ScratchNLP: Constructing Scratch programs from natural language commands

Tina Quach[1], Kara Luo[1], and Willow Jarvis[1]

June 5, 2018

[1] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 02139
quacht, luok, wjarvis @mit.edu
Code available online: https://github.com/willowjar/scratchNLP.git

1 Introduction

With the current emphasis on computational thinking starting from a young age, there has been significant efforts to lower the barrier of entry to programming so that it is accessible to all. Particularly, visual, block-based programming languages like Scratch support creative learning and computational thinking through the construction of interactive programs [1]. While block-based programming languages are widely popular and have been incorporated into many educational programs, they may be tedious to operate and are inaccessible to people with visual impairments. A natural language interface for Scratch will make the introduction to computational thinking more intuitive and accessible, especially if combined with a speech interface. The ability to program using natural language interfaces could also translate to interactions with voice assistants and smart devices. In this paper, we present ScratchNLP, a system that translates natural language to its equivalent Scratch program.

In ScratchNLP, the user can enter sentences that describe the behavior of the program such as:

```
when the program starts make a variable called x
delete variable x
make a variable called x and a variable called y
set x to random number between 10 and 20
set y to 4 times six
change y by negative 3
```

We have written a set of syntactic and corresponding semantic rules to translate these natural language commands into Scratch's internal JSON representation of a program. Users can then give a command that will indicate the end of the program and generate an SB2 file that may be uploaded into a Scratch editor as a new project.

1.1 Introduction to Scratch

Scratch is a visual, block-based programming language and on-line community that supports novices' (especially children's) engagement with creative learning and computational thinking through projects they make and share [1]. Scratch is designed to have a "low floor, high ceiling", which means that its barrier to entry is low, but it can support a variety of interactive media projects of different complexity levels. Scratch projects can execute programs that play sounds, manipulate images, respond to events, and interface with LEGO programmable bricks. By allowing novice users to easily generate programs that have visual and audio components, Scratch engages novice users and helps them design projects that connect with their interests. In contrast to more conventional programming languages which require significant programming experience before such complex programs could be created, Scratch provides abstractions that allow users to focus more on the logical process behind programs, rather than being burdened with the framework and syntax of a language.

Additionally, Scratch guides the users in the process of creating programs through the visual language of the blocks. The blocks' shape and color provide constraints and visual cues for where a block could and should go. The curated vocabulary of blocks is designed to be simple yet expressive. The introductory and contained nature of Scratch lends itself to being used in conjunction with natural language instructions as a powerful interface for creative computing.

1.2 Why natural language?

Here are three reasons why one may wish to utilize a system that translates natural language to its equivalent Scratch program.

- **A conversational interface is more accessible.** A main feature of Scratch as an introductory programming language is its visual and block-based nature. Unfortunately, visually-impaired children are not able utilize this drag-and-drop version of Scratch. A conversational interface for programming could lower the cognitive load of interacting with blocks on a screen and support blind children's engagement with computational thinking and creative learning.
- **Natural language is familiar and intuitive.** Natural language does not require extensive training, as it is ubiquitous and familiar [2]. However, there is a trade-off between familiarity and clarity. Natural language itself is complex and often contains ambiguity. Thus, a pure natural language programming system would be difficult to implement and could be confusing and unproductive to the users as they struggle to specify the intended behavior of the program[3]. However, if natural language is constrained by a simple programming language, a balance between flexibility and clarity could be achieved.
- **Natural language can be concise.** While Scratch is able to reduce the burden of learning the syntax of a programming language by providing descriptive blocks, there are limitations to how flexible the blocks can be without being redundant. For example, consider the statement "x >= y". While it is a standard operation, its equivalent implementation in Scratch is an or-statement with (x > y) and (x = y) as its arguments. A natural language interface would be able to handle certain redundancies and variations in user-defined instructions without making the programming more difficult for the user or requiring a more robust programming language.

2 Previous Work

Research on programming with natural language provides a foundation for our development of a natural language interface for creating Scratch programs.

Programs exist in our daily lives in the format of the commands, instructions, and recipes we tell one another. People often engage with the vocabulary necessary for expressing computational concepts in programs. One of the fundamental concepts used in computer programming is a control statement—a statement that describes the order in which actions are to be carried out, and provides instruction about when or how often other instructions are to be followed. While there has been debate as to whether or not non-programmers will naturally utilize control statements, findings from a study done in 1985 demonstrate that non-programmers can and do use control statements when giving instructions [4]. These findings establish the fundamental basis for using natural languages to construct computer programs, as untrained users would spontaneously issue instructions similar to computational commands.

Capindale and Crawford found that natural language is an effective means of database queries, regardless of the user's aptitude with computers. More importantly, the study found that feedback from the system helps users understand the language limitations and learn how to avoid or recover from errors [5]. ScratchNLP aims to accomplish the more complex of providing a natural language interface for creating a program rather than answering questions or retrieving information from a database.

More recently, several empirical studies have been conducted to investigate the effectiveness of natural language when used in the context of programming. A study conducted by Biermann et al. at Duke University tested the effectiveness of a natural language programming system called NLC. The NLC system, like the system presented in this paper, utilizes semantic rules to interpret instructions provided. The study showed that after students were given sample sentences and practice problems, the students were able to quickly adapt to programming within the subset of English implemented by NLC [2]. While the results from the study described are promising, the loop and conditional definitions of the system were implemented only to a limited degree at the time of the experiment, and only a simple repeat command was mentioned to the test subjects.
Another study completed by Good et al. describes specific challenges of natural language program generation. The challenges described in the paper stem from the inherent differences between natural languages and computer languages. Natural languages are often redundant and afford multiple ways of expressing a single idea. The communication of concepts through natural languages also often rely on humans' ability to generate synonyms and infer information based on the context of the statement. There are several specific challenges that are pointed out in this study, a few of which are listed below.

1. The users would use synonyms in place of rule and phrase key words that the grammar uses to parse and interpret the instruction.
2. The users would use incorrect syntax of rules and phrases.
3. The users would add additional words when giving an instruction.
4. The users would omit rule and phrase key words when giving an instruction.
5. The users would use one keyword in place of another.
6. The users would wrongly assume syntax rules that do not exist [3].

In this next section, we will describe how our system addresses these challenges using a more flexible syntax and a dynamic vocabulary set.

3 Methodology & Implementation

Our system for constructing Scratch programs from natural language is designed to receive and respond to a series of textual inputs and generate a JSON specification of the project that gets compiled into the SB2 format, standard for Scratch programs.

3.1 Methodology

Before implementing the ScratchNLP, we made a series of design choices that simplified the system and provided the foundation for a creative programming experience.

3.1.1 Narrowing Scope

We began by narrowing the scope of our problem. In Scratch, there are nine categories of blocks: Motion, Looks, Sounds, Events, Control, Sensing, Operators, Variables, and Custom Blocks. Before implementing the system, we chose to support five of the nine categories:

1. “Events” - allow the user to build programs that are capable of reacting to events like key presses and mouse clicks.
2. “Control” (See Loops and Conditionals) - allow the user to include loops and if-else logic
3. “Operators” - allow the user to complete arithmetic and logical operations within their program.
4. “Sound” - allow the user to play and dynamically manipulate sound
5. “Data” - allow the user to create and use variables

"Events", "Control", "Operators", and "Data" are necessary for engaging with computational thinking concepts. We chose the "Sound" category such that the programs created would be perceivable by both sighted and unsighted users.

3.1.2 Designing a Grammar

Given these categories, we considered how users express and describe programs and commands in natural English. We manually generated example sentences that correspond to supported Scratch commands. This approach produced commands that closely aligned with the way commands appear on Scratch blocks as they are intended to be descriptive.

Beyond natural language commands, the way humans describe the interactions and relationships between these commands is also flexible. Thus, we must consider how entire programs are described in natural language. This exercise demonstrated that as descriptions become more natural, they also become more ambiguous, and obscure the key information to be used in the program. Unnatural descriptions tend to be more precise. Consider the Scratch program in Figure 1 for example.

Here we explore four ways to describe the program, ranging from natural to unnatural.

1. When the space key is pressed, play the hi na tabla sound and wait a quarter of a second, and then play sound hi tun tabla and wait another quarter of a second. Do **that** three times.

Figure 1: A Scratch program that plays drum sounds when the space key is pressed.

2. When the space key is pressed, play the hi na tabla sound, wait a quarter of a second, play sound hi tun tabla, and wait another quarter of a second. Do all instructions three times.

3. When the space key is pressed, repeat the following three times. Play the hi na tabla sound. Wait 0.25 seconds. Play the hi tun tabla sound. Wait 0.25 seconds. That's it.

4. When the space key is pressed, create a repeat 3 loop. In the loop, play the hi na tabla sound. Wait 0.25 seconds. Play the hi tun tabla sound. Wait 0.25 seconds. Outside of the loop, ...

In the first statement, `that` is used to refer to the 4-block command sitting inside the loop. However, from the sentence alone, it is possible to assume that `that` actually refers only to the last command, `wait 0.25 seconds`, or any subset of the commands. The second statement clearly states that all instructions are repeated, but it is unclear about what subset of commands actually respond to the space key press. The third statement disambiguates the the context in which the commands are run it (when the space key is pressed), and it uses `That's it` to make it clear what is contained in the scope of the repeat loop. The fourth is the most unnatural instruction, and it most closely mirrors the Scratch blocks. In this system, we chose to design a grammar that supports phrasings like the third example instruction to balance between natural language and clarity.

The constrained context of building a Scratch program enables us to identify instructions that would be ambiguous in other contexts and use them as concise expressions of lengthier commands. For example, the user could say `faster` to communicating the idea of increasing the tempo of the outputted sound. The user input `faster` gets translated to the Scratch internal representation `[changeTempoBy:, 10]`.

Lastly, we designed our grammar to support the different phrasings of the same concept. For example, *"the sum of x and y"*, *"x plus y"*, *"x added to y"* are evaluated to the same statement, consisting of the 'x' block and 'y' block nested in a '+' block. We also aimed to allow users to include or exclude determiners before nouns and to use synonyms.

In the future, this system will eventually receive input from a speech recognizer to create a conversational, voice-based interface for creating Scratch programs. With this in mind, we chose to assume that user inputs would not include punctuation and would be case insensitive.

3.2 Implementation

To understand user input, the system uses a grammar that is defined by syntactic and lexicon rules that each have a corresponding semantic rule[1]. As depicted in Figure 3, the system first uses regular expressions to extract unrecognized words that must be added to the grammar before the system

[1]The parser used with the grammar and semantic rules is 6.863 - Spring 2018 - Semantics Interpreter as provided in lab 3 software.

can parse the input. The regular expression looks for unrecognized words that either represent a number or is the name of a variable, list, or message. Then, the grammar is used to generate a parse tree for the specific user input (See Figure 2). The semantic rules are evaluated on each parse tree to generate a script. This occurs for each user input. A series of user inputs is used to construct and modify a JSON specification of a Scratch project. At any point, the user may query for the JSON specification or ask for a SB2 file to be compiled.

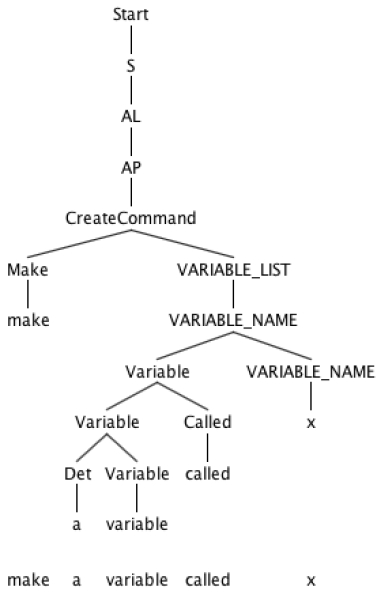

Figure 2: Parse tree generated for the command "make a variable called x"

While the aim of the system is to provide a flexible programming interface, there are several restrictions on how a user may interact with the system.

1. **Data structures are expected to be declared and created before they are used in other commands.** Like most conventional programming languages, the system expects a statement creating a variable or a list with an assigned name before this name could be used to reference this variable or list in the context of other instructions.

2. **The user instructions are expected to be *complete*.** A *complete instruction* is one that defines the context in which an action should be completed (if any) as well as the complete set of actions that are attached to this context. If there are multiple actions to be taken, all of them should be provided in this complete instruction.

   Three examples of *complete instructions* are provided below.

   ```
   "make a variable called x"
   "play the meow sound"
   "if the timer is greater than 2 then reset the timer thats it"
   ```

   Each of the sample instructions provided above can behave as a standalone command. In the case of the third conditional statement, both the condition to test on (timer is greater than 2)

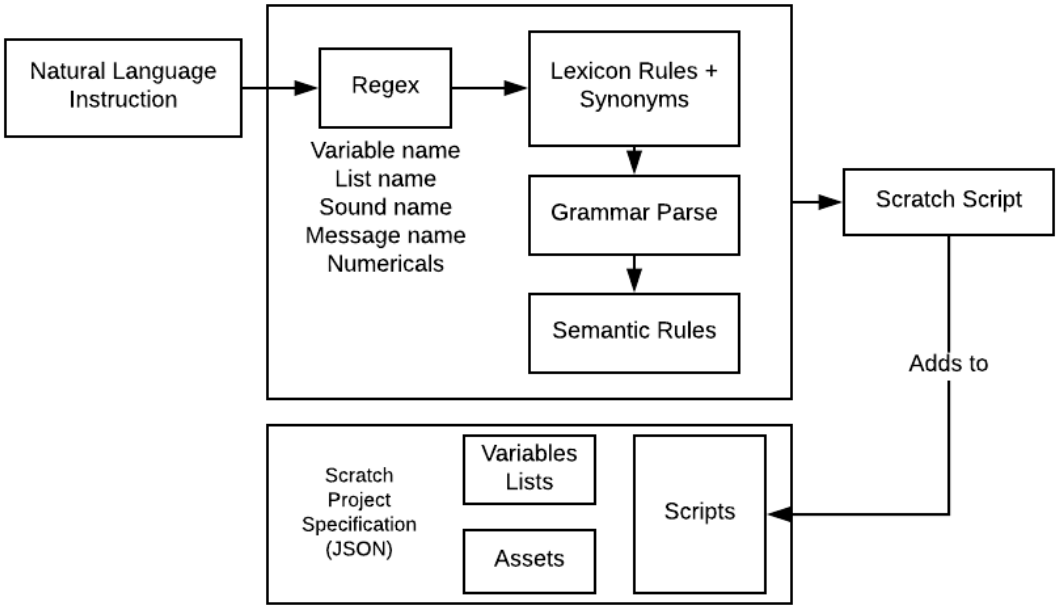

Figure 3: Overall System Architecture: Textual natural language user input is translated into a Scratch script through a Python system built on regular expressions, syntactic, and semantic rules.

and the corresponding action to be taken (reset the timer) are clearly defined. The use of the trigger word `"thats it"` is discussed in section 3.6 in further detail.

In contrast, three *incomplete instructions* are listed below.

```
"if x is greater than 3"
"x is not equal to y"
"repeat add 1 to x"
```

In the first incomplete instruction, the condition is declared for a conditional command, but no action is attached to it.
While the second failed statement may be considered a completely valid boolean phrase, there is no action attached to the statement. To make this boolean phrase a *complete instruction*, the user could use it as a condition in a conditional command and attach an action as the response for when the statement evaluates to true.
The third failed case is a fragment of a loop command where the duration of the loop is not clearly defined. The user may replace this command with *"repeat add 1 to x 5 times"* or *"repeat add 1 to x forever"*, either of which would be considered a *complete instruction*.

3. **The user-provided instructions are expected to be case-insensitive, without punctuation, and with correct spelling** [2] If our natural language system is to be compatible

[2]In certain cases, the correct spelling of a word would involve the use of apostrophes, which conflicts with the requirement that there be no punctuations in the user provided instruction. In these cases, the no-punctuation rule takes precedence. This is due to the fact that the `inner_parse` function of the parser we use is unable to handle single quotation and double quotations in the instruction.

with a speech interface, our system must be case-insensitive to handle variation in the speech recognizer. Furthermore, our system must not depend on punctuation that requires users to voice punctuation explicitly by saying 'comma' or 'period' out loud to maintain the natural interface.

The output of the system would be a JSON file, Scratch's internal representation of the program, that includes the variables and lists that are created by the user, as well as the script to be run. This JSON file is zipped with existing assets to generate a single `.sb2` file that can be loaded into Scratch as a working program.

3.3 Syntactic-Semantic rule pairs

Every syntactic rule has an associated semantic rule that specifies what parts of the natural language input encode arguments and how to handle them. The semantic rules are applied to each parse tree constructed by the grammar. For example, the instruction *"add x to the list l"* will trigger a call to an internal helper function with the arguments passed in the correct order, in this case, $appendToList(l, x)$

To support multiple variations of the same command, we define multiple syntactic rules to correspond to the same function via nearly identical semantic rules. In this way, our system can compute on only the meaningful arguments in the user input and drop filler words. For example, in the instruction "set the first element of the list called numbers to 1", the auxiliary phrase "the list called" does not provide any additional information, and the user may naturally choose to exclude it from the instruction.

3.4 Generalize to various user inputs

Given a set of syntactic and lexical rules, our parser can only parse a sentence if every token is accounted for in the rules. However, a programming system needs to support numbers and variable names – which cannot be specified in the grammar ahead of time as there is an intractable number of possible variable names and numbers that may be specified when building the Scratch program. Furthermore, users may unintentionally use synonyms in communicating instructions, so ScratchNLP is designed to handle synonymous phrases as well.

3.4.1 Handle names and number phrases

The user may give names to variables, messages, or lists so they can refer back to and appropriately modify them. The user may also choose to include numbers in their instructions to set a variable to a particular value, index into a list, or perform arithmetic operations, comparisons, or define bounds for random number generation.

This presents a challenge as the grammar needs to be able to handle and parse these numbers and constant names correctly, and it is impossible to predict which numerical values or constant names will appear in the instructions before they are received. To handle these unknown words, the system first uses regular expressions to detect these phrases. The system then creates lexicon rules to add these names to the grammar based on their name type. Lastly, all unknown tokens are assumed to be number phrases and added to the grammar via lexicon rules. At the end of this process, the system is ready to parse the input.

3.4.2 Recognize synonyms

When engaging with natural language systems, users often use synonyms in place of rule and phrase key words[3]. To make our system more robust against such linguistic variations, we use Python NLTK (Natural Language Toolkit) to find synonyms for keywords substitutable by synonyms [6].

The lexicon rules contain a list of terminal phrases that the system accepts. In general, these rules take the form of declaring a base word nonterminal in association with a list of synonyms (or a single word for reserved words like *variable* and *sprite*) that can replace the base word with little or no change to the meaning of the command. For example, the lexicon file contains a mapping of the nonterminal *Make* to the list *["make", "create"]*. Such a rule allows for both *"make a variable called x"* and *"create a variable called x"* to be accepted and be considered equivalent.

ScratchNLP includes a component that generates synonyms for a set of predefined nonterminals for the specified part of speech, and adds these synonym sets (synsets) as lexicon rules to the grammar file. This allows synonyms to be recognized and parsed correctly instead of being labeled as unknowns by the regular expression component.

The synonyms are identified by querying for all synsets (as defined by NLTK) that contain the specified words that are also matched in the word's part of speech. All words in those synsets are gathered without repetition. Then this synonym gathering component adds all synonyms of a word to the lexicon file as terminals mapped from the original non-terminal.
[3]

Not all keywords can be substituted by its synonym. For example, a list is behaviorally different from a set or a dictionary, though all of them are storage structures and could be synonyms for each other in terms of natural language. This could introduce confusion to the user and have unexpected consequences. Another example is the keyword "volume", which in our context, is how loud a sound is. However, it common use case is referring to how much liquid there is so we would expect that most, if not all, synonyms of volume would be useless to our system. Incorrect synonym are manually excluded from the lexicon.

Below are a few examples of synonyms commands. The first command is the regular form our written lexicon expects. Anything after the first command are examples of equivalent commands using synonyms. The last item is the output of evaluating these commands using the semantic rules. From the examples below, we can see that these words that were deemed appropriately substitutable by their synonyms range from nouns to verbs to adverbs. The ability to recognize synonyms make our system robust to users whose natural tendencies or memory of the commands do not match that specified by our original limited vocabulary.

```
play the meow sound
play the meow audio
play the meow auditory', 'sensation
{'variables': {}, 'lists': {}, 'scripts': [['doPlaySoundAndWait', '
    meow']]}

make variable called x subtract 5 from x
```

[3]The parser used by our system does not allow for terminal phrases with multiple words. Occasionally during the synonym searching using NLTK, we would find multi-word synonym to a keyword we are interested in. For example, "take off" is found to be a synonym of "subtract" and "clock time" is found to be a synonym of "time." To handle such multi-word synonyms, our synonym generator segments the multi-word synonym into multiple semantic rules joined together by a syntactic rule that maps the original nonterminal of the keyword to the synonymous phrase.

This requires no change or modification of previously written syntactic, semantic, or lexicon rules, and thus is modular in terms of functionality. The system currently supports synonym phrases with up to 2 words, as from the 210 synonyms found, only 1 (less than 0.5%) was a synonym with more than two words.

```
produce variable called x deduct 5 from x
make variable called x take_off 5 from x
produce variable called x take_off 5 from x
yield variable called x deduct 5 from x
{'variables': {'x': 0}, 'lists': {}, 'scripts': [['changeVar:by:', '
    x', ['*', 5, -1]]]}

change volume by 5
vary volume by 5
alter volume by 5
{'variables': {}, 'lists': {}, 'scripts': [['changeVolumeBy:', 5]]}

repeat play the meow sound forever
reiterate play the meow audio perpetually
echo play the meow audio everlastingly
recur play the meow sound always
{'scripts': [[111, 146, ['doForever', [['playSound:', 'meow']]]]]}

stop all sounds
discontinue all sounds
quit all sounds
block off all sounds
{'scripts': [[111, 146, ['stopAllSounds']]]}
```

3.5 Differentiate between commands and key words

When parsing and interpreting natural language instructions, there may be confusion as to whether a particular phrase is intended as a key word (for example, a variable name or a message name), or as part of the instruction. Consider the example "make a list called a". When parsing this command,the first "a" should be parsed as a determiner whereas the second "a" should be interpreted as a list that the user is creating. To handle cases like these, the user's command is first matched to regular expressions that select for the tokens as marked by the question mark in the following phrases "variable called ?", "the ? sound". Then, these tokens are added to the grammar under the appropriate tags (in the example given above, the tags will be "VARIABLE_NAME" and "NAME_OF_SOUND").

In particular, the system expects the term *"called"* or *"named"* before the names of variables and lists when they are being declared.

3.6 Handling nesting in complex logic

The main difference between natural language and drag-and-drop programming is that all the instructions will be given sequentially in time, regardless of the relation between one statement and another. However, the user may want to generate more complex code structures, like nested loops, using natural language.
Consider the instruction

"repeat the following 5 times [Action A] repeat the following 6 times [Action B]"

There are two possible interpretations for this command.
The first of which includes two separate loop structures, the first one loops over [Action A] 5 times,

and the second one loops over [Action B] 6 times.

```
Repeat 5 times:
        Action A
Repeat 6 times:
        Action B
```

The second interpretation has one nested loop structure. Wherein each passing of the loop involves performing [Action A] once, and repeating [Action B] 6 times. Overall, [Action A] is performed five times, whereas [Action B] is performed a total of 30 times.

```
Repeat 5 times:
        Action A
        Repeat 6 times:
                Action B
```

In such a case, the ambiguous instruction will produce two different interpretations with different behavior. Therefore, it is necessary to incorporate trigger words at the end of a loop instruction to indicate the end of list of instructions to loop over. However, in normal utterances, it is less natural to say "end the if loop", which would require the user to have a clear understanding of the structure of the program already. To make our system more natural to novice coders, we chose to the phrase **thats it** to signal the end of a loop or a conditional command.

This system chooses the phrase `thats it` (with the apostrophe removed, as the system expects input without punctuation) as the trigger word. To illustrate how this works, let us review the example given above. To generate code that follows the form of the first piece of code, the instruction will follow the form *"repeat the following 5 times [Action A]* ***thats it*** *repeat the following 6 times [Action B]* ***thats it****"*. In contrast, to produce the second piece of code the instruction will look like *"repeat the following 5 times [Action A] repeat the following 6 times [Action B]* ***thats it thats it****"* where the two `thats it` phrases close off the inner and outer loop, respectively.

As demonstrated above, we are able to distinguish between two loops behaving in a linear manner or a nested manner. For similar reasonings, the trigger phrase *thats it* is implemented in conditional commands (if commands, if-else commands, etc.) and event commands (when [Event A] do [Action A]) as well to allow for unambiguous program generation.

3.7 Create non-linear programs

In a Scratch project, a statement *block* corresponds to a single instruction. Multiple blocks can be snapped together vertically to create a script referred to as a *stack*. Within each stack, code is executed linearly top to bottom, but the stacks themselves are distributed in space and can be nonlinear. Furthermore, Scratch programs are event driven and can run code asynchronously. Every event handler must be specified as its own stack. For example, the user may use the following expressions to specify a program that allows you to play with the volume of a sound as it plays the sound (Figure 4).

```
when the space key is pressed play the cave sound thats it
when the up arrow key is pressed louder thats it
when the down arrow key is pressed quieter thats it
when the program starts play the meow sound 10 times thats it
```

In our system, the user does not explicitly treat every event handler as a stack. Our system is designed to manage stack creation for them. Every time an event handler is specified, a stack is

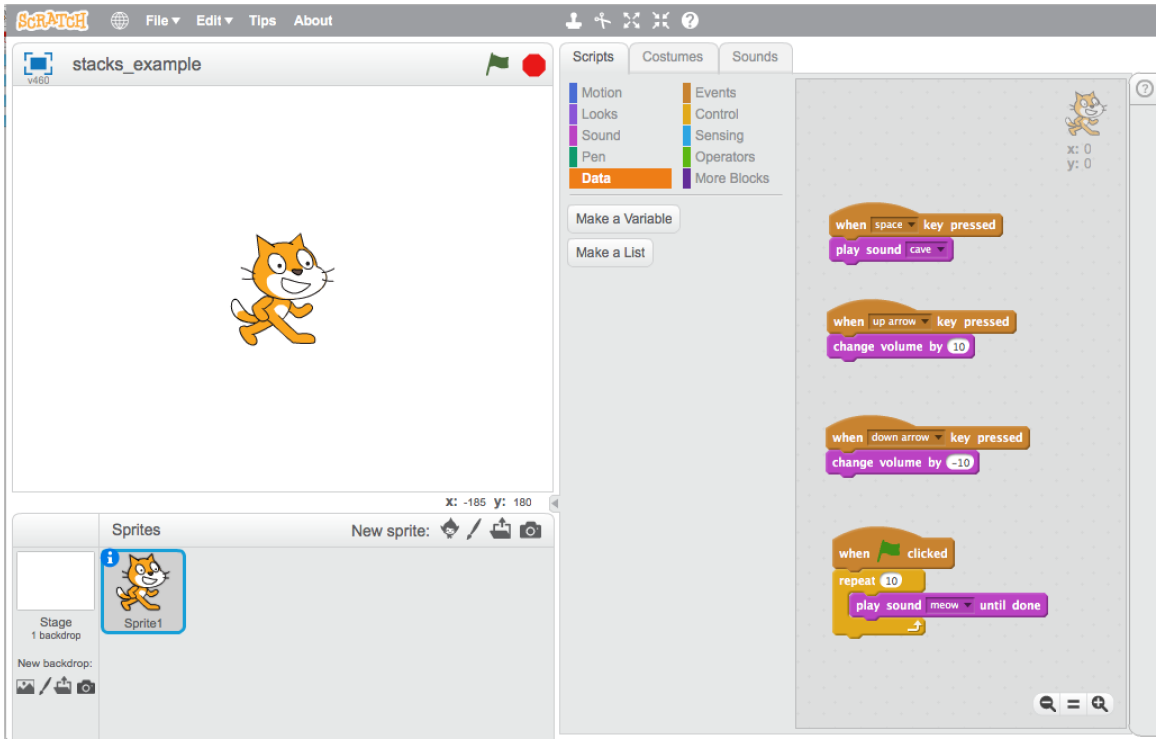

Figure 4: A Scratch project consists of sprites that each have associated assets and code in the form of blocks. Each sprite is an object in Scratch which performs functions controlled by scripts. Here is an example project with a single sprite that has code using event handlers that exist as separate stacks.

created for the handler and a stack is made for all previous instructions that do not yet belong to a stack.

3.8 Flexibility in instructions

There are many different ways to phrase the same concept. Consider the following cluster of instructions that indicate a summing instruction (all of which are currently supported by the system).

```
a plus b
a added to b
sum of a and b
```

Previous studies have shown that asymmetric instruction support may confuse users [3]. If there is a certain amount of flexibility supported for one specific type of command (in this case, addition), then in similar commands (in this case, subtraction commands), users will naturally assume the same amount of flexibility. For this reason, we provide support for the following subtraction instructions, as well as analogous instructions for multiplication and division.

```
a minus b
a subtracted from b
difference between a and b
```

Guided by this concept of symmetry within flexibility, the system is able to guide the users to provide more predictable commands.

4 Testing

Every input into the system, if parseable, generates a parse tree. The first level of testing involved making sure that *the syntactic rules generate the correct parse tree(s).* The second level of testing involved verifying that the *decoration of this parse tree via the semantic rules generates the correct output.* The output follows the form of an object specifying the variables, lists, and scripts in the project being built. The third level of testing aimed to see if *the grammar is flexible and intuitive.* For this, we employed incremental testing using the command line interface. We entered commands line by line and verified the contents of the output as those that match the Scratch programs generated by the Scratch block-based programming interface. The last level of testing involved properly generating an SB2 file that can be uploaded to Scratch's online project editor.

4.1 Creating test sentences

In order to test our system, we developed multiple sets of inputs that each correspond to the creation of a specific Scratch program. The programs developed are based on what programs we expect users to want to create, as well as by the constrained set of Scratch block types we designed the system to support. These sentences served as our initial test set. The test sentences were designed to use every kind of command supported in the grammar at varying levels of complexity. For example:

- `make a variable called x` is a simple command that add `x` to the Scratch project JSON. `set x to 1` adds a single instruction `["setVar:to:", "x", 1]` to the script.
- `set x to a random number between 10 and 20` adds additional complexity, nesting a reporter block inside of a command like so: `["setVar:to:", "x", ["randomFrom:to:", 10, 20]]`.
- We also included sentences that evaluated nested commands. For example, `play the meow sound 10 times` employs nesting `["doRepeat", "10", [["doPlaySoundAndWait", "meow"], ["changeTempoBy:", "10"]]]`.

For the full set of test sentences see the Appendix A.

4.2 Generating proper parse trees

1. **Does our system parse all the sentences we expect them to?** Using a test set allows us to easily verify that our system can parse the sentences it should parse.
2. **Does our system successfully parse commands that don't make sense as Scratch commands?** When optimizing for the parser to work on the test sentences it is easy to create rules that broaden the kinds of sentences that could be generated. However, the grammar may become too permissive, possibly allowing and generating programs that do not correspond to real Scratch programs. We defined a set of test sentences that should not be permitted by our syntactic and semantic rules. We discovered that our system permits the following invalid inputs.

   ```
   set timer to timer is greater than 0
   ```

Our system should reject setting the timer to a boolean phrase since the timer is a numeric quantity that corresponds to the time elapsed since the project began playing. Instead, our system mistakenly interprets the timer to be a variable, which can be set to a boolean phrase.

```
when the green flag is clicked when the program starts play the
   drum sound thats it thats it

if x is greater than or equal to 2 then when the the down key is
    pressed reset the timer thats it thats it
```

Our system mistakenly accepts the two inputs above, which nest event handlers. This will not generate a valid Scratch project because events must be top level commands and should not be allowed to be enclosed in a conditional.This occurs because our grammar uses recursive rules in order to allow commands, known as Action Phrases, to be part of the definition of event handlers, which are also a kind of Action Phrase.

```
make a variable called x
when i receive x reset the timer and play the drum sound thats
   it
```

This program is accepted by our system. However, it needs to be disambiguated because, in the phrase 'when i receive x', 'x' could refer to the contents of the variable 'x' or just be a message string containing the letter 'x'. Our system fails to detect this ambiguity, and chooses to treat 'x' as a string.

3. **How much ambiguity is introduced when using the natural language grammar defined for our system?**
   This is tested by examining the number of valid parse trees that are generated from a single utterance.
   When the system encounters multiple parses for a given sentence, the system picks the first one. However, there is no predefined process that orders the potential pase trees. Therefore, ambiguity in parsing resulted in inconsistencies in terms of which trees to decorate. For example, the sentence *"set volume to 1"* means that we want to set the volume of the sound outputted in the Scratch project to 1. However, the statement is ambiguous and the first parse generated interprets volume to be the name of a variable in a 'set variable' command `[['setVar:to:','volume', 1]]` rather than a special attribute of the Scratch project's sound player. Yet in certain cases, the parse selected would work as intended. For example, *"set volume to 5"* generates `[['setVolumeTo:', 5]]`.

   A potential solution in the future lies in presenting options to the user to resolve the ambiguity.

4.3 Flexible and intuitive grammar based on natural language

4.3.1 Is the grammar we've defined flexible enough?

While the system was able to successfully map English expressions to program instructions, several usability issues that Good et al. had mentioned in their study [3] are apparent in this system as well.

A few common mistakes that result in unparseable instructions are listed below.

1. Dropping the `thats it` key word or having an incorrect number of `thats it` key words after nested commands.

2. Dropping the `called` or `named` key word when declaring a new variable.

3. Forgetting to initialize a variable before it is referred to elsewhere.

The cases mentioned above are trade-offs in the system design that provide clarity in the logic, and thus are difficult to avoid. Listed below are specific examples of failed instructions.

1. if 1 plus 1 equals two make a variable called x
2. if 1 is greater than 2 make a variable called x
3. if 1 is greater than 2 then make a variable called x

Marking the end of an if statement is not something that happens in natural language. Thus, it is likely that users will forget to close their conditional and control statements. This issue resulted in the following user test sentences failing. These were sentences that were created without referencing the grammar.

A user may naturally want to create variables using the following sentences. However, these fail because the regular expression expects the phrase "variable called $VARIABLE\_NAME$", not "variable $VARIABLE\_NAME$". Not supporting the sentences below simplifies our system, but reduces the flexibility of the grammar.

1. make the variable x
2. make a variable x

It is also worth noting that many of the variations currently supported by the system were added after failed instructions were identified in the development process. Since the test sample is not complete, further empirical studies are needed to help create a more robust grammar.

4.3.2 Is the system still usable when the user is not referencing the grammar to determine what forms are supported by the grammar?

When using the system, we found ourselves checking the grammar whenever sentences we expected to parse did not. It quickly became clear that our initial iteration of the grammar was too restrictive. A typical user will have a reference for what commands are supported by the system, however we want to avoid the need to reference the grammar. By coming up with programs without referencing the grammar, we generate example programs that encode intuitive expressions of computational thinking concepts. In this way, we subjectively evaluated whether the grammar was flexible enough and what intuitive forms should be supported.

In the future, we'd like to conduct user testing with people who are not familiar with Scratch or are new to programming. These tests would allow us to better evaluate the grammar's flexibility and its affect on the usability of the system.

5 Evaluation

The system, as described in this paper, is able to take in a set of natural language instructions and generate a Scratch project from it. The system successfully supports Scratch commands from five different categories.

Processing phrases that use contextual information involves complex operations. This system also struggles with *complete instructions* that are composed of a list of smaller sequential instructions. The use of sequential key words like *"first", "second", "then", "finally"* is common in the process of giving a multi-step instructions. However, in human languages, there is no strict rule that requires

each statement to be a single action. Each of these instructions can then be composed or more sub-statements. Consider the following script.

```
first make a variable called x
second set x to 7
finally divide x by 1
```

Now consider this equivalent (single) command below.

```
first make a variable called x second set x to 7 then divide x by 1
```

Since the system should be able to handle both types of commands - sequential word followed by a single action and a sequential word followed by a list of actions, further ambiguity is introduced in the parsing process. The system currently would generate multiple parses when given a sentence the input *"first make a variable called x second set x to 7 then divide x by 1"*. Specifically, it would produce the parse: `[first make a variable called x] [second set x to 7] [then divide x by 1]` and `[first make a variable called x] [second set x to 7 then divide x by 1]` to list just two. While these two parses may evaluate to the same result logically, the Scratch program requires correctly placed brackets around lists of instructions. Thus, having three separate instructions will produce different bracketing results than having a single instruction followed by a list of two instructions (with extraneous brackets surrounding the list). In this case, Scratch would not be able to generate code blocks correctly for the parse with extraneous brackets.

One key modification made to our system is the addition of a synonym generator, From the 25 keywords that were identified as appropriately substitutable by its synonyms, our synonym generator found 180 single-word synonyms, 29 double-word synonyms, 1 four-word synonyms. Our synonym generator doesn't handle adding synonyms that more than 2 words long, so out of 210 synonyms generated, all of them were added to our system except one. This averages to about 8.4 synonyms found per word. In a given command, if there are x words that were identified as being appropriately substitutable by its synonyms, it would mean that there would be approximately 8.4^x such additional sentences per command that our grammar can accept when users use synonyms. Our system has about 11 types of commands and each type of commands has about 5 rules. Thus by adding synonym support we would be able to parse and handle approximatly $55 * (8.4)^x$ more command sentences.

6 Conclusions

By designing and implementing ScratchNLP, a system that translates natural language commands into Scratch programs, we are able to partially address several issues previous studies have brought to light regarding natural language programming. Specifically, ScratchNLP supports synonym matching, determiner dropping, and variations in expressing the same command. However, the development process of ScratchNLP also revealed several challenges of disambiguating human languages whose correct parses depend largely on context. We have also learned about the variation in ways people express computational concepts and the difficulty of capturing all such variations even in a relatively constrained context like Scratch programming.

While natural language instructions may be easier for a novice to understand, the mathematical and logical clarity decreases. It was necessary for us to make design decisions and develop several strategies for creating a grammar that balances flexibility and specificity. As a basis, a grammar of syntactic and corresponding semantic rules used by ScratchNLP is able to successfully parse logically complex commands (given several simple rules that the user should follow) to generate Scratch programs using over 50 different Scratch blocks.

While test results described in this paper have been relatively promising, the system has not yet been evaluated by a novice programmer. Thus, our conclusions about the clarity and flexibility of the grammar must take into account a novice's perspective. Additional user studies must be conducted with our target users to observe how they describe Scratch programs and computational thinking concepts having not seen the vocabulary. Then, after providing them with samples of the vocabulary and grammar, the users should describe programs again. This second set of responses is crucial to the understanding of what expressions of concepts are intuitive and should be supported by our grammar.

Investing in learning about natural language interface for creating programs becomes even more relevant in an era of creative computing and smart wearables, phones, and appliances. Voice assistants like Amazon Alexa, Apple's Siri, Google Assistant, Microsoft's Cortana, have popularized screenless, voice based interface for controlling our environments and learning information. Currently customizations to these voice assistants and their behavior on devices must be done via code or web interfaces. We envision a future in which you can program your environment just by conversing with your environment. We started with the small context of a Scratch program to show that a natural language interface is an effective method for lower the barrier to computing. By widening the context, we may allow anyone who can type or talk to create programs for any smart device.

7 Future Work

While the system is able to interpret a wide set of natural language instructions, there are several different approaches to improve the robustness of the system.

7.1 Handle ambiguity in user instructions.

There are several different types of ambiguity that the system may encounter.

Ambiguous instructions and multiple parses. There is an inherent trade-off between flexibility and clarity in natural language programming. To allow for more flexible user instructions, the system may produce multiple parses and interpretations for one single command. For example *"set volume to 5"* may be interpreted as a sound command that sets the volume of the audio to 5. It may also be interpreted as the initialization of a variable called `volume` to a value of 5.

Ambiguity in logic statements. Another type of ambiguity involves logical statements. Consider the following boolean phrase *"A and B or C"* . In natural languages, it is unclear whether this statement should be interpreted as *"(A and B) or C"* or *"A and (B or C)"* .

Order of operation in arithmetic expressions. While the order of operation is clearly defined in mathematics, describing such operations in natural language is more challenging and nuanced. For example, when constructing such a statement it may be natural to say "add 3 to 5 times 6". In the sequential order in which this statement is given, this will be evaluated to `(3 + 5) * 6`. Yet if the correct order of operation dictates that this be evaluated to `3 + (5 * 6)`.

In these types of ambiguity errors, it is important to present different possible interpretations to the user, and request the user to disambiguate.

7.2 Increase support for synonym usage.

Dynamically recognize synonyms. The system currently generates synonyms for a predefined list of lexical terms utilizing synsets from the NLTK WordNet interface. However, there are limitations to such a method of synonym generation. In addition to generating synonyms that are unfit in the context of our grammar, it also relies on predefined lexical terms, and therefore it will not contain an exhaustive list of words that users may choose to use (as users often use unpredictable words and sentence structures). In future implementations, the synonym words may be generated in a more dynamic manner - when a foreign word is recognized, the semantic difference between such a word with known and defined words would be calculated to approximate the meaning of such a word. If there is no existing word with similar semantic meaning, then this foreign word could be dropped. To determine similarity in semantic meaning, one approach might be to search through potential synsets for an out-of-vocabulary word from user's input and compare that to every synset of our keywords that this out-of-vocabulary word could be a synonym of. We can use NLTK's API to get a score that indicates path similarities between two synsets in terms of word senses. Though this approach would be dynamic and more flexible, however, the time it takes to search through all potential synsets and comparing using similarity may take longer.

Conduct empirical studies to understand the users' linguistics choices. The premise of natural language programming presents additional difficulties as there may be specific terminology used in the computer science context. For example, "set", "dictionary", "list", "array", etc. all have specific meaning in a programming context, and it is unclear how a novice will utilize them or refer to them differently than expected. Therefore, additional empirical tests should be conducted with users from varying computing backgrounds to identify words and phrases that are often used interchangeably.

Support the use of pronouns to reference antecedents. In conversation and natural language, people use pronouns such as 'it', 'that', 'this' to concisely refer to what has already been expressed without needing to repeat it. ScratchNLP could be extended to recognize 'it', 'that', and 'this' as special key words that indicate a reference to a different node in the parse tree. This will likely introduce more ambiguity that would have to be resolved with a system in which the user disambiguates.

7.3 Provide users with meaningful feedback for error resolution.

As previous studies have shown, useful system feedback is able to help clarify system limitations for the users and expedite the learning process for a user [5]. Therefore, in addition to the disambiguation process mention in section 7.1, this system should be able to provide informative error messages.

Provide predictive suggestions for phrases and terms. Since the grammar defined by the system is finite, the system may use the lookahead set from a shift-reduce parser to guide the user with possible ways to complete their instruction given a partial instruction that the system has already received from the user [7].

Pre-screening for potential errors. Currently, the system acts as an interface between natural languages and the Scratch programming language. Therefore, the user is unable to check the correctness of their program until it has been completed and loaded into Scratch. To improve this, the system could impose some basic restrictions on the arguments involved in the users instructions, and provide some preliminary feedback as the user is defining the program. For example, the case of *"divide x by 0"* should not be allowed, as a general rule. Therefore, in the corresponding semantic

rule, the second argument (denominator) should be constrained to values that are nonzero.

7.4 Provide ways to simplify the programming process further.

Programming with natural languages may be intuitive, but detailing the program in a sequential manner (especially if the user chooses to use speech input) may be a long process. It seems natural that, like more conventional programming languages, this system should allow for the definition of functions or macros that the user may refer to to simplify the process further.

7.5 Other ways to expand the system.

- In its current form, the system supports only a subset of functionalities that Scratch provides. Further work should be completed to support more types of commands (such as mouse tracking and motion commands) to take advantage of the Scratch platform.
- Expand the system to provide speech-to-text as a method of providing user input, rather than the current REPL that is being used.
- Allow users to debug the program through natural language. Users should be able to query the state of the project, easily preview and execute the project during the creation process.
- Support a non-linear project creation flow. Currently, every instruction a user inputs is used to add to a program, forcing a linear process for creating a program. Users must also be able to navigate and modify the program.

References

[1] About Scratch
`https://scratch.mit.edu/about`

[2] Alan W. Biermann, Bruce W. Ballard and Anne H. Sigmon. *An experimental study of natural language programming*. Int. J. Man-Machine Studies (1983) 18, 71-87

[3] Judith Good and Kate Howland. *Programming language, natural language? Supporting the diverse computational activities of novice programmers*. Journal of Visual Languages & Computing, Volume 39, April 2017, 78-92

[4] Kathleen M. Galotti and William F. Ganong III . *What non-programmers know about programming: Natural language procedure specification*. Int. J. Man-Machine Studies (1985) 22, 1-10

[5] Ruth A. Capindale and Robert G. Crawford. *Using a natural language interface with casual users*. Int. J. Man-Machine Studies (1990) 32, 341-362

[6] NLTK WordNet Interface `http://www.nltk.org/howto/wordnet.html`

[7] Frank Pfenning, Rob Simmons, and Andre Platzer. Lecture Notes on Shift Reduce Parsing. Carnegie Mellon University

A Test Sentences

A.1 Test Programs

```
when the program starts make a variable called x thats it
delete the variable x
make a variable named z and create a variable called y
set z to random number between 10 and 20
set y to 4 times six
change y by negative 3

make a variable called x
when the green flag is clicked make a list called list and repeat
   play the bang sound 5 times and if x is equal to 1 then broadcast
    alas thats it thats it
when the down arrow is pressed broadcast downwego thats it
finally broadcast Iamdone

when the space key is pressed broadcast hello 3 times then create a
   new variable called count thats it
do the following 5 times add 1 to count thats it
if count is greater than zero then play the bing sound and change
   volume by 3 thats it

create a variable named y
repeat the following 5 times increment y by 20 then decrement y by 1
    thats it
broadcast hey y times
then make a new list called l
make a variable called name and set name to three then multiply name
    by the sum of three and 4
add name to l

first set the volume to 50 percent
second play the drum sound
third decrease the volume by 30
finally broadcast shush

make a list called z
make a variable called  y
set y to 1
add y to z
make a variable called x
set x to the first item in z

make a variable called bananas and set bananas to 10
make a variable called strawberries
set strawberies to 5
make a list called fruit
add strawberries to fruit
add bananas to fruit
```

```
when the program starts play the cave sound thats it
when the right arrow key is pressed add 1 to bananas thats it
when the right arrow key is pressed add 5 to strawberries thats it
```

A.2 Successful Test Sentences with Expected Parses

```
 when i receive hello play the meow sound thats it
 [[16, 5, [["whenIReceive", "hello"], ["playSound:", "meow"]]]],
{'variables': {}, 'lists': {}, 'scripts': [[['whenIReceive', 'hello
    '], ['doPlaySoundAndWait', 'meow']]]}


when the green flag is pressed make a variable called z and set z to
     1 thats it
{'variables': {'z': 1}, 'lists': {}, 'scripts': [[['whenGreenFlag'],
     None, ['setVar:to:', 'z', 1]]]}


when timer is not less than 1 broadcast hello and play the meow
    sound thats it
{'variables': {}, 'lists': {}, 'scripts': [[['doWaitUntil', ['not',
    ['<', 'timer', 1]]], ['broadcast:', 'hello'], ['
    doPlaySoundAndWait', 'meow']]]}


if x is less than the sum of 1 and 3 then broadcast hello thats it
[['doIf', ['<', ['readVariable', 'x'], ['+', '1', '3']], [['
    broadcast:', 'hello']]]]


if x is less than the sum of 1 and 3 then broadcast hello and add 3
    to x thats it
[['doIf', ['<', ['readVariable', 'x'], ['+', '1', '3']], [[['
    broadcast:', 'hello'], 'changeVar:by:', 'x', '3']]]]

if x is not less than three then broadcast hello thats it
 [["doIf", ["not", ["<", ["readVariable", "x"], "three"]], [["
     broadcast:", "hello"]]]]

repeat the following steps 160 times make a variable called tea and
    set tea to a random number between 1 and 5 thats it
{'variables': {'x': 0, 'tea': 0}, 'lists': {}, 'scripts': [['
    doRepeat', 160, [['wait:elapsed:from:', 0.1], ['setVar:to:', 'tea
    ', ['randomFrom:to:', 1, 5]]]]]}

when i receive the message called x decrement x by 3 thats it
[['whenIReceive', 'x'], ['changeVar:by:', 'x', ['*', '3', -1]]]

make a variable called x
make a variable called z
if x is less than z then broadcast hello thats it
```

```
{'variables': {'x': 0, 'z': 0, 'tea': 0}, 'lists': {}, 'scripts':
   [['doIf', ['<', ['readVariable', 'x'], ['readVariable', 'z']],
   [['broadcast:', 'hello']]]]}
```

A.3 Test Sentences for Synonyms

```
Generate a variable named z
Set z to a random number between 3 and 5
When the space key is pressed add 1 to z
Create a list named list
Append z to list
Remove element zero from list

play the meow sound
play the meow audio
play the meow auditory sensation
{'variables': {}, 'lists': {}, 'scripts': [['doPlaySoundAndWait', '
   meow']]}

make variable called x subtract 5 from x
make variable called x deduct 5 from x
bring forth variable called x take off 5 from x
produce variable called x take off 5 from x
yield variable called x deduct 5 from x
{'variables': {'x': 0}, 'lists': {}, 'scripts': [['changeVar:by:', '
   x', ['*', 5, -1]]]}

change volume by 5
vary volume by 5
alter volume by 5
{'variables': {}, 'lists': {}, 'scripts': [['changeVolumeBy:', 5]]}


repeat play the meow audio forever
repeat play the meow sound perpetually
repeat play the meow audio everlastingly
repeat play the meow sound always
{'scripts': [[111, 146, ['doForever', [['playSound:', 'meow']]]]]}

stop all sounds
discontinue all sounds
quit all sounds
hold back all sounds
block off all sounds
{'scripts': [[111, 146, ['stopAllSounds']]]}
```