

B.Eng. Dissertation

Talk-to-Code: Hands-Free Program Editing

By

Archana Pradeep

In partial fulfilment of the
requirements for the Degree of
Bachelor of Engineering (Computer Engineering)
National University of Singapore

2019/20

B.Eng. Dissertation

Talk-to-Code: Hands-Free Program Editing

By

Archana Pradeep
National University of Singapore

2019/20

Project No: H0201370
Advisor: Dr. Ooi Wei Tsang
Deliverables:
 Report: 1 Volume

Abstract

Talk-to-Code program editing is a project which focuses on adding voice-enabled editing capabilities to a previous project "Coding by Dictation" (Nicholas, 2017). This project aims to make it easier for programmers, with Repetitive Strain Injuries such as Carpal Tunnel Syndrome that prevent them from typing code, to write computer programs using speech. The project works with natural speech as input and converts this into pre-designed structured commands that is then parsed by an Abstract Syntax Tree.

Unlike the previous project, this Talk-to-Code software is implemented as a VSCode plugin. Existing voice-programming software only supports text-level editing capabilities. The project focuses on designing and implementing program-level editing capabilities for the system. This report explains the challenges of using voice to edit programs and discusses the different aspects and guidelines of human-computer-interaction that has been incorporated into the design of the edit commands so as to enhance the overall user experience with the Talk-to-Code system.

Subject Descriptors:

Human-centered computing — Sound-based input
Computing methodologies — Speech recognition
Software and its engineering — Parsers

Keywords:

Voice-based programming

Implementation Software and Hardware:

NodeJS 10.16.3

VSCode 1.39.1

Acknowledgement

I would like to thank my supervisor, Prof. Ooi Wei Tsang for his expert advice and encouragement throughout my final year project. In his busy schedule, he ensured that he was meeting us every week to give input and hear our updates. His positive energy is what kept me going through the trials and tribulations of this dissertation.

I would also like to thank the 5 participants of my user test - Kyle, Jiachao, Varun, Wei Jun and Amrut who were extremely patient and understanding while facing issues unique to testing using video calls.

Lastly, I would like to thank my friends and family for their immense support. To Lawrence, my fyp teammate and close friend who always encouraged me when my project wasn't making enough progress. To Varun, for ensuring that I didn't go insane in the last 4 years and making sure I got employed. And finally, to my mom and dad without whom none of this would have been possible and for supporting me as I faced the highs and lows of an engineering degree.

Table of Contents

Title	i
Abstract	ii
Acknowledgement	iii
1 Introduction	1
1.1 Project Description	1
1.2 Aims of the project	1
1.3 Why use voice to program	2
1.4 Why editing code with voice is challenging	2
1.5 Contributions	3
2 Relevant Literature	5
2.1 Existing Voice-enabled Programming Software	5
2.1.1 Tavis Rudd’s PyCon talk and VoiceCode.io	5
2.1.2 Google Docs type with voice feature	6
2.2 Relevant Research	6
2.2.1 Spoken Programs	7
2.2.2 VoiceCode	7
2.2.3 Program-editing Behavior	8
2.2.4 Text-editing Behavior	9
2.2.5 Voice User Interface	9
2.2.6 Hands-free Programming design guidelines	10
3 Project Overview	11
3.1 Overview	11
3.2 Background	12
4 Design Considerations	14
4.1 Google Cloud Speech API Disambiguation	14
4.2 Modification to the edit commands	15
4.3 Commands Designed	15
4.4 Mapping code lines to structured commands	16
4.5 Binary Search	17
4.6 Navigating the code	17

5	Implementation details	19
5.1	Backspace	19
5.2	Delete commands	20
5.2.1	Delete Line	20
5.2.2	Delete Function	21
5.2.3	Delete Block	21
5.3	Comment and Uncomment Commands	22
5.3.1	Comment line and Uncomment line	23
5.3.2	Comment block and Uncomment block	24
5.4	Rename Commands	25
5.4.1	Rename variable	25
5.4.2	Rename function	26
5.5	Insertion Commands	26
5.5.1	Insert before line	27
5.5.2	Insert before block	27
5.6	Cut,Copy and Paste Commands	28
5.6.1	Cut line and Copy line	28
5.6.2	Cut block and Copy block	28
5.6.3	Paste above or below line	28
5.7	Other edit commands	29
5.7.1	Find and replace	29
5.7.2	Typecast variable	29
5.8	Undo	30
6	User Testing	31
6.1	Experimental Design	31
6.2	Participants	31
6.3	Procedure	32
6.4	Results	33
6.4.1	Insights from rankings of edit commands	33
6.4.2	Insights from post-task survey	35
7	Conclusion	36
7.1	Future Work	36
	References	37
A		A-1
A.1	Final list of edit commands	A-1
A.2	Programming tasks for user test	A-2
B		B-1
B.1	Survey answers represented by pie-chart	B-1

Chapter 1

Introduction

1.1 Project Description

The Talk-to-Code software focuses on creating an end-to-end program that can convert speech input to a program. Talk-to-Code program-editing focuses on adding editing capabilities to the newly ported over VSCode extension version of the Talk-to-Code software. The project aims to allow users to perform normal text-editing and additional code-level editing with voice input. While previous iterations of this project was done in Python and Java, this iteration is mostly written in JavaScript and Typescript since we intend to use Talk-to-Code as a plug-in to IDEs for more ease of access. Previous iterations also do not have editing capabilities at all and the user must undo or delete using the keyboard to make changes. The commands designed as part of this project are close to natural speech so as to provide a better user experience to programmers.

1.2 Aims of the project

Repetitive Strain Injury (RSI) is a potentially tiring condition that stems from overusing the hands to perform any repetitive task like typing, clicking a mouse and so on. Since programmers spend a good proportion of their working hours using a computer regularly, they are susceptible to suffering from RSI like Carpal Tunnel Syndrome (CTS). Talk-to-Code was built to ensure

that programmers suffering from RSI who can't use their hands to code had some other medium to do their work and not affect their productivity. While the existing Talk-to-Code project deals with being able to program by voice input, there is no way of going back and editing the program without typing. Since programmers typically spend 20% of time refactoring and editing existing code, this project would help them reduce typing significantly allowing RSI patients to recover faster.

1.3 Why use voice to program

Voice is a natural and intuitive medium to communicate with. Since we are trying to reduce movement of the wrists to help programmers with RSI to heal while still continuing to code, gestures using arms and hands are eliminated from possible alternatives. Using eyes to navigate has often been used as a complement to voice programming tools, however using eye movement alone is exhausting to do as according to (Majaranta, Aula, & R  ih  , 2004) users have to spend some time focusing on each key to activate the key and the rate of typing is at a few words per minute.

Another aspect we considered when choosing voice was also the usability of the software. Voice input is able to handle complex directions with lesser effort than using some form of eye-based typing or eye-based screen navigation.

1.4 Why editing code with voice is challenging

From my literature review discussed later in chapter 2 (Begel & Graham, 2005) I gained the insight that there are numerous challenges in using voice to edit a program

1. **Exact commands are tedious in natural language:** Spelling out all the words and symbols or describing the text in detail is an exhausting task for the user. However, abstracting out too many details and talking about your intent at a high-level is difficult for the computer to work with.
2. **Programming languages are designed to be unambiguous:** Programming lan-

guages are structured to be precise and mathematical leaving no space for ambiguity. The placement of a punctuation or a white space are paramount to the proper interpretation of a program by compilers.

3. **No spoken form of Programming languages:** Unlike natural languages which originally only had a spoken form from which a written form was derived much later, programming languages have only ever had a written form. Thus most programmers find it difficult to verbalize code. The numerous details present in the written form of a program must now be gleaned by the software from verbal input.
4. **Speech recognizing tools are not suited to programming tasks:** Speech recognizing tools' widest use case is normal text creation and editing. Hence, speech recognition models are trained for transcribing natural language and for word processing tasks. However programming languages differ from natural language in that they have different structures like blocks, classes, functions and expressions which could all be edited and navigated through quite differently.

1.5 Contributions

My contributions to the project has been to implement and design every aspect that pertains to editing code. Below is a detailed list of my contributions to Talk-to-Code:

1. Studied programmer editing behaviour through previously conducted studies and designed a list of editing commands that resemble closely with natural language.
2. Worked on top of the VSCode extension built by my teammate, to code out eighteen edit commands.
3. Enabled undo feature for all the edit commands to go back to previous state of code.
4. Built a list of preferred bi-grams that were commonly used but misunderstood keywords to improve accuracy of the Google Speech Recognizer API.

5. Designed and implemented unit tests for the edit commands and performed manual testing for overall system to ensure that editing works on any kind of structure in a c program.
6. Designed and implemented a faster way to ensure that the code users see is mapped well to the structured command list that is maintained internally.
7. Implemented navigation commands to steer around speech that has not yet been parsed into a structured command to give more leeway to the user to change their code as they are talking
8. Conducted user studies to qualitatively measure the effectiveness and usefulness of the implemented edit commands.

Chapter 2

Relevant Literature

2.1 Existing Voice-enabled Programming Software

To get a better understanding of where voice programming stands today, I explored some existing technology.

2.1.1 Tavis Rudd's PyCon talk and VoiceCode.io

Tavis Rudd's 2013 PyCon talk (Tavis Rudd, 2013) on his voice programming software using python and Dragon NaturallySpeak is one of the first well-known attempts at programming through voice. His approach was to map one-syllable words to keyboard shortcuts or longer phrases. His "grammar" however, contains mostly unusual words like "slap" for newline and "pa" for space and so on. He also only uses normal text editing commands like selecting all and deleting. The advantage of Rudd's software is that it is extensible to any programming language. However, the software has a steep learning curve, he claims it would take 2 months to remember the words.

Similarly, with VoiceCode.io (Ben Meyer, nd) the commands are short unusual words ,like "doon" for the down arrow and "snake" for underscore, as illustrated in figure 2.1 that is not user friendly. It also uses Dragon as its speech recognizer and is program-language-agnostic. However, it does not just work on code it helps with overall navigation and operation of every aspect of the computer like opening a file or searching something on the browser. VoiceCode.io's

editing capabilities are limited to just navigation, selection and deletion as with normal text editing.

jQuery

Desired:

```
$("#my-class").fadeOut();
```

Spoken:

dolly prexter spine my class
ricky dockmel fade out prekris semper

Notes:

- "dockmel" is shorthand for "dot camel"

Figure 2.1: Sample voice command and corresponding code in JQuery using VoiceCode.io

Through Rudd's software and VoiceCode.io I've learnt that my edit commands need to be close to natural language for better usability. I was also convinced after seeing the demos for both that a program-level edit is imperative to better user experience for programming.

2.1.2 Google Docs type with voice feature

The Google Docs' type with voice feature is quite fast and accurate in terms of speech recognition since it has Google Speech API running as its speech recognition engine and it allows an extensive amount of editing using speech including delete, insert, cut, copy, manipulating tables etc. It also has very intuitive, natural-language-like edit commands. However, the editing in Google Docs only works for text-editing and can't do program editing like deleting a function or commenting a block of code.

2.2 Relevant Research

Before implementation, I set out to design the edit commands. To design these commands, I read a combination of papers for text editing and program editing to understand both general editing and program editing to discern if there was any overlap between the two and to identify what behavior was exclusive to program editing.

2.2.1 Spoken Programs

The paper about designing SpokenJava, a way to speak Java, (Begel & Graham, 2005) although did not cover editing a program, spoke in detail about the importance and ambiguities of using natural language when programming. As part of their research, Begel (2006) designed a study to observe how programmers verbalize code. My takeaway for designing the edit commands came from the paper's conclusion which claimed programmers do not write programs linearly and that they plan and write different components of the program before piecing it together. This means that being able to insert before lines and copying/cutting and pasting functions and blocks would be important commands to have. I also learnt that it would be important to keep the words in the command short due to prosody level ambiguity that the paper talks about, where spaces between words could affect how it is parsed by the speech recognizer.

2.2.2 VoiceCode

This paper (Desilets, Fox, & Norton, 2006) introduces a system for programming-by-voice called VoiceCode. VoiceCode also has the same intention as our project which is helping RSI patients. There are few features VoiceCode claims to have which helped me think about the program flow for my edit commands.

1. **One command can do multiple things like navigation and insertion:** This feature allows a user to input one command which has a normal command with other types of commands like navigation or edit commands embedded into it. For e.g: "client array index i **jump out** equals zero".
2. **Using code templates:** VoiceCode provides templates of common blocks of code like if blocks, for blocks etc that users can then modify further to suit their requirements using commands like "then" and "add argument". This feature gives users leeway to make decisions, on how they want the block of code to be, much later.
3. **Flexibility in phrasing:** To allow for a less steeper learning curve when using the software, VoiceCode supports different ways of conveying the same intent. This ensures

that the user does not have to remember a set syntax. However, considering that the users are mostly programmers, this feature is not necessary.

4. **Two broad classifications of error correction:** The paper puts error correction into two categories - "Not what I said" and "Not what I meant". This can be thought of as a mistake by the software in understanding the user's speech and understanding the user's intent. In Talk-to-Code's case I try to address both. I address the mistake in what the user said (i) by adding most used phrases to the speech recognizer's API to detect it better (ii) by allowing users to delete and say parts of the command again if the command has not yet been parsed as explained in section 3.2.6 and 3.2.7

2.2.3 Program-editing Behavior

A paper conducted a study on text editing behavior of programmers to design more flexible structured editors (Ko, Aung, & Myers, 2005) . In the study programmers were asked to complete five maintenance tasks on a 503-line Java painting program in a 70-minute period using the Eclipse 2.0 IDE. Three were debugging tasks, requiring single-line changes, and two were enhancements which took up 20% of their time, requiring the creation and modification of classes, algorithms, and variables. Although the study was language specific, it put into perspective the different components of a program and how they are edited differently. One of the conclusions the paper drew was that programmers use a small set of editing patterns to achieve code modifications.

The study found that although functions and variables are semantically different, they were edited the same way (p. 1559). Another part of the study was to convert an if to a while loop. The results showed that the programmers didn't just edit parts of the existing code but deleted the entire block and wrote it again (p. 1559). This meant that for "keyword structures" such as while, if-else, for, the edit command had to be designed such that the user could delete the block rather than line-by-line. The paper also mentioned commenting out code(p. 1560) which also facilitated in designing comment and uncomment commands that work on lines and also on logical blocks of code.

Although the paper uses normal typing for their study, it helped me design my initial set of commands by the categories the paper divided different components of the code into - like names, methods, lists, comments etc. I also got the idea of having a typecasting command from this paper.

2.2.4 Text-editing Behavior

A paper on voice-based, eyes-free word processing interface EDITalk (Ghosh, Foong, Zhao, Chen, & Fjeld, 2018) conducted a Wizard-of-Oz study on 12 people to observe their text-editing patterns. Although EDITalk is meant to be eyes-free where people get audio feedback on their text edits, the paper's division of editing into three tiers with one of them being core text-editing features was insightful. The paper suggests that the core text-editing operations performed in their study were insert, delete and replace (p. 3). This helped me design my delete, rename, search-and-replace and insert commands.

2.2.5 Voice User Interface

Voice user interfaces (VUIs) allow users to interact with a system through voice commands. VUIs don't necessarily have to be visual, they can be auditory or tactile feedback to the user. In this project, the input is voice but the feedback is visual. An article on VUIs (Goossens, 2018) talks about the anatomy of a voice command which consists of :

1. **Intent:** The purpose of the voice command
2. **Utterance:** How the voice command is phrased
3. **Slot:** Required or optional variables for more detailed execution of command

While designing the edit commands I did not give much leeway for utterance since our primary user group is programmers and they are used to speaking a fixed syntax/grammar. However the intent and slot were aspects I gave thought to while designing the edit commands. I ensured that each command has just one purpose and the slot was either the line number or the cursor position which the code retrieved without user input.

The article also talks about having an error strategy in place in case the speech recognizer does not parse the voice command correctly. Unlike what the paper suggested of having the software suggest what the user meant, I decided on making an information pop up on the editor to show the user the wrong command they entered.

2.2.6 Hands-free Programming design guidelines

A paper which evaluates design guidelines for hands-free programming (Murad, Munteanu, Clark, & Cowan, 2018), gives detailed analysis of ten widely accepted design guidelines. The following list contains some of the guidelines which I applied to designing the edit commands:

1. **Visual feedback from the system:** Past research studies mentioned in this paper explains that users feel some frustration if they can't see what they've just said to the software. They don't know if they need to pause or go back and correct something. When testing the edit commands, there were instances where the command didn't get parsed correctly and nothing was reflected on the editor. It was then that I decided to have a message pop-up which showed the user what command the software received.
2. **Preventing user errors:** Studies conducted by papers that (Murad et al., 2018) uses for evaluation, found out that to build user's trust in the interface, there has to be as few errors with the functioning of the system. Errors on the Natural Language processing side were found to be less impairing to the overall user experience. Testing with both unit tests and manual tests, ensures that the commands behave as expected in Talk-to-Code.
3. **Aim for lower cognitive load:** Eleven papers mentioned in this paper, stress on the importance of making it easier to remember the speech commands. Four papers also found that providing a user with many options might overwhelm them. Thus I designed my edit commands to be quite short and similar in structure: [action] [block/function] at line [line] or just [action] line [line], for users to remember them better.

Chapter 3

Project Overview

3.1 Overview

As mentioned in the introduction, Talk-to-Code focuses on providing a VSCode extension that programmers can use to code with speech. My project deals with editing the program and also a few navigation commands that facilitate in the editing process.

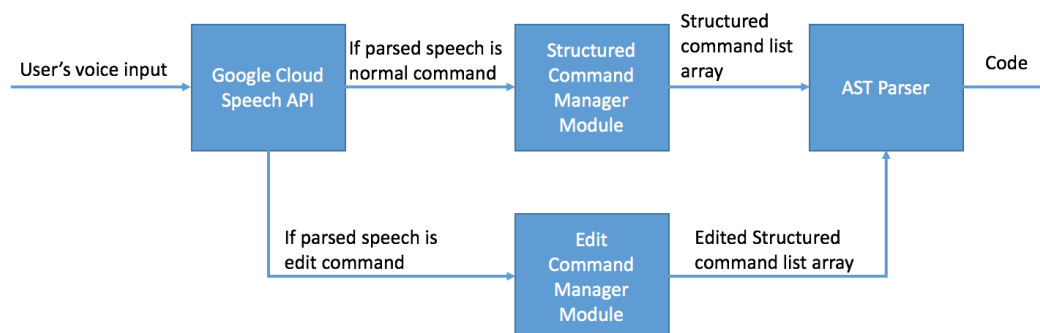


Figure 3.1: High-Level Workflow of Talk-to-Code

The application starts with user speaking to an editor when they have activated the extension. Assuming that there is already some code the user has spoken out, once the user says an edit command, it goes to the extension's main module where it is identified that it is an edit command. If the edit command has a line number associated with it, it searches for the line number in an array that has mapped the line numbers the user sees to the index of the

structured command list array that is stored internally. If such an index cannot be found, then the user is told via a message pop-up that the line does not have code on it. Else if the index can be found, then the respective method pertaining to that edit command modifies the structured command list array to reflect the edit.

3.2 Background

The Talk-to-Code software has 3 major components to it. The speech recognizer, the Abstract Syntax Tree(AST) and the Structured Command Manager that converts speech to structured command. The Abstract syntax tree and structured command manager were designed/implemented by my teammates. This section aims to shed some light on these terms and what the components do before getting into the design and implementation details of the edit commands later on.

1. **Speech Recognizer:** This is the module that transcribes voice input into text. In the case of Talk-to-Code, the speech recognizer is Google’s speech-to-text API.
2. **Abstract Syntax Tree:** To understand the AST, it is important to first understand what a parse tree is. A parse tree,also called a derivation tree (Gudivada & Arbabifard, 2018), is a graphical representation that illustrates how strings in a language are derived using the language’s grammar rules.

In programming, the parse tree represents the program in source-code form. The parse tree is usually relatively large compared to the source text since it represents the entire derivation, with a node for every grammar symbol in the derivation (Hathhorn, 2012). Hence, the parse tree takes up a lot of memory since each node and edge must be allocated space by the language’s compiler. The compiler also spends a lot of time traversing these nodes and edges during compile time, so a more succinct way to represent the program was devised.

This efficient way to represent the code resulted in the AST. It retains the essential structure of a parse tree but does away with extraneous nodes that serve no purpose in

the rest of the compiler. ASTs have been used in many practical compiler systems and syntax-directed editors.

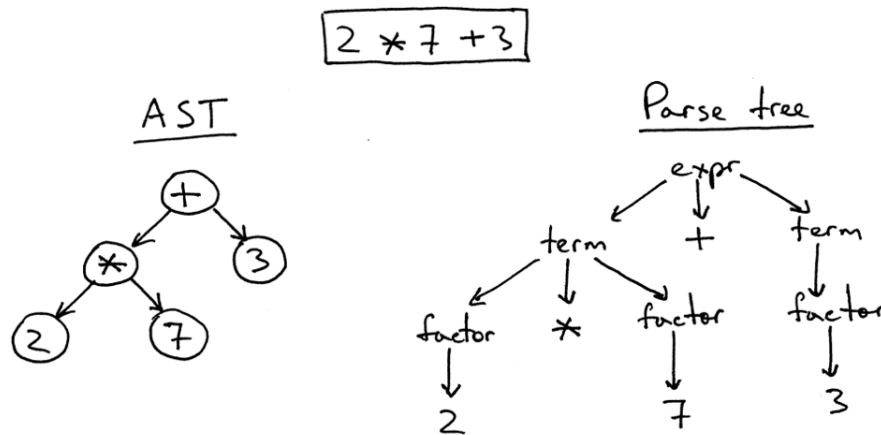


Figure 3.2: An abstract syntax tree vs. parse tree for a simple arithmetic expression

3. **Structured Command Manager** This module sits in between the speech recognizer and the AST in the overall workflow. The structured command manager takes in the text form of the voice input as given by the Google speech-to-text API and passes it into the `get_struct()` function which then breaks down the string into words that are then replaces with relevant tokens. These tokens are then joined together to form the structured command that is stored internally in a structured command list and passed to the AST to parse into code.
4. **Structured Command:** Structured command is a form of the input string with distinguishable tokens that is easier for the AST to then parse into code. For e.g "declare int first equal 1" as voice input looks like *#create int #variable first #value 1 #dec_end;;* in structured command. The various tokens and the ending ";;" helps the AST parse without ambiguities.

Chapter 4

Design Considerations

This section discusses the design of the edit commands and other auxiliary functionalities that were designed and implemented through the course of this project when I realised it would complement the edit commands and enhance the overall quality of user experience.

4.1 Google Cloud Speech API Disambiguation

While working on designing the edit commands, I tried to test them out with the speech recognizer module to see if the commands were parsed correctly. I ran into issues with using numbers and blocks such as if and while. Since a lot of my edit commands relied heavily on line numbers, it was important that the speech recognizer parsed these as numbers and not as words. I explored various configurations to the Google Cloud Speech API:

1. **Enhanced Model:** There are currently two models offered by Google Cloud speech API, `phone_call` and `video`. While the `phone_call` model is optimized for audio input coming through a phone I assumed it would be less sensitive to background noise. However, the `phone_call` model was less accurate than the default since its sampling frequency (8kHz) was half that of the default. It parsed integer as "in did your" and declare as "did Clare" and so on. Since the `video` model had the same sampling frequency (16kHz) there wasn't a difference in the accuracy of the parsing.
2. **Adaptation Boost:** This feature allows for scoring homonyms so that a homonym that is

more frequently desired gets a higher preference or "boost". I initially tried out boosting, however you must maintain a long list of words and their various scores and for the most part the boost score of the word depended on context so I decided against using this.

3. **Speech Context:** This feature allows the user to specify words or phrases that are preferred. I decided to use this configuration and define bi-grams such as "line 2" and "begin if" since they were constantly being parsed as "line two" and "begin is" respectively. It also took up less effort to maintaining this list.
4. **Class Tokens:** These are predefined classes of different data types that you might want the speech recognizer to identify such as addresses, phone numbers, ordinals etc. However, when I tried to use this to recognize numbers better, it was only available with the enhanced model- `phone_call` mentioned in the first bullet point.

4.2 Modification to the edit commands

Even after entering preferred phrases via the `speechContext` configuration on the Google cloud speech API, all my commands that end with "on line [line]" kept getting mistaken as "online [line]" almost 70% of the time. Hence all these commands were changed to "at line [line]" and this proved to be more distinguishable for the speech recognition API. My goal for the edit commands was ensure that for the majority of the time that the user is working with the extension, they don't end up repeating their commands multiple times.

4.3 Commands Designed

With the help of insights gathered from papers and articles mentioned in section 2 and my own experience as a programmer, the edit commands designed in Figure 4.1 include a mix of normal text-editing commands like deleting a line and program-editing commands like renaming a function.

	Designed Edit Commands
1	Delete line [line]
2	Delete [word] on line [line]
3	Delete [while/if-else/for/do-while] block starting on line [line]
4	Delete function [function-name]
5	Rename variable [name] on line [line] to [new-name]
6	Rename function [function-name] to [new-function-name]
7	Insert [word] after [reference-word] on line [line]
8	Typecast expression on line [line] to [dtype]
9	Comment line [line]
10	Comment [while/if-else/for/do-while] block starting on line [line]
11	Comment function [function-name]
12	Uncomment line [line]
13	Uncomment [while/if-else/for/do-while] block starting on line [line]
14	Duplicate [while/if-else/for/do-while] block starting on line [line] to line [new-line]
15	Duplicate line [line]
16	Move function [function-name] up to line [line]
17	Move [while/if-else/for/do-while] block starting on line [line] to line [new-line]
18	Move line [line] to line [new-line]
19	Move line [line] out of [while/if-else/for/do-while] block
20	Change post Increment on line [line] to pre-increment (and vice-versa)
21	Change variable data type of [variable] to [dtype]
22	Change [operator] to [new-operator] in if/while condition

Figure 4.1: List of Edit Commands Designed Initially

4.4 Mapping code lines to structured commands

When the Talk-to-Code extension was initially implemented, the user simply saw the structured command list. Only when the user said "display code" did the structured command list go through the AST parser and display code to the user. Hence at that point, when the user said "delete line 2", I knew that line 2 on the editor mapped directly to index 1 on the structured command list array.

However, as the project progressed, this "display code" command was removed and the user automatically saw the code rather than the structured command list. The issue with the initial edit commands for this feature was the mapping. The AST leaves blank lines in between the

code automatically as seen in figure 4.2. Hence, when the user says "delete line 2" there is no direct mapping between line 2 and an index in the structured command list. I therefore built a `mapLinesToCode()` function that maps the code and structured command list by ignoring blank lines, library import statements etc. and another function called `checkIfFunctionPrototype()` since the structured command list for creating a function corresponds to both the function prototype and the actual declaration of the function. These functions result in an array that looks like `[3,5,7,8,9,10]` for the code in figure 4.2.

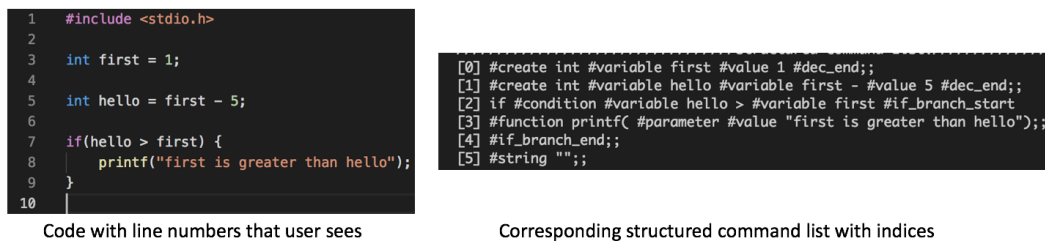


Figure 4.2: Picture of difference in indices as seen by user vs structured command list array internally

4.5 Binary Search

For the edit commands that use line numbers, the mapped array is used to find the corresponding structured command list index to manipulate. To speed up the process of searching for the line number, I decided to implement a binary search. Since the assumption is that the line numbers in the mapped array will always be increasing, it implies that they are sorted in ascending order. This brings searching for the line number down from linear time of $O(n)$ to $O(\log n)$ where n is the size of the mapped array.

4.6 Navigating the code

The way Talk-to-Code works is- the user can see what they are speaking until it becomes a parseable sentence. For e.g: If the user says "declare integer" it is not yet parseable, if they say a variable name after that it becomes parseable. This is illustrated in figure 4.3.

```

1  #include <stdio.h>
2
3  int i;
4
5  begin loop condition i equal 0 condition i less than 5

```

The for loop on line 5 while it is still being spoken and not parseable

```

1  #include <stdio.h>
2
3  int i;
4
5  for (i = 0; i < 5; i++){
6
7
8  }

```

The for loop once the user dictates the whole command

Figure 4.3: An example of "for" loop before being parseable and after

It was only after I started implementing my edit commands, that I realised how important it was to navigate across the code. While my teammate implemented a go up and down feature, I decided that we also needed a way to go left and right on the same line. The go left and right commands were designed with the fact that we are programming with voice in mind. Hence, go left and right do not just go left and right by one character space on the line but rather by one word. This is because, with the Google Cloud Speech API, it is hard to spell out words due to the ambiguity of some of the letters like 'see' for 'c' or 'be' for 'b' and so on.

The go left and right only works on the line that has not been parsed to a structured command yet since for the user to edit a structured command, they must have knowledge of how each structured command looks. This feature helps convert an *if* block to *while* loop or change the conditions within the *for* loop with a lot of ease.

Another set of navigation commands I designed and implemented were scroll up and scroll down. After resizing my window to a much smaller size, I found out that the editor doesn't move with your code. So the user could be typing at line 10 but he/she can only see line 3-7 on the window. I thought this would also help when programmers review their code as it is during the review process that users make some last minute edit changes. I also thought scrolling up and down using the mouse, although a minute strain to their hands, defeats the aim of making sure RSI patients get complete rest while still being able to program. The scroll up and down unlike all the other commands I've implemented deals directly with the VS Code API rather than manipulating internally stored cursor variables or the structured command list.

Chapter 5

Implementation details

This chapter will cover the implementation details of the edit commands and other additional functionalities. It also explains why I implemented the commands the way I did and how the user experience is affected with each of the commands.

5.1 Backspace

To supplement the navigation commands in the previous chapter, I also implemented a backspace command that deletes a user-defined number of words on the line using "backspace [n]". This was also made with similar reasoning in mind as that of the go left and right commands that the user uses voice and thus character level editing is too detailed. It takes less effort to delete a word and say a corrected one than to delete character by character. This command works in tandem with the go left and right commands only on text that the user can see hasn't been parsed yet.

One of the challenges with implementing the backspace command was with the current speech array. It is an array of strings of different lengths divided by the pause the user took between saying each phrase. This array stores everything the user has said that cannot be parsed into a structured command yet. However when the user says a command like "backspace 3" and the last/most recent string inside the current speech only has three words the backspace has to be carried on to the next string inside the current speech and so on.

5.2 Delete commands

This section will cover the design decisions made to implement the delete commands. There are 3 delete commands in Talk-to-Code for 3 levels of abstraction in a program structure. They are:

5.2.1 Delete Line

This command allows a user to delete a line of code. The way that it is done internally, is by first checking if the line number provided by the user has code, if not the user sees an error message as shown on figure 5.1. If this check passes, the index corresponding to that line number is found in the structured command list. Once it is found, a simple splice() function deletes it from the structured command list and it is then sent over to the Abstract Syntax Tree for parsing to code. The delete command does not check if the code after the delete is syntactically correct because I think the job of the delete command is to delete regardless and it is up to the user to make sure that the code remains syntactically correct.

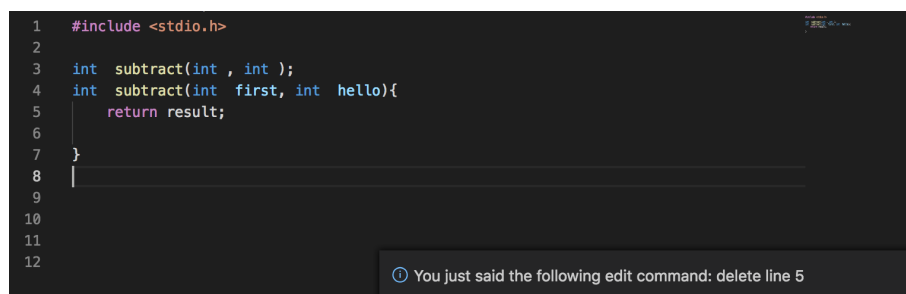


```
1 #include <stdio.h>
2
3 int subtract(int , int );
4 int subtract(int first, int hello){
5     int result = first - hello;
6
7     return result;
8 }
9
10
11
12
13
14
```

ⓘ Sorry! No code on line 6. Please provide a line number that has co...

ⓘ You just said the following edit command: delete Line 6

Figure 5.1: Error message after user attempts to delete line 6 which has no code



```
1 #include <stdio.h>
2
3 int subtract(int , int );
4 int subtract(int first, int hello){
5     return result;
6 }
7
8
9
10
11
12
```

ⓘ You just said the following edit command: delete line 5

Figure 5.2: User deletes line 5 correctly

5.2.2 Delete Function

Delete function deletes a function mentioned by the user. Since the edit command are primarily built for C code, the assumption is that there are no overloaded/overridden functions. The delete function command returns an error message to the user if the function is not found, so as to provide a user a way to debug as shown in figure 5.3



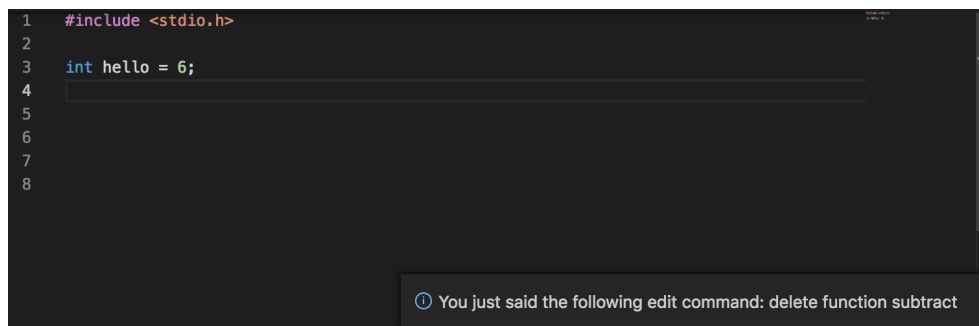
```
1 #include <stdio.h>
2
3 int hello = 6;
4
5 int subtract(int , int );
6 int subtract(int first, int hello){
7     int result = first - hello;
8
9     return result;
10 }
11
12
13
14
```

① Sorry! Function with name hello not found.

① You just said the following edit command: delete function hello

Figure 5.3: User tries to delete a function called hello

If successful in deleting, the user receives visual feedback from the editor and can see that the function has been deleted and also sees what command they just said. This helps reduce the confusion in the user as to whether they can say the next command or if their interaction with the software is going as planned.



```
1 #include <stdio.h>
2
3 int hello = 6;
4
5
6
7
8
```

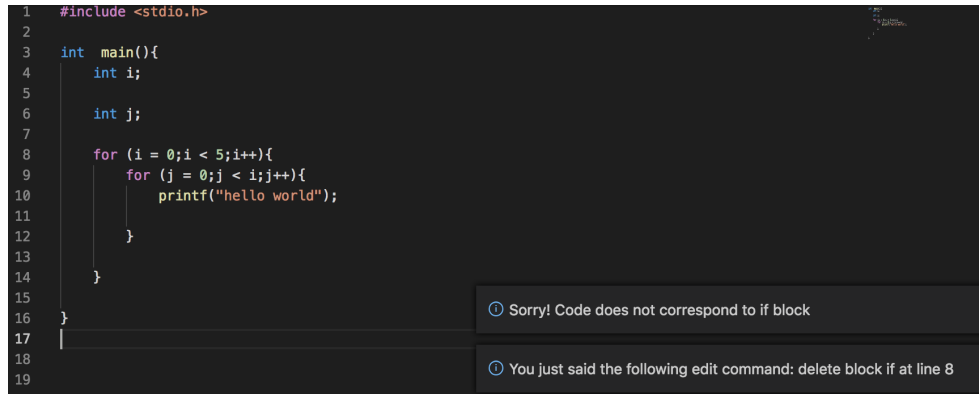
① You just said the following edit command: delete function subtract

Figure 5.4: User successfully deletes function subtract

5.2.3 Delete Block

This function deletes a specific block starting at a user-defined line. If the start of the block cannot be found at the specific line, the user receives an error message. Furthermore, the delete

block function is complex in that it has to take into consideration nested blocks of code. I keep track of nested blocks using a counter. The counter is incremented for every block of the same type after the block to delete and decrement every time I see an end block until the counter hits zero.

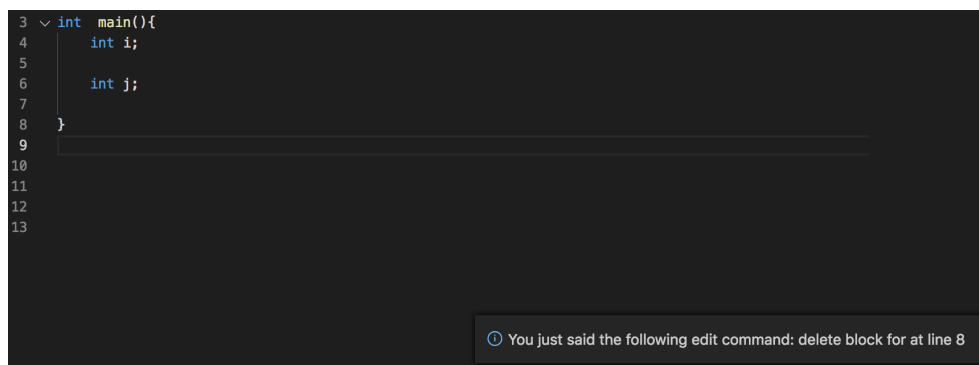


```
1 #include <stdio.h>
2
3 int main(){
4     int i;
5
6     int j;
7
8     for (i = 0; i < 5; i++){
9         for (j = 0; j < i; j++){
10             printf("hello world");
11         }
12     }
13
14 }
15
16 }
17
18
19
```

① Sorry! Code does not correspond to if block

① You just said the following edit command: delete block if at line 8

Figure 5.5: User tries to delete an if block when there is a for block on line 8



```
3 int main(){
4     int i;
5
6     int j;
7
8 }
9
10
11
12
13
```

① You just said the following edit command: delete block for at line 8

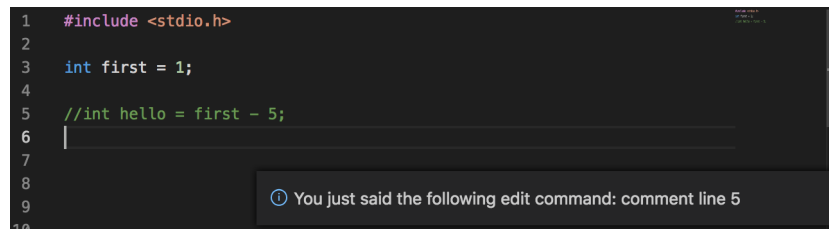
Figure 5.6: User successfully deletes for block on line 8

5.3 Comment and Uncomment Commands

This section covers the design and implementation details of comment and uncomment commands and also the user experience considerations when the command does not work as the user intended for it to, so as to relieve the cognitive load mentioned in chapter 2. I made the decision to not have a "comment function" command since users can just not call the function if they don't want to include it in the program flow.

5.3.1 Comment line and Uncomment line

Comment line works in the same way as delete line works. The key difference is that instead of splicing the line in the structured command list, `#comment` and `#comment_end` tokens are appended to the beginning and end of the line.

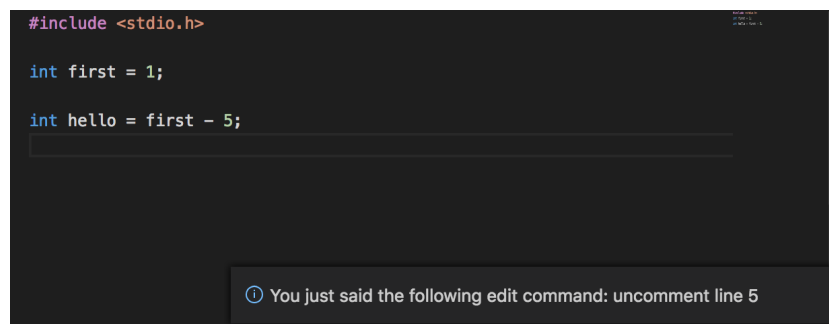


```
1  #include <stdio.h>
2
3  int first = 1;
4
5  //int hello = first - 5;
6
7
8
9
10
```

① You just said the following edit command: comment line 5

Figure 5.7: User successfully comments line 5

Uncomment line works in contrast to comment line and takes away the comment tokens from a line if present (figure 5.8), else displays an error message to the user (figure 5.9).



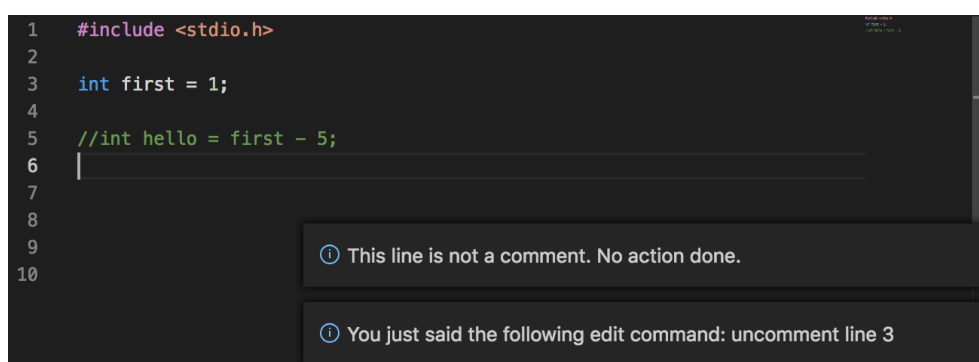
```
#include <stdio.h>

int first = 1;

int hello = first - 5;
```

① You just said the following edit command: uncomment line 5

Figure 5.8: User successfully uncomments line 5



```
1  #include <stdio.h>
2
3  int first = 1;
4
5  //int hello = first - 5;
6
7
8
9
10
```

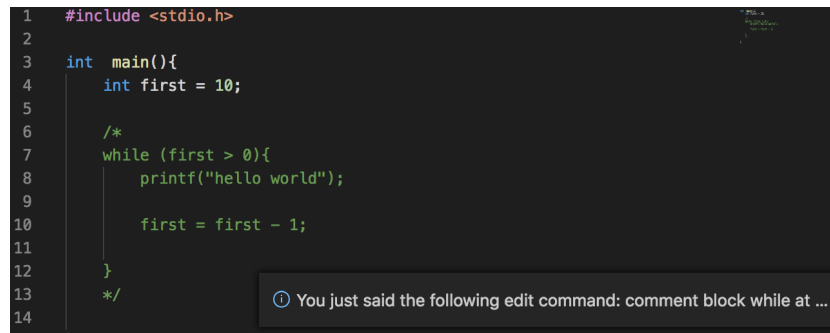
① This line is not a comment. No action done.

① You just said the following edit command: uncomment line 3

Figure 5.9: Error message displayed when user tries to uncomment line 3.

5.3.2 Comment block and Uncomment block

Commenting and uncommenting block finds the start and end of a block in the same way as delete block does. Once the block is found, comment tokens are appended to the first index and last index of the structured command list elements corresponding to the block or they are removed for uncomment block.

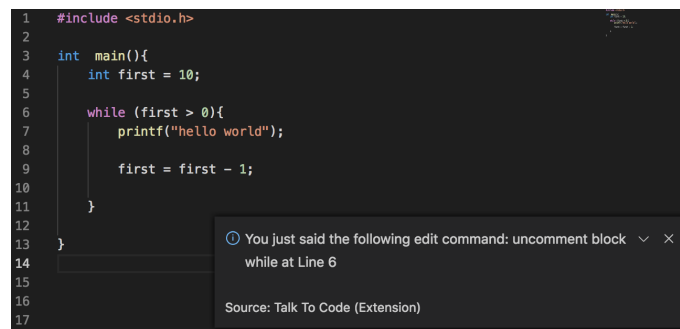


```
1  #include <stdio.h>
2
3  int main(){
4      int first = 10;
5
6      /*
7      while (first > 0){
8          printf("hello world");
9
10         first = first - 1;
11     }
12     */
13
14 }
```

① You just said the following edit command: comment block while at ...

Figure 5.10: User successfully comments out while block

The uncommenting of a block's command requires a line number, to allow users more leeway with the uncommenting command, the line number to refer to the commented block can either begin from where the comment starts as shown on figure 5.11 or it can start from where the actual block starts. If the block the user is referring to is not commented, an error message is displayed to the user similar to that of uncomment line.



```
1  #include <stdio.h>
2
3  int main(){
4      int first = 10;
5
6      while (first > 0){
7          printf("hello world");
8
9          first = first - 1;
10     }
11 }
12
13
14
15
16
17
```

① You just said the following edit command: uncomment block while at Line 6

Source: Talk To Code (Extension)

Figure 5.11: User successfully uncomments while block using start of comment at line 6

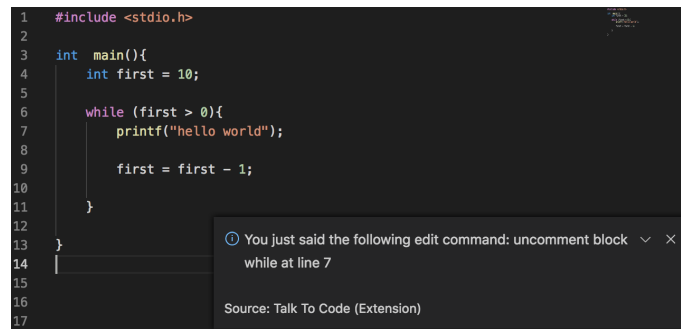


Figure 5.12: User successfully uncomments while block using start of comment at line 7

5.4 Rename Commands

This section talks about renaming functions and renaming variables in the file. This section deals with how I incorporate design guidelines I learnt from chapter 2 and also how I deal with the unique challenges with implementing these two commands.

5.4.1 Rename variable

This function renames all mentions of this variable across the file. However, I realised that the variable should be renamed within a specific scope like a function or a block. So the rename variable now looks at where the cursor is currently and finds the closest declaration of the variable from the cursor. Then the rename variable finds the scope of that declaration and renames all the variables of that name within the scope.



Figure 5.13: User renames variable first to goodbye in subtract(); cursor position: line 25

5.4.2 Rename function

Rename function does not worry about scope, it instead just searches for all mentions of the function name with a token `#function` or `#function.declare` and replaces it with the new function name. If the function name cannot be found, it returns an error message to the user.

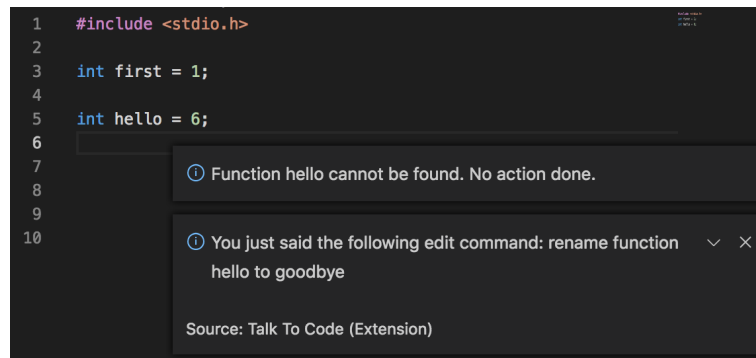


Figure 5.14: User tries to rename a function hello which does not exist.

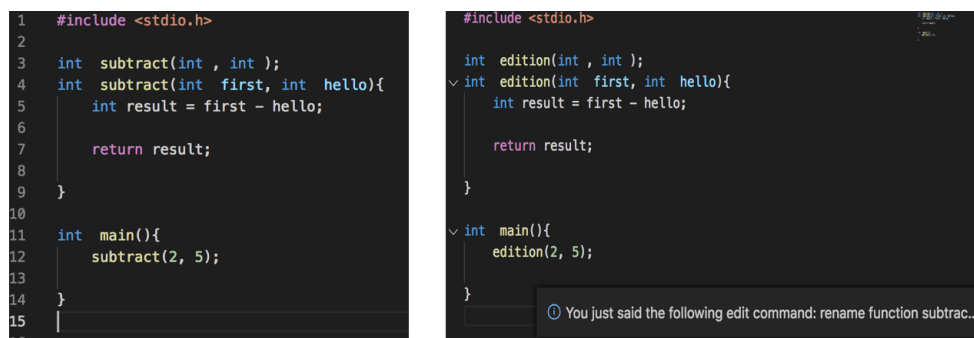


Figure 5.15: User successfully renames function subtract to edition.

5.5 Insertion Commands

Insertion commands include insertion before a line or insertion before the nearest block of code of predefined type (nearest if block if command is "insert before if block").

5.5.1 Insert before line

Insert before line moves cursor to before a reference line. This command helps to navigate the code without having to move up or down by just one line at a time. Like other commands, if the line that acts as the reference line does not contain any code, the user gets an error message. The reason I have to check for this is because the software does not have a lot of control over the code that the user sees in the sense that the AST decides where the spaces between the lines of code are inserted and so the only way to manipulate the code seen on the screen is through manipulating the structured command list. Thus the reference line must contain code which can then be mapped to an index on the structured command list.

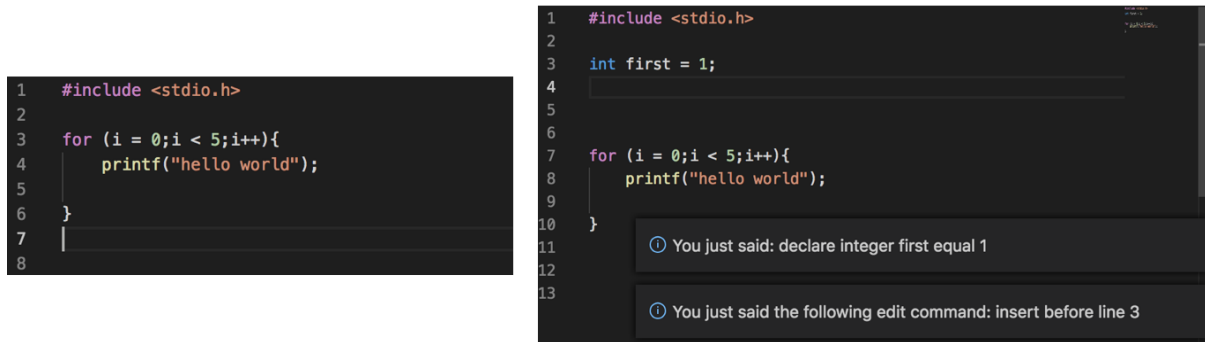


Figure 5.16: User successfully inserts before line 3

5.5.2 Insert before block

Insert before block inserts before the closest block of that type to the cursor, this allows the user to make local changes as and when they are typing since I learnt from research studies that programmers do not program linearly. The insert block first find the nearest start of that block type from the cursor position and then uses that as reference to place the new cursor position.



Figure 5.17: User successfully inserts before nearest if block from cursor initially on line 14.

5.6 Cut, Copy and Paste Commands

These commands help duplicate lines and blocks of code and also to move around these blocks of code. These functions are meant to be useful for the programmer when refactoring or changing program flow of the code.

5.6.1 Cut line and Copy line

Cut line deletes the line from the structured command list and saves it to a buffer while copy line does not delete anything but just saves the line to the buffer. This buffer is cleared every time cut and copy are invoked.

5.6.2 Cut block and Copy block

Cut and copy block works in the same way that all other block commands work in that it finds the start and end of the block while taking into consideration nested blocks and functions. This chunk of code is then saved to a buffer. The cut block also deletes this chunk of code from the structured command list and this reflects on to the editor once it passes through the AST.

5.6.3 Paste above or below line

Paste above or below line uses the data stored in the buffer from the cut and copy commands and inserts them into the structured command list at the mapped index from the reference line of code. Note that the buffer stores structured commands, not the actual code. If the buffer is

empty, the paste command shows an error message to the user stating that the buffer is empty.

5.7 Other edit commands

These are additional edit commands I felt were important after doing my literature review.

5.7.1 Find and replace

Finds and replaces a word across the entire document. Since rename variable only works within a certain scope depending on the variable's declaration, I thought it would be useful to have a global find and replace word function. I have also observed that this feature exists in most text editors hence I thought it would be good to have a familiar edit command.



Figure 5.18: User successfully finds all instances of the word hello and replaces it with goodbye.

5.7.2 Typecast variable

Typecast variable is an edit command that is used to typecast a variable to another data type maybe while refactoring or just while changing the data type of the original variable. While this command will be used less frequently than the other commands, I learnt from my literature review that it is useful to have.



Figure 5.19: User successfully typecasts variable first to double.

5.8 Undo

The undo command ("Scratch that") works by keeping a snapshot of the code before the edit command and restoring the previous state once the undo command is invoked. The initial idea for undoing edit commands was to use the edit commands themselves for undo for e.g. uncomment to undo comment or rename variable to undo a previous rename variable. However, from a software design perspective, it would not have been consistent with the undo logic for the non-edit commands and hence I decided to keep snapshots.

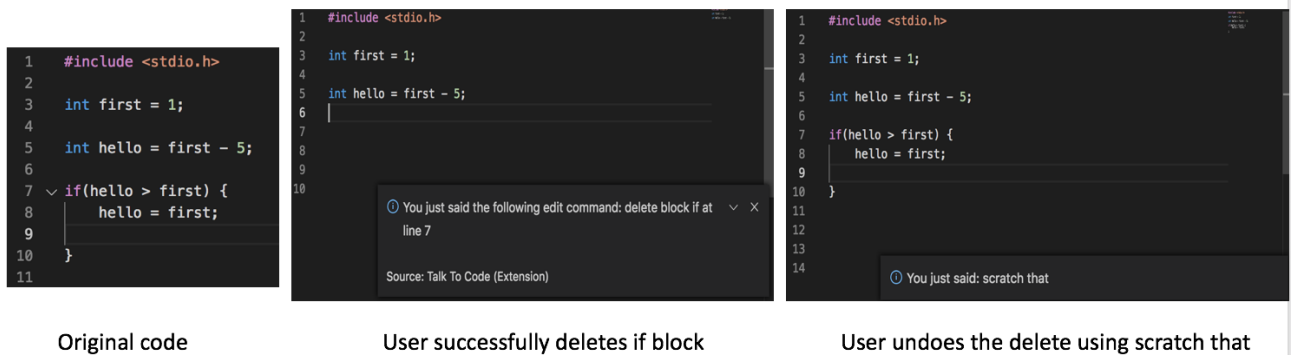


Figure 5.20: User successfully undoes the delete block command

Chapter 6

User Testing

I conducted a user test to see how easy and intuitive the interaction of programmers was with the Talk-to-Code system while editing the program. This user test had only 5 participants due to the covid-19 situation and the test was conducted remotely over zoom and google hangouts.

The user test was conducted to mainly answer the following questions:

1. How useful are the edit commands that were designed and implemented?
2. Was the user experience with edit commands enhanced due to the pop up messages?
3. Did the length and intuitiveness of the edit commands reduce cognitive load on the users?

6.1 Experimental Design

The test involved 2 ranking exercises, 3 programming tasks and one survey with 4 questions at the end of the programming tasks. Due to time constraints, not all commands were tested using the programming task. There is also variation among the participants in their choice of edit commands for each task.

6.2 Participants

The 5 participants (all male) were students at the National University of Singapore. The average years of programming experience was 5.5 years. They have all programmed in C before as they

all either majored in Computer Engineering or Computer Science. I only chose participants who were programmers because they are our biggest user group. Beginner programmers are less likely to use voice to program.

6.3 Procedure

1. Participants were given a list of edit commands and were asked to rate them from most useful to least useful based on their experiences as a programmer.
2. Participants were then given editing tasks with pictures showing before and after state of a piece of code and their goal was to make the current state of the code look like the after state using the edit commands. One such task is shown in figure 6.1
3. Participants were once again asked to rate the usefulness of the edit commands after having experienced the software
4. Participants were then surveyed for how they felt about the user experience, what confused them about the editing aspect of the project and any suggestions they would like to make.

BEFORE	AFTER
<pre>int swap(int , int); int swap(int first, int hello){ int temp = first; first = hello; hello = temp; return 0; } int main(){ int first = 1; int hello = 2; if(hello > first) { hello = first; } }</pre>	<pre>int swap(int , int); int swap(int first, int hello){ int temp = first; first = (double) (hello); //hello = temp; return 0; } int main(){ int first = 1; int hello = 2; /* if(hello > first) { hello = first; } */ }</pre>

Figure 6.1: Programming task to get participant to try out edit commands.

6.4 Results

6.4.1 Insights from rankings of edit commands

Prior to ranking I found out from the participants that their criteria for ranking the edit commands was how often they perform actions similar to the edit commands on a day-to-day basis. From figure 6.2 it can be seen that the lowest ranked command was delete function with only 20% of the participants finding it very useful. Next comes find and replace with 40% of the participants finding it very useful since they misunderstood that rename variable and find and replace does the same thing. On the other end of the spectrum were the insert before line command with 60% of users finding it very useful hence placing it in the first 6 positions in the rankings and 40% of the users placing it in the middle 6 positions of the rankings. After

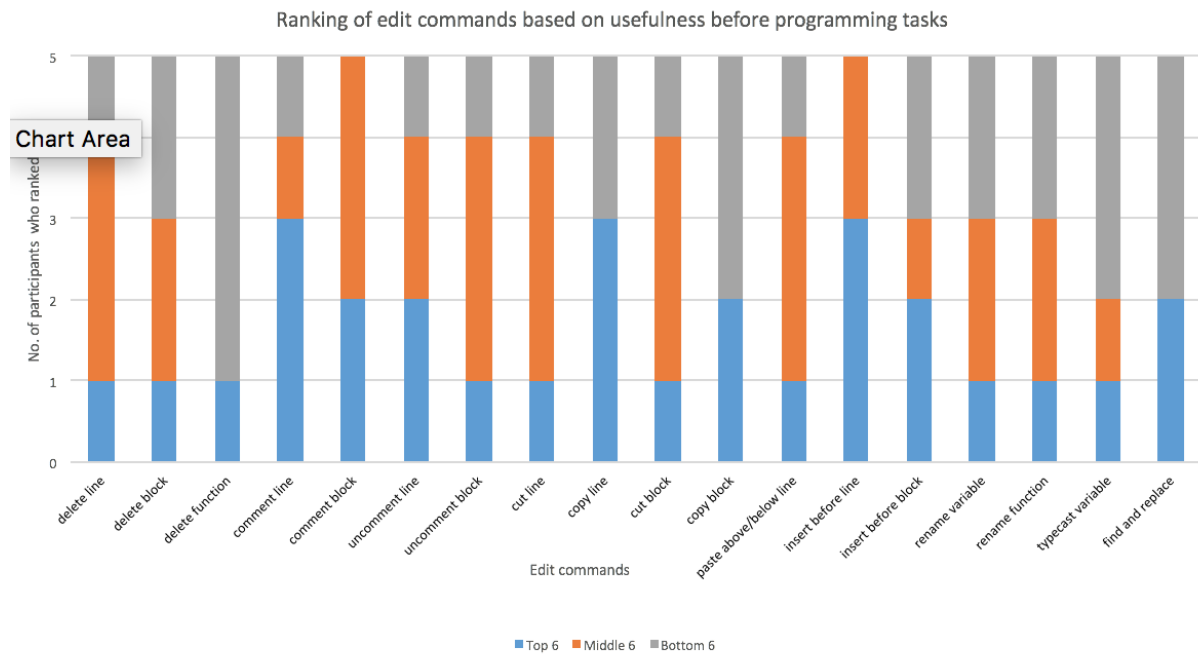


Figure 6.2: Ranking of edit commands by perceived usefulness before programming task

the 3 programming tasks, the participants' general perception of the commands and relative rankings did not change considerably in most cases. With delete function still ranking close to the bottom with only 20% ranking it in the top 6 and the other 80% of participants ranking it in the bottom 6 as shown on figure 6.3. I believe this is because they did not see themselves using

a delete function command as frequently as the other commands and also the programming task that was meant to get the participants to use delete function was done by all of them using rename function and delete line commands.

The typecast variable command's ranking also reduced with 0% placing it in top 6 as opposed to before the programming task where 20% of the participants placed it in the top 6. When I asked users why they ranked typecast so low, 80% of the users said they mostly program in python where typecasting is not used that much and they have only really used typecast in C or Java but they don't use these languages very frequently.

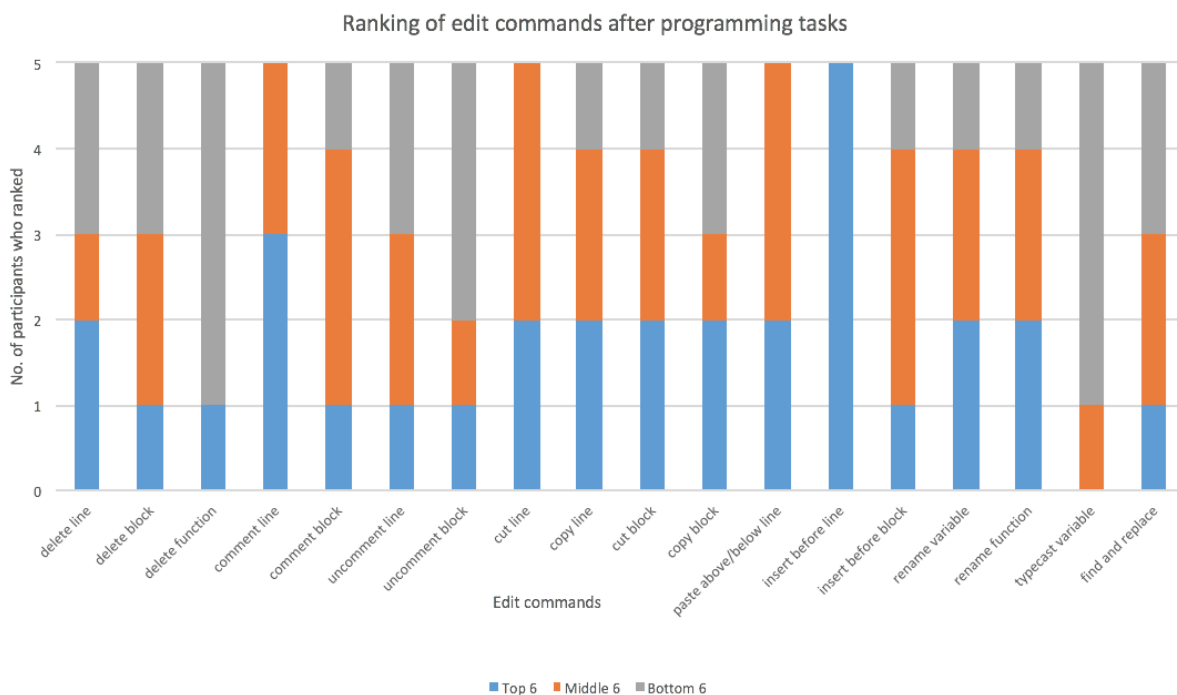


Figure 6.3: Ranking of edit commands by perceived usefulness after programming task

On the other hand, the insert before line command was placed in top 6 by 100% of the participants because they found it very useful in navigating across the code and jumping to lines. Comment line, cut line and paste above/below line also saw no ranks in the bottom 6. What was interesting about cut line however was that 80% of the participants said they used cut line even to delete a line just in case they want to add it back in later using paste. This made me realize that this is perhaps the reason delete line was ranked so low among the edit commands.

During the programming tasks, I realised that majority of participants used find and replace instead of rename variable even after they were informed that rename variable only renames within a scope. And as opposed to the claim made by (Ko et al., 2005) that programmers prefer to delete an entire function or block and rewrite it again, the participants in the user test renamed a function and deleted lines instead of deleting the entire function.

The one common misunderstanding that I noticed while conducting the test was participants trying to use lines where there was no code to perform their edit command for e.g insert before line 7. They had a hard time understanding why they couldn't edit a blank line since it is controlled by the AST. I believe this was detrimental to the overall user experience but can be avoided with good user documentation.

6.4.2 Insights from post-task survey

From the post-programming task survey, 100% of participants said that the edit commands were intuitive and easy to remember. The participants attributed the ease of remembering to the consistency in the structure of the edit command. Thus I was successful in reducing cognitive load on the users. Without asking, 80% of the participants commented on the pop-up messages for errors and showing what the speech recognizer parsed and thought the pop-ups weren't invasive. They said it helped ensure that the software was not a black box and that the pop-up messages help to debug or see any mistakes made. Thus I was successful in reducing the frustration and uncertainty of the users with the software.

The users were asked if undoing the edit commands enhances their experience with the system and 100% of the participants said yes. The participants recognized that parsing speech may not always be accurate and the edit commands might not do what the user intended for it to do if the parsing of the command went wrong. Hence the undoing helps increase their trust with the system.

Lastly, when asked if they would use the talk-to-Code software in its current state, only 60% of participants said they would. The other 40% thought the software was good for smaller programs but not well-developed for a large codebase.

Chapter 7

Conclusion

To summarise, this thesis covers the design and implementation of edit commands and some auxiliary navigation commands that facilitate the editing capabilities of the Talk-to-Code software. This was the first time that a voice-programming software has program-level editing capabilities built into its repertoire of commands. That being said, there is still quite lot of ground to be covered in terms of editing capabilities and the languages that these edit commands can work with.

7.1 Future Work

The program currently only has edit commands restricted to lines, functions and blocks of code like while/for/if/else etc. Once object oriented programming languages are enabled on Talk-to-Code there is potential to have class-level edits or edits that facilitate inheritance.

Another feature that the edit command could have is suggestions for edits. For e.g: if the user tries to delete a line that does not have code, the software could prompt the user with a suggestion asking if they meant to edit another nearby line instead. The edit commands could also have auto-correcting abilities. If the user says an edit command in the wrong order or format, the system could automatically correct it for a more seamless interaction.

The software could additionally record words the user edited to new words so that this can be fed back into the speech recognizer for a more robust speech parsing experience.

References

- Begel, A., & Graham, S. (2005). Spoken programs. Vol. 2005 (pp. 99 – 106), 10, 2005.
- Ben Meyer (n.d.). Voicecode.io.
- Desilets, A., Fox, D., & Norton, S. (2006). Voicecode: An innovative speech interface for programming-by-voice (pp. 239–242), 04, 2006.
- Ghosh, D., Foong, P., Zhao, S., Chen, D., & Fjeld, M. (2018). Editalk: Towards designing eyes-free interactions for mobile word processing (pp. 1–10), 04, 2018.
- Goossens, F. (2018). Designing a vui – voice user interface.
- Gudivada, V. N., & Arbabifard, K. (2018). Chapter 3 - open-source libraries, application frameworks, and workflow systems for nlp. In V. N. Gudivada, & C. Rao (Eds.), *Computational analysis and understanding of natural languages: Principles, methods and applications*, Vol. 38 of *Handbook of Statistics* (pp. 31 – 50). Elsevier.
- Hathhorn, C. (2012). Engineering a compiler, second edition by keith d. cooper and linda torczon. *ACM Sigsoft Software Engineering Notes*, 37, 01, 2012, 36–37.
- Ko, A., Aung, H., & Myers, B. (2005). Design requirements for more flexible structured editors from a study of programmers’ text editing (pp. 1557–1560), 04, 2005.
- Majoranta, P., Aula, A., & Rähkä, K.-J. (2004). Effects of feedback on eye typing with a short dwell time (pp. 139–146), 01, 2004.
- Murad, C., Munteanu, C., Clark, L., & Cowan, B. (2018). Design guidelines for hands-free speech interaction (pp. 269–276), 09, 2018.
- Nicholas, L. Z. Z. (2017). *Talk-to-code: Coding by dictation*.
- Tavis Rudd (2013). Using python to code by voice. [YouTube video]. Accessed Sep. 14, 2019.

Appendix A

A.1 Final list of edit commands

	Designed Edit Commands
1	Delete line [line_number]
2	Delete block [block_name] at line [line_number]
3	Delete function [function_name]
4	Comment line [line_number]
5	Comment block [block_name] at line [line_number]
6	Uncomment line [line_number]
7	Uncomment block [block_name] at line [line_number]
8	Typecast variable [variable_name] as [data_type] at line [line_number]
9	Rename variable [variable_name] to [new_variable_name]
10	Rename function [function_name] to [new_function_name]
11	Insert before line [line_number]
12	Insert before [block_name] block
13	Cut line [line_number]
14	Cut block [block_name] at line [line_number]
15	Copy line [line_number]
16	Copy block [block_name] at line [line_number]
17	Paste above/below line [line_number]
18	Find [word] and replace with [new_word]

Figure A.1: New list of edit commands

A.2 Programming tasks for user test

BEFORE	AFTER
<pre>1 #include <stdio.h> 2 3 int main(){ 4 subtract(5, 2); 5 6 } 7 8 int i; 9 10 int j; 11 12 for (i = 0; i < 5; i++){ 13 for (j = 0; j < i; j++){ 14 printf("hello world"); 15 } 16 } 17 18 } 19 20 int subtract(int , int); 21 int subtract(int first, int hello){ 22 int result = first - hello; 23 24 return result; 25 } 26 }</pre>	<pre>#include <stdio.h> 1 2 3 int main(){ 4 int j; 5 6 edition(5, 2); 7 8 int i; 9 10 for (i = 0; i < 5; i++){ 11 for (j = 0; j < i; j++){ 12 printf("hello world"); 13 } 14 } 15 } 16 17 } 18 19 int edition(int , int); 20 int edition(int first, int hello){ 21 int goodbye = first - hello; 22 23 return goodbye; 24 } 25 }</pre>

Figure A.2: Programming task 1

BEFORE

```

int swap(int , int );
int swap(int first, int hello){
    int temp = first;

    first = hello;

    hello = temp;

    return 0;
}

int main(){
    int first = 1;

    int hello = 2;

    if(hello > first) {
        hello = first;
    }
}

```

AFTER

```

int swap(int , int );
int swap(int first, int hello){
    int temp = first;

    first = (double ) (hello);

    //hello = temp;

    return 0;
}

int main(){
    int first = 1;

    int hello = 2;

    /*
    if(hello > first) {
        hello = first;
    }
    */
}

```

Figure A.3: Programming task 2

BEFORE

```

#include <stdio.h>

int subtract(int , int );
int subtract(int first, int hello)
{
    int result = first - hello;

    return result;
}

int main(){
    int first = 1;

    int hello = 6;

    while (hello > first){
        hello = hello - 1;
    }
}

```

AFTER

```

#include <stdio.h>

int swap(int , int );
int swap(int first, int hello){
    int temp = first;

    first = hello;

    hello = temp;

    return 0;
}

int main(){
    int first = 1;
}

```

Figure A.4: Programming task 3

Appendix B

B.1 Survey answers represented by pie-chart

Are the edit commands intuitive to use and remember?

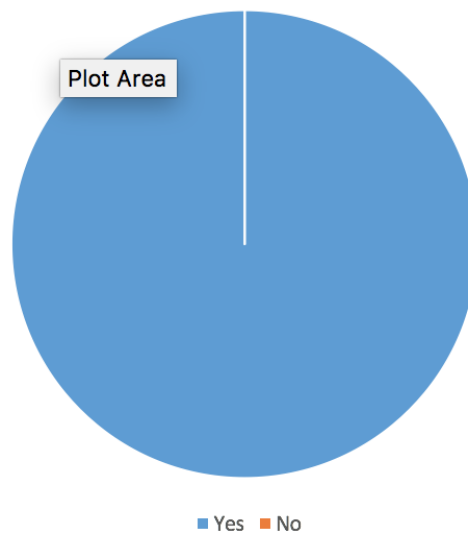


Figure B.1: Programming task 1

Does the pop-up messages enhance the interaction with the software?

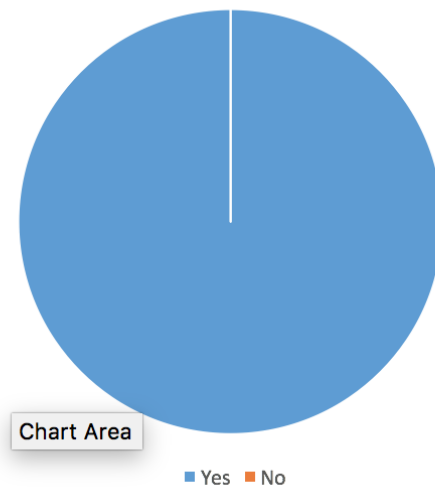


Figure B.2: Programming task 2

Will you use the Talk-to-Code software if you have to use hands-free programming

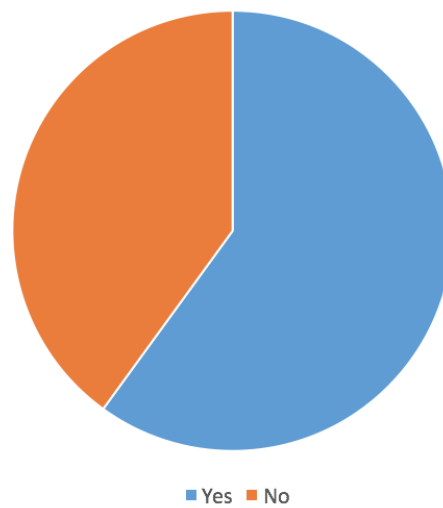


Figure B.3: Programming task 3