiling affected the timing.

The benchmarks are run on the Google Chrome browser on an Intel i7 laptop.

Timings are reported by whatever the *timer* block reports. The harness is as follows: A *reset timer*, the program itself (with or without a *warp* block wrapped around it), and a *report* block that mentions the time via the *timer* block.

Tables 1, 2, and 3 show the timing results of benchmarking programs using four different ways: interpreter, compiled (inside the *warp*) with *Compile + Run*, compiled again with *Run only*, and interpreter's *Warp*.

In Table 2 when benchmarking the matrix multiplication tasks, the compiled version was too fast to be able to determine if the interfaced lagged. However, using the Table 3 results it is possible on larger-sized matrices that the interface does not lag and is fast, but would be immensely slow on Snap!.

#### *Discussion of Benchmark Results*

The compiler *warp* expectedly performs simple tasks around the same speed as the interpreter *warp* block and also for large tasks outperforms the interpreter *warp*.

For the simple tasks, the program is small enough in terms of the number of instructions that any overhead introduced from the compiler and its harness doesn't beat the inefficiencies of the *warp* block. Sometimes, as seen in the plain linear loop program without the *say* block, the *warp* block is faster, though the initial program is already quick to complete anyway. It is believed that any code that involves rendering the Sprite's speech bubble the linear loop program with the *say* block caused the massive slowdown. The compiler harness and the compiler itself inside the *warp* block are already considerable in size, and also the interpreter needed to power each Snap! block, so as a result they even each other out.

On the other hand, for large tasks like the matrix multiply and matrix squaring, the problem is big enough to jump over any overhead hurdle and run faster than the overhead inside the interpreter when running each block. The compiler code achieved 4.5x and about 7.36x speedup than the interpreter *warp*, and about 303x more compared to using only the interpreter.

#### *Liveliness of Snap! with Compiler*

With the current settings of 300 yields skipped over or until timeout, many tasks do not lag the interface as bad as what interpreted *warp* does. This does allow the user to interact with other parts of the Snap! interface without having to wait for the program to complete running, therefore allowing one to check their work while debugging or editing other block series.

Task	Interpreter Time (seconds)	Compiled Warp Time (seconds)		Interface Lag during Compiled Run (Yes / No)	Interpreter Warp Time (seconds)
		First Time - Compile + Run	Run Only		
Linear Loop Program adding 1 to variable, 10,000 times + Saying the added value	177.4	19.7	18.5	Yes	19
Linear Loop Program adding 1 to variable, 10,000 times	166.6	0.5	0.5	No	0

**Table 1: Timing benchmarks for simple linear loop program tasks**

Task	Interpreter Time (seconds)	Compiled Warp Time (seconds)		Interface Lag during Compiled Run (Yes / No)	Interpreter Warp Time (seconds)
		First Time - Compile + Run	Run Only		
Two 2x2 Matrix Multiply	0.2	0	0	Could not determine	0
Two 3x3 Matrix Multiply	0.7	0	0	Could not determine	0

**Table 2: Timing benchmarks for large matrix multiply tasks**

Task	Interpreter Time (seconds)	Compiled Warp Time (seconds)		Interface Lag during Compiled Run (Yes / No)	Interpreter Warp Time (seconds)
		First Time - Compile + Run	Run Only		
3x3 Matrix Power to 100 via Matrix Multiply Looping	66.7	0.2	0.2	No	0.9
3x3 Matrix Power to 1000 via Matrix Multiply Looping	666.7	2.2	2.2	No	16.2

**Table 3: Timing benchmarks for large matrix squaring tasks**

However, if one were to add blocks that relied less on computation and control, but more on graphics like move or say, the interface starts to lag as much as the warp block.

However, because the compiler runs all the blocks on one JavaScript program, rather than as an interpreter which goes through a series of tokens, adding blocks to the middle of a series of Snap! blocks inside warp while that series on blocks are running would not work. The code inside the warp block would have to be recompiled by reaching the block again to trigger the JIT compiler.

As a result, using a compiler can help with introducing the speed of the program while not heavily sacrificing the liveliness of Snap! for computational tasks.

## CONCLUSION

The introduction of a compiler is one step forward towards introducing a speedup for Snap!, such that it can introduce a new feature of programming that can help them mitigate the various inefficiencies that come with Snap! This project also created a framework to continue implementing a compiler for Snap!. The results show that there is some feasibility in continuing the development and potential optimization of this compiler, especially to introduce speed while not sacrificing liveliness as much as possible.

On the other hand, while future programmers would not be aware of the abstraction of the compiled code in the background, we can say the same for the interpreter as well. However, based on our initial need finding, we could already see that Snap! programmers are already frustrated with the slowdowns introduced as a result of the interpretation and also the

constant lag of using the warp block or Turbo mode, which means waiting for the return of control back to the Snap! interface to interact with it.

A big major hurdle in getting the compiler completed was the sheer size of the Snap! code base and the many different functions that power the Snap! interface. Since Snap! is a very niche language and no documentation exists to detail its creation, it requires intense code reading and subsequent analysis using breakpoints to see if the JavaScript runner reaches that point, and then see how a certain interaction behaves for that code area. Regardless, the work that has been laid out here should create a firm groundwork to identify where and how to meet the needs of the Snap! programming language in a compiled context.

Since some of the features are so tight nit into the interpreter, it may not even be possible to cover all the possible features that Snap! has. Even the Scratch variant, Leopard, still has some blocks still in development.<sup>8</sup> One idea would be to have the compiler share some implementation with the interpreter, to support those blocks. One way would be to compile the parts before the interpreted block, have logic that would pause the code and push the block that can only be used in interpretation into the Snap! context stack, and then continue running the code down.

As the compiler gets more developed, one future usability study we would like to explore is having a group of students use the compiled version of Snap! in their projects. These students would use Snap! for their learning and programming. This would reveal where in large Snap! programs the feature

<sup>8</sup><https://leopardjs.com/manual>

would work and would not work, and either give recommendations to those who use the tools or provide workarounds.

## REFERENCES

- [1] Brian Harvey and Jens Möning. 2020. *Snap! Reference Manual*.  
<https://snap.berkeley.edu/snap/help/SnapManual.pdf>
- [2] Josh Pullen, Florrie Haero Miller, and adroitwhiz. 2020. Leopard. (2020). <https://leopardjs.com/>
- [3] Jens Möning. 2008. Snap! - A Visual Programming Language inspired by Scratch. (2008).  
<https://github.com/jmoenig/Snap>
- [4] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. 2019. ChocoPy: A Programming Language for Compilers Courses. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E (SPLASH-E 2019)*. Association for Computing Machinery, New York, NY, USA, 41–45. DOI:  
<http://dx.doi.org/10.1145/3358711.3361627>

## Appendix

### A) Compiler Code

#### A1. Reporter Blocks

```
case 'reportOr':
  return this.compileInfix('||', inputs);
case 'reportAnd':
  return this.compileInfix('&&', inputs);
case 'reportIfElse':
  return '(' +
    this.compileInput(inputs[0]) +
    ' ? ' +
    this.compileInput(inputs[1]) +
    ' : ' +
    this.compileInput(inputs[2]) +
    ')';
```

```
case 'doReport':
  return 'current_process.popContextTilFunc();' + '\n' +
    'return ' + this.compileInput(inputs[0]);
```

#### A2. Some of the Control Loop Blocks

```
case 'doForever':
  if (!this.yield_enabled) {
    throw "Forever loop requires 'yield' to be enabled";
  }
  var body = inputs[0].inputs()[0];
  var while_body = "";
  if (body) {
    while_body = "\n" + this.compileSequence(body);
  }
  return "while (true) {\n" +
    while_body + "\n" +
    "yield;\n" +
    "}";
case 'doUntil':
  var goalCondition = this.compileInput(inputs[0]);
  var body = inputs[1].inputs()[0];
  var loop_body = "";
  if (body) {
    loop_body = "\n" + this.compileSequence(body);
  }
  return "while (!(${goalCondition})) {\n" +
    loop_body + "\n" +
    "yield;\n" +
    "}";
case 'doRepeat':
  var counter, body;
  counter = this.compileInput(inputs[0]);
  body = inputs[1].inputs()[0];
  if (isNaN(counter) || counter < 1) {
    counter = 0;
  }

  var repeat_body = "";
  if (body) {
    repeat_body = "\n" + this.compileSequence(body);
  }
  var yield_keyword = "";
  if (this.yield_enabled) {
    yield_keyword = "yield;\n"
  }
  return `for (let i = 0; i < ${counter}; i++) {\n` +
    repeat_body + "\n" +
    yield_keyword +
    "}";
```

```
case 'doForEach':
  var upvar, list, pre_body;
  [upvar, list, pre_body] = inputs;
  upvar = upvar.inputs()[0].blockSpec;

  var assignment_to_code_list;

  list = this.compileInput(list);

  assignment_to_code_list = list;

  var body = pre_body.inputs()[0];

  var for_body = "";
  if (body) {
    for_body = this.compileSequence(body);
  }
  // vars.changeVar(upvar, dta.step);
  var proc_context_vars = "current_process.context.variables"
  var yield_keyword = ""
  if (this.yield_enabled) {
    yield_keyword = "yield;"
  }
  return `${proc_context_vars}.addVar("${upvar}");
let list = ${assignment_to_code_list};
current_process.assertType(list, 'list');
let list_size = list.length();
for (let i = 1; i <= list_size; i++) {
  if (list.isLinked) {
    ${proc_context_vars}.setVar("${upvar}", list.at(1));
    list = list.cdr();
  } else {
    ${proc_context_vars}.setVar("${upvar}", list.at(i));
  }
  current_process.pushContext(null, current_process.context);
  ${for_body}
  ${yield_keyword}
  current_process.popContext();
};
```

### A3. Custom Blocks

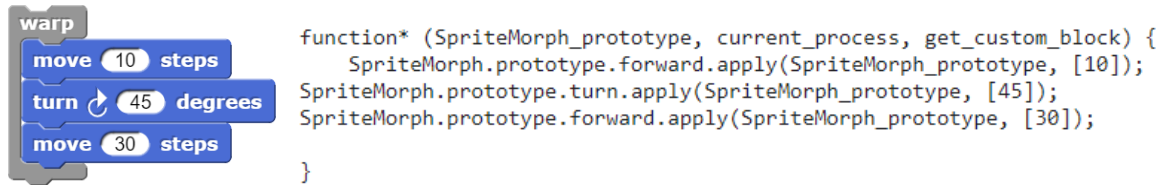
```
JSCompiler.prototype.buildCustomBlock = function(blockName, block_definition, inputs) {  
  /*  
    DESIGN:  
    1. Check if the Function is compiled or not (acting like a linker), then compile  
    2. Caller style of pushing a New Context and Assigning Variables  
    3. Call the function  
    4. Check if how many procedure call so far (this exists in Process already) and then "yield"  
    LAST. Pop the context  
  */  
  var function_name = this.process_text(blockName);  
  
  if (block_definition.to_compile) {  
    block_definition.to_compile = false;  
  
    try {  
      this.num_custom += 1;  
      var func_code = this.compileFunctionJSCode(block_definition, inputs);  
      console.log(function_name, "\n", func_code);  
      eval(`block_definition.compiled_function = ${func_code}`);  
      this.num_custom -= 1;  
    } catch (error) {  
      block_definition.to_compile = true;  
      throw error;  
    }  
  }  
  
  var func_params = "SpriteMorph_prototype, current_process, get_custom_block"  
  if (inputs.length > 0) {  
    func_params = `${func_params}, ${this.compileInputs(inputs)}`;  
  }  
  
  var get_custom_code = `get_custom_block(SpriteMorph_prototype, ${function_name}, ${block_definition.isGlobal})`;   
  return `(yield *(${get_custom_code}).compiled_function(${func_params}))`;  
}
```

### A4. Skipping Yields

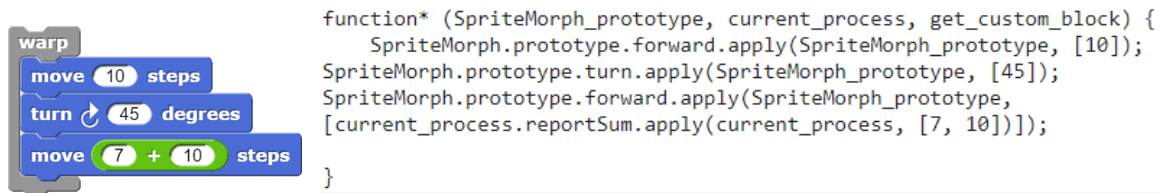
```
var yield_skips = 300;  
while ((yield_skips > 0) && ((Date.now() - this.lastYield < this.timeout))) {  
  gen_output = gen_code.next();  
  if (gen_output.done) {  
    break;  
  }  
  yield_skips--;  
}
```

## B) Compiler Generated Code

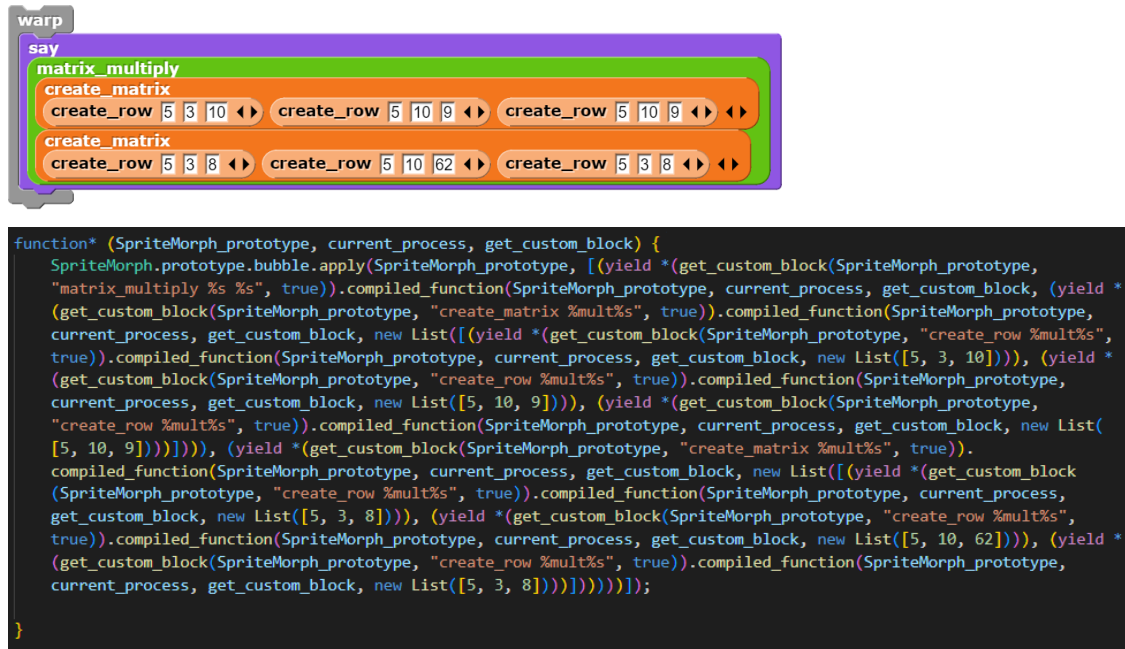
### B1. Basic movement code



### B2. Movement code with addition operation

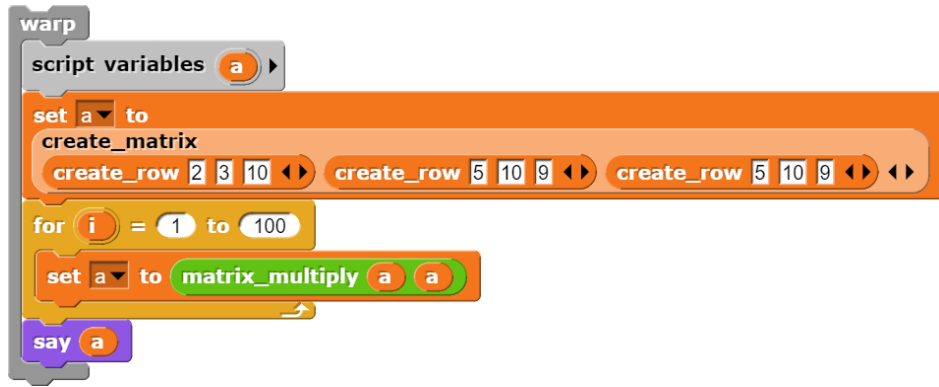


### B3. Matrix Multiply





## B4. Matrix Squared



The Scratch script for B4. Matrix Squared is as follows:

```

warp
script variables a
set a to
  create_matrix
    create_row 2 3 10
    create_row 5 10 9
    create_row 5 10 9
for i = 1 to 100
  set a to matrix_multiply a a
say a
  
```

```

function* (SpriteMorph_prototype, current_process, get_custom_block) {
  current_process.doDeclareVariables.apply(current_process, [new List(["a"])]);
  current_process.setVarNamed("a", (yield *(get_custom_block(SpriteMorph_prototype, "create_matrix %mult%s", true)).
  compiled_function(SpriteMorph_prototype, current_process, get_custom_block, new List([(yield *(get_custom_block
  (SpriteMorph_prototype, "create_row %mult%s", true)).compiled_function(SpriteMorph_prototype, current_process,
  get_custom_block, new List([2, 3, 10])), (yield *(get_custom_block(SpriteMorph_prototype, "create_row %mult%s", true)).
  compiled_function(SpriteMorph_prototype, current_process, get_custom_block, new List([5, 10, 9]))], (yield *
  (get_custom_block(SpriteMorph_prototype, "create_row %mult%s", true)).compiled_function(SpriteMorph_prototype,
  current_process, get_custom_block, new List([5, 10, 9])))]))));

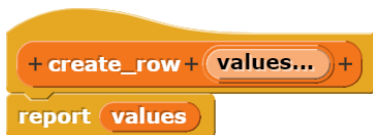
  let dir = Math.sign(100 - 1);
  current_process.context.variables.addVar("i");
  for (current_process.context.variables.setVar("i", 1); (dir * current_process.context.variables.getVar("i")) <= (dir *
  100); current_process.context.variables.changeVar("i", dir)) {
    current_process.pushContext(null, current_process.context);

    current_process.setVarNamed("a", (yield *(get_custom_block(SpriteMorph_prototype, "matrix_multiply %s %s", true)).
    compiled_function(SpriteMorph_prototype, current_process, get_custom_block, current_process.getVarNamed("a"),
    current_process.getVarNamed("a"))));

    yield;
    current_process.popContext();
  };
  SpriteMorph_prototype.bubble.apply(SpriteMorph_prototype, [current_process.getVarNamed("a")]);
}
  
```

## B5. Matrix Multiply Custom Blocks

create\_row



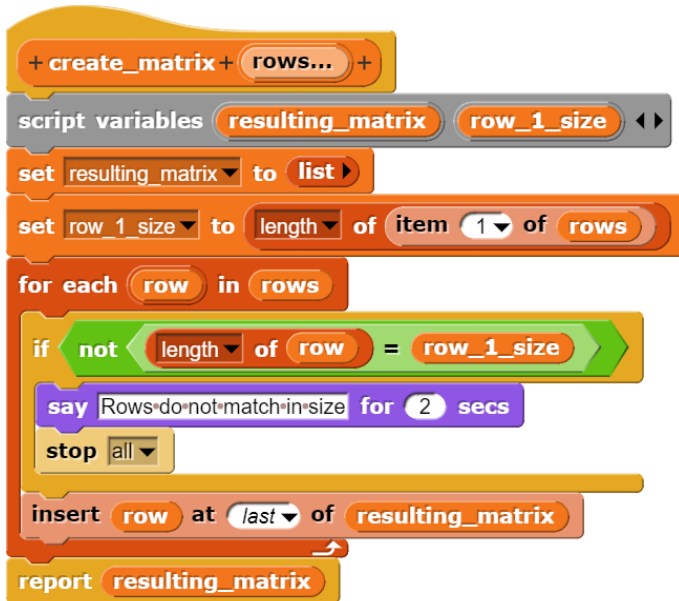
```

function* (SpriteMorph_prototype, current_process, get_custom_block, values) {
  yield;
  current_process.context.variables.addVar("values", values);

  current_process.pushContext(null, current_process.context, 1);
  current_process.popContextTillFunc();
  return current_process.getVarNamed("values");

  current_process.popContext();
}
  
```

## create\_matrix



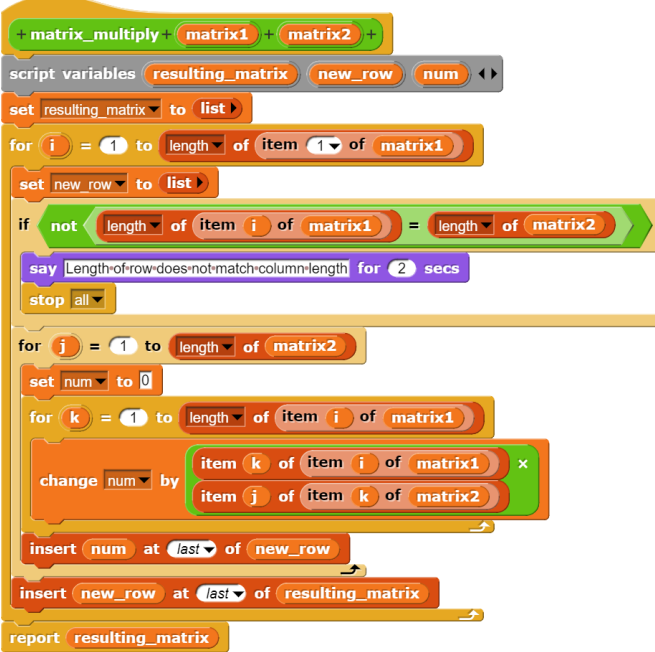
```
function* (SpriteMorph_prototype, current_process, get_custom_block, rows) {
  yield;
  current_process.context.variables.addVar("rows", rows);

  current_process.pushContext(null, current_process.context, 1);
  current_process.doDeclareVariables.apply(current_process, [new List(["resulting_matrix", "row_1_size"])]);
  current_process.setVarNamed("resulting_matrix", current_process.reportNewList.apply(current_process, [new List([])]));
  current_process.setVarNamed("row_1_size", current_process.reportListAttribute.apply(current_process, ["length",
  current_process.reportListItem.apply(current_process, [1, current_process.getVarNamed("rows")]]));
  current_process.context.variables.addVar("row");
  let list = current_process.getVarNamed("rows");
  current_process.assertType(list, 'list');
  let list_size = list.length();
  for (let i = 1; i <= list_size; i++) {
    if (list.isLinked) {
      current_process.context.variables.setVar("row", list.at(1));
      list = list.cdr();
    } else {
      current_process.context.variables.setVar("row", list.at(i));
    }
    current_process.pushContext(null, current_process.context);
    if (current_process.reportNot.apply(current_process, [current_process.reportEquals.apply(current_process,
    [current_process.reportListAttribute.apply(current_process, ["length", current_process.getVarNamed("row")]),
    current_process.getVarNamed("row_1_size")]])) {
      current_process.doSayFor.apply(current_process, ["Rows do not match in size", 2]);
      current_process.doStopThis.apply(current_process, ["all"]);
    }
    current_process.doInsertInList.apply(current_process, [current_process.getVarNamed("row"), "last", current_process.
    getVarNamed("resulting_matrix")]);

    yield;
    current_process.popContext();
  }
  current_process.popContextTilFunc();
  return current_process.getVarNamed("resulting_matrix");

  current_process.popContext();
}
```

## matrix\_multiply



```

function* (SpriteMorph_prototype, current_process, get_custom_block, matrix1, matrix2) {
  yield;
  current_process.context.variables.addVar("matrix1", matrix1);
  current_process.context.variables.addVar("matrix2", matrix2);

  current_process.pushContext(null, current_process.context, 1);
  current_process.doDeclareVariables.apply(current_process, [new List(["resulting_matrix", "new_row", "num"])]);
  current_process.setVarNamed("resulting_matrix", current_process.reportNewList.apply(current_process, [new List([])]));

  let dir = Math.sign(current_process.reportListAttribute.apply(current_process, ["length", current_process.reportListItem.apply(current_process, [1, current_process.getVarNamed("matrix1")])]) - 1);
  current_process.context.variables.addVar("i");
  for (current_process.context.variables.setVar("i", 1); (dir * current_process.context.variables.getVar("i")) <= (dir * current_process.reportListAttribute.apply(current_process, ["length", current_process.reportListItem.apply(current_process, [1, current_process.getVarNamed("matrix1")])])); current_process.context.variables.changeVar("i", dir)) {
    current_process.pushContext(null, current_process.context);

    current_process.setVarNamed("new_row", current_process.reportNewList.apply(current_process, [new List([])]));
    if (current_process.reportNot.apply(current_process, [current_process.reportEquals.apply(current_process, [current_process.reportListAttribute.apply(current_process, ["length", current_process.reportListItem.apply(current_process, [current_process.getVarNamed("i"), current_process.getVarNamed("matrix1")])]), current_process.getVarNamed("matrix2")])])]) {
      current_process.doSayFor.apply(current_process, ["Length of row does not match column length", 2]);
      current_process.doStopThis.apply(current_process, ["all"]);
    }

    let dir = Math.sign(current_process.reportListAttribute.apply(current_process, ["length", current_process.getVarNamed("matrix2")]) - 1);
    current_process.context.variables.addVar("j");
    for (current_process.context.variables.setVar("j", 1); (dir * current_process.context.variables.getVar("j")) <= (dir * current_process.reportListAttribute.apply(current_process, ["length", current_process.getVarNamed("matrix2")])]); current_process.context.variables.changeVar("j", dir)) {
      current_process.pushContext(null, current_process.context);

      current_process.setVarNamed("num", 0);

      let dir = Math.sign(current_process.reportListAttribute.apply(current_process, ["length", current_process.reportListItem.apply(current_process, [current_process.getVarNamed("i"), current_process.getVarNamed("matrix1")])]) - 1);
      current_process.context.variables.addVar("k");
      for (current_process.context.variables.setVar("k", 1); (dir * current_process.context.variables.getVar("k")) <= (dir * current_process.reportListAttribute.apply(current_process, ["length", current_process.reportListItem.apply(current_process, [current_process.getVarNamed("i"), current_process.getVarNamed("matrix1")])]); current_process.context.variables.changeVar("k", dir)) {
        current_process.pushContext(null, current_process.context);

        current_process.incrementVarNamed("num", current_process.reportProduct.apply(current_process, [current_process.reportListItem.apply(current_process, [current_process.getVarNamed("k"), current_process.reportListItem.apply(current_process, [current_process.getVarNamed("i"), current_process.getVarNamed("matrix1")])]), current_process.reportListItem.apply(current_process, [current_process.getVarNamed("j"), current_process.reportListItem.apply(current_process, [current_process.getVarNamed("k"), current_process.getVarNamed("matrix2")])])])]);

        yield;
        current_process.popContext();
      }
    }
    current_process.doInsertInList.apply(current_process, [current_process.getVarNamed("num"), "last", current_process.getVarNamed("new_row")]);

    yield;
    current_process.popContext();
  }
  current_process.doInsertInList.apply(current_process, [current_process.getVarNamed("new_row"), "last", current_process.getVarNamed("resulting_matrix")]);

  yield;
  current_process.popContext();
};
current_process.popContextIfFunc();
return current_process.getVarNamed("resulting_matrix");

current_process.popContext();
}

```