Voice Programming in Computer Science Education

March 6, 2018

1 Introduction

Our goal is to create a voice enabled platform that uses machine learning to allow anyone to learn the fundamentals of computer science. Currently, we seek to reduce the amount of cognitive load required to be introduced to programming concepts, which would give more students opportunities in computer science. Voice programming looks to benefit both new and experienced programmers. Google Blockly is a simple language that allows us to create a robust programming grammar that can be used of people of all ages. In addition to discovering solutions to voice recognized programming, our project will help teach people who are unable to use a traditional mouse and keyboard.

1.1 What are the results or outcome of the project?

2 HCI Principles

Cognitive load, the amount of thinking required to complete a task, was a main focus when making design decisions. According to "The Limits of Speech Recognition" by Ben Shneiderman, certain applications of speech recognition can be successful as long as the designers of the application focus on creating an efficient and reduced grammar. Our grammar is the structure of language that is recognized by our speech recognition and it is built to be as simple as possible. Simplicity for the speech recognition aims at reducing cognitive load by reducing redundancy. In Cognitive Load Theory by John Sweller, the redundancy effect is when redundant information causes a user to understand the content less than if the redundant content was removed; therefore, using a simple grammar without redundancy, should decrease the amount of cognitive load needed to use voice programming in computer science education.

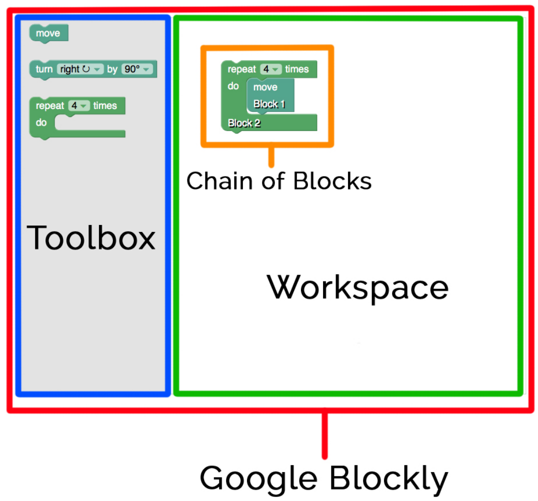

Figure 1: Google Blockly workspace diagram

3 Blockly

Google Blockly is a JavaScript library that creates a visual block programming environment where core computer science logic can be taught, such as conditionals and looping. Blockly is run in the web browser, which gives the opportunity for mobile use. We chose Google Blockly over other block based languages due to its easy to work with and has a customizable API.

Blockly has a “workspace”(See figure 1), which is where all of the blocks in the program are located. A “chain of blocks” (See figure 1) is used to refer to a series of connected blocks in the workspace. In Google Blockly, there is the ability to allow several chains of blocks to be run while in the workspace. The order of execution causes the topmost chain of blocks. The topmost chain is computed based off the highest vertical location of the first block in a chain. Normally, users interact with Blockly through looking at the workspace and adding blocks to the workspace by using a menu that holds the allowed blocks for a program, which we will refer to as the “toolbox” (See figure 1). In order to move blocks from the toolbox to the workspace, users must click and drag using a mouse.

The toolbox is where available blocks can be found. Google Blockly has a large library of blocks, which can be grouped as:

- **Logic**: if statements and equality testing
- **Loops**: while loops, for loops, for-each loops and breaking from a loop
- **Math**: incrementing variables, basic math and geometry operations, suma-

tion and randomized numbers

- **Text**: displaying text, appending text, searching through text, substring, printing and prompting
- **Lists**: creating a list with variables, length, testing if empty and searching through a list
- **Color**: changing the color and random coloring
- **Variables**: creating a variable, setting a variable's initial value and changing a variable's value
- **Functions**: a chain of blocks with a name and possibly a return value

4 Speech

To reduce cognitive load, we focused on creating a concise and fixed grammar, meaning there is only one way to say a command, which is a term we use to describe a manipulation performed by speech. While speech recognition can be used for a conversation style of programming, we wanted to mimic the standard idea that computer languages have rigid grammars.

To create a rigid grammar, we focused on how to make the grammar as simple as possible. It was important to use verbiage that came natural as well. For example, instead of having two commands to either move a block below/above another block, we only supply the user to moving a block below another one. Reducing the grammar by even one command simplifies tasks, because instead of saying, "move block 1 above block 2" the user can simply use the opposite command and say "move block 2 below block 1".

The user has the following commands:

- Add - "get a ______ block"
- Connect - "connect bock ___ under block ___"
- Connect inside - "connect block ___ inside block ___"
- Separate - "seperate block ___"
- Change field - "change ___ in block ___ to ___"
- Delete - "delete block ___"
- Run program - "run the program"
- Next level - "go to the next level"
- Stay on level - "stay on this level"

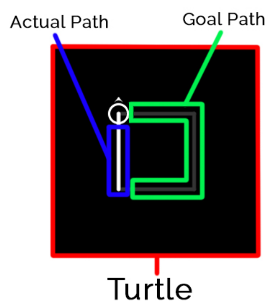

Figure 2: Turtle workspace diagram

4.1 How do the user interface commands described earlier map to the commands in the grammar?

5 Turtle

Turtle, is a Blockly game, that has a turtle that can be moved the a smaller subset of commands. The objective of the game is to trace the path shown to the user with the provided blocks in the toolbox.

- **move**: moves the turtle forward
- **turn**: turns the turtle, can decide between left or right turning and angle of turning 1,45,72,90,120,140 degrees
- **repeat**: repeats the blocks found inside, can decide to repeat either 2,3,4,5 or 360 times
- **pen**: can decide between up/down, up stops the turtle from drawing a line, down makes the turtle's movements draw lines again

Turtle is a game that breaks the concepts of computer science into individual levels. Each level reinforces already taught topics and introduces new topics, the amount of cognitive load required is reduced by splitting concepts between levels. An example of the teaching would be as follows:

1. User is exposed to how to move the turtle forward and also turn the turtle to make a square. In order to make the square, the user is only give move and turn blocks, meaning they need four move blocks and three turn blocks to complete the task.
2. User is given a repeat block in the toolbox. Now the user is tasked at completing the same task as before, but with only using one repeat block, one move block and one turn block.

A user has now, in two levels, been first introduced at simply how to move and turn the turtle in the Google Blockly environment and then naturally transitions into looping. Since the task is simply "create a square", it is not difficult to utilize the new blocks provided, given knowledge of previous levels.

6 Speech recognition

We perform speech recognition using the Google Speech API in Google Chrome. The API provides us with what it recognizes as the user's speech. We chose this option over CMU's Sphinx because we wanted to get better recognition. So we chose to use Google Speech API which has been trained on lots of data over CMU's Sphinx engine which we could train on a limited and non-representative set of data that we would generate.

7 Suggestions

We provide the user a list of suggestions to teach them the grammar. We do this because the grammar is a rigid, fixed grammar. Furthermore, we provide these suggestions because we don't give any training in the grammar to the user. Instead, we hope that they can pick it up by following the suggestions. The suggestions mechanism provides generic suggestions, not specific to the block IDs or the workspace. We do not suggest how the user should solve the program, as the user might mistakenly believe. Instead, we simply teach them how to use the grammar. However, the appearance of particular suggestions is triggered by certain states of the workspace. For example, when an empty repeat block is on the canvas, we suggest that the user "Connect block 2 inside block 1" with these exact block IDs, regardless of what the block ID of the repeat block is. Finally, we provide an example in Table 1 of the suggestions box with the corresponding workspace before and after adding the first block. We modify the suggestions list when we believe it could be useful for the user. For example, when there is a repeat block on the canvas, we suggest that the user connects a block inside of the repeat block, to make use of the repeat block. However, our current suggestions system is quite limited.

8 Corrections

As previously described, we use the Google speech API built into Google Chrome to capture user commands. However, the API has a hard time understanding commands from our grammar. We hypothesize that the API expects ordinary English, and as a result, phrases like "get a turn block" or "change 4 in block 3 to 5" are often recognized incorrectly by webkitspeechcrecognition. We developed Algorithm 1 to correct incorrectly recognized phrases ("recognitions") to what we hope are the intended commands ("utterances").

Table 1: Example of suggestions before and after adding the first block to the workspace.

when	workspace	suggestions
before		"Get a move block"
after		"Get a move block" "Delete block 1" "Run the program"

The correction algorithm first takes in a recognition r, a workspace W, and a max modification factor (described later) λ. First, the recognition r is converted into a phoneme sequence ρ using a modified version of CMU's pronouncing dictionary [1]. Then we generate a set C of all possible commands that a user can specify given the workspace. For each command c, we convert it into a phoneme sequence γ and compute its edit distance e to ρ. We iterate over C to find the minimum edit distance e^* (breaking ties arbitrarily) and corresponding minimum edit distance command c^*.

Furthermore, if μ is too far from the original recognition sequence ρ, we reject the correction μ and notify the user that we didn't understand their command. We added this feature to avoid using a "correction" from a completely unrelated sentence picked-up or recognized by the speech API. Our notion of "too far" is satisfied when

$$e^* > \lambda * length(\rho)$$

As such, there is no fixed maximum number of edits, as this could give different performance for different string lengths.

8.1 Correction Algorithm Parameters

The correction algorithm requires the following parameters to run:-

- **Recognition**: The recognition of the utterance by the speech API.
- **Workspace**: The state of the workspace that contains information about the numbers and types of blocks currently present on the workspace.
- **Maximum Modification Factor**: The maximum percentage of modification (i.e. correction) allowed to the original recognition. The methodology to get calculate this number, is described in section 8.2.

Algorithm 1 Correction Algorithm

procedure CORRECT(r, W, λ)
  $\rho \leftarrow stringToPhoneme(r)$
  $C \leftarrow generatePossibleCommands(W)$
  $e^* \leftarrow \infty$
  $c^* \leftarrow r$
  $\gamma^* \leftarrow \rho$
  for c in C **do**
    $\gamma \leftarrow stringToPhoneme(c)$
    $e \leftarrow findMinEditDist(\rho, \gamma)$
    if $e < e^*$ **then**
      $e^* \leftarrow e$
      $c^* \leftarrow c$
      $\gamma^* \leftarrow \gamma$
    end if
  end for
  if $e^* \leq \lambda * length(\rho)$ **then**
    return c^*
  else
    return r
  end if
end procedure

8.2 Calculation of Maximum Modification Factor

8.2.1 Data Collection

The data was gathered manually by speaking into the speech engine and recording the utterance, recognition and correction (if it exists) in text form. The correct and incorrect utterances, along with their recognitions and corrections were stored as CSV files.

8.2.2 Statistical Procedure

The data containing the accuracy corresponding to each threshold value from 0.00 to 1.00 (with a step size of 0.01) was collected by running the analysis on both positive and negative data examples.

For the correct examples, the accuracy was obtained by counting the total number of correct corrections. In other words, we need to count the total number of times the recognition was corrected to the original utterance. For the incorrect examples, the accuracy was obtained by counting the total number of times corrections were not made - since corrections are always valid, mapping an incorrect utterance to a valid command is not correct.

Lastly, the maximum modification factor was calculated by using the threshold value that corresponded to the maximum average accuracy of the positive and negative examples. It is important to note that the average was calculated by weighing the categories equally, and not the examples.

8.2.3 Limitations

1. We had approximately 100 positive examples and 200 negative examples, and this is not the expected distribution of real-world data. In practice, we would expect a lot more positive examples than negative.

2. The negative examples were mostly random utterances that might be picked up by the microphone in a loud environment. In other words, these utterances didn't even come close to being recognized as a valid speech command. While this data might be useful when a lot of vocal activity is going on around the user, this does not serve any purpose in adapting the speech recognition to a user that is using the speech interface in a quiet environment (which is the expected behavior).

3. Sample size of approximately 300 examples is low.

8.3 Give an example of a common case that the corrections algorithm, as currently implemented, fails on

9 Layout

As the user writes a program, certain locations on the workspace fill with blocks. This presents a challenge: what should happen when blocks overlap with one another? For our system to be practically useful, it must avoid introducing such visual impairments, which may prevent the user from issuing commands (e.g. if they cannot see a block ID) or which may introduce unnecessary cognitive load.

Blockly by default places new blocks at the top left of the workspace, even if the new block will overlap an old block. Similarly, when Blockly connects one block to another, the resulting chain might overlap another chain.

We devise simple layout algorithms for each case. To make this formal, we view the workspace as a 2D plane W where the origin is the top left corner, the x axis ranges from 0 to $\textsc{Width}(W)$, and the y axis ranges from 0 to $\textsc{Height}(W)$. Let B denote the set of blocks on the workspace and let m denote a small, constant margin. In our implementation, we choose $m = 20$ (pixels) because it is small enough to preserve space on the workspace, and large enough to prevent Blockly from automatically connecting nearby blocks.

Adding Blocks We place new blocks by finding the vertically lowest free position on the workspace and placing the new block there, exactky m pixels right of the toolbox. This is summarized by Algorithm 2.

Moving Blocks Suppose the user has just connected one block to another, and the resulting chain now overlaps n chains. We repair the layout by moving the n conflicting blocks individually via Algorithm 3.

Algorithm 2 Place New Block

```
procedure PlaceNewBlock(B, m)
    ℓ ← m                                    ▷ stores lowest y on the workspace
    for b ∈ B do
        if y_b + height(b) ≥ ℓ then
            ℓ ← y_b + height(b) + m
        end if
    end for
    return (m, ℓ)
end procedure
```

Algorithm 3 Relayout Existing Block

```
procedure Relayout(b, B, W, m)
    x ← m
    while x < width(W) do
        y ← m
        while y < height(W) do
            if CanMoveWithoutConflict(b, (x, y)) then
                Move(b, (x, y))
            end if
            y ← y + m
        end while
        x ← x + m
    end while
    Move(b, (m, m))
end procedure
```

10 Related Work

10.1 What is the related work in speech interfaces for programming?

10.2 What is the related work in speech interfaces for computer science education?

11 Future Work

References

[1] Alex Rudnicky. Cmudict. https://github.com/Alexir/CMUdict, 2015.