B. Comp. Dissertation

**Talk-to-Code: Coding by Dictation**

By

Lam Zhen Zong Nicholas

Department of Computer Science

School of Computing

National University of Singapore

2017/2018

B. Comp. Dissertation

**Talk-to-Code: Coding by Dictation**

By

Lam Zhen Zong Nicholas

Department of Computer Science

School of Computing

National University of Singapore

2017/2018

Project No: H0201280

Project Supervisor: Assoc Prof. Ooi Wei Tsang

Deliverables:

Report: 1 Volume

Contents

[1 Introduction 1](#_Toc497843603)

[1.1 Project Objectives 1](#_Toc497843604)

[2.1 Command Based Approach 1](#_Toc497843605)

[2.2 Natural Language Based Approach 2](#_Toc497843606)

[3 Progress made so far 6](#_Toc497843607)

[3.1 Project Overview 6](#_Toc497843608)

[3.1.1 Program Flow 7](#_Toc497843609)

[3.2 Speech Recognizer Module 8](#_Toc497843610)

[3.2.1 Experiment with Basket of keywords 8](#_Toc497843611)

[3.2.2 Experiment with script for a sample program 9](#_Toc497843612)

[3.3 Structured Language 10](#_Toc497843613)

[3.4 Word Corrector Module 12](#_Toc497843614)

[3.4.1 Word Corrector Phase One 13](#_Toc497843615)

[3.4.2 Word Corrector Phase Two 13](#_Toc497843616)

[3.5 Word Parser Module 14](#_Toc497843617)

[3.6 Structured Command Parser Module 14](#_Toc497843618)

[4 Plan for the next semester 15](#_Toc497843619)

[4.1 Improving the Structured Language 15](#_Toc497843620)

[4.2 Adding more constructs to the Structured Language 15](#_Toc497843621)

[4.3 Improving Word Corrector Module 15](#_Toc497843622)

[4.4 Improving overall program and feedback 16](#_Toc497843623)

[5 References 16](#_Toc497843624)

# 1 Introduction

## 1.1 Project Objectives

Coding by Dictation is a continuation of the previous project ‘Talk-to-Code: From Structured Command to Source Code’ (Gao, 2016) which creates a foundation for creating a hands-free natural language programming application. This application allows users to write program code by just dictating the program to the computer via voice input.

This project aims to enable people with disabilities or conditions such as the Carpal Tunnel Syndrome (CTS) to write computer programs. CTS (also known as Repetitive Strain Injury) is a medical condition which affects the median nerves of the hand, causing pain and discomfort to the user. Research has shown that majority of people diagnosed with CTS tend to perform repetitive tasks involving the use of fingers or hands (Crouch, 1995). Thus, it is of no wonder that programmers fall under this category, as they perform repetitive typing tasks to write computer programs.

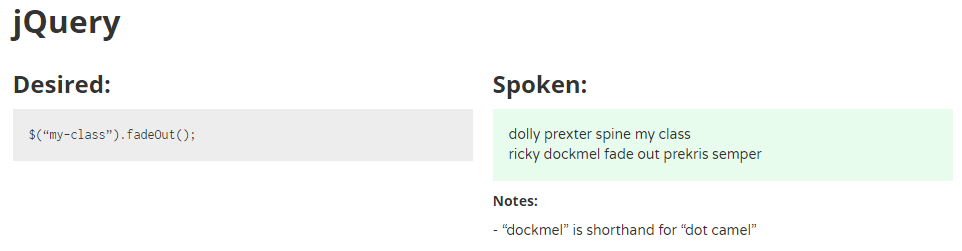
This project will enable programmers to write program code hands-free, therefore reducing the risk of contracting CTS, or even enable CTS patients to write code once again.   
2 Literature Review

To obtain a more complete picture of the activities in the area of voice programming, one needs to research on two approaches to voice programming: “natural language based approach” and “command based approach”, in order to achieve an in-depth understanding of voice programming, which could greatly help in the development of this project. We would look at one example for each approach in the following subsections to understand the existing works regarding voice programming better, and learn from them to develop a better system.

## 2.1 Command Based Approach

For command based approach at voice programming, we would be looking at the program called VoiceCode.io by Ben Meyers, a programmer who was suffering from CTS (Voicecode.io, 2017). VoiceCode is a program where users can use their voices to control the computer in real-time, where users can use their voices to draft emails, switch from application to application or navigate within an application.

VoiceCode uses an external program Dragon Dictate to convert the user’s speech into text. VoiceCode works by taking in commands from user. For example, the command “Doon twenty-one” presses the down-arrow key 21 times. “Doon” refers to the down-arrow key, while the “twenty-one” is considered a variable to signify 21 times. The figure below shows another example of the spoken command for jQuery.



*Figure 1: VoiceCode tutorial example of spoken command for jQuery*

One of the strengths of VoiceCode is that it can be used for any programming language as it provides low-level building blocks for the programming process. However, it has a rather steep learning curve due to the use of non-natural language words (as seen from figure above). It could potentially take plenty of time and memory work to be able to remember the language used to issue the correct command for the program code. This is not ideal as the user could have lower productivity levels as additional time is needed to learn the new language presented by VoiceCode, where that time could be spent on the thought process for an algorithm instead.

From this, I have learnt that a good voice programming application should preferably use a more natural English language approach, so as to smoothen the learning curve to the voice programming application.

## 2.2 Natural Language Based Approach

As for natural language based voice programming approach, we have the program Spoken Java by Andrew Begel and Susan L. Graham (Begel and Graham, Spoken Programs, 2005). Spoken Java is a voice based programming software which is relatively similar to the Java language, but coding is done by voicing out the program code in a natural way instead of typing. Similarly to our cause, the motivation for developing Spoken Java is to help programmers who are suffering from repetitive stress injuries (or CTS as mentioned earlier), by changing the mode of programming from using hands to using voice, which could alleviate the stress put into typing. Even though Spoken Java was developed around the year 2005, the findings and works of it are still very much relevant today, and we can still derive useful lessons and insights from it to further develop our voice based programming system.

According to the paper by Begel and Graham, existing voice based programming systems were not natural enough for programmers to enter and edit program code. They hope to design a better alternative to that, which should preferably follow closely to how programmers would verbalize the program naturally. In another research paper by Begel, he identified some challenges associated with developing voice-based programming (Begel, Spoken Language Support for Software Development, 2006).

1. **Speech is ambiguous in nature**

Speech has 3 aspects which renders it ambiguous, firstly, it consists of homophones, words that have similar pronunciation but have different spellings and meanings (e.g. “i” and “eye”, “to” and “two”). Secondly, sentences might sound confusing, they might sound like asking a question or they might sound like making a statement due to the inconsistent prosody in the voice’s pitch and modulation (e.g. “we eat waffles for breakfast” can sound like a question and also a statement depending on the prosody used). Thirdly, speech consists of stop words such as “um”, “uh”, and “erm” could add on to the difficulty for interpretations. All these aspects exponentially raises the difficulty for a computer to accurately understand a human speech and converts it accurately.

1. **Programming languages and tools were not meant to be ambiguous**

Programming languages were designed to be easily readable and written by the computers (and not humans). Computer requires programs to be precise and containing all necessary punctuations so as to make program analysis, compilation and running very efficient and feasible in its implementation. This is contrary to the nature of speech which is inherently ambiguous, therefore making it difficult for us to bridge the translation process between speech and programming languages.

1. **Speech tools are not suitable for programming tasks**

Speech recognizers are designed to translate speech into word text in order to create and edit text documents. The training of these speech recognizers were also done in conjunction to understand specific natural languages to support word processing tasks. Despite programming languages being similar to natural languages, they are not the same. Word processing tasks are also very different from what is required of programmers, who create, edit and navigate through programs. Thus, speech tools are not suitable for programming.

1. **Using voice for software development is something new for programmers**

Programmers are very used to programming with a keyboard for a long time, thus it might be difficult for them to adapt to switching to voice-based programming. It is something new for them and some time and effort is needed to learn this new style of programming. This is another dimension of challenge for the programmers.

Begel and his team subsequently designed a study to find out how programmers verbalize code. They have asked ten expert programmers to read a page of Java code. Their programmers were chosen from different stratums: half of them know Java programming, the other half do not. Half of them had English as their native languages, while the other half had not. Half of them learnt programming from U.S.A., while the other half learnt outside U.S.A. As each programmer had his or her own style, they verbalize the speech differently in their own manner. The results of the experiment are summarized below.

1. **Lexical level – Spoken words are hard to write down**

Several classes of lexical ambiguity were discovered during the experiment. Firstly, the issue with the use of homophones where the same word is recognized by a speech recognizer differently (e.g. “for” recognized as “4”, “fore” or “four”). Next, the issue where capitalization was not explicitly verbalized, which could cause problems as many programming languages are case-sensitive. Another problem is with the ambiguity of whether spaces between words are intended (e.g. user saying “drop stack process” for “dropStackProcess”). It is unclear where should we put the spaces between the words, and thus it is not clear whether “drop stack process” is spoken for 1 or 3 identifiers.

1. **Syntactic level – Structural ambiguities**

In many cases where a punctuation is found, it is omitted by the user when spoken. For example, the dot in “object.stack”, the parentheses in a method call like “e.printStackTrace()”, the comma between arguments in a method call, or a semicolon to terminate the statement. This is of concern as punctuations are important in programming languages to precisely mark out different structures of the program.

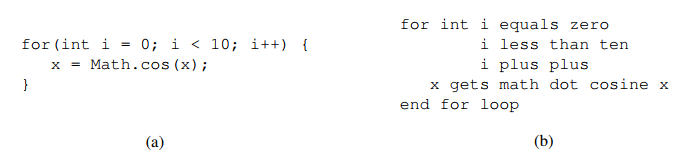
1. **Semantic level – Abstraction by programmers**

It is very natural and common for programmers to speak in higher-level code instead of lower-level code constructs. Despite the subjects being given instructions to verbalize the program code instead of paraphrasing the visible patterns within the code, some of them went on doing the latter. For instance, they say “set all the fields of array to null” instead of verbalizing it explicitly. Begel states that using high-level speech can potentially improve the productivity of the programmer as a line of phrase can generate many lines of useful code, but that is something worth looking at in the future.

1. **Prosody level – Disambiguation**

Some subjects also used prosody as a means to perform disambiguation, though that is only employed by the English native speakers. An example would be “array sub i plus plus” which could translate to array[i]++ or array[i++]. English native speakers utilizes the pause to indicate the grouping of the terms in that expression to differentiate between the two cases mentioned earlier. Many non-native English speakers would use their own native language’s prosody when speaking English, and this could potentially change the meaning of the words altogether.

Using the results from their experiments, they designed and created the program Spoken Java, which is a modified dialect of Java to match closely to how programmers would verbalize program code. They have also made improvements to their program to address the issues raised during the study.



*Figure 2: Part (a) shows Java code when writing a for loop. Part (b) shows the same for loop entered with Spoken Java (carriage returns written for clarity only)*

Figure 2 shows an example of how a Java program might be entered in Spoken Java. Spoken Java has an associated program analysis framework called Harmonia, which analyzes texts which were translated from voice by a speech recognizer. Harmonia has lexical analysis capabilities which is able to handle homophones, miscapitalized words and arbitrarily concatenated words. The program uses contextual information to perform disambiguation and choose the proper interpretations, and finally converting Spoken Java into Java code. Spoken Java is also defined by a lexical specification which allows for different ways of verbalizations to match the same construct.

However, it is noted that in Begel’s study, the programmers did not receive any feedback of their current state or progress when verbalizing the code, and they know nothing about the program’s correctness. This prevents them from understanding the aspect on how a programmer would verbally edit the code. Moreover, the program was spoken in a linear fashion, which is vastly unlike how most programmers would write code. Some programmers prefer modular programming, while others prefer planning the interface first before the actual coding. All these styles would require a speech system to accept partial code, and supports the analysis of spoken programs which are incomplete that might eventually be formulated into a usable solution. This is currently unsupported by Spoken Java.

Furthermore, many coding does not involve reading off from a script, but writing coding on the fly. This action of spontaneous voice-based coding could potentially show us what programmers would say to edit or manipulate code segments, what kind of errors and ambiguities would surface from a non-linear code entry method.

From this example, we have gained useful insights about the challenges of voice-based programming and the ambiguities associated with speech. It gives us many ideas on how we should define our structured language as well, in order to eliminate ambiguities. It has also thought me about the importance of feedback in a programming application, and how to develop a system that resembles a programmer’s thought process more closely. At this point, it is worthy to note that our program Coding by Dictation is not exactly the same as Spoken Java as our program is not totally natural-language based, but it aims towards a semi-command-based approach with natural language components.

# 3 Progress made so far

## 3.1 Project Overview

This project focuses on creating an end-to-end program which converts a user’s voice into C program code. The ideal situation would be to have a program which allows programmers to speak in a human and natural way, and translates that to code subsequently. However, it is noted that there can be many different ways to verbalize the same piece of code, and that natural English commands were rather ambiguous in the sense that there can be many different interpretations of that natural English command. (Chew, 2016)

For instance, in order to declare an integer variable with value zero, one programmer might verbalize “Create integer variable with value zero” while another programmer might say it as “Declare an integer variable equal zero”. Both programmers are trying to write the exact same piece of code through different ways of saying it, but it is not easy to have a program that is able to decipher and know that these two different verbalizations should result in the generation of the exact same piece of code.

As for the latter case regarding the ambiguity of the natural English commands, the English command “declare integer x, y, z” can be interpreted (in code) as “int xyz;” or “int x, y, z;”. This ambiguity can result in the 2 different interpretations as outlined above, and this causes problems as the program will not know which piece of code to generate and what the user actually intends to write.

In order to eliminate ambiguities, this project will introduce a structured and fixed way of expressing programming constructs, while incorporating some natural English in my structured language as well. The structured language will be presented in a later section in this paper. The system architecture of the Coding by Dictation application at the current stage is depicted in the diagram below.

Speech Recognizer Module

Word Corrector Module

Word Parser Module

Structured Command Parser Module

English text (structured language)

Processed text

Structured Command

Voice input from user

Program Code

*Figure 3: System Architecture of Coding by Dictation application*

### 3.1.1 Program Flow

We will first have an overview of how the Coding by Dictation application works, before elaborating on the details of each component in the sections to follow. The application will first take in voice input from the user. Let’s say the user says “*declare float money equal three point two end declare*”. Assuming that the Speech Recognizer module misinterprets that as “*declare float money equal three point two and declare*” and that will be the English text passed on to the Word Corrector module. Word Corrector module will recognize and correct “*and declare*” to “*end declare*” and pass on the corrected processed text to the Word Parser module, where the text will be converted to the structured command “*#create float #variable money #value 3.2 #dec\_end;;*” This structured command is then passed into the Structured Command Parser module which will translate that to C program code “*float money = 3.2;*”. The table below shows some examples of how the program flow will be for different voice inputs.

|  |  |  |  |
| --- | --- | --- | --- |
| **Text recognized by Speech Recognition module** | **Processed text after correction by Word Corrector module** | **Structured command output by Word Parser module** | **C Program code output by Structured Command Parser** |
| declare float money equal three point two and declare | declare float money equal three point two end declare | #create float #variable money #value 3.2 #dec\_end;; | float money = 3.2; |
| mummy equal mummy plus two end equal | money equal money plus two end equal | #assign #variable money #with #variable money + #value 2;; | money = money + 2; |

Some of the terms found in this section might be unfamiliar at this point of time, all of these terms will be explained in greater details in the following sections which examine deeply into each individual component of the application.

*Figure 4: Examples of program flow for different voice inputs*

## 3.2 Speech Recognizer Module

My Speech Recognizer Module converts voice input from the user into English text, using the python Speech Recognition library (Zhang, 2017). The python Speech Recognition library supports several APIs in which we can use to perform the speech recognition. Out of the several APIs, I have picked 3 of them (mainly Google Speech Recognition, Google Cloud Speech API, and Microsoft Bing Voice Recognition) to carry out further testing and experiment before deciding on one of them to use for my Coding by Dictation application.

### 3.2.1 Experiment with Basket of keywords

After coming up with an initial draft of my own structured language, I have picked out some of the keywords and conducted an experiment with 10 subjects. The subjects were asked to record their voice into an audio file which consists of the keywords. These audio files are then passed into my speech recognizer module with the use of 3 different APIs as mentioned above. The summarized results of the experiment are shown below in Figure 5 (those in yellow signifies the highest scoring API).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keywords** | **Google Speech Recognition** | **Google Cloud Speech API** | **Microsoft Bing Voice Recognition** | **Total** |
| Equal | 7 / 10 | 9 / 10 | 6 / 10 | 22 / 30 |
| If Then Else | 5 / 10 | 5 / 10 | 5 / 10 | 15 / 30 |
| End If | 0 / 10 | 5 / 10 | 0 / 10 | 5 / 30 |
| Declare integer | 5 / 10 | 7 / 10 | 0 / 10 | 12 / 30 |
| Size | 6 / 10 | 9 / 10 | 8 / 10 | 23 / 30 |
| Index | 9 / 10 | 10 / 10 | 10 / 10 | 29 / 30 |
| Create Function | 4 / 10 | 9 / 10 | 6 / 10 | 19 / 30 |
| Return Type | 1 / 10 | 8 / 10 | 2 / 10 | 11 / 30 |
| Parameter | 3 / 10 | 9 / 10 | 1 / 10 | 13 / 30 |
| Call | 5 / 10 | 5 / 10 | 10 / 10 | 20 / 30 |
| For | 0 / 10 | 0 / 10 | 0 / 10 | 0 / 30 |
| Plus plus | 5 / 10 | 6 / 10 | 7 / 10 | 18 / 30 |
| While | 0 / 10 | 4 / 10 | 0 / 10 | 4 / 30 |
| Switch case end switch | 0 / 10 | 0 / 10 | 0 / 10 | 0 / 30 |
| Dot | 2 / 10 | 4 / 10 | 0 / 10 | 6 / 30 |

From the figure above, it can be noted that Google Cloud Speech API performs consistently better than the other two in most of the cases of keywords. The difference between Google Speech Recognition and Google Cloud Speech API lies within the fact that the latter allows for a list of preferred phrases to be passed into the recognition process, which greatly enhances the accuracy of the speech recognition. On the other hand, Microsoft Bing Voice Recognition seem to be only accurate for a few keywords and is largely inaccurate for most of the keywords involved.

*Figure 5: Summarized results of Speech Recognizer module with basket of keywords*

### 3.2.2 Experiment with script for a sample program

The previous experiment focuses more on keywords, or very short phrases, as a starting point for our analysis. We realized that the actual recognition program will require the user to speak in complete sentences, for instance, to declare a variable, the user would say “declare integer count equal zero” instead of just “declare integer” as per our previous experiment. In order to affirm that Google Cloud Speech API is the most accurate API to use for our speech recognition, another experiment will be conducted. In this experiment, a script was written for a sample program to find the maximum number in an integer array, using my structured language (which will be presented in a later section in this paper). The same 10 subjects used in the previous experiment were tasked to read from the same script and their voices are recorded. Similarly, the audio files are then passed through the Speech Recognition module and the results are tabulated as follows in figure 6.

Note: The script consists of 64 words. The results below is the sum for 10 subjects, and the numerator represents the number of correctly recognized words for 10 subjects, while the denominator represents the total number of words for 10 subjects.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Google Speech Recognition** | **Google Cloud Speech API** | **Microsoft Bing Voice Recognition** | **Total** |
| Total | 389 / 640 | 423 / 640 | 247 / 640 | 1059 / 1920 |

The numbers above reaffirms that using Google Cloud Speech API produces the most accurate results over the other two APIs. Moreover, I have also observed from the raw data of the results that Microsoft Bing Voice Recognition is more suitable for recognizing proper complete English sentences than user-defined structured language. As such, I have decided on the use of Google Cloud Speech API for my speech recognizer module.

*Figure 6: Summarized results of Speech Recognizer module with script for sample program*

## 3.3 Structured Language

I have defined a set of structured language in order to express programming constructs in an unambiguous manner so that the computer program can accurately translate them into program code which is intended by the programmer. To enable programmers to remember and use the structured language in a fluent manner, some natural English constructs are also incorporated into the structured language as well. The current state of the structured language is defined below in figure 7.

|  |  |  |  |
| --- | --- | --- | --- |
| **Programming Constructs** | **Structured Language** | **Example speech** | **Example code** |
| Verbalizing words of different forms | Integer: just verbalize the number | Ten  One hundred and thirty four | 10  134 |
| Float: verbalize the number and use point for the decimal place | Thirty four point two | 34.2 |
| String: String (string content) end string | string hello end string | “hello” |
| Character: character (character content) | character x | ‘x’ |
| if-else conditions | begin if – then – else - end if  or  begin if – then – end if  (When dealing with nested if-loops, use end if to close the if loop) | begin if A then B begin if C then D else F end if G end if | if (A) {  B  if (C) {  D  } else {  F  }  G  } |
| Variable declaration | declare (variable type) (variable name) end declare  or  declare (variable type) (variable name) equal (literal) end declare | declare integer x end declare | int x; |
| declare float tax rate equal one point zero seven end declare | float taxRate = 1.07; |
| Array declaration | declare (variable type) array (variable name) size (number) end declare  (Using ‘with’ is optional) | declare integer array sequence size ten end declare  declare integer array sequence with size ten end declare | int sequence[10] |
| Variable assignment | (variable name) equal (literal) end equal | x equal ten end equal | x = 10 |
| Array assignment | (variable name) array index (number) equal (literal) end equal | sequence array index three equal four end equal | sequence[3] = 4 |
| Function declaration | create function (function name) return type (variable type)  0 or more: [parameter (variable type) (variable name)]  begin – end function  (Using ‘with’ is optional) | create function search return type integer parameter integer lower parameter integer higher begin A end function  create function search with return type integer with parameter integer lower with parameter integer higher begin A end function | int search(int lower, int higher) {  A  } |
| Function call | call function (function name)  0 or more: [parameter (variable name)]  end function  (Using ‘with’ is optional) | call function search parameter lower parameter higher end function  call function search with parameter lower with parameter higher end function | search(lower, higher) |
| For loops | for loop condition  (variable name) equal (literal)  condition  (variable name) (comparison operator) (literal)  condition  (variable name) (operator) begin – end for loop | for loop condition i equal one condition i less than sum condition i plus plus begin end for loop | for (i=1; i<sum; i++) {  } |
| While loops | While (variable name) begin – end while  While (variable name) (comparison operator) (variable / literal) begin – end while | while x begin B end while  while x less than two begin B end while | while (x) {  B  }  while (x < 2) {  B  } |

This set of structured language will be edited from time to time in order to improve speech recognition as well. For example, the Speech Recognition module has problems trying to decipher the word “if”. In most cases, the word “if” is omitted by the Speech Recognition module totally. The language is then changed for that particular construct, in the sense that the user is required to say “begin if” instead, as the Speech Recognition module has a higher accuracy in recognizing “begin if” as compared to “if” alone. More adjustments were also made during the implementation process, where signs of ambiguity were discovered in the structured language.

*Figure 7: Structured language for Coding by Dictation*

## 3.4 Word Corrector Module

Our Speech Recognition module is not perfect, we cannot 100% translate voice into text accurately. There will be instances where the Speech Recognition module misinterprets the audio. As a result, there is a need for us to process the input text into processed text, and this is where we utilize our Word Corrector module. There are 2 phases in this Word Corrector module. The first phase focuses on correcting the common errors which are misinterpreted by the Speech Recognition module, whereas the second phase focuses on using contextual information about previously declared variables to perform the correction of variable names used in the future part or lines of the coding process.

### 3.4.1 Word Corrector Phase One

In the first phase of the Word Corrector module, we correct the common errors which were misinterpreted by the Speech Recognition module. We utilize the raw data results retrieved from our earlier experiments on the basket of keywords and script for the sample program to perform our word correction. For instance, we realized that the end constructs are often misinterpreted (i.e. “end function” misinterpreted as “and function”), therefore we will attempt to correct “and” into “end” if the word “and” appended with the following word will form a valid end construct. As such, the Word Corrector module mainly targets the correction of program keywords only. Currently, the module uses a hardcoded algorithm to perform the correction of common errors, which will be improved upon in future. This will be discussed later in another section of the paper.

### 3.4.2 Word Corrector Phase Two

The second phase of the Word Corrector module uses contextual information about previously declared variables to perform the correction. After declaring variables, we will expect them to be used again, and variables which are not declared should not appear in the code at all as it will cause a syntactical error for the program. Thus, declared variable names will be stored in a list and subsequent use of variable names will reference this list to correct any erroneous variable names. This correction is done by finding the most similar word in the list to the word recognized by the Speech Recognizer module, and replaces that word with the most similar word found. The most similar word is found with the aid of the jellyfish python library, which does approximate and phonetic matching of strings (Turk and Stephens, 2016).

Firstly, we would convert all of the words into their phonetic encodings using American Soundex, and then compare the resulting phonetic encodings with Jaro-Winkler Distance to get a similarity index between two words. These are done with the help of the jellyfish python library. As the jellyfish python library provides more than just one way to doing phonetic encoding (namely: American Soundex, Metaphone, NYSIIS) and string comparisons (Levenshtein Distance, Damerau-Levenshtein Distance, Hamming Distance, Jaro Distance, Jaro-Winkler Distance), there is a need to test which pair of algorithms deliver better results. It is preferable to have higher similarity indices for closer matching words phonetically, and differentiable similarity indices between different pair of words (i.e. no ties in similarity index). After much testing, I have found that American Soundex and Jaro-Winkler Distance are the best fit for this purpose.

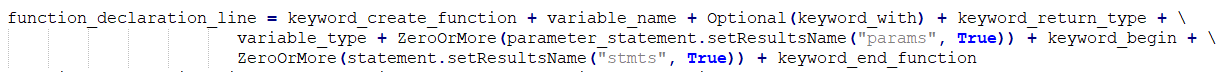
It is also worth to note that the way which we use to find the closest matching pair of words is actually a heuristic and is not an algorithm. This is due to the fact that it is difficult to truly tell which word is closer to a particular target word. For instance, if we have already declared the variables “eye”, “ai”, “aye” and we need to correct the word “i”, which of these 3 variables would we deem as closest matching to the word “i”. However, this heuristic is good enough for the purposes of this program in most instances, after testing the program with our script for the sample program from our earlier experiment.

## 3.5 Word Parser Module

This module translates processed text from the Word Corrector module into structured command for the Structural Command Parser. This module uses the pyparsing python module (McGuire, 2016) which allows parsing with user-defined grammar rules, in this case, our structured language, and converts it into structured command. More on structured command will be covered in the next module in the following section. The usage of a parsing library like pyparsing here is to ensure the robustness of the system and allows for easy editing of the structured language whenever necessary. This is a stark improvement from the initial implementation of the parser where parsing is done in a hardcoded manner.

We have also introduced the concept of context free grammar in defining our structured language’s grammar rules, after realizing that the initial set of language defined is not really properly designed and errors are not properly detected. With context free grammar, the grammar rules are now more well-defined and that new constructs or edits can be done more easily to the parser now.

The figure below shows an example of the function declaration construct using pyparsing and using context free grammar.



*Figure 8: Python code for function declaration construct*

From the figure, we can see that the structured language defining the function declaration construct can be easily changed and it is properly defined from the start keyword to the ending keyword involved. Any part of the construct can be replaced by some other keyword if we want to change the keyword to use for this construct, and we can also add in extra keywords or remove some terms in the construct should the need arises. Furthermore, context free grammar allows for recursive definitions, which allow us to create nested constructs like nested loops for example. In the case of nested loops, a for loop will allow for statements to be inserted into the for loop body, and a statement can be another for loop as well. All these increases the robustness and flexibility of the system.

## 3.6 Structured Command Parser Module

Structured command generated from the Word Parser module is converted into program code here. This module is basically a wrapper module which runs the Java program from the previous project ‘Talk-to-Code: From Structured Command to Source Code’ (Gao, 2016) as mentioned earlier. His program is able to construct a compilable and runnable C program with its Abstracted Syntactical structure, after accepting his self-defined structured command input, which we have converted from our structured language. This module completes our end-to-end program and generates C program code.

# 4 Plan for the next semester

For the next semester, I will continue to improve on the project to make it easier for programmers to code with their voices, so as to empower CTS patients to write code once again and also to reduce the risk of contracting CTS. The areas of improvements and their details will be elaborated on in the following subsections in this paper.

## 4.1 Improving the Structured Language

The current structured language might be a little bit hard to remember by heart and may still lack some natural English language aspects, in the sense that it is not really spoken in a natural English manner. Thus, the plan for next semester would be to continue redefining and editing the structured language, in order to make the speaking more natural, while retaining the unambiguous domain of the language so that the translation to program code is not ambiguous. There will also be attempts to improve the structured language to use keywords which can be more easily recognized by the Speech Recognition module to make the recognition process more accurate.

## 4.2 Adding more constructs to the Structured Language

The current structured language does not contain every programming construct available, and is restricted to only a limited set of constructs. The plan for the following semester would be to introduce more constructs to the program. Some of the constructs which were planned to be added are as follows:

1. Switch – Case construct (The switch statement)
2. The use of logical “and” and logical “or” when performing comparisons
3. Shorthand assignment operators (such as “+=”, “-=”, “\*=”, “/=”)
4. Allowing the use of symbols like percentage (%), dollar ($) and ampersand (&)

The sets of constructs to be added are not exhaustive. There will be many more other constructs which could potentially be added as well, and the more commonly used constructs would take precedence in the order of my implementation in future.

## 4.3 Improving Word Corrector Module

As mentioned earlier, the Word Corrector module’s phase one currently uses a hardcoded algorithm to correct common errors, and we will improve on this aspect in the next semester. A hardcoded algorithm is not robust enough and is not easy for us to edit or extend the current state, as such a better algorithm is necessary to improve our module further. One possible way of doing so would be to make use of the word similarity heuristic, as presented in Word Corrector module’s phase two, to execute the correction. We could possibly use a list of keywords and check if the recognized word from the Speech Recognition module has a close similarity to any of the keywords in the list and perform the correction. In the next semester, I will look at the plausibility of employing such a method to perform the correction and also investigate and research for more alternative methods to improve the correction, to check if there is any better way of doing the correction.

## 4.4 Improving overall program and feedback

The current Coding by Dictation application is a simple program which converts voice input into program code, and it has very minimal User Interface (UI). These aspects would be worked upon in the next semester to give the user a better coding interface. Moreover, there is only a limited set of feedbacks given to the user according to what the user has input into the program. There is a need to give proper and more detailed feedback to the users, which will greatly enhance the coding efficiency of the programmer. If time permits, I will also look into allowing the user to edit the program as that is more similar to the thought process of a programmer. A programmer does not code by writing out a whole piece of code on paper first before typing it into the computer. The thought process would include going back in the middle of coding to declare a variable which is needed to be used at the current stage of coding, thus the need for user to add or edit code previously entered. This will create a more complete program which would resemble closely to how programmers would usually type code.

# 5 References

Begel, A. and Graham, S.L. (2005) Spoken Programs. University of California, Berkeley.

Begel, A. (2006) Spoken Language Support for Software Development. University of California, Berkeley.

Chew, Y.X. (2016). Hands Free Programming. National University of Singapore, 2016

Crouch, Tammy (1995). Carpal Tunnel Syndrome and Repetitive Stress Injuries: The Comprehensive Guide to Prevention, Treatment and Recovery. Frog, Ltd. Berkeley, California.

Gao, R.S. (2016). Talk-to-Code: From Structured Command to Source Code. National University of Singapore, 2016

McGuire, P. (2016). Pyparsing (Version 2.1.9) [Software]. Available from <http://pyparsing.wikispaces.com/Download+and+Installation>

Turk, J. and Stephens, M. (2016) Jellyfish (Version 0.5.6) [Software]. Available from <https://github.com/jamesturk/jellyfish>

Voicecode.io. (2017). Advanced Voice-Control, speech to code, program by voice, stop RSI. [Online] Available at: <https://voicecode.io/>

Zhang, A. (2017). Speech Recognition (Version 3.7) [Software]. Available from <https://github.com/Uberi/speech_recognition#readme>