

CodeBlox

18-500 Spring 2020

Authors: Melodee Li, Eric Maynard, Aarohi Palkar
Electrical and Computer Engineering, Carnegie Mellon University

Abstract—CodeBlox is a set of blocks representing syntax that users, primarily children, can piece together to form programs. These programs are sent to an interpreter, which displays a GUI of the configuration of blocks, the output, errors that occurred and their locations, if any. The design of this project is split into three main parts: the interpreter/GUI, the embedded communication protocol, and the electronics within the tiles. The electronics comprise of a battery powered circuit that interacts with its environment entirely through IR and visible light sensors and emitters. Our final code can be found [here](#).

Index Terms—Circuit, Embedded, Interpreter, PCB, GUI, Sensor, Syntax, IR, Programming Teaching Tools

1 INTRODUCTION

Programming has traditionally been confined to a written form. Even programming tools aimed for kids, such as MIT Scratch, are usually intangible “drag and drop” programs. Because of this, coding is mostly only accessible to reading/writing learners. We hope to solve this dilemma by liberating coding from its written medium through the use of tangible building blocks. Kids will be able to connect these smart blocks together and build real programs. This will better engage visual and kinesthetic learners and allow children to be introduced to the field of programming at a younger age. Additionally, we hope to make coding more collaborative and exciting by allowing children to work with these tiles together in an educational environment.

In CodeBlox, users will learn how to code using tiles with syntax terms. A user should be able to piece tiles together to form programs. Once the user hits a “run” button, the program will be sent to a laptop with the CodeBlox interpreter which will then execute the code. The language should include basic programming control sequences, such as “loop”, “if”, variables, and mathematical operators; it should also include syntax to output the result of a program to a computer screen. Some example programs a user should be able to write using CodeBlox is printing out numbers and running a while loop.

Before COVID-19, we were going to use a robot to

perform the actions specified by the CodeBlox programs. However, due to constraints presented to us from this new situation, we have replaced the robot (and its on-board interpreter) with a interpreter and GUI on a laptop. The GUI displays the tiles read, the output of the program, and any errors that occur. Since we no longer had the robot to display output, we added a “print” syntax to our language, to visualize the output the robot would otherwise be receiving.

2 DESIGN REQUIREMENTS

The following are some requirements we outlined during our design process, followed up with the results of whether we met these requirements in whole, partially, or not at all.

2.1 Child Safety - Success

Description: Children may handle the final product in unexpected ways, and our design must defend against misuse. Thus, the design of the tiles’ interior as well as the interaction between tiles must be safe; there should be no way to short any two components of any part of the system and harm the user.

Results: Our project uses no external electrical connections and each tile is powered by an on-board battery. The tiles could then be totally sealed to prevent tampering with by the child.

2.2 Latency - Fail

Description: The entire system must also run with low latency. External research has demonstrated that the average human attention span after a stimulus is 8 seconds.[4] Therefore, the system should take 8 seconds to run after the “run” button is pressed on the master tile. This includes the tiles communicating the tile topology between themselves, the master communicating the full topology to the interpreter, and the GUI displaying the results.

Results: When bench-marking our results we discovered that our compilation time varied with the configuration of our tiles. When placed in a grid, our tiles compiled much faster than when placed in a line. In a line it takes 6.17 seconds to compile 9 tiles,

whereas in a perfect square, it only takes 4.3 seconds to compile those same 9 tiles. Seeing that the linear formation is our worst-case configuration, we did our benchmarking that way. Below in Figure 1 is a graph showing compilation time v.s the number of tiles. It shows a very linear runtime behavior, which we can extrapolate to guess that the compilation of 20 tiles would take 14.222 seconds. This exceeds our goal of 8 seconds. However, it is important to note that we are powering our microcontroller with a 1MHz internal clock. At our given voltage, we could increase it to 4MHz, thus quadrupling our compilation time. If we did this, we would drop our compilation time to 3.56 seconds, which is well within our target goal.

Compilation Time vs. Number of Tiles in a line

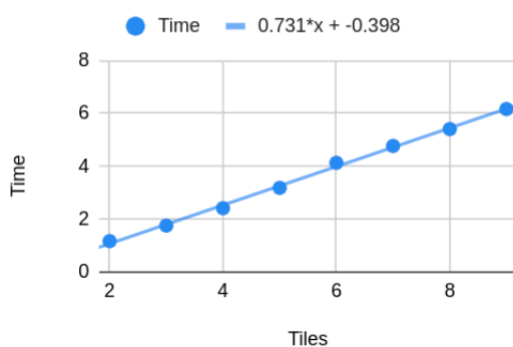


Figure 1: Compilation time vs. the number of tiles placed in a single line. Compilation time varies linearly with increasing number of tiles.

2.3 Programming Language - Partial Success

Description: The programming language should be designed in a way that would be intuitive for children while being robust to program with. Therefore, the language should support the following constructs:

- booleans and numbers
- loops (e.g. "while")
- conditionals (e.g. "if", "else")
- output in the form of print statements
- variables and assignments
- math (e.g. comparisons, numbers, operations)
- functions in zero or one variable

The language does not have to support:

- strings
- parentheses

- higher order mathematics
- functions with 2+ arguments

Results: The programming language CodeBlox uses supports booleans and numbers, loops and conditions, print statements, variables and assignments and a variety of mathematical operations. We also included a dynamic typechecking system in order to teach the children about invoking mathematical operations with the correct types. However, we did not get around to implementing functions, since we ended up adding a GUI feature to display the output of the program instead of simply relaying output to a robot. Because of this extra GUI addition, the implementation of functions was pushed behind, leading to it only being half completed (designed, but not implemented) by the end of the semester.

2.4 Battery Life - Success

Description: The toy should support 15 hours of continuous usage over a 3 month period. If our product meets these requirements, it could be used in settings such as CMU's SPARK Saturdays[5] for a semester's worth of classes without needing to replace the batteries. To test this, we will do a component-by-component energy analysis.

Results: Our tiles draw 18.5 uA of quiescent current. This means they could last for 6 years in sleep mode. When being used, they only draw a few milliAmps for the few seconds that the tiles are compiling. Thus we have very much met our power goals.

2.5 Pad Syntax Accuracy - Success

Description: Measuring the encoding should have a 100% accuracy, because it would be frustrating for a child to recompile their code multiple times for an error that is not their fault.

Results: With paper syntax pads, we were getting about 95% accuracy in measurements. However, when we tested a more rigid material like cardboard or wood, we measured 100% accuracy.

2.6 Inter-Tile Messages Accuracy - Success

Description: The data sent between tiles should also have a 100% accuracy rate. This requirement is necessary because any small error can cause readings that a child would be very confused by.

Results: We tested our system on several different topologies, and 20 out of 20 of our compilations contained all of the correct tiles and their locations. Thus, we believe that we have 100% accuracy in regards to reading tile topology.

2.7 Syntax Pad Construction - Success

Description: The syntax pads should be easy to construct with household items. We want replacement pads to be reliably made using only materials found at stores like Walmart.

Results: Although paper or cardboard pad construction doesn't yield the best results in terms of reading accuracy, a parent could definitely make a new pad using cardboard, paper, tape and sharpie markers in a pinch.

3 ARCHITECTURE OVERVIEW

3.1 System Configuration

We built several connectable tiles with custom PCBs and a microcontroller on each board. These tiles communicate with each other and are able to determine their relative location to each other and communicate with one master tile. This master tile determines the absolute location of all tiles and communicates this topology to the interpreter via serial communication. The interpreter then executes the code, and then display the results in a GUI.

3.2 Tiles

Each tile represents one syntax term. The syntax term a tile represents is determined by a pad that is placed on top of the tile. The tile itself is generic and can be configured by any syntax pad. The tile houses a micro-controller and sensors which read which pad is above the tile and communicates this to neighboring tiles, with the ultimate goal to transmitting this information back to the master tile.

To read the pad on top of the tile, the tile has 6 reflective sensors mounted on top. These reflective sensors measure whether the spot on the pad placed directly above the sensor is dark or light. By placing 6 colored spots on the pad, the tile can read this 6 bit encoding to determine which pad is placed on top of it. This allows for 64 different pad types.

To accomplish tile to tile communication, we use IR LEDs to transmit data back and forth between adjacent tiles. Each tile has 4 onboard IR emitters and 4 IR receivers, with one emitter and one receiver per side of the tile. Tiles pulse the IR emitters to communicate with adjacent tiles.

Each tile is powered with its own battery. Since the batteries create a limit on how much power these tiles can use before the batteries die, the tiles spend most of the time in power-saving sleep mode. A tile will then only be awake if its neighbor sends a wake-up IR signal

to indicate that a compilation has started.

Mechanically, these tiles attach to each other using magnets. By orienting the magnets as such in Figure 3, two tiles only connect in the correct orientation, or if flipped 180 degrees. Because our design is symmetric, a 180 degree flip is still valid.

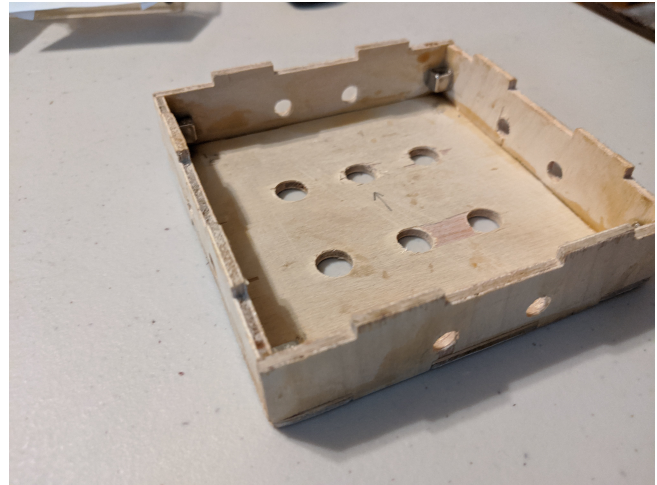


Figure 2: An upside-down view of a constructed tile enclosure, without the bottom wood piece. The six holes on the top are for the reflective sensors to read the 6 bit encoding of the pad placed on top of the tile. The two holes on each side are for the IR emitters and receivers.

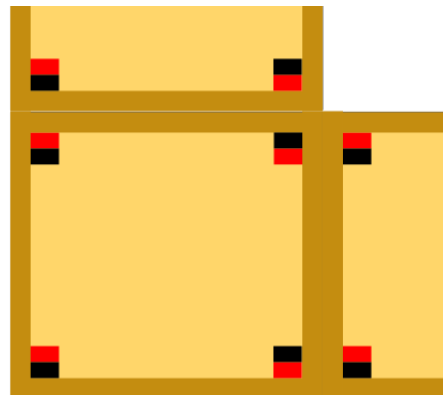


Figure 3: Orientation of the magnets inside each tile. Opposite polarities attract, making the two tiles only connect correctly in a horizontal orientation (with respect to the orientation of the six reflective sensors on top of the tile)

3.3 Pads

Each pad is double sided. The top side contains the syntax associated with the pad. The bottom side of the pad contains the binary encoding of the syntax. The bits are represented by squares, where a white square encodes a "1" and a black square encodes a "0".

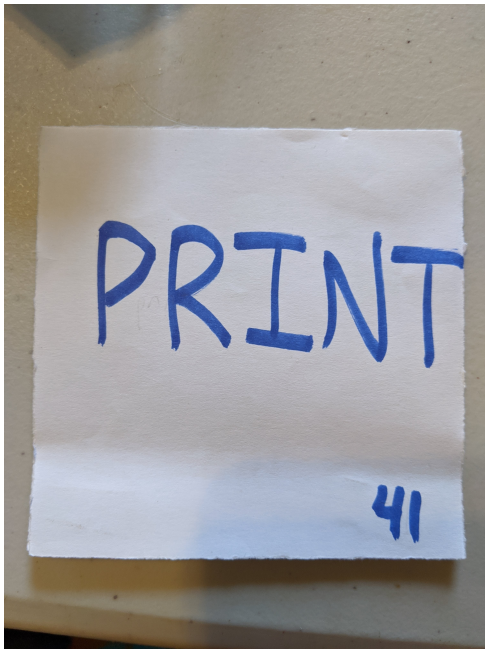


Figure 4: Top side of a syntax pad, displaying the syntax and its corresponding encoding value. The pad can be easily made using paper and a sharpie, as represented in this picture. The encoding value, in this picture 41, is unique to a syntax, in this picture "print", and the binary encoding of 41 will be colored in the back.

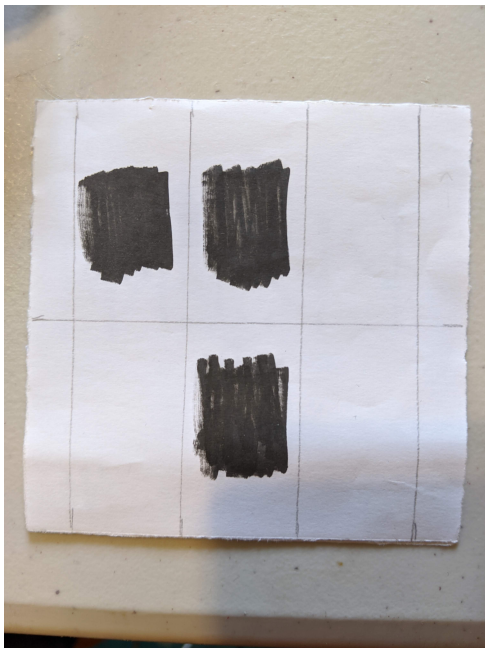


Figure 5: Bottom side of a syntax pad, displaying the binary encoding through sharpied-in squares. Black squares represents a "0" while a white square represents a "1". In this picture, the encoding is read from bottom, 101001, which corresponds to a 41, a "print" command.

3.4 GUI

The GUI resides on the laptop where the user receives output. Once the interpreter has executed all the code in the the CodeBlox program, the GUI receives four inputs: the input blocks to the program, the file in which the interpreter writes its output (more detail about the interpreter architecture can be found in the subsystems section), whether any errors occurred during execution and which tile caused the error. A sample output of the GUI with and without error is shown in Appendix G.

4 SYSTEM DESCRIPTION

4.1 Electrical Subsystem

The electronics of each tile are built upon a custom made PCB. The PCB contains through-hole components. Appendix D shows the schematic of this circuit and the PCB layout.

The circuit consists of an Atmega328p as the micro-controller, 4 IR communication circuits for inter-tile communication, 6 reflective sensor circuits to read the top tile encodings, and support for ISP programming and serial communication for debugging.

See Appendix D for an image of the circuit schematic and images of the actual soldered PCB.

Most of the details about the electrical system can be seen in the schematic and in the Design Trade Studies section below.

4.2 Embedded Subsystem

4.2.1 Topology Formation

Once a user constructs their program and presses the "run" button, the tiles will begin communicating with each other and figuring out the configuration of the program. This methodology can be broken up into three parts:

1. Starting from the master tile, form a tree graph amongst the tiles, with parent-child relationships. The master tile is at the top of the hierarchy.
2. Starting from the "leaf" tiles, forward the locations and configurations of tiles from children to parents. By the time the forwarding reaches the master tile, it will receive the full topology of the tiles.
3. The master tile sends the topology to the interpreter via serial.

The lifecycle of a tile (let's call it Tile A) during this methodology can thus be broken down into five states:

1. Tile A waits for another tile to request to be its parent.

2. Tile A receives a parent request and immediately sends its own configuration to the parent. It ignores any other parent requests.
3. Tile A requests for all of its adjacent tiles (besides its parent) to be its child.
4. The tiles that agree to be Tile A's child will send the configurations they have learned to Tile A. As soon as Tile A receives any configuration, it will asynchronously forward it to its parent.
5. When Tile A receives a "done" message from a child, it assumes that the child is done sending information. Once Tile A receives "done" messages from all of its children, it sends a "done" message to its parent.
6. Tile A resets back to its original state.

Let's say Tile A is sending the topology it knows to its parent, and it is currently sending information about Tile B. When Tile A sends a topology, the message is composed of four words: a word saying that the message is a "tile info" message, a word sending the x coordinate, a word sending the y coordinate, and a word sending the encoding. The x-coordinate and y-coordinate are of Tile B relative to the position of Tile A, with Tile A being at coordinate (0,0). These coordinate messages will represent a signed binary number, with the most significant bit indicating the sign of the number. The encoding that Tile A sends will contain the binary encoding of Tile B's syntax.

4.2.2 Inter-tile messages

Each tile spends most of its lifecycle in a power-saving sleep mode. In the Atmega328p environment, we are putting these tiles in this sleep mode using the *LowPower.h* library.^[1]

A tile (let's call it Tile A) will wake up when a neighboring tile (Tile B) sends a high IR signal. This signal will trigger an interrupt that will cause Tile A to wake up and start reading the rest of Tile B's messages. Tile B will wait an entire word for Tile A to wake up, and then begin communicating with it.

Each "word" of transmission consists of 10 bits:

- **Bit 0: Synchronization Bit.** This high bit will mark the beginning of a message (and the end of a previous message) to allow Tile A to synchronize with Tile B.
- **Bits 1-8: Message Bits.** These bits contain the message that Tile B is sending.
- **Bit 9: End bit.** If another word follows this word, this bit is the same bit as the next word's Synchronization Bit.

Messages can be composed of 1-4 words. The first word of a message describes the type of message being sent, and the subsequent words (if any) describe the contents of the message. The following are different types of messages that could be sent:

- **Alive message.** A tile would send this message to indicate that it is still alive and present. This message contains a single word describing that the message type is "alive".
- **Parent request message.** A tile would send this message to request to be its neighbor's parent. This message contains two words. The first describes that the message type is "request parent". The second describes the location of the tile relative to the neighbor that it is sending the message to. The purpose of the second word is so a tile can figure out if it was placed upside-down. For example, if a tile receives from its top side a "Request Parent - Bottom" message, the tile will realize that the neighbor is actually below it, not above it. Then it will flip its internal orientation accordingly.
- **Tile information message.** A tile would send this message to send a particular tile's information. This message contains four words. The first describes that the message type is "tile information". The second and third describe the x- and y-coordinates of the tile, and the fourth describes the syntax encoding.
- **Done message.** A tile would send this message to indicate that it has sent all of the tile information it knows, and that it will discontinue communication with the neighbor. This message contains a single word describing that the message type is "done".

A demonstration of how the communication protocol would be implemented with a sample program can be found in Appendix C.

4.3 Interpreter and GUI Subsection

The interpreter is written in Python. The syntax of the Codeblox language is similar to that of Python's; however, the size of programs is limited by the number of tiles a user currently possesses.

First, the top level program on the computer parses the stream from the master tile as an $m \times n$ array that can be parsed by the interpreter. This array is then passed to the interpreter, which runs the code defined by the tiles sequentially, running through each row.

Since the only kind of output we have is displaying to the screen (print), the interpreter writes all print statements to a local file, including any possible errors. Once the interpreter finishes running (more on non-terminating programs later), the GUI begins to run.

The GUI is implemented as a pygame program with three features: first, it displays a virtual representation of the user's CodeBlox program. The benefits of this is twofold: the user is able to visualize any potential errors with their code, and if the virtual representation is different from their tiles, the user knows that the error is caused by the serialization process, and not their code.

As mentioned, the GUI also displays error statements and outputs. A box at the bottom is used to display output to the user, and if there is any error in the program, the GUI displays a message saying what kind of error it is in the output box, as well as highlights the faulty tile in the virtual representation of the program. This was designed keeping the user in mind, since it is very beneficial to the user to know why their program isn't working, which tile they should fix and what they should do to fix the issue.

The following describes the formal specification of the CodeBlox programming language [3]:

Types: `bool` | `num`

Expressions:

Conditional:

```
if( $e : \text{bool}$ )( $e'$ )
if( $e_1 : \text{bool}$ )else( $e_2$ )
while( $e : \text{bool}$ ) $e'$ 
```

Variables:

```
 $v_1, v_2, v_3, v_4, v_5$ 
```

Operator:

```
( $e_1 : \text{num}$ ) + ( $e_2 : \text{num}$ ) :  $\text{num}$ 
( $e_1 : \text{num}$ ) - ( $e_2 : \text{num}$ ) :  $\text{num}$ 
( $e_1 : \text{num}$ )  $\times$  ( $e_2 : \text{num}$ ) :  $\text{num}$ 
( $e_1 : \text{num}$ )  $\div$  ( $e_2 : \text{num}$ ) :  $\text{num}$ 
```

Comparator:

```
( $e_1 : \text{num}$ ) = ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
( $e_1 : \text{num}$ )  $\neq$  ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
( $e_1 : \text{num}$ ) < ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
( $e_1 : \text{num}$ )  $\leq$  ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
( $e_1 : \text{num}$ ) > ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
( $e_1 : \text{num}$ )  $\geq$  ( $e_2 : \text{num}$ ) :  $\text{bool}$ 
```

Logic:

```
( $e_1 : \text{bool}$ ) and ( $e_2 : \text{bool}$ ) :  $\text{bool}$ 
( $e_1 : \text{bool}$ ) or ( $e_2 : \text{bool}$ ) :  $\text{bool}$ 
```

Commands:

```
print( $e : \tau$ ) : ()
```

Nop:

```
nop
```

The interpreter first receives a stream of $m \times n$ tile codes, representing the position and code of the tiles in the current CodeBlox program. It then calls `run_code()` on this stream. It interprets and runs each line at a time. More details about implementation can be found in the software diagram in Appendix F. If an error is encountered, it stops execution and uses the in-built python exception handling mechanism to write an error to the output file and terminate. If no error is

encountered, the program runs to completion.

Once the interpreter is done running, as described above, the GUI now runs in a pygame program. There are four inputs to the GUI program: the blocks themselves, the output file where the output of the interpreter is written, whether an error was encountered in the interpreter and the location of such error. The GUI reads the input blocks as draws a virtual representation of them on the screen. Then, it reads the output file and writes the output on the screen. If an error was encountered, first it highlights the tile corresponding to the location passed in as the fourth parameter. Then, in the output section, it prints out the type of error, from the output file.

We also perform simple error checking on the tiles. The following are the types of errors that can be thrown:

Errors:

SYNTAX : Syntax error.

TYPE: Type mismatch error.

INDENT : Unexpected indentation error

DIVZERO : Division by zero error.

OVERFLOW: Program has run for too many iterations.

Since the GUI runs after the interpreter terminates, something we were limited by was what would happen if the user wrote an infinite loop. Since CodeBlox programs are meant for young children to learn programming, we believe that teaching the children that an infinite loop was happening (and thus throwing an overflow error) was more beneficial than the program looping and not providing any output. Since we are not controlling a robot any more as well, we do not run into a situation where a program would necessarily need to infinite loop. Therefore, after 50,000 iterations of a while loop, the interpreter erases all output to the output file, and terminates with an overflow error. Now the GUI can be called and the child can see that they inadvertently wrote a loop that does not terminate.

5 DESIGN TRADE STUDIES

5.1 Optical vs Electrical Communication

When determining how these tiles would communicate, we initially thought that direct electrical contacts would be the ideal medium to communicate across. A direct electrical connection allows for short rise times and low energy communication.

However, as we learned from our Build18 project, these electrical contacts are very finicky and unreliable. It is challenging to form a solid electrical connection between 4 contacts without using some form of a socket. Additionally, electrical connections need to be external to the tile, and thus require additional construction

beyond the PCBs.

To address these issues, we decided to use IR emitters and receivers to communicate data. These do not require a robust connection between neighboring tiles nor do they require any additional construction beyond assembling the PCBs. However, they do draw more current and have slower rise times than direct electrical connections.

5.2 Power Supply

We originally planned to power all the slave tiles from the master tile, but realized that this had severe safety implications. At the time, we anticipated that each tile would draw 100mA of current, and thus, powering 30 tiles would require a 3 amp power supply. This posed a fire hazard and risk for shock.

Our first solution was to switch to using onboard batteries to power the tile. We chose two unregulated AAA batteries to power the circuit. We chose to not put a voltage regulator on the batteries because of the voltage regulators' high quiescent current. Instead, we lowered the brown-out detection from the microcontroller to 1.8volts and lowered the clock speed of the microcontroller to 1MHz to allow an operating voltage range of 1.8 to 3 volts. This allows the two AAA batteries use almost all of their energy before the circuit no longer functions. All components on the board were spec'ed to this voltage range.

Because our tiles operate within a voltage range instead of at a fixed voltage, our emitters, receivers, and reflective sensors needed to operate within a voltage range as well. Experimentally, this proved fine for the IR circuitry, but for the reflective sensors, we measured inconsistent values for different voltages. To address this issue, we powered the reflective sensors with their own voltage regulator, which was only powered while measuring the sensors. Thus, we obtained the consistency of a voltage regulator without the high quiescent current cost.

5.3 Internal or External Clock

When designing our communication protocol, we considered whether we wanted to use an external clock or synchronize using the tile's internal clock. If we used an external clock, we would need two emitters and two receivers per side. Having two emitters on the same side of the tile would increase the chance of interference and increase the cost per tile.

By synchronizing using the internal clock, we can get away with only one receiver and one emitter per side. We chose this paradigm because it's cheaper and causes less interference. However, as a consequence of

synchronizing using the tile's internal timer, we had strict timing deadlines to meet and had to use a slower transmission rate.

5.4 Pulses vs Continuous Emission

We originally intended for the IR emitters to be on for the entire duration of a clock cycle when emitting a logical 1. However, this would require 20mA per side for an entire clock duration. If all 4 sides are transmitting and we were reading our reflective sensors (+120mA), our current draw would be 200mA. This is too high to be powered by batteries for an extended period of time. Our solution was to only emit the IR emitters and reflective sensors in brief pulses that consume a small fraction of this energy.

5.5 Interrupts vs. Polling

Originally, our embedded protocol would poll each IR photo-transistor to see if its value had changed. This required continuous IR emission to work. To work with pulses, we enabled external interrupts to handle signal changes. We originally did not do this because interrupts aren't surfaced in the Arduino IDE for the Atmega328p, but we discovered that the micro-controller does support these.[2] Before using interrupts, we had to leave the signal on for a whole clock cycle, but with interrupts, we could send a brief pulse of only 100ns in width at a 1kHz rate and have the interrupt handler not miss any pulses.

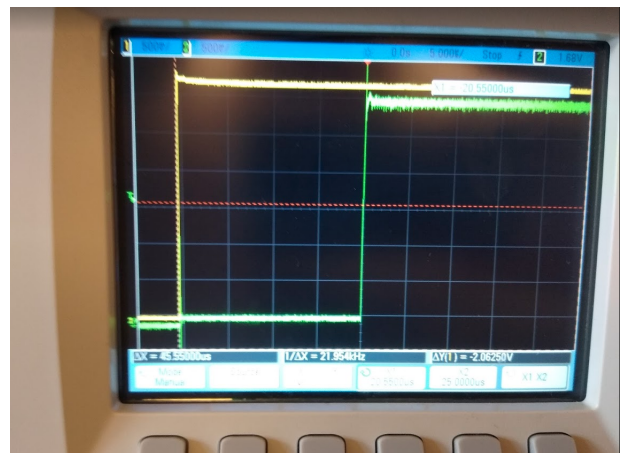


Figure 6: Detecting pulses and responding in 20uS.

5.6 On/Off Switch vs. Sleep Mode

Since we designed each of our tiles to be individually-powered, we needed a mechanism to make sure that the tiles wouldn't always be fully on, or else we would be wasting power and wouldn't meet our battery life requirement. We debated over two solutions: 1) each tile would have an on/off switch, or 2) each tile would be in a power-saving sleep mode when it is not being

used. We eventually settled on the sleep mode solution, because although the switches may save more energy in the long run, they create a more tedious user experience. In sleep-mode, we measured a current draw of 18.5 μA on the micro-controller, which is nearly negligible.

5.7 Clock Speed

We arrived at a minimum clock speed of 1kHz computing the load time of 30 linearly stacked tiles and choosing a clock speed that could reduce the time to 8 seconds, which is our requirement.

Consider an extreme scenario, in which 20 tiles are lined up in a row. Total message sent =

$$\begin{aligned} &= n + (1 + 3(1)) + (1 + 3(2)) + \dots + (1 + 3(n)) \\ &= n + n + 3(1 + 2 + \dots + n) \\ &= 2n + 3\left(\frac{n(n+1)}{2}\right) \\ &= O(n^2) \end{aligned}$$

Worst case runtime is $O(n^2)$, where n is the number of tiles.

Let $n = 20$. Then the total number of messages sent is $2(20) + 3\left(\frac{20(20+1)}{2}\right) = 670$ messages. Our word size is 8. Therefore the number of clock cycles required is $670 \times 8 = 5360$.

Our constraint is that the total time should take 8 seconds.

Assuming all other operations take negligible time, our clock speed would have to be $\leq \frac{8\text{sec}}{5360\text{cycles}} = 1.493$ ms/cycle.

5.8 Interpreter Language

After moving from the robot to an interpreter housed on a laptop, we were not constrained to writing our interpreter in C or C++. We compared C/C++ against python and a functional programming language such as OCaml and evaluated the pros and cons of each. These pros and cons can be found below:

1. C/C++
 - + Fast
 - Difficult to use higher level data structures
2. python
 - + Easy to understand and write
 - + Familiar
 - + Easy deserialization library
 - + Pygame for GUI
 - Slow
3. OCaml

- + Can do recursive computation easily
- + Compilers usually written in a functional language
- Unfamiliar
- Less support for GUI

Overall, we decided to work with python since we would not be writing large programs that the latency would contribute significantly to the endpoint latency of the system, and since python gave us the most support with the GUI.

6 PROJECT MANAGEMENT

6.1 Schedule

We have split up our project into four mostly parallel tasks: PCB design and soldering, embedded protocol design and implementation, tile enclosure design and assembly, and interpreter/GUI design. Our Gantt chart (see Appendix B) reflects these splits accordingly.

6.2 Team Member Responsibilities

Due to COVID-19, the team decided to divide up responsibilities such that Aarohi worked on the interpreter/GUI independently, while Eric and Melodee worked on the circuits, firmware, and tile construction together. We assigned these responsibilities so that Aarohi wouldn't have to work on any physical part of the project, since she was not in the same location as Eric and Melodee.

- PCB Design: Eric was responsible for the PCB design since he had the most experience with PCB design, especially related to the Build18 project. He tested each component to make sure they behaved as expected, and tested the PCBs when they arrived.
- Soldering: Eric and Melodee worked together in soldering the components to the PCBs, using a soldering iron that Eric had.
- Embedded Protocol: Eric and Melodee developed an embedded protocol for communication between the master and slave tiles. Melodee fleshed out the protocol in Python, and then they worked together on writing the C++ firmware, which they then uploaded to the tile microcontrollers. Melodee also integrated the master tile to the interpreter via serial communication.
- Tile Construction: Eric and Melodee cut out wood pieces for the tile enclosures, drilled holes for the sensors, and used wood glue to glue them together. They then used epoxy to glue magnets to the corners of the tiles, and then used a hot glue gun to glue the PCBs to the wood.

- **Pad Construction:** Melodee constructed the pads with paper, tape, and sharpie.
- **Interpreter:** Aarohe was responsible for developing the interpreter logic. Aarohe worked on fully defining these and implementing them in the interpreter, as well as rewriting the majority of the interpreter to make it more robust to requirements such as conditional statements and loop exit conditions. She also added dynamic type checking and error checking to the interpreter. Aarohe was also responsible for parsing the stream from the master tile.
- **GUI:** Aarohe was responsible for creating a simple GUI to view the output of the CodeBlox program, as well as a virtual representation of the tiles along with displaying information regarding errors. Aarohe compared pygame with tkinter for creating the GUI, and ultimately decided on pygame since it gave her the most control over text representation.

6.3 Budget

See appendix E for an itemized breakdown of our budget. Our total cost of all materials is \$493.42.

7 RELATED WORK

CodeBlox is based off the original CodeBlox, a project from Build18 2019. We have improved communication between the tiles to make it more reliable, and changed the communication protocol to account for this change. We've also added more functionality to the interpreter, which now supports error checking and a dynamic typechecker.

Some related work to CodeBlox is MIT Scratch or MIT AppInventor, which support drag-and-drop syntax "building blocks" to create a program. CodeBlox takes this to the next level by bringing these blocks to the physical world.

We also compared our work to that of Dave Touretzky's from the Carnegie Mellon Robotics Department. [6] This work, named Kodu, is very similar to ours, in that it uses physical tiles to teach children programming. However, Kodu largely focuses on using colors and shapes in order to reinforce programming paradigms, instead of focusing on strict python-like syntax like CodeBlox. Additionally, Kodu was fortunate to be able to test their tiles with second grade students, something CodeBlox was unable to do.

8 SUMMARY

Overall, we believe that our project was very successful. We were fortunate to be able to continue

the project with minimal re-scoping, even when the class transitioned to online instruction. Although the system is slower than our initial goals and the language doesn't support functions, these design goals are achievable if we decide to continue with this project.

8.1 Lessons Learned

We can identify several lessons that we have learned throughout the semester.

- **Microcontroller with more memory.** As we were writing the firmware for the tiles, we tried to structure our code in a way that would take advantage of C++'s function lambdas. However, we realized that this uses up a lot of the Atmega328p's memory, and we had to restructure our implementation. If we had used a microcontroller with more memory, we wouldn't have to be so memory-conscious.
- **Greater visibility between IR sensors.** Since we couldn't use a laser cutter to cut our wood, and since the soldered IR sensors could be slightly shifted from tile to tile, it is possible for us to construct a tile such that the IR sensors aren't lined up with the side-holes of the tile. This could cause a lot of IR emitter pulses to not be received by the IR receivers. If we were to reconstruct these tiles, we would have each side of the tile have one large hole that both the IR emitter and receiver can emit/read through.
- **Indicator light during compilation.** We considered adding an indicator light to the master and/or slave tiles, so that a child would receive feedback that the system is compiling. Having compilation lights on the slave tiles could also indicate to a user that the tile is being included in the topology. This idea was not a high priority for us, so it ultimately did not happen. In the next iteration of CodeBlox, we believe that having an indicator light would improve the user experience, especially since we would want to return to using the robot instead of a GUI.
- **Surface mount components.** Due to COVID-19, the ability to use surface mount components wasn't in our control, since we didn't have a reflow oven at hand. But in the next iteration we would use surface mount instead of through-hole components. We've realized that soldering many PCBs is a tedious task, and we would save a lot of time if we learn how to use a reflow oven.
- **Multiple serial cables.** We only bought one serial cable to debug the tiles with, which forced us to only read serial output from one tile at a time. Debugging would be easier and faster if we had more than one serial cable.

- **Designing all parts of the programming language.** A major reason why functions took a while to implement and were left unfinished was that they were not scoped well at the beginning. We were unable to determine whether functions should be treated as expressions (which are not allowed to appear by themselves in a line in our programming language), or as a command/variable assignment statement. Since in our language variables are allowed to modify state (not a pure programming language), functions could be categorized as both expressions and commands, which would mean restructuring the interpreter, something we could have avoided if we had specified out functions in full detail before implementation.

References

- [1] “Arduino Low Power - How to run ATMEGA328P for a year on coin cell battery”. In: *Home Automation Community* (Feb. 2020).
- [2] “Arduino Pin Change Interrupts”. In: *The Wandering Engineer* (Aug. 2014).
- [3] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.
- [4] Kevin Mcspadden. “You Now Have a Shorter Attention Span Than a Goldfish”. In: *Time* (May 2015).
- [5] *SPARK Saturdays*. 2020.
- [6] David S. Touretzky. “Teaching Kodu with Physical Manipulatives, Vol 5 No. 4”. In: *ACM Inroads* (Dec. 2014).

Appendix A

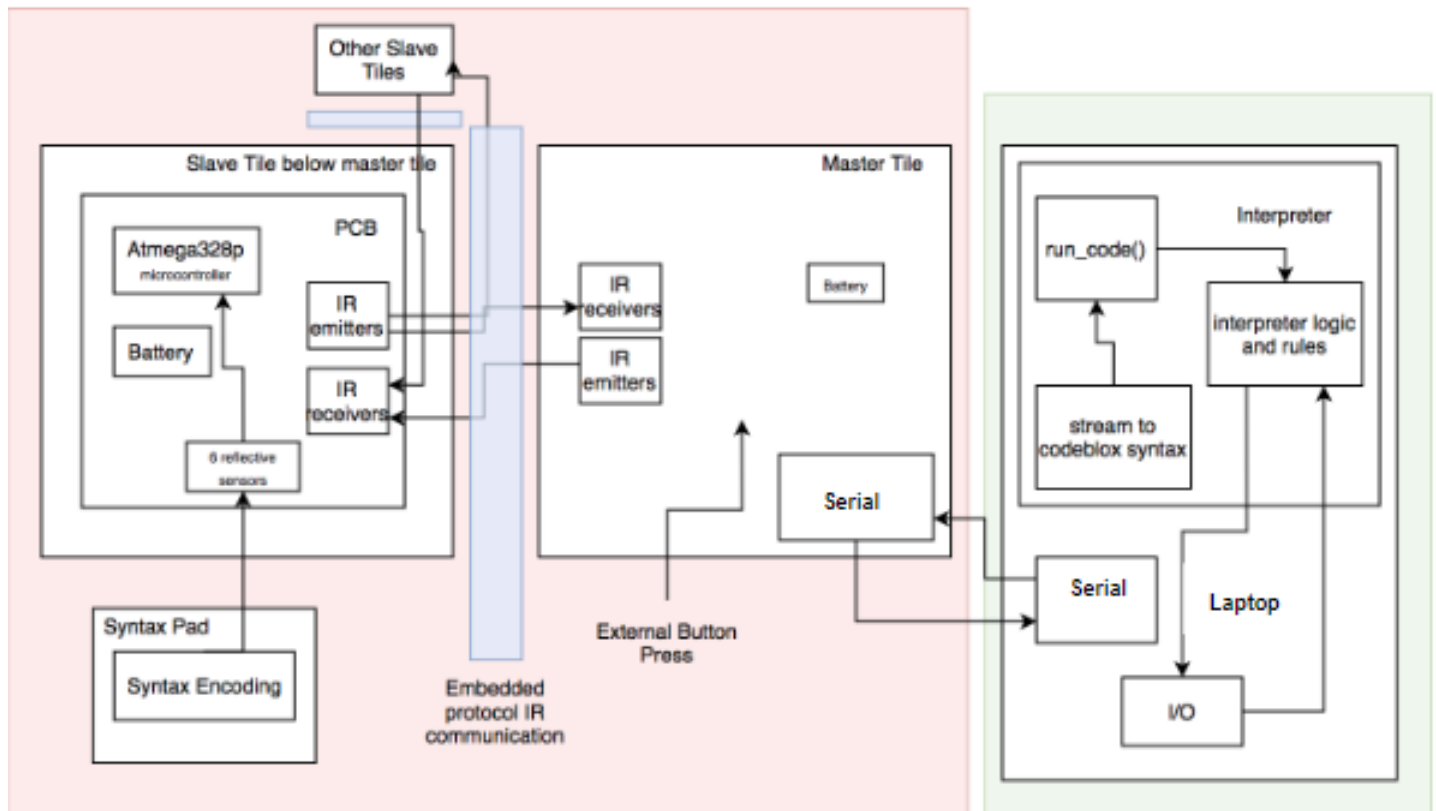


Figure 7: Overarching system diagram for CodeBloX. Contains three major components: slave tiles, the master tile, and the interpreter/GUI. Communication between slave tiles and the master tile is performed through IR, and communication between the master tile and the interpreter is performed through serial communication. Hardware parts are in the red box, and software parts are in the green box. The master tile, slave tiles, and syntax pads were assembled by us. Everything in the diagram was bought except for the syntax pad materials and laptop.

Appendix B

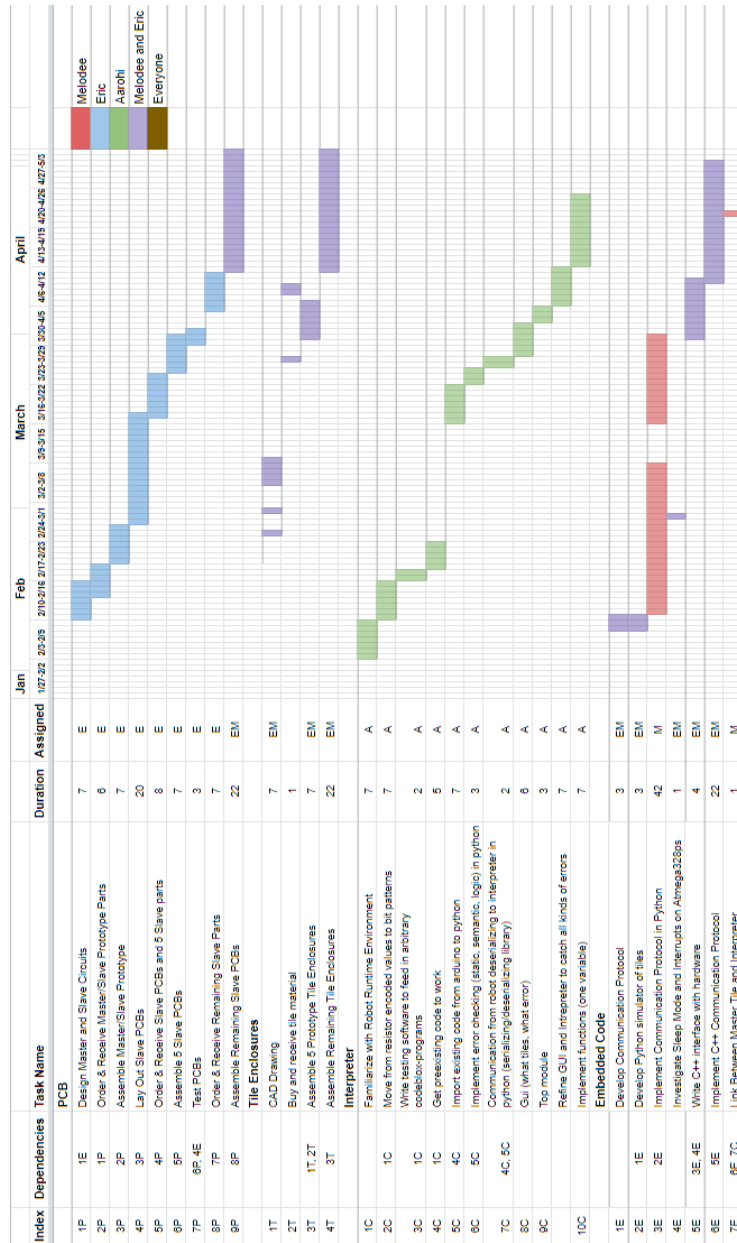


Figure 8: Gantt Chart. The tasks were split primarily into PCB, Embedded Communication and Interpreter/GUI, each of which was handled by Eric, Melodee and Aarohi respectively. An additional task, Tile Enclosures, was performed by Eric and Melodee

Appendix C

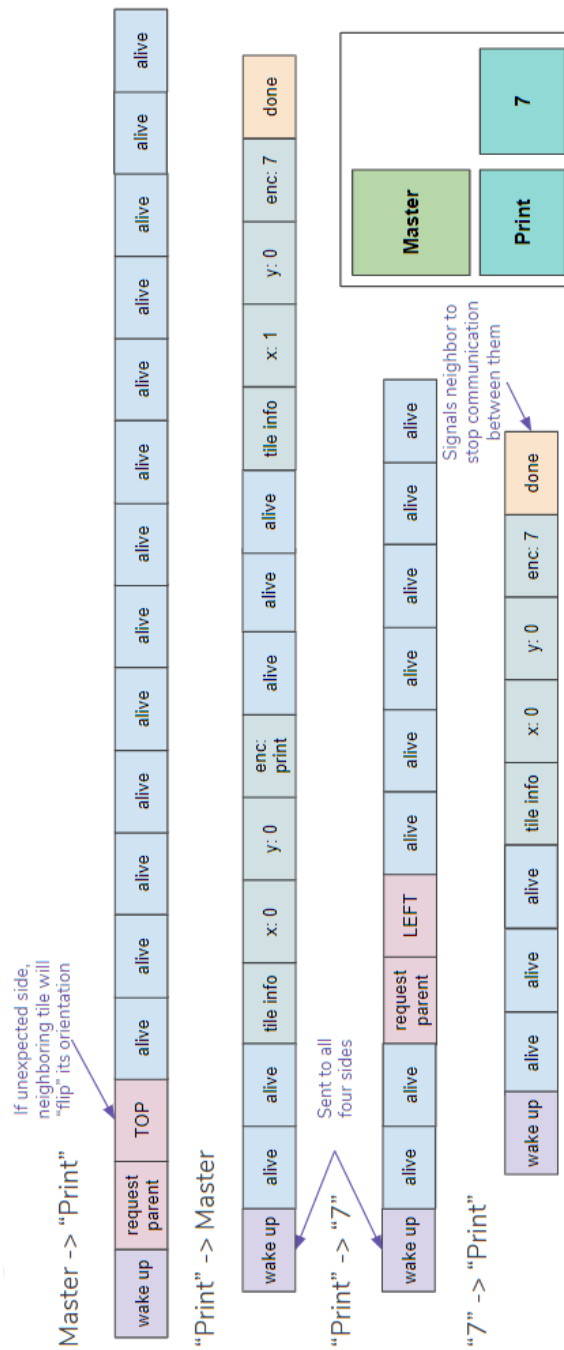


Figure 9: Example of possible program, with messages that would be sent during compilation. Each colored rectangle represents a word. The example program consists of a master tile, a "print" tile below it, and a "7" tile to the right of the "print" tile.

Appendix D

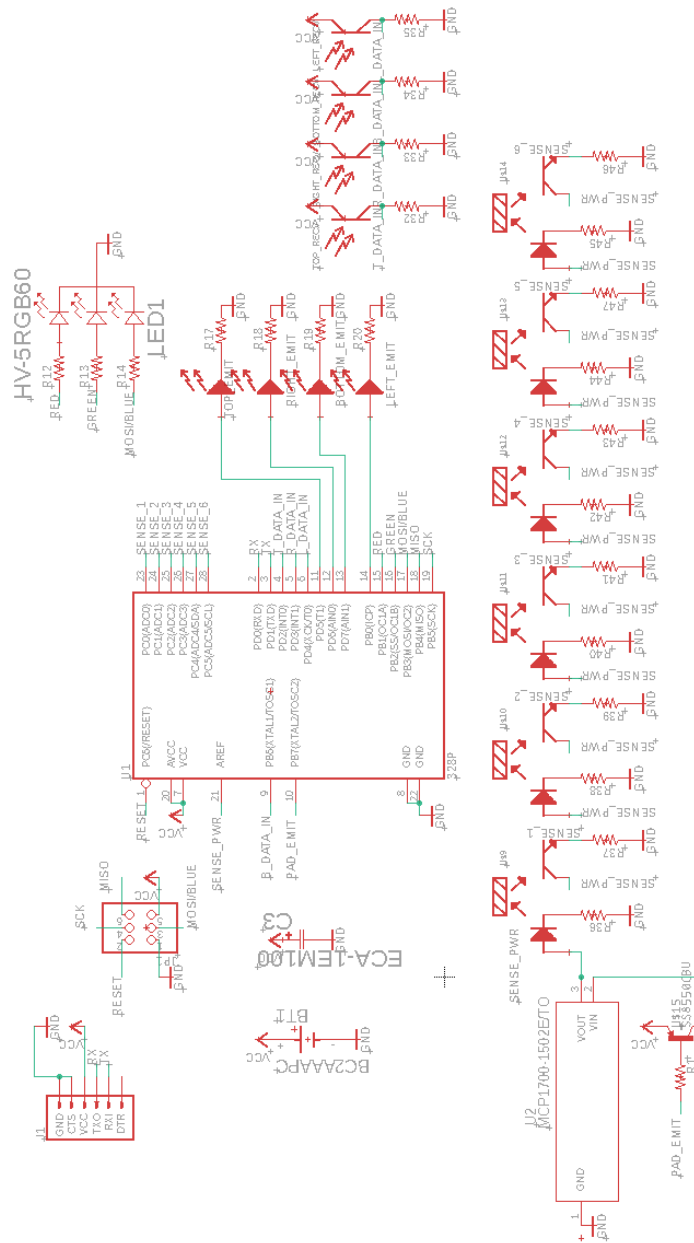


Figure 10: This is the circuit schematic for the tile. The circuit is composed of 4 IR emitter and receiver circuits, 6 reflective sensor circuits which are powered by a voltage regulator which is controlled by a PNP BJT. These circuits are controlled by an atmega328p microcontroller with ISP and serial interfaces for programming and debugging. This is all powered by two AAA batteries.

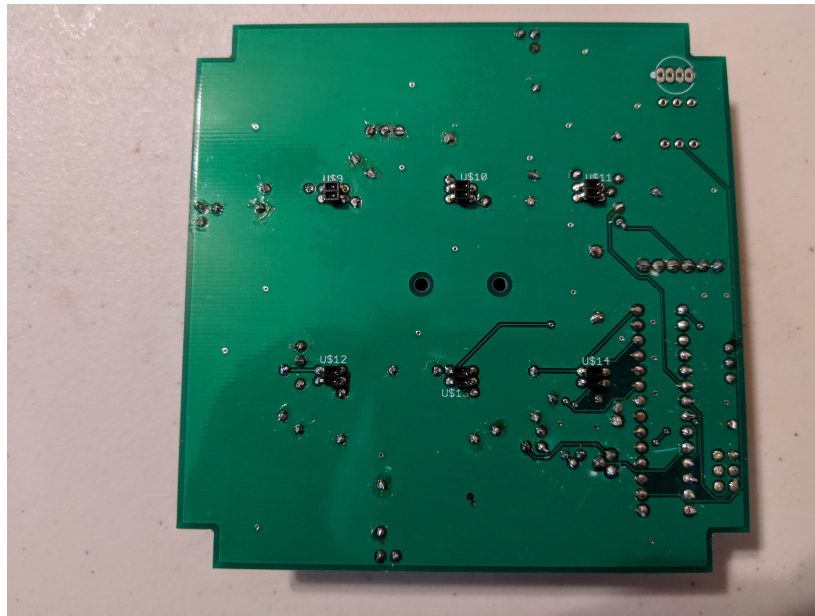


Figure 11: Top side of soldered PCB. This shows the 6 reflective sensors which read the pad encoding.

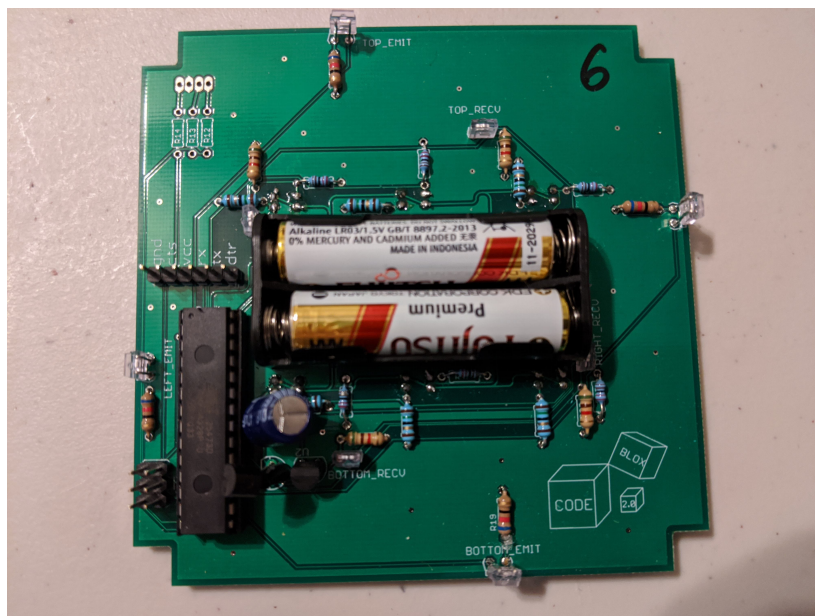


Figure 12: Bottom side of soldered PCB. You can see the IR circuits at the edges of the tile for communication and the battery and micro-controller towards the center.

Appendix E

Name	Link	Number	Cost per unit	Total cost
IR Emitter	https://www.digik	120	0.1552	18.624
IR Phototransisto	https://www.digik	120	0.1296	15.552
Reflective Senso	https://www.digik	180	0.6079	109.422
Atmega328P	https://www.digik	20	1.84	36.8
Electrolytic Capa	https://www.digik	30	0.128	3.84
Indicator LED	https://www.digik	30	0.821	24.63
Voltage Regulator	https://www.digik	30	0.31	9.3
PNP BJT	https://www.digik	30	0.1664	4.992
batteries	https://www.digik	60	0.2646	15.876
battery holder	https://www.digik	30	1.049	31.47
1x6 header	https://www.digik	30	0.232	6.96
2x3 header	https://www.digik	30	0.108	3.24
2.2k res	https://www.digik	180	0.0275	4.95
15 res	https://www.digik	180	0.0275	4.95
62 res	https://www.digik	120	0.0162	1.944
5.1k res	https://www.digik	120	0.0162	1.944
USB C breakout	https://www.sparl	1	8.95	8.95
PCB's	https://www.pcbw	40	2.6	104
Neodymium Cub	https://totaleleme	2	19.99	39.98
Velcro	https://www.ama	1	16	16
1/8" plywood		2	15	30
				493.424

Figure 13: Budget: Total cost = \$493. The budget created accounted for the creation and assembly of 30 tiles. The electrical components on board the tiles are scaled accordingly.

Appendix F

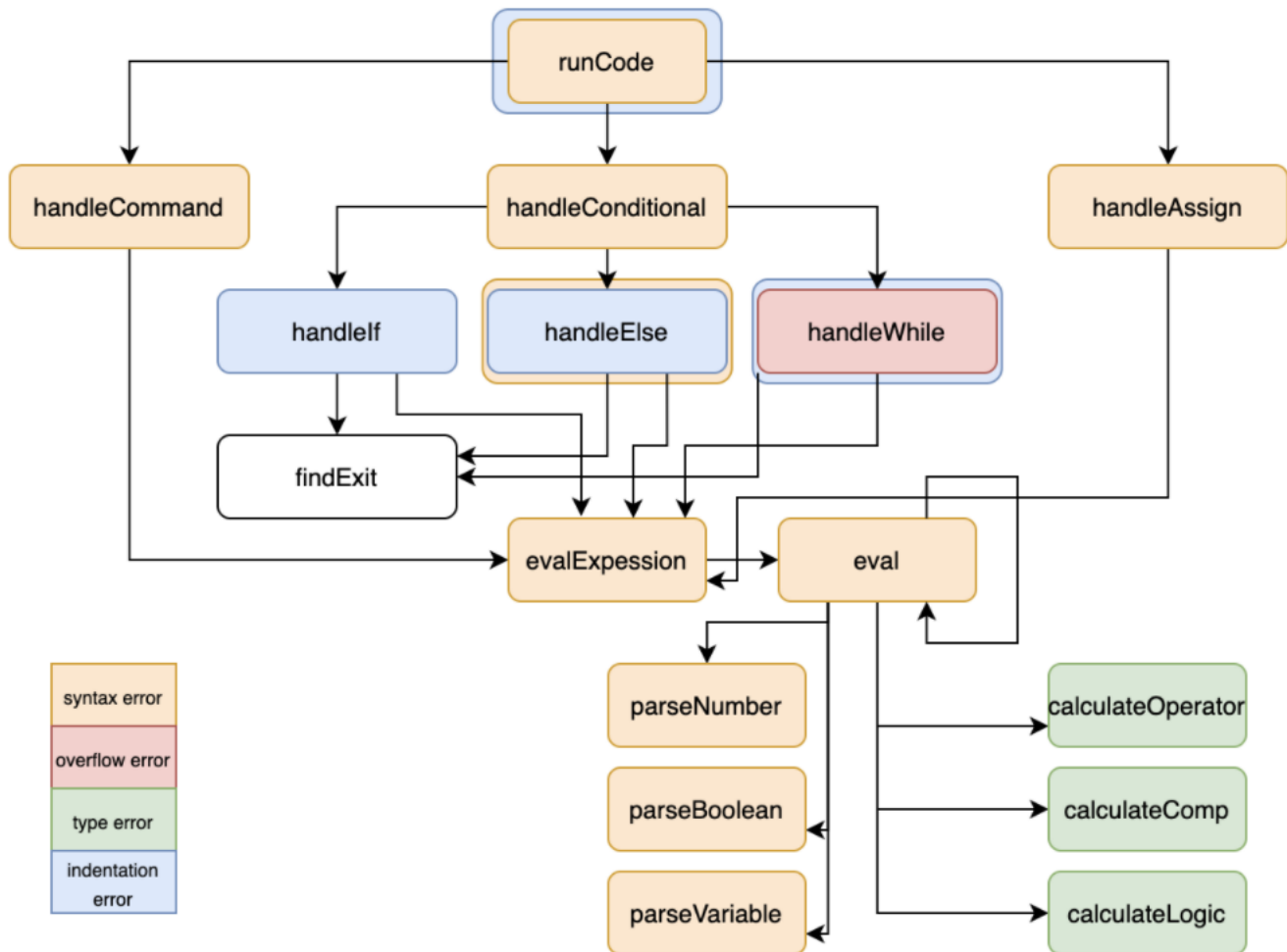
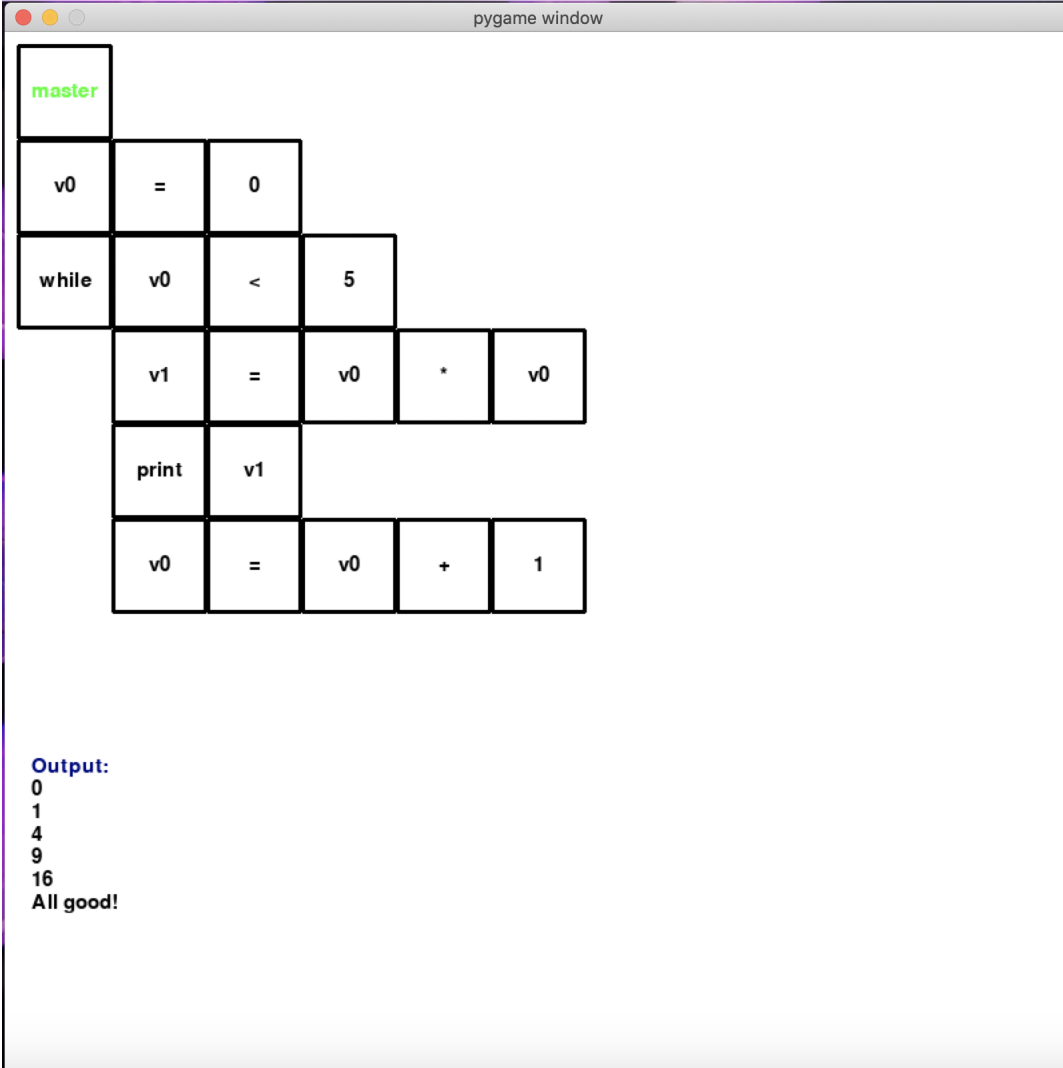


Figure 14: The specification of the interpreter. There are three main functions that each represent the three things that are valid statements in CodeBlox: commands, conditionals and assign statements. All functions evaluate expressions using the `evalExpression` function, which calls the `eval` function, which recursively computes a value using helper functions. All functions then return the next row the interpreter then has to process. For commands and assign statements, this is simply the next row. However, since conditional statements are allowed to jump and loop, an exit position (representing the position the indentation realigns with the caller conditional) is returned instead. The different places different types of errors can be encountered is also seen in the diagram.

Appendix G



The image shows a window titled "pygame window" containing a Python program and its output. The program is displayed in a grid-like format with tokens separated by spaces. The output is listed below the program.

```
master
v0 = 0
while v0 < 5:
    v1 = v0 * v0
    print v1
    v0 = v0 + 1
```

Output:
0
1
4
9
16
All good!

Figure 15: A program that prints out the squares of numbers from 0 to 4, inclusive. There are no errors in this program, so once the interpreter finishes running, the GUI plots this program along with the output. The "All good!" at the end lets the user know that there was no error in their program.

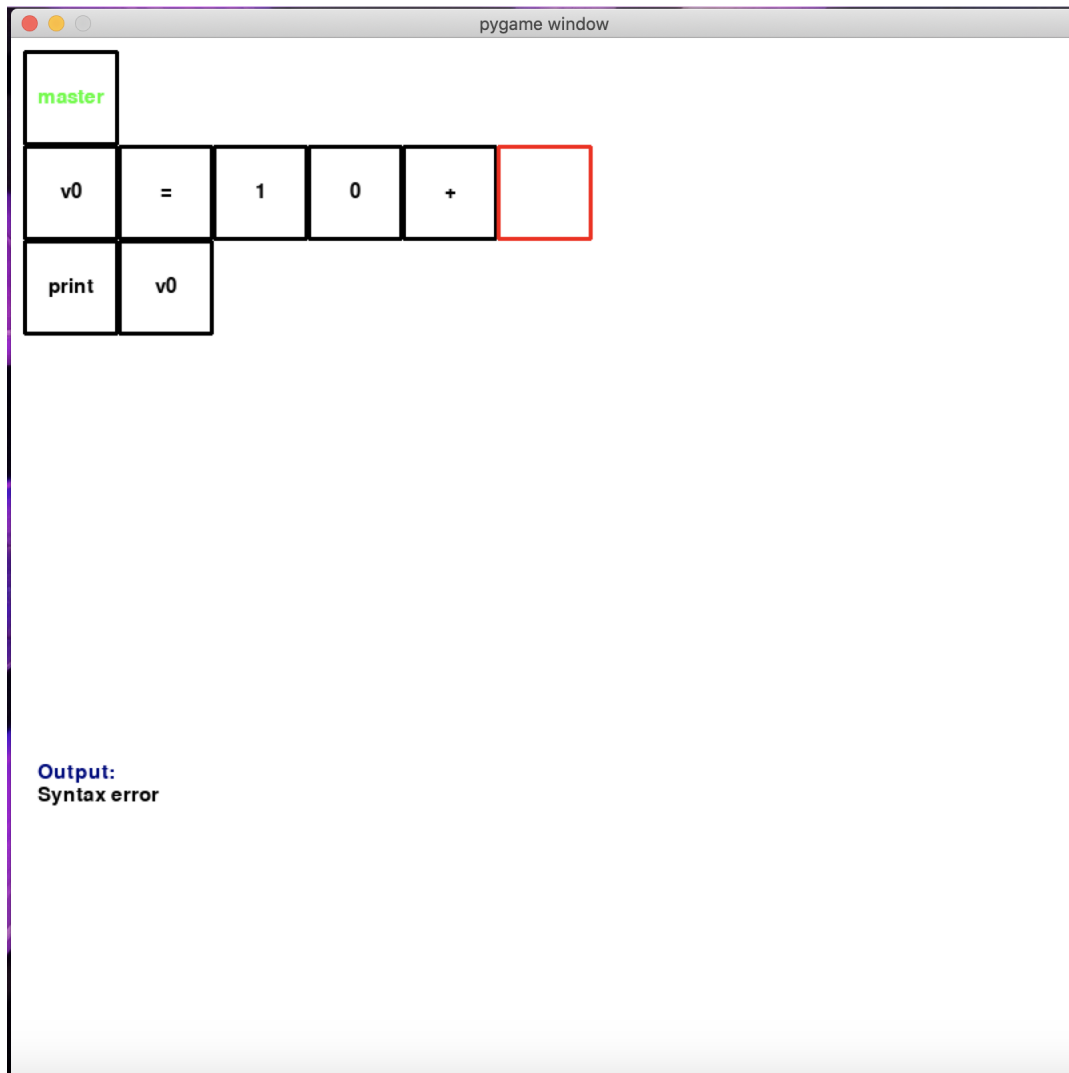


Figure 16: A program that attempts to do addition, but there is no second argument to add. The interpreter recognizes this incomplete syntax and throws an error where it was expecting another argument. The GUI then draws a red box around where the interpreter expected an argument, indicating to the user that the error occurred at that location. The output also says "Syntax error". Notice that the "All good!" message is missing, due to the error.