Disability Assistive Programming:

Using Voice Input to Write Code

by

Hunter Lee

Honors Thesis Project

Appalachian State University

Submitted to the Department of Computer Science

and The Honors College

Bachelor of Science

May, 2019

Approved by:

_____

James B. Fenwick Jr, Ph.D, Thesis Director

_____

Richard E. Klima, Ph.D, Second Reader

_____

Alice A. McRae, Ph.D, Departmental Honors Director

_____

Jefford Vahlbusch, Ph.D, Dean, The Honors College

Abstract

In today's society, the development and integration rate of technology into everyday life is at an all time high. As this trend continues to grow, it is becoming an increasingly important and valued skill for people to be able to design, produce and modify computer code to suit their needs. More often than not, the people who could get the most out of writing and customizing their own code for interacting with the world are those who are physically the least able to do so. People with disabilities such as blindness, Parkinson's disease, Amyotrophic Lateral Sclerosis, or those who have been in severe accidents are often times left unable to utilize various computer functions. Whereas most of these functions have been adequately adapted to use things such as voice commands, the ability to write customized code has seen very little attention.

Providing coding accessibility to people with disabilities may seem like a single purpose application at first glance, however the possibilities this affords these users are endless. With the ability to program using voice recognition software, people who are unable to adequately interact with a keyboard or other input device will be able to write or customize programs to assist with their often unique personal needs. For instance, someone who is blind could program a series of sensors to help them navigate their house with ease or let them know when various issues arise through the use of customized alert software. Furthermore, amputees can create software that assists in various home automation functions specific to them, that would otherwise require the assistance of another person. In a society that is becoming ever more dependent on technology, ensuring that people with disabilities can

interact and help shape it is critical.

Although speech to text software already exists, using such software for coding would be inefficient and frustrating. Some speech to code software already exists however, the rhetoric used to control these programs is often far more complex than it need be. The overall vision of this project is to create a program that allows for quick and intuitive speech to code transcription for the Java programming language with significantly fewer words needing to be spoken. The purpose of this thesis is to lay the groundwork for this endeavor and to act as a proof of concept for future work. Specifically this thesis will develop the research knowledge on the aspects involved in creating a speech to code program and will provide the methodology used in the research and creation of a simple proof of concept program.

Acknowledgments

I have been incredibly fortunate to have such a strong support system to help me throughout my thesis work. First I would like to thank Dr. Fenwick, whose guidance and mentorship throughout this process helped lead me towards my goals. His willingness to support my endeavors and to learn alongside me over the past few months made this project possible. I would also like to thank the rest of my thesis committee for supporting and challenging my work to ensure it is at its best. Your contributions are greatly appreciated. Most importantly I would like to thank all of my friends and family for their never-ending love and support for which I am forever grateful. I have learned so many lessons from them, their patience and encouragement have enabled me to be where I am today.

Contents

List of Figures and Tables

Chapter 1

Introduction

In today's society, the development and integration rate of technology into everyday life is at an all time high. As this trend continues to grow, it is becoming an increasingly important and valued skill for people to be able to design, produce and modify computer code to suit their needs. More often than not, the people who could get the most out of writing and customizing their own code for interacting with the world are those who are physically the least able to do so. People with disabilities such as blindness, Parkinson's disease, Amyotrophic Lateral Sclerosis, or those who have been in severe accidents are often times left unable to utilize various computer functions. Whereas most of these functions have been adequately adapted to use things such as voice commands, the ability to write customized code has seen very little attention.

Providing coding accessibility to people with disabilities may seem like a single purpose application at first glance, however the possibilities this affords these users are endless. With the ability to program using voice recognition software, people who are unable to adequately interact with a keyboard or other input device will be able to write or customize programs to assist with their often unique personal needs. According to the Center for

Disability Rights, the number one unmet need that people with disabilities have is being able to live independently [1], which is something this software could help with. For instance, someone who is blind could program a series of sensors to help them navigate their house with ease or let them know when various issues arise through the use of customized alert software. Furthermore, amputees can create software that assists in various home automation functions specific to them, that would otherwise require the assistance of another person. In a society that is becoming ever more dependent on technology, ensuring that people with disabilities can interact with it is critical.

Although the implications of developing a system of this sort could have the most impact on the lives of people living with disabilities, it would also benefit the rest of the programming world as well. Many corporations and independent coders alike may eventually opt to start coding via speech recognition instead of manual keyboard input. In theory, developing software in this manner will inherently be much faster, being able to collect input at the rate of the spoken word instead of having to be thoughtfully typed out. This in turn means faster software development which, from a business standpoint, means more profit. Independent coders on the go could also begin developing code in transit between locations without having to carry around a heavy device to type on.

This thesis lays the groundwork for the development of a voice-enabled coding tool. Currently the proof of concept implementation is able to collect speech through the use of the Java Speech Recognition API and the open source Sphinx 4 Speech Recognition Library. It then processes the speech using a rudimentary language grammar which outputs to a .txt file. This file acts as the input into a custom JFlex lexical analyzer and CUP parser for a

functional subset of the Java programming language.

Chapter 2 delves deeper into the backgrounds of the Sphinx 4 Speech Recognition Library, JFlex lexical analyzer and CUP parser, and will also define much of the terminology that will be used throughout the essay. Chapter 3 explains the methodology used in creating, testing, and improving the functions of the speech to code program (STC). Chapter 4 briefly describes the outcome of this project and its capabilities as they are at the completion of the proof of concept STC program. Finally, Chapter 5 discusses the conclusions that can be drawn from the project as well as a plethora of future work ideas.

Chapter 2

Background

2.1 Terminology

This thesis uses a variety of high level and technical terms that may be unclear. To ensure that the discussion in the following sections is clear and understandable some key terms are defined below.

- **Black Box:** A device, system or object that is viewed in terms of its inputs and outputs without any knowledge of its internal workings.
- **Grammar:** A set of rules that defines the structure of a language; that is to say what can and cannot be in the language.
- **Hit Rate:** The rate at which the software gives the correct output for the given input.
- **Lexical Analyzer (Lexer):** A piece of software that breaks down input into a series of tokens that the parser can use to validate syntax.
- **Non-Terminal:** A symbol within a given grammar rule that can be replaced by the contents of another grammar rule.
- **Parser:** A piece of software that takes a list of input tokens and analyzes them to ensure

they conform to the rules of the given grammar.

- **Rule:** A definition for a non-terminal symbol; in other words, the symbols that can replace a non-terminal.
- **Terminal:** A token that can not be replaced by anything and only represents itself.
- **Utterance:** Any form of audio input.

2.2 Sphinx4 Speech Recognition Library

In order to translate speech to text, the audio of the speaker must be processed through a speech recognition library. The Sphinx 4 Speech Recognition Library was chosen to accomplish this task. Sphinx 4 is a flexible open source speech recognition framework developed at Carnegie Mellon University. This software was identified as an ideal speech recognition framework to use because it is already based in Java, a familiar and popular programming language, and utilizes the Java Speech Recognition API. As to not overcomplicate things, the Sphinx 4 Speech Recognition Software will be referenced as a black box that takes audio input and a grammar and gives back corresponding text translations as shown in Figure 1 below.

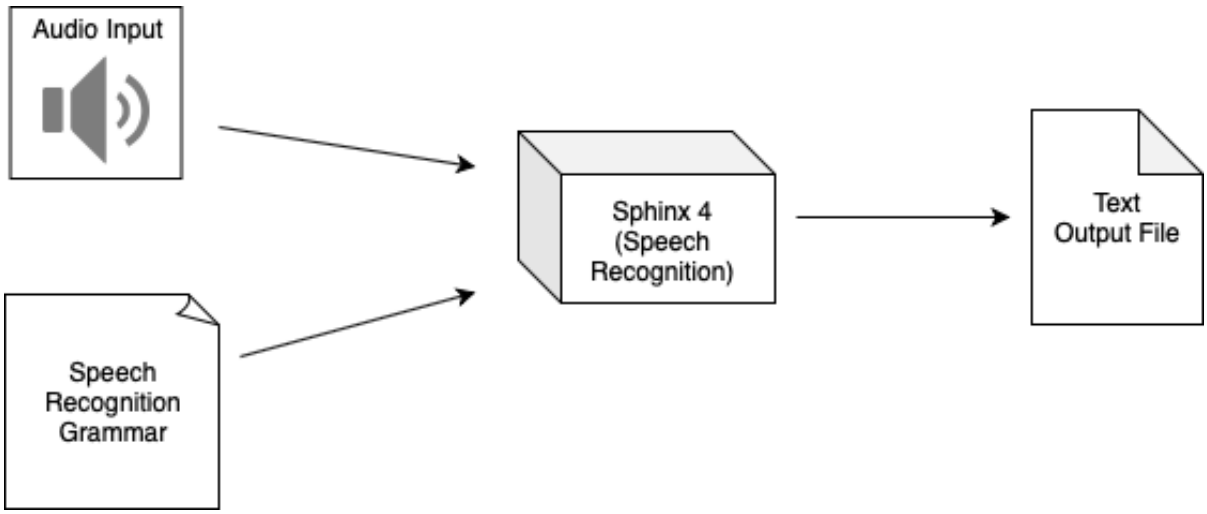


Figure 1: Visualization of the Sphinx4 black box

2.3 JFlex Lexical Analyzer and CUP Parser

To ensure that the given output from the STC program is of valid Java syntax, we process the output through a Java parser. The tool used for Java parsing is comprised of two parts: the JFlex lexical analyzer and the CUP parser. The lexer changes the text into a series of terminal and non-terminal symbols which it then passes to the parser. The parser in turn checks to see if the syntax for the input symbols matches a given symbol sequence in the parser grammar. The JFlex and CUP combination is ideal for this project because they are designed to be used for Java and are also written in Java. The JFlex/CUP tool suite is also used in the Programming Language Translation course at Appalachian State University.

Chapter 3

Methodology

There were two main threads of development for this project: the speech recognition software (SRS) and the lexer/parser software (LPS). Interestingly, these two developmental processes often mirrored each other despite being completely partitioned for the majority of the process. In the following chapter the development and testing processes for each of the components will be explained and discussed as they were created: individually at first and then brought together towards the end.

3.1 Speech Recognition Software

The basis for the speech recognition software came from open source Sphinx 4 demo code [5] that was then modified to fit the needs of the project. The majority of work in this area revolved around testing the software to find what its limits were and how to utilize it effectively. The following section describes the research and testing portions of the development process while the section thereafter describes the parameter augmentations that were chosen for maximum recognition accuracy.

3.1.1 Research and Testing

The first step in the process was to figure out how Sphinx 4 worked. This involved adding the Sphinx 4 libraries into Eclipse's resource directory to make the Sphinx 4 procedures accessible and then changing the build path to include the Sphinx 4 resources. For more detailed information on the configuration process see Appendix A.1.

After getting the Sphinx 4 software configured correctly, the testing process began. The default setup of the demo code runs the general Sphinx 4 library, meaning the software tries to match the audio utterances to any words that it has in its extensive database. This method proved to be fairly inaccurate with a hit rate of only 65%. Searching for a solution to this problem led to two possible methods.

The Sphinx 4 Frequently Asked Questions page contained a section suggesting the use of software called SphinxTrain [2]. SphinxTrain is a subset of the software that is used to train the Sphinx libraries to understand new words or pronunciations. The issue with this is that in order to train the software you have to go through a roughly ten minute training process in which you feed it audio and text so it can learn your specific style of speech. Although this may be necessary for adding non-standard word tokens in the future, such as args and other variable names, for the purposes of the STC program this option is impractical and time-consuming because each user has to train the Sphinx separately.

The other option proved to be a far more viable solution. This solution was to give the Sphinx software a custom grammar (or dictionary as the CMU documentation refers to it). This involved creating a list of acceptable words in grammar file. The words used to run the tests are depicted below.

```
grammar grammar;
public <testwords>  = (apple | book | cat | door | echo | fence | gorilla | happy | input| jester);
```

Figure 2: Example STC dictionary

Upon creation of the custom dictionary, tests were conducted to see if there was a noticeable difference in hit rate. Tests were performed in a quiet location with no interruptions. For the tests, four participants were asked to read the words normally two times, once utilizing the general Sphinx 4 grammar and once with the custom grammar provided above. The results are as follows:

Hit Rates of General and Custom Grammars in Sphinx 4

Participant Description	General Grammar Hit Rate	Custom Grammar Hit Rate	Overall Improvement in Hit Rate
Male, Soft-spoken	60%	95%	35%
Male, Loud	65%	95%	30%
Female, Soft-spoken	35%	80%	45%
Female, Loud	35%	80%	45%

The test concluded that the hit rate with a custom grammar is 38.75% more accurate on average than that of the general library grammar. It is interesting to note that the software has a more difficult time understanding female voices than it does male voices. Upon further research, this is a reoccurring issue among all voice recognition software and not specific to the Sphinx Suite [7]. The other unintended consequence of using a custom grammar to look up utterances is a recorded speed increase of roughly 200% which is only expected to decrease slightly as the custom dictionary gets larger.

3.1.2 Development

After the testing stage was completed, the development portion of the speech recognition software began. The demo code sufficed for capturing audio utterances and handling the bulk of the processing. The major addition required was a custom grammar to run audio input through. By narrowing down the program specifications to include only the standard “Hello World” example code, the size of the grammar is equally succinct. The “Hello World” grammar is as follows:

```
 7 grammar grammar;
 8
 9 public <access>  = ( public | private );
10 public <name>  = ( class | hello world )+;
11 public <methods>  = ( main method );
12 public <quote>  = ( quote );
13 public <brace> = (left brace | right brace);
14 public <parenthesis> = (left parenthesis | right parenthesis);
15 public <brackets> = (left bracket | right bracket);
16 public <main> = (static | void | main | string | arguments );
17 public <methodBody>  = (print line);
18 public <termination> = (end | semicolon);
```

Figure 3: Example word categories for a custom Sphinx 4 grammar

The access category named <access> on line 9 will be used as an example. The category has two options for utterances, *public* and *private,* and follows the standard format for a line in the grammar:

public <name_of_category> = (*word* | *word*);

A specific concept that should be noted is that some utterances require multiple words to be spoken at one time. The <brace> category on line 13 for example requires the user to say the words “left” and “brace” without pausing between utterances. Failure to follow the word “left” with the word “brace” to create a complete <brace> utterance will cause the program to either force the utterance to match a different single-word category or default to the first multi-word category beginning with a matching utterance.

The name of the category is optional but is helpful when attempting to dictate a “soft” parser for the grammar. It is referenced as a soft parser because it only suggests what it wants to hear but will also comply if the input utterances are drastically different. Defining named categories allows us to create this “soft” parser by combining the categories into rules such as in Figure 4 below.

```
public <syntax> = <access>{1} <name>{2} <brace>{2};
```

Figure 4: Example grammar rule for custom Sphinx 4 grammar

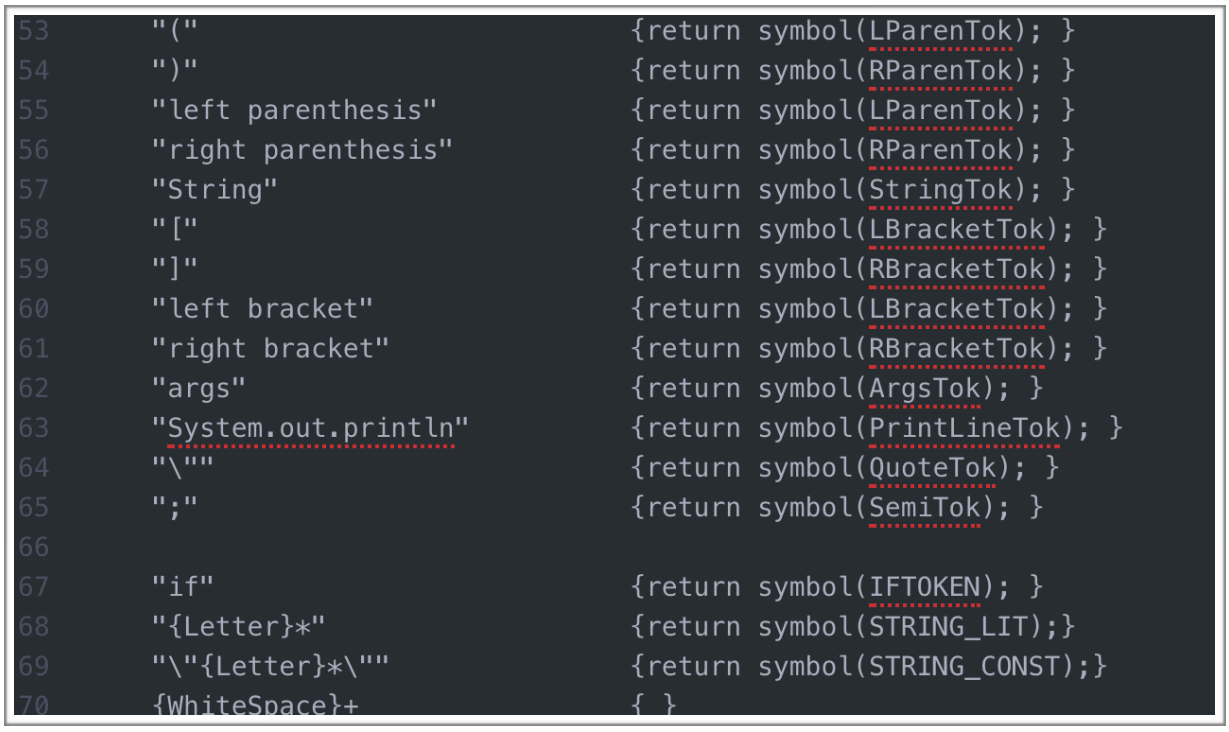
The addition of something such as the rule in Figure 4 would try to match the audio utterances of an access word followed by two name words followed by two brace words. In the future this system could be used to increase the hit rate and speed of audio input. However it comes at the cost of reduced flexibility as it bears the issue of the software trying to force its input into a singular format.

3.2 Lexer/Parser Software

During the development of the speech recognition software, the LPS was being developed simultaneously. This involved less research but a more extensive amount of coding and debugging. The following sections will discuss the creation of the lexer and the parser grammar.

3.2.1 Lexer Development

The job of the lexer is to take in a sequence of characters generated by the STC program, analyze them and return the corresponding set of output tokens. Lexer implementation is based on a finite state machine but tools like JFlex can simplify this task. Although using JFlex is typically a relatively straight forward process there is a specific design choice that needs more explanation. Figure 5 shows the bulk of the word to token association part of the lexer and will be the point of discussion for the remainder of the section.



```
53      "("                         {return symbol(LParenTok); }
54      ")"                         {return symbol(RParenTok); }
55      "left parenthesis"          {return symbol(LParenTok); }
56      "right parenthesis"         {return symbol(RParenTok); }
57      "String"                    {return symbol(StringTok); }
58      "["                         {return symbol(LBracketTok); }
59      "]"                         {return symbol(RBracketTok); }
60      "left bracket"              {return symbol(LBracketTok); }
61      "right bracket"             {return symbol(RBracketTok); }
62      "args"                      {return symbol(ArgsTok); }
63      "System.out.println"        {return symbol(PrintLineTok); }
64      "\""                        {return symbol(QuoteTok); }
65      ";"                         {return symbol(SemiTok); }
66
67      "if"                        {return symbol(IFTOKEN); }
68      "{Letter}*"                 {return symbol(STRING_LIT);}
69      "\"{Letter}*\""             {return symbol(STRING_CONST);}
70      {WhiteSpace}+               { }
```

Figure 5: Example section from the implemented JFlex Lexer

In particular the concept that may raise questions is the repetition on lines 53 and 55 of Figure 5, as well as similar redundancies with subsequent tokens. The purpose behind this is to ensure that the flexibility of the speech recognition software isn't compromised. For instance, during future modifications functionality can be added to automatically take the words "left parenthesis" and map them to the character "(" in the STC program before sending it to the lexer. Notice that the current definitions on lines 53 and 55 already account for this future change by mapping the input "(" and the multi-word sequence "left parenthesis" to the same *LParenTok* token. This design choice also allows for the LSP text input to come from either a speech-based programming system or a standard keyboard input system which is used to verify the LSP works as intended.

3.2.2 Parser Development

Development of the CUP parser consisted of creating a subset of the Java grammar that understands the requirements of the basic “Hello World” program. This involved creating a set of terminals, a set of non-terminals, and a set of grammar rules. Figures 6 and 7 below will show sections of each of the aforementioned steps as an example of their formatting.

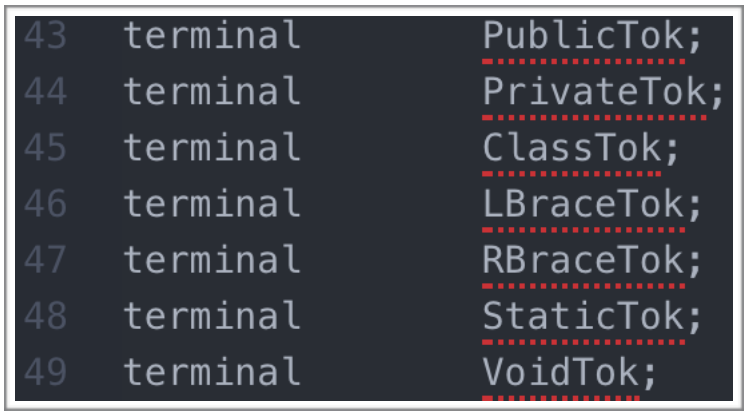


Figure 6: Depiction of CUP terminal formatting

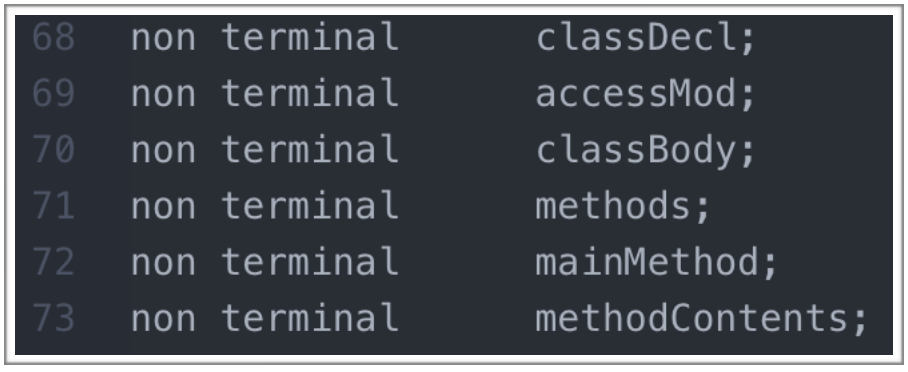


Figure 7: Depiction of CUP non-terminal formatting

```
76  non terminal Program;
77
78  Program     ::= classDecl
79                      {:System.out.println("Program Successfully Parsed");:}
80                  | STRING_LIT
81  ;
82
83  classDecl   ::= accessMod ClassTok Name LBraceTok classBody RBraceTok
84                      {:System.out.println("classDecl is good");:}
85  ;
86
87  accessMod   ::= PublicTok
88                      {:System.out.println("accessMod is good");:}
89                | PrivateTok
90                      {:System.out.println("accessMod is good");:}
91  ;
92
93  classBody   ::= methods
94                      {:System.out.println("classBody is good");:}
95  ;
96
97  methods   ::= mainMethod
98                      {:System.out.println("methods is good");:}
99  ;
```

Figure 8: Depiction of CUP grammar rules for Java Hello World program

The code above in Figure 8 demonstrates how the parser matches a sequence of terminals and non-terminals to their respective tokens or rules. It is important to note that by utilizing CUP for parsing, small snippets of code can be executed at the end of each non-terminal rule. This can be seen in line 79 of Figure 8 as the Java code imbedded in matching {: and :} symbols.

At the top of Figure 8, the “program” is defined to be a single *classDecl*. If CUP is able to satisfy this definition, then it will execute the subsequent code to print the success method. To see if CUP can satisfy the *classDecl* requirement, it must look into the *classDecl*

rule. Once CUP begins processing through the rule it will try to match the contents of the *accesMod* rule, a ClassTok, a Name token, an LBraceTok, the contents of a *classBody* rule and a RBraceTok. To match this it will have to process through the *accesMod* and *classBody* rules and so on until all tokens are matched or an error is returned.

Chapter 4

Results

The combination of the SRS and LPS developmental processes results in a project that can take audio input, interpret the audio with an accuracy ranging between 80% - 95%, and output the interpretation as a .txt file filled with spoken utterances. This output is then subsequently processed through the JFlex/Cup software to ensure that it adheres to Java syntax requirements. The JFlex/CUP system currently supports a subset of the Java programming language that allows for a fully functional “Hello World” program. Figure 9 depicts a visual representation of this process.

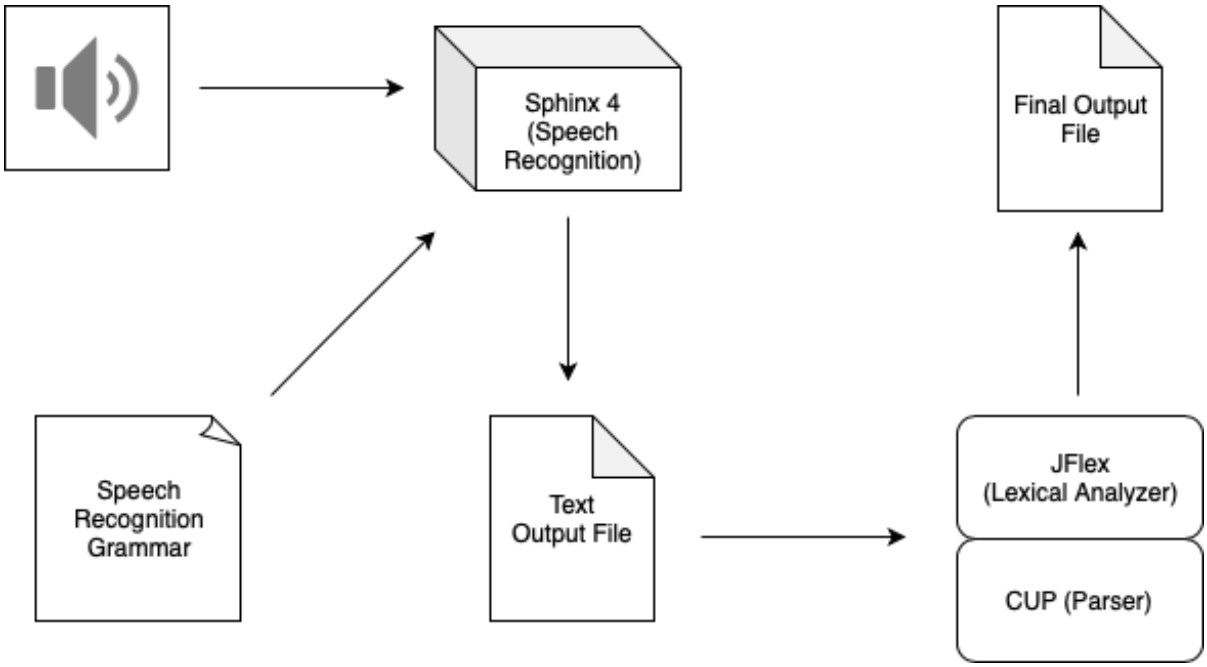


Figure 9: Visual representation of the full STC program process

Currently, the JFlex/CUP output simply indicates whether or not the given input from the SRS is valid for the Java programming language, however it does not write the program out for the user. This step was determined outside of the scope of a proof of concept project as ensuring the input is gathered and parsed correctly automatically ensures the integrity of the code. From that point the original text file would simply be read utterance for utterance and mapped to its respective symbol if needed.

Chapter 5

Conclusion and Future Work

The proof of concept project for a speech to code program is a success. The program successfully takes in spoken utterances as audio input, transcribes it to text accurately, and validates its syntax for a basic Java program. With the groundwork being laid for a more full-featured STC program spanning the entirety of the Java programming language is possible. Future work can involve extensive broadening of the grammars for both the speech recognition software and the lexer/parser software to accomplish this goal. It is also possible to automate the flow of data between the SRS and the LPS instead of needing to run it manually. Additional features to increase functionality include:

- Code navigation

  The ability to navigate through written code to quickly view, augment, and maintain code. For instance, if an utterance is misheard, after parsing the user could verbally navigate through the new code using keywords like “Up”, “Down”, or “Search” to find and fix errors.

- Computer generated voice prompting for inputs

  Once more extensive speech recognition dictionaries are in place, this would add the

ability for the program to suggest what it needs to hear from the user next.

- Staggered rejection and/or clarification requesting for bad input

  Given unknown or incorrect input, the speech recognition program could halt additional input and ask for clarification.

- Auto-completion and formatting of common structures

  The ability for the program to take in partial commands such as “print” and return the correctly formatted “System.out.println(…);.

Bibliography

[1]A. Pulrang. “Top 5 Disability Issues.” Internet: http://cdrnys.org/blog/disability-politics/top-5-disability-issues/, [Jan 16, 2019].

[2]Carnegie Mellon University. “Sphinx-4 Frequently Asked Questions.” Internet: http://my.fit.edu/~vkepuska/ece5527/Projects/Fall2011/Burgos,%20Wilson/sphinx4-1.0beta6/sphinx4-1.0beta6/doc/Sphinx4-faq.html, Mar 6, 2010 [Dec 15, 2018]

[3]CMUSphinx. “Frequently Asked Questions (FAQ).” Internet: https://cmusphinx.github.io/wiki/faq/, [Dec 15, 2018].

[4]C. O’Donovan. “Changing Grammar of LiveSpeechRecognizer at Runtime.” Internet: https://sourceforge.net/p/cmusphinx/discussion/sphinx4/thread/6afa2348/, Sept 26, 2017 [Dec 15, 2018]

[5]goxr3plus. “Java Speech Recognizer Calculator.” Internet: https://github.com/goxr3plus/Java-Speech-Recognizer-Tutorial--Calculator, Apr 23, 2017 [Dec 15, 2018]

[6]N. Shmrev. “How to Reduce Noise in Sphinx4 Application.” Internet: https://stackoverflow.com/questions/29059287/how-to-reduce-noise-in-sphinx4-application, Mar 15, 2015 [Dec 15, 2018].

[7]R. Tatman. “Google’s speech recognition has a gender bias.” Internet: https://makingnoiseandhearingthings.com/2016/07/12/googles-speech-recognition-has-a-gender-bias/, July 12, 2016 [Jan 16 2019].

Appendix A

-Link to a helpful Sphinx4 setup guide: https://youtu.be/R8vsXKFTee0

-Link to the full Github Repository: https://github.com/hunterlee52/Speech-to-Code-Proof-of-Concept/

-Link to Demo Video: https://www.youtube.com/watch?v=5WLa-5m84Tc