

CM3203 Final Report

Majel: Voice Assistant Style Interface
For Command Line Terminal

Abstract

Traditionally, the only way for a user to interact with a command line interface on a computer is via typing commands with a keyboard. This has issues ranging from simple efficiency to accessibility concerns. In this project, I will aim to resolve these issues by implementing an alternative way of interacting with a CLI via voice control. To this end, I will also attempt to detail a grammar structure that will process any possible command, and implement a system to tailor the grammar to the user's system. I will attempt to develop a 'voice assistant' style interface that will be able to run any possible typed command as a spoken one. The project presents a working 'proof of concept' program that will act as a platform for further development.

Acknowledgements

Prof. David Marshall for his support throughout as project supervisor.

Table of Contents

| | |
|------------------------------------|-----------|
| Abstract | 1 |
| Acknowledgements | 1 |
| Table of Contents | 2 |
| 1. Introduction | 4 |
| CLI Issues | 4 |
| SR Issues | 5 |
| Aims and Objectives | 6 |
| 2. Background | 7 |
| Command Line Interface | 7 |
| Speech Recognition and CMU Sphinx | 8 |
| JSGF Grammar | 10 |
| Python | 11 |
| 3. Approach | 12 |
| Basic Speech to Text | 12 |
| Getting Microphone Input | 14 |
| A CLI Language Model | 15 |
| Grammar Structure for CLI | 18 |
| Populating Dynamic Grammars | 19 |
| Running Commands | 19 |
| 4. Implementation | 20 |
| Backend | 21 |
| setup_dict_grammar() | 21 |
| get_dictionary() | 25 |
| create_grammar() | 26 |
| add_to_grammar() | 27 |
| update_folder_grammar_dictionary() | 28 |
| Frontend | 29 |
| main() | 29 |
| get_command() | 31 |

| | |
|---|-----------|
| word_to_character() | 32 |
| run_command() | 35 |
| change_directory() | 36 |
| Grammar Structure | 38 |
| 5. Results and Evaluation | 40 |
| Unit Testing | 40 |
| Unit Test 1 - Initialization | 40 |
| Unit Test 2 - Running Typed Commands | 43 |
| Unit Test 3 - Running Spoken Commands | 44 |
| Unit Test 4 - Updating Dynamic Grammar Files and Phonetic Dictionary File | 45 |
| Unit Test 5 - Adding Aliases | 48 |
| Unit Test 6 - Applying Modifiers | 50 |
| Unit Test 7 - Handling 'Dangerous' Commands | 51 |
| Unit Test 8 - Overriding Pronunciation | 52 |
| Usability Testing | 55 |
| Conclusion | 57 |
| 6. Further Work | 58 |
| 7. Reflections on Learning | 60 |
| References | 61 |

1. Introduction

There are two disparate concepts that I am attempting to bring together in this project; the command line shell of a Linux operating system and speech recognition.

Command Line Input (CLI) is among the oldest methods of human computer interaction. The user types commands on a keyboard which are then interpreted by the computer, running programs with parameters as specified. The most common form of CLI today is the command shell, a means of interacting with the files and programs on a computer. These exist in all popular desktop operating systems, including Windows, OSX and various Linux distributions. Given their relative complexity and difficulty to use, command shells are not typically the default means of interacting with an operating system in lieu of a graphical desktop interface.

Speech Recognition (SR) describes the process of converting audio data of speech into a human and computer readable format, normally text. This is usually a nontrivial, computationally complex process, with several different methods available. While it was once considered an obscure and unwieldy form of HCI, it has become commonplace in recent years due to the popularity of 'voice assistant' interfaces on smartphones and Internet of Things speaker devices.

CLI Issues

In order to use a CLI, the user must be able to use a keyboard to enter commands. This means that users with certain disabilities that hinder or prevent the use of a keyboard will struggle to use CLI. Solutions to this problem such as virtual keyboards tend to be slow and imprecise. A speech based input for CLI as I am proposing would be closer in speed and efficiency to standard typing.

Another issue with CLI is that it requires the user to learn what exactly each command does and how to properly use them, meaning that the barrier to entry for new users of a CLI is high compared to a GUI interface. This is particularly true of multiparameter, complex commands, for example compiling a C program with the GNU C Compiler, or use of the source control system Git via CLI. With my proposed speech based input system, the user will be able to set up alias' for commands that are more explicit in their function such as 'change directory' for 'cd' and 'execute filename' for './filename'.

Since CLI requires the user to type commands, use of them can take the user's focus away from some other task to look at the terminal window. With a speech based input, the user will be able to stay focused on another task while they are inputting a command using their voice, increasing the efficiency of their computer usage.

SR Issues

In most forms of SR that are encountered on a daily basis, the transcription of the audio data is done 'online' and not on the device that recorded the data. While this is efficient for small low specification devices such as phones or IoT speakers, it requires a stable internet connection to work. For my program, I plan to instead do the transcription locally, and only use an internet connection to get pronunciations of new words.

Another issue faced by common SR systems is that they must be able to process almost any possible input from the entirety of the language; by massively limiting the possible set of words to be recognised, the speed of transcription should be increased.

Possible Applications and Uses

The simplest application of combining command line and speech recognition technology is in a voice assistant analogue to launch programs; for example speaking the command 'firefox' to launch the eponymous web browser. Another simple application would be for a system administrator who must perform a set routine of tasks on a regular basis could set up a number of alias commands such as 'copy log files' or 'make backups'; actively speaking these commands aloud would aid in maintaining the routine, and also allowing them to focus on another task. Speech commands could also help designing interfaces for useful programs that are inaccessible due to their command line nature; for example speaking the command 'execute my program' is much more intuitive and understandable than the typed command './myprogram'. A good example of this would be the command line interface for the source control software 'git'; saying 'get the newest updates' is much more descriptive than 'git pull origin master'. Use of SR would also allow for executing CLI commands in hardware setups that do not allow for a keyboard such as IoT devices.

There are serious limitations to the application of SR; even with noise cancellation, it is much less effective in noisy environments, and is in some cases rendered unusable. The higher potential of error when compared with a standard typed input also means

that SR might not be desirable in critical systems, for example a system interpreting a command to 'move' a file as 'remove' could have a catastrophic impact. These issues could be minimised via use of high quality input devices (microphones), but the cost of purchase and installation of such devices may outweigh the potential benefit.

Aims and Objectives

In this section, I will list the aims and objectives of the project, in terms of features I wish to implement.

- Allow for the successful execution of any program installed on the system, given that it's name, arguments and user defined inputs are said correctly.
- Majel will be able to be launched via a 'wake word' at any time.
- Majel will work entirely offline, with no calls to online APIs for recognition.
- Some form of natural language interface will be implemented, with the user not having to explicitly state the name of the program, only what they want to have happen.
- Processing of input will be very responsive, with the command being executed as quickly as possible.
- The input will be checked and the user prompted if the input command could have been misinterpreted, especially when moving, copying and deleting files, or other critical commands.
- Majel itself will have a command line interface that will mimic the functionality of other shell interfaces (BASH, fish ect).
- Users will have the option of extending the set of commands Majel will recognise by importing their command line history.
- The system will use calls to the web based Sphinx dictionary creation service to obtain pronunciation data about new words it encounters.
- Users will be able to override the pronunciation of words with their own.
- Majel will be lightweight, requiring only the installation of a small number of Python packages to function on a standard Ubuntu based distribution.

2. Background

Command Line Interface

Command Line Interface (CLI) evolved from teleprinter (TTY) machines, where instead of messages being communicated between two people in TTY, CLI commands are communicated from a person to a computer. They predate GUI interfaces and provide a more powerful and efficient way of interacting with a computer. In a CLI, the user types a command and then submits it, where it is interpreted by the computer and executed. Any output from the command is usually printed to the terminal interface that the user typed the command to.

A popular command line shell for Linux operating systems is the *Bourne-again shell* or BASH shell[1]. A simple command in BASH is defined as “a sequence of words separated by blanks, terminated by one of the shell’s control operators” [2]. The general anatomy of a simple command is the following:

```
Prompt command arg1 arg2 arg3 ... argN
```

Where:

- Prompt is information provided by the shell program that provides information such as the current working directory within the file system, the username of the current user and the host name of the machine.
- Command is provided by the user and refers to a program installed on the machine or a command ‘built in’ to the shell.
- Arg1 ... argN are arguments provided to the command that inform it’s execution, such as files or folders to be run on or additional instructions that are specific to the command being run.

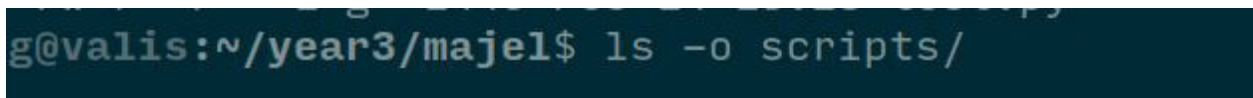


Figure 2.1 - Screenshot of the BASH shell showing the Prompt, Command and two Arguments

In the above example (Figure 2.1), the Prompt displays the user 'g' on machine 'valis' is in the working directory '~/year3/majel'. The Command for the program 'ls' is being executed with the two arguments '-o' changing how the program operates and the directory 'scripts/' as it's target.

Speech Recognition and CMU Sphinx

Speech Recognition (SR) is the conversion of spoken speech into a text string that can be interpreted by a computer. It has a long history with the earliest advances made in the 1960s. It's most apparent and popular use currently is in 'virtual assistant' products such as Apple's Siri, Amazon's Alexa and Google's Google Assistant. It's other main use is in out load dictation of text. While the earliest models required 'training' on a particular voice in order to function, modern systems use hidden Markov acoustic models and n-gram statistical models to match the input sounds to likely words. CMU Sphinx [3] is one such system, and it's Python implementation 'PocketSphinx' is used by this project.

CMU Sphinx is a freely available speech recognition system created at Carnegie Mellon University. It exists in several iterations, but the version used for this project is 'Pocketsphinx' a "lightweight recognizer library written in C" [4] that has a Python interface [5]. It is recommended by the developer in situations where "speed or portability" [6] are required such as this project. The following section is adapted from the Sphinx documentation[7].

Sphinx uses a standard structure of speech that consists of the following:

- Phones - "more or less similar class of sounds"
- Diphones - "parts of phones between two consecutive phones"
- Triphones,Quinphones - "phones in context" that 'describe slightly different sounds"
- Senones- "short sound detectors" used "to compose detectors for triphones"
- Subwords - syllable-like "reduction-stable entities" that remain the same even when speech becomes fast.
- Words - formed of subwords. They "restrict combinations of phones"
- Utterances - formed of words and fillers ("um", "uhhh", breathing, cough). nThey are not necessarily the same as sentences.

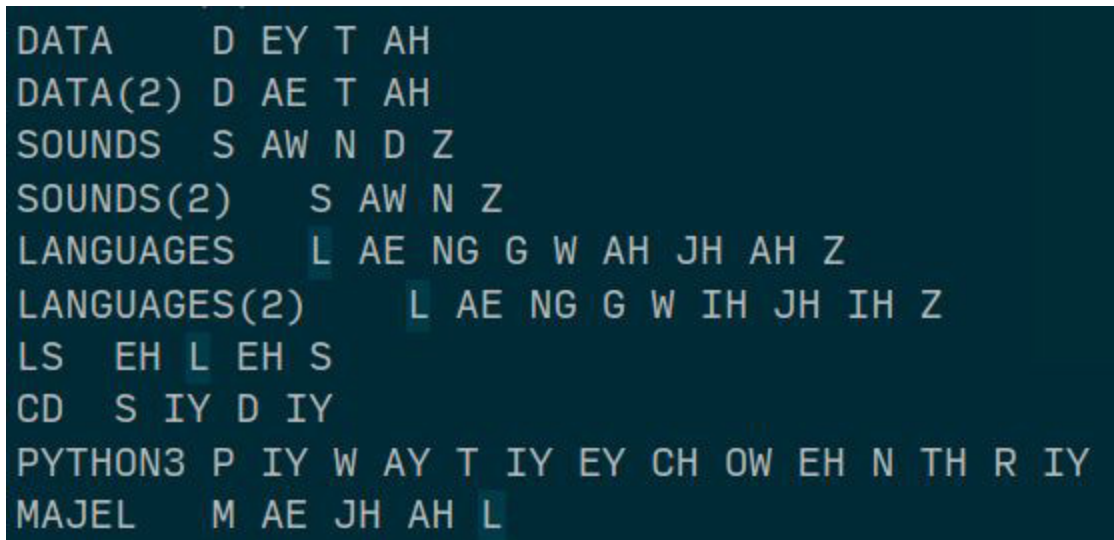
The recognition process used by Sphinx involves two key concepts, features and models. Features are an optimisation, defined in the Sphinx documentation as: “Numbers that are calculated from speech usually by dividing the speech into frames. Then for each frame, typically of 10 milliseconds length, we extract 39 numbers that represent the speech. That’s called a feature vector.”

A model “describes some mathematical object that gathers common attributes of the spoken word”. The model of speech used by Sphinx is called a Hidden Markov Model or HMM. In this “(a) process is described as a sequence of states which change each other with a certain probability”.

Other models used by Sphinx are:

- Acoustic model - “contains acoustic properties for each senone.”
- Phonetic Dictionary - “contains a mapping from words to phones” (Figure 2.2)
- Language model - “used to restrict word search. It defines which word could follow previously recognized words (remember that matching is a sequential process) and helps to significantly restrict the matching process by stripping words that are not probable.” This can be generated using a large number of sentences containing the words to be recognised.

NOTE: A keyword list or a grammar can be used instead of a language model - the final version of this project uses a grammar instead, for reasons that will be detailed later.



```
DATA      D EY T AH
DATA(2)   D AE T AH
SOUNDS    S AW N D Z
SOUNDS(2)  S AW N Z
LANGUAGES  L AE NG G W AH JH AH Z
LANGUAGES(2) L AE NG G W IH JH IH Z
LS        EH L EH S
CD        S IY D IY
PYTHON3   P IY W AY T IY EY CH OW EH N TH R IY
MAJEL     M AE JH AH L
```

Figure 2.2 - Screenshot excerpt from the Phonetic Dictionary used by Majel. Note that for some words many possible phone combinations are listed.

JSGF Grammar

Instead of a language model, a grammar file can be used to restrict word search. Sphinx supports the Java Speech Grammar Format (JSGF)[8] for this purpose. This format adapts some of the conventions of the Java programming language such explicit public/private declarations and import rules.

The vertical bar '|' means that the rule is defined by a set of alternatives. For example the following JSGF grammar:

```
#JSGF V1.0;

grammar planet;

public <planetname> = Earth | Mars | Jupiter | Venus;
```

Would generate the following possible phrases only:

- "Earth"
- "Mars"
- "Jupiter"
- "Venus"

An important property of JSGF that is used extensively by this project is the ability to import rules from other grammars. For example, assuming the above grammar was defined in a file called "planets.gram" (in the same directory) we could define another grammar as :

```
#JSGF V1.0;
Grammar greet;
import <planets.gram>
Public <greet> = Hello[ <planet.planetname>]! We come in peace!
```

Which would generate the following sentences only:

- “Hello! We come in peace!”
- “Hello Earth! We come in peace!”
- “Hello Mars! We come in peace!”
- “Hello Jupiter! We come in peace!”
- “Hello Venus! We come in peace!”

Note that the square brackets means that the content is optional, so no planet name appears in the first generated sentence.

A key step in the development of this project is the creation of an unchanging ‘master’ grammar that represents any possible command. This master grammar will import dynamic, changing grammars that define programs, files names, file extensions and aliases, allowing for the system to be able to process a changing set of possible commands.

Python

I will use the Python programming language version 3.6 to implement the project. I have chosen this language since I am experienced in it and there exists many helpful tools for implementing speech recognition available for it. These tools include some non standard modules, available from Python's package manager 'pip':

- **SpeechRecognition**[11] - This library provides an easy to use wrappers for a number of speech recognition services, including PocketSphinx, as well as for recording and accessing audio files using Pyaudio. It performs the former function via 'Recogniser' objects and the latter via 'Microphone' and 'Audio'. It vastly simplifies the recognition process, and allows for the transcription result to be returned as a string. It requires the following two modules to do this.
- **Pocketsphinx**[9] - This is the python interface to the C based PocketSphinx implementation. PocketSphinx is a cut down 'lightweight' version of CMU Sphinx, that is "specifically tuned for handheld and mobile devices, though it works equally well on the desktop". It handles the transcription of speech from audio data, using a phonetic dictionary, an acoustic model and a language model or grammar.
- **Pyaudio**[10] - This provides python bindings for PortAudio, an audio I/O library. It handles the recording and playback of audio on a variety of platforms.
- **Pyjskf** [12] - This is a JSKF compiler, matcher and parser for Python. Used to create and append the 'dynamic' grammars mentioned above. It can read and write grammars from files, as well as add rules and match strings to the grammar.

All of the modules listed here are platform independent, meaning that the program should work on most modern Linux distributions. Additionally, it would not be difficult to port the system to another OS, as long as they are compatible with Python.

3. Approach

Basic Speech to Text

The first step I took to implement the project was to simply take spoken English from an audio file and print the content that is spoken in the file to the screen as text. I sourced several kinds of speech files, including excerpts from audiobooks and speeches. I used the 'en-US' language package that is provided with Pocketsphinx python; this package includes an acoustic model, phonetic dictionary and language model for American English. I wrote a simple python program to do this, 'speech_test.py'.

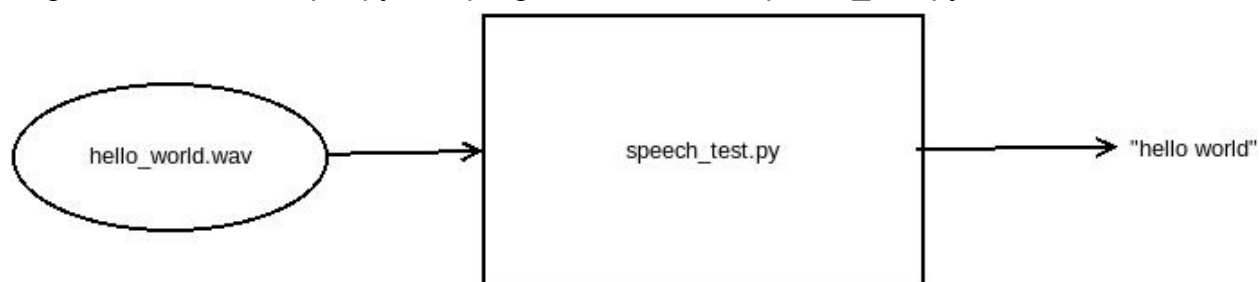


Figure 3.1 - Diagram showing the input and output of the speech_test.py program

| File Description | Actual File Contents | Transcribed by speech_test.py |
|--|---|---|
| <i>Extract of low quality radio audio, transmitted from the surface of the moon.</i> | "That's one small step for man, one giant leap for mankind" | "have you all and to a boil with on" |
| <i>Extract of medium quality audio, from a freely available audiobook.</i> | "There is a spectre haunting Europe, the spectre of Communism" | "most bacteria scalding europe the specter of communism" |
| <i>Extract of high quality audio, from a freely available audiobook.</i> | "Count Dracula had directed me to go to the Golden Krone Hotel, which I found, to my great delight" | "count dracula had directed me to go to the golden crown hotel which i found to my great delight" |

Table 3.1 - Results of transcription of audio files of different qualities.

As shown in Table 3.1, the accuracy of the transcription was dependent largely on the quality of the speech in the audio file. In the lowest quality audio, only the general 'shape' of what was spoken was detected, with none of the actual words being properly transcribed. The medium quality file gave better results, with the last few words of the file being transcribed correctly, albeit with a different spelling from the actual text. In the highest quality audio file there was a difference of only a single word that can be explained by the 'en-US' language package not containing the uncommon word 'krone'.

I also found that the accuracy of transcription was correlated with the length of the audio input. In some cases, limiting the transcription to the first few seconds of the file would increase the accuracy, in other cases it appeared that cutting the end of a sentence reduced the accuracy.

I also learnt that the transcription would not consider semantic elements of the speech such as the end of sentences. This has an adverse effect on the transcription, where it seemed that the last word of a sentence would sometimes affect the transcription of the first word of the following. This was likely not to be an issue for this project, given the lack of semantic elements in a CLI input.

Getting Microphone Input

The next step was to devise a method of collecting spoken word data from the user. The Speech Recognition Python module includes a 'Microphone' object that uses the default microphone for the system. This object has a 'listen' method that can store audio input. I modified the `speech_test.py` program to source the audio from this object rather than the file input, and then ran the program speaking the following phrases, using the inbuilt laptop microphone and a higher quality headset microphone.

| Spoken Phrase | Transcription with laptop microphone | Transcription with headset microphone |
|--|--|---|
| "Hello world" | "Hello wolf" | "Hello world" |
| "The quick brown fox jumped over the lazy dog" | "a quick run folks jumped of lazy dog" | "brown at fox jumped over the lazy dog" |
| "Space the final frontier; these are the voyages of the starship enterprise" | "face the final frontier these villages starship enterprise" | "the space the final frontier and these avoidance of the starship enterprise" |

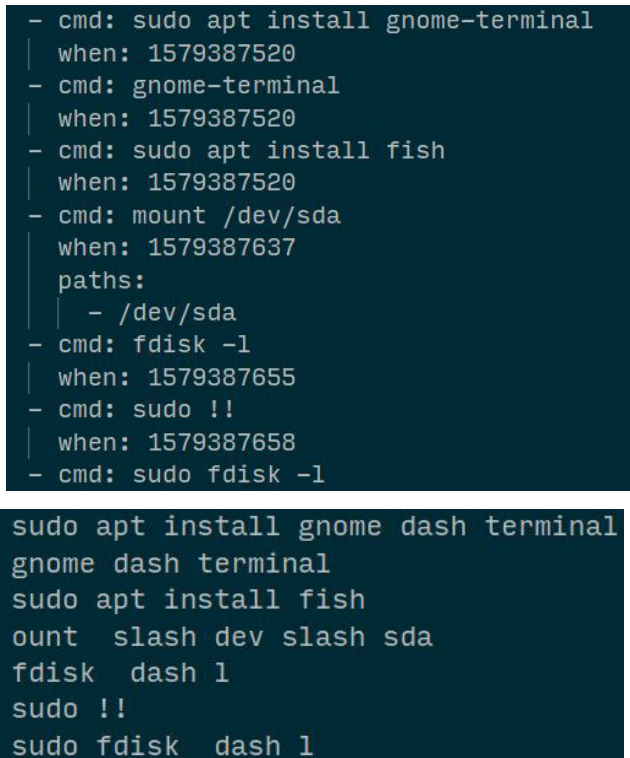
Table 3.2 - Results of transcription using different input methods of different phrases

As with the audio files, the results in Table 3.2 show that quality of the recording affects the accuracy of transcription. While the transcription using the headset was not perfect, it did recognise the majority of the words being spoken (even if they were transcribed incorrectly), unlike the laptop mic which seems to skip some sections.

Using the inbuilt microphone on my laptop in a busy lab resulted in the system being unable to distinguish the speech input at all. I found that the best results were when using the high end headset microphone in an otherwise silent room.

A CLI Language Model

The next step was to create a language package for CLI commands; this would limit the possible input of data to only possible command inputs. Following the documentation for Sphinx[12] I would use a web service [13] to create the phonetic dictionary and language model, but would use the same acoustic model as 'en-US'. This is because the acoustic model only describes how to interpret the spoken 'sounds' which will be the same for the CLI inputs as for English. In order for the web service to work it needs a 'corpus' of 'sentences' to generate the language model and phonetic dictionary. To get this list, I wrote a python script, "clean_fish.py" that takes a list of CLI inputs (for example a shell history file, Figure 3.2) and returns a suitable corpus file(Figure 3.3). Part of this program converts special characters in the CLI inputs ("/","-","|") into how they would be spoken aloud ("slash","dash","pipe").



The figure consists of two screenshots of text. The top screenshot shows a shell history file with entries for installing gnome-terminal, fish, mounting /dev/sda, and running fdisk. The bottom screenshot shows the output of a script that cleans these entries, replacing special characters with their spoken equivalents (e.g., 'slash', 'dash', 'pipe').

```
- cmd: sudo apt install gnome-terminal
| when: 1579387520
- cmd: gnome-terminal
| when: 1579387520
- cmd: sudo apt install fish
| when: 1579387520
- cmd: mount /dev/sda
| when: 1579387637
| paths:
|   - /dev/sda
- cmd: fdisk -l
| when: 1579387655
- cmd: sudo !!
| when: 1579387658
- cmd: sudo fdisk -l
```



```
sudo apt install gnome dash terminal
gnome dash terminal
sudo apt install fish
ount slash dev slash sda
fdisk dash l
sudo !!
sudo fdisk dash l
```

Figure 3.2, Figure 3.3 - Screenshots from the shell history file and the resultant corpus file.

The web service returns two files, one a '.dic' that contains the phonetic dictionary and the other a ".lm" (Figure 3.4) that contains the language model.

```
-0.8986 </s> -0.3010  
-0.8986 <s> -0.2077  
-2.5888 A -0.2424  
-2.5888 ADD -0.2780  
-2.2878 APT -0.2214  
-2.5888 B -0.2827  
-2.5888 C -0.2424  
-2.5888 CAPITAL -0.2988  
-2.5888 COMMIT -0.2424  
-2.5888 CP -0.2780  
-2.5888 CURL -0.2780  
-1.3847 DASH -0.2816  
-2.5888 DIFF -0.2780  
-2.2878 F -0.2424  
-1.2878 FILENAME -0.2160  
-2.5888 FIND -0.2780  
-2.1117 FIREFOX -0.1936
```

Figure 3.4 - Screenshot excerpt from a language model file generated by the online tool.

The language model approach was not at all as successful as I had imagined. It appears that the structural difference between the ‘real’ language expected by the web service, and the ‘language’ of the commands is too large. Additionally, the language model made no distinction between the various component parts (program name, file or folder paths, ect). This meant that the accuracy of the transcription was very low. Transcriptions would commonly contain either none or several program names, for example “sudo ls” being transcribed as “ls ls ls”. It was clear at this point that a more structured method was required.

The first solution I devised was splitting the input of the command into three types of input, using three different language models (Figure 3.5) . The first input would take the command followed by the user saying “argument” or “filename” where those components would be in their command. The program would then prompt the user for exactly those arguments and filenames and substitute them into the command.

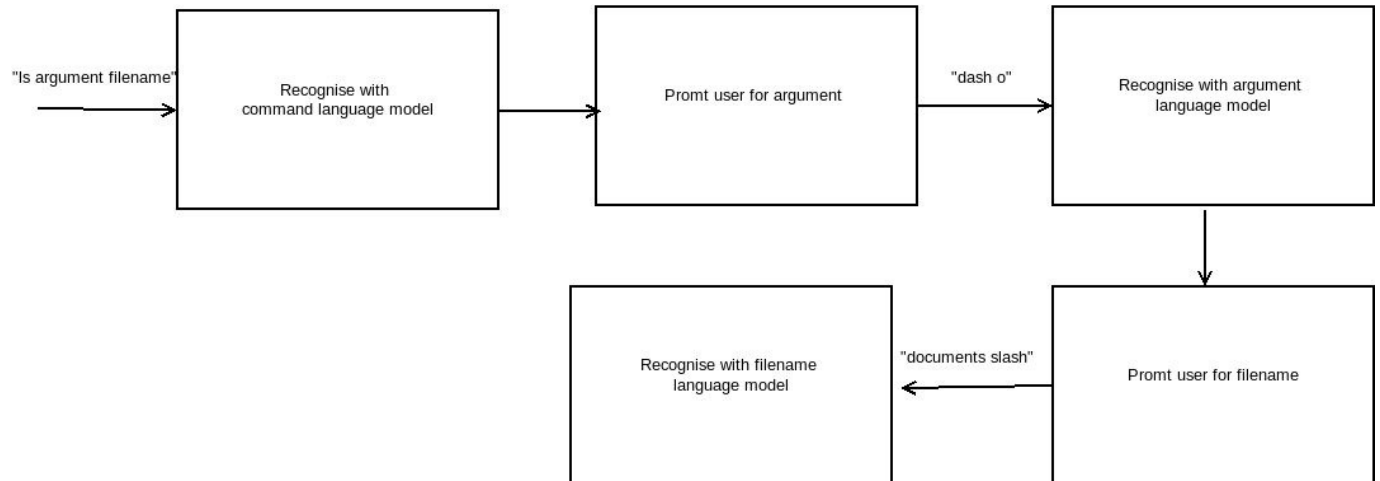


Figure 3.5 - Diagram depicting the process of the three language model solution

While this approach did help alleviate the issues of the original, single language model approach, it came with its own problems. Firstly, it took a long time to input any command, with the user having to wait for each recogniser step to complete before the next stage. Errors in any of the three recognition steps could result in a nonsense command, meaning the user would have to start the input all over again. It was not completely worthless however, as I would reuse the basic concept of separate structures for each command component later in the 'dynamic' grammars.

Another issue with the language model approach was that the .lm files could only ever contain words that had appeared in the corpus used to generate them was based on. This would mean that a new language model would have to be generated every time the user wanted to run a program or access a file or folder that did not exist or was not visible to the program when the lm and dictionary files were generated.

Despite the above problems, it was already apparent that the smaller set of possible words was hugely improving performance compared to using the en-US language pack, with transcription taking less than a second rather than several seconds depending on the length of the input data.

Grammar Structure for CLI

It was clear at this point that a more structured, defined method of restricting word search was needed, that would be able to expand to include previously uninvoked programs, filenames and extensions. This new method would also have to be supported by pocketsphinx. With this in mind, I decided to use a grammar structure in the form of a JSGF grammar file. I used as a basis a simple context free grammar for English [14] that describes a sentence as consisting of a Noun Phrase (NP) and Verb Phrase (VP). I defined a command as consisting of a Command Phrase and a recursive Parameter Phrase.

The original version of this grammar was as follows:

```
#JSGF V1.0;
grammar command;

public <command> =[<sudo>]<prog>[<option>]*[<file>]*;

<prog> = LS| FIREFOX | PYTHON;
<option> = (DASH A| DASH V | DASH C)[<file>];
<file> = FILENAME;
<sudo> = SUDO;
```

Here, a command consists of a program name optionally preceded by “sudo” and optionally followed by a command line option or a filename. The final version of the ‘command’ grammar contains no set terminal rules for programs or files, instead referencing the single public ‘root’ rule of one of the ‘dynamic’ grammars. The root rule of the dynamic grammars then represent the possible options for programs, files and folders. For more detail on the final version of this grammar, see Chapter 4, Implementation.

The grammar approach was from the start more successful compared to the language model. It was much easier to diagnose problems with the grammar given that I had complete control over its content, unlike the language model which I could only influence indirectly. The grammar also made it impossible for the system to recognise inputs that did not remotely resemble a possible command; such inputs could now be detected.

The move to the grammar approach also meant that the language model creating web tool was no longer necessary. I could now use the simpler service [15] that returns only the phonetic dictionary.

Populating Dynamic Grammars

The next step was to devise a way to automatically populate the 'dynamic' grammars that would store the program names, file names and extensions, folder names and aliases used in the program. My initial approach for the program grammar was to use the in-built BASH command 'compgen' to get a list of all the compiled binaries in the system's path. This proved problematic however, as with over 400 programs some of which had long and unwieldy names, the accuracy of transcription was greatly affected. To remedy this, I wrote code that reads CLI history files (BASH and fish history) and counts the number of occurrences of each program in the list. I then only included those programs that had at least one appearance i.e had been typed by the user at least once. This cut the program grammar down to a much more manageable 130 or so programs.

I then moved on to the grammars for folders, files and file extensions. I wrote code that got all folder names from the current directory and all folder names from the level below the current directory. One quirk of this step was that names which started with certain characters such as "." or "_" could not have their pronunciations generated by the web service, so these had to be filtered out. I then wrote similar code to handle files and their extensions, and the 'alias.txt' that stores the user defined aliases.

Running Commands

I had originally intended for Majel to work in conjunction with an existing shell such as BASH, piping the output of my program into the shell's input. There were major issues with this however, namely in that using the command 'cd' to change directory would not be preserved once exiting the Python script, instead leaving the shell in the location that the program was originally invoked from. This was because of the way that the stack of processes works on Linux systems, with the stack of processes exiting back in the original directory location, rather than the location that the process was when it was ended.

To fix this, I instead had Majel itself handle the execution of commands via the 'subprocess' module for python. I also wrote functions to emulate the behavior of BASH's 'cd' command. The end result of these changes was effectively a simple shell program that was entirely independent from BASH. Another positive effect of this

change was that I could now implement my own control commands for Majel that could be run from within the program such as 'majel_timeout' that sets the number of seconds the program waits for microphone input.

4. Implementation

The project's implementation consists of two separate parts. The first 'majel' is the main script that handles the frontend implementation; command prompt, obtaining speech, recognising and running commands. The second, 'setup' is a collection of backend functions which handles the creation and updating of the dynamic grammars, as well as the creation and editing of the phonetic dictionary file. The 'majel' script is run to actually start the program, with functions from setup imported and called when needed.

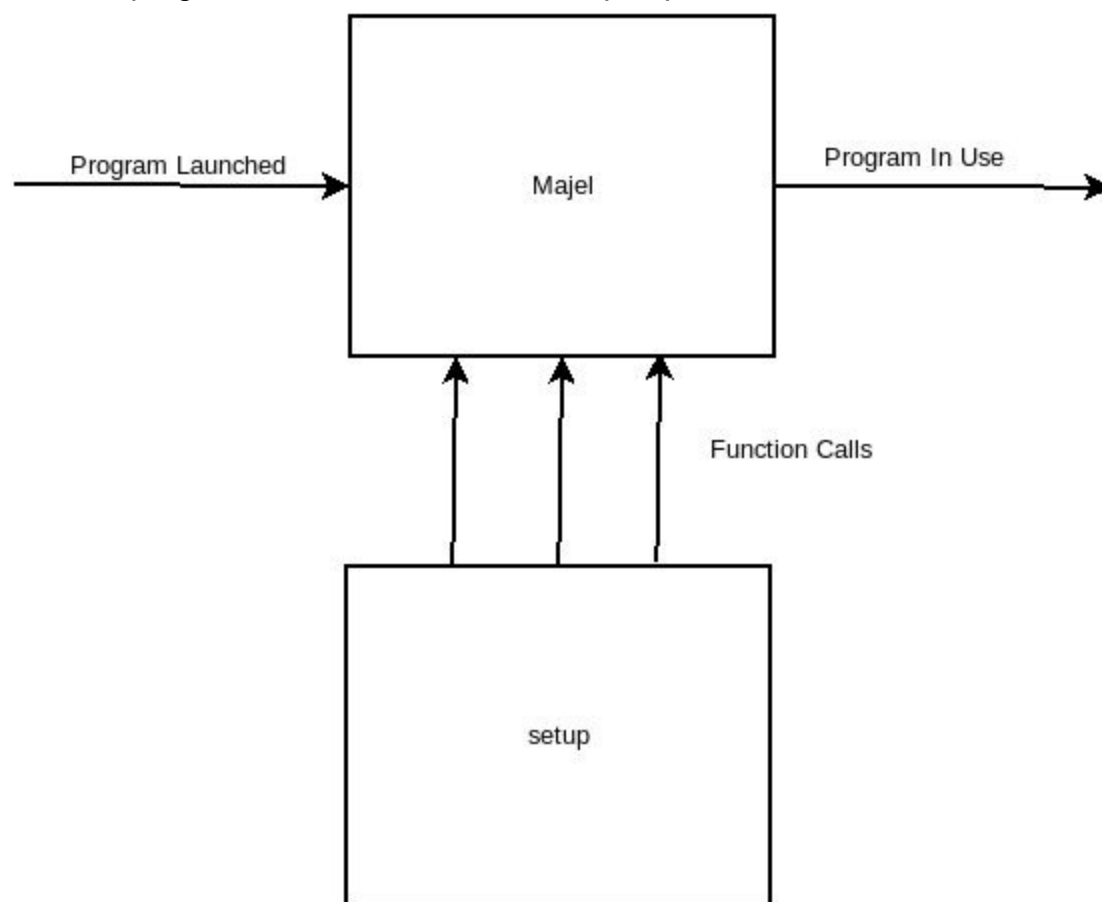


Figure 4.1 Overview diagram of program modules

In this section I will highlight the key function structures of each module, describing each on a code level.

Backend

`setup_dict_grammar()`

This is the core function of the setup script, and is called when the main program is run as 'majel setup'. It handles the creation of lists of programs, files, folders and aliases, the creation of the phonetic dictionary containing all of these lists and finally the dynamic grammars. This process is shown in Figure 4.2.

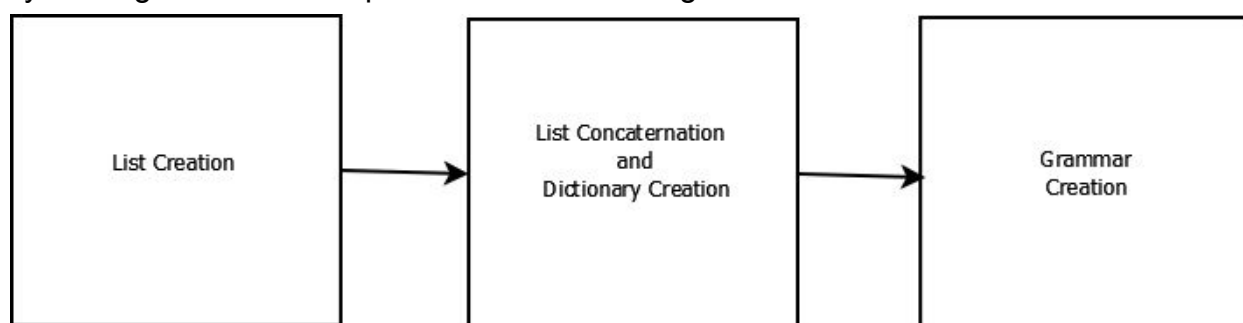


Figure 4.2 - Diagram showing the process of `setup_dict_grammar()`

Figure 4.2 shows the creation of text files listing the various components used by the program; programs, file names, file extensions, directory names and aliases. The function `compare_prog_list()` compares the list of all programs with the programs that have been previously invoked in the BASH history file, and returns a list that contains only those programs that have been run at least once.

As shown in Figure 4.3 all words to be used in the program are combined into a single file ("folders_out.txt") which is then uploaded to the web service to create the phonetic dictionary, "master.dict". This is an efficiency consideration as it reduces the number of calls to the web service required.

Finally, the dynamic grammar files are created; before this happens, each word list is compared to the words in the dictionary file and any words that do not appear in the dictionary are removed from the lists. This is to ensure that words that are not able to be processed by the dictionary creation service are not erroneously added to the grammars. All temporary files used by this function are then deleted. This code is shown in Figure 4.4. One of the resultant grammar files is shown in Figure 4.5.

```
#get the installation location
path = get_path()
# execute script that populates a file progs.txt
os.system('%s/scripts/compngen.sh' % path)

# get content of progs.txt
with open("%s/scripts/progs.txt" % path, 'rt') as f:
    data = f.readlines()
# formatting
data = [s.replace("\n", "") for s in data]

# use cmd history files to get the most common commands used
master = generate_prog_list()

# returns list of most common programs
prog_list = compare_prog_list(master, data)

# writes program list to file
write_list_to_file(prog_list, "%s/scripts/progs_out" % path)

#get list of aliases
alias_list = get_alias()

#write alias list to file
write_list_to_file(alias_list, "%s/scripts/alias_out" % path)

# gets folder names from the given directory
folder_list = get_directory()

# writes folder list to file
write_list_to_file(folder_list, "%s/scripts/folders_out" % path)

#get file names and extensions from the given directory:
(file_list, ext_list) = get_filenames()
print(file_list, ext_list)
#write file list to file
write_list_to_file(file_list, "%s/scripts/file_out" % path)
#write file extensions list to file
write_list_to_file(ext_list, "%s/scripts/exts_out" % path)
```

Figure 4.2 - Code excerpt from `setup_dict_grammar()` that shows the creation and writing to files of lists containing programs, files, folders, file extensions and aliases.

```
# combines all word lists into one
print("combining word lists...")
combine_files("%s/scripts/folders_out.txt" % path,
              "%s/scripts/progs_out.txt" % path)
combine_files("%s/scripts/folders_out.txt" % path,
              "%s/scripts/cmd_out.txt" % path)
combine_files("%s/scripts/folders_out.txt" % path,
              "%s/scripts/file_out.txt" % path)
combine_files("%s/scripts/folders_out.txt" % path,
              "%s/scripts/exts_out.txt" % path)
combine_files("%s/scripts/folders_out.txt" % path,
              "%s/scripts/alias_out.txt" % path)
master_path = "%s/languages/cmd2/master.dict" % path
# use web service to create dictionary
print("getting dictionary...")
get_dictionary("%s/scripts/folders_out.txt" % path, master_path)
print("done!")
```

Figure 4.3 - Code excerpt from setup_dict_grammar() showing the combining of the word list files and the creation of the phonetic dictionary file via get_dictionary().


```
words_in_dictionary = get_words_from_dictionary()
#print(words_in_dictionary)

#ensure that all words in grammars are in the dictionary
prog_list = compare_lists(prog_list, words_in_dictionary)
file_list = compare_lists(file_list, words_in_dictionary)
ext_list = compare_lists(ext_list, words_in_dictionary)
folder_list = compare_lists(folder_list, words_in_dictionary)
alias_list = compare_lists(alias_list, words_in_dictionary)

create_grammar(prog_list, "progs", "%s/grammars/progs.gram" % path)
create_grammar(file_list, "files",
               "%s/grammars/files.gram" % path)
create_grammar(ext_list, "exts",
               "%s/grammars/exts.gram" % path)
create_grammar(folder_list, "folders",
               "%s/grammars/folders.gram" % path)
create_grammar(alias_list, "alias", "%s/grammars/alias.gram" % path)

# remove temporary files
os.remove("%s/scripts/folders_out.txt" % path)
os.remove("%s/scripts/progs_out.txt" % path)
os.remove("%s/scripts/cmd_out.txt" % path)
os.remove("%s/scripts/exts_out.txt" % path)
os.remove("%s/scripts/file_out.txt" % path)
os.remove("%s/scripts/alias_out.txt" % path)

os.remove("%s/scripts/progs.txt" % path)
```

Figure 4.4 - Code excerpt from `setup_dict_grammar()` showing the comparison of each list to the dictionary, dynamic grammar creation and temporary file

```
#JSGF V1.0;
grammar progs;
public <root> = (<rule0>|<rule1>|<rule2>|<rule3>|<rule4>|<rule5>|<rule6>|<rule7>);
<rule0> = LS;
<rule1> = PYTHON3;
<rule2> = CD;
<rule3> = MAJEL;
<rule4> = GIT;
<rule5> = APT;
<rule6> = EXIT;
<rule7> = PIP;
```

Figure 4.5 - Screenshot from “progs.gram” showing one of the grammar files created.

get_dictionary()

This function takes in a list of words as a parameter and returns a phonetic dictionary as a file “words.dict”. It uses a web based service provided by the developers of Sphinx to do this. The file uploading and downloading is handled using the ‘requests’ python module. In addition to the word list file, a ‘hand’ file is also uploaded, containing any user defined pronunciations that should not be overwritten by what the web service generates.

```
def get_dictionary(file_read, file_write="words.dict"):
    """
    takes a text file of a list of words(file_read) and returns
    a dictionary file (file_write) describing
    how to understand that word aloud.
    """
    file_hand = "%s/scripts/hand.txt" % path
    url = "http://www.speech.cs.cmu.edu/cgi-bin/tools/logios/lextool.pl"
    print("reading %s..." % file_read)
    files = {'wordfile': open(file_read, 'rb'), 'handfile': open(file_hand, 'rb')}
    r = requests.post(url, files=files) # get HTML response of file upload
    for lines in r.text.split(">"): # find download link
        if "<!-- DICT " in lines:
            dl_link = lines
    dl_link = dl_link.replace("<!-- DICT ", "") # strip download link
    dl_link = dl_link.replace(" --", "")
    #print(dl_link)
    dict_response = requests.get(
        dl_link, allow_redirects=True) # get dict file
    print("writing %s to file..." % file_write)
    open(file_write, 'wb').write(
        dict_response.content) # write contents of dict
```

Figure 4.6 - Code excerpt showing the get_dictionary() function

create_grammar()

This function takes a list of words and creates a 'root' grammar with each word in the list as a private terminal rule and a public root rule that expands to each terminal rule. It then saves this grammar to a ".gram" file, such that it can be loaded by the 'master' grammar. This function makes use of the 'RootGrammar' and 'PublicRule' objects from the 'pyjskf' module for the creation of the grammar and its rules. Since all words returned by the speech recognition component are fully capitalised, each element of the grammar likewise must uppercase. There is also a check here that no special characters are erroneously added to the grammar.

```
def create_grammar(word_list, name, gram_file):  
    """  
    read a list in a text file ('`word_list`') and create  
    a grammar ('`name`') file ('`gram_file`') for that list,  
    such that the speech can one of any of the elements of the list  
    """  
    upp_list = list()  
    grammar = RootGrammar(name=name, case_sensitive=True)  
    i = 0  
    for lines in word_list:  
        rule_name = "rule" + str(i)  
        upp = lines.upper().strip()  
        #print("upp is",upp)  
        if upp != "" and upp != "{" and upp != "}" and upp != "." and upp[0] != "_":  
            r = PublicRule(rule_name, upp, case_sensitive=True)  
            grammar.add_rule(r)  
            upp_list.append(upp)  
            i = i+1  
  
    with open(gram_file, 'wt') as g:  
        print(grammar.compile(), file=g)
```

Figure 4.7 - Code excerpt showing the create_grammar() function.

add_to_grammar()

This function takes a grammar file containing a root grammar and a list of words and returns a grammar file with that list of words appended as private terminal rules. It is used by the functions that deal with the addition of new programs, files, folders and aliases as the program is running. Due to a limitation of the 'pyjsgf' the rules from the input grammar must be copied and added to a new grammar, rather than having new rules be appended to the input grammar. The new rules are then added to this new grammar and it is saved, overwriting the input grammar. The function ensures that each terminal rule in the output grammar expands to a unique string, by checking that the new rule expansions are not in 'old_rules_text'.

```
def add_to_grammar(grammar_path,file_path,gram_name):
    """
    loads a ``Grammar`` at grammar_path and tries to add rules to it
    from the file in file_path then returns the new ``Grammar``
    """
    old_gram = parser.parse_grammar_file(grammar_path)
    with open(file_path,'rt') as f:
        word_list = f.readlines()
    #remove root rule from old grammar
    old_gram.remove_rule(old_gram.get_rule_from_name("root"))
    # get list of rules from old grammar
    old_rules = old_gram.rules
    new_gram = RootGrammar(name=gram_name, case_sensitive=True)
    # add existing rules to new grammar
    i = 0
    old_rules_text = list()
    for rules in old_rules:
        exp = rules.expansion.text.upper()
        old_rules_text.append(exp)
        if exp not in word_list:
            rule_name = "rule" + str(i)
            r = PublicRule(rule_name,exp,case_sensitive=True)
            new_gram.add_rule(r)
            i += 1
    # add new rules to new grammar
    for lines in word_list:
        rule_name = "rule" + str(i)
        exp = lines.upper().strip()
        print("upp is ",exp)
        if exp not in old_rules_text and exp != "" and exp != "{" and exp != "}" and exp != ".":
            r = PublicRule(rule_name, exp, case_sensitive=True)
            new_gram.add_rule(r)
            i += 1
    # compile new grammar back to file
    new_gram.compile_to_file(grammar_path,compile_as_root_grammar=True)
```

Figure 4.8 - Code excerpt showing the add_to_grammar() function

update_folder_grammar_dictionary()

This function is called when the user changes to a new directory. It acts in a similar way to `setup_dict_grammar()`; it gets a list of new folders in the current directory and updates the folders grammar and the phonetic dictionary with any new words. Given that it calls the `get_dictionary()` function, it requires that the system is connected to the internet to function. There exist similar functions for the updating of file names, file extensions and aliases.

```
def update_folder_grammar_dictionary(p):
    #print(p)
    if os.path.exists("%s/grammars/command.fsg" % path):
        os.remove("%s/grammars/command.fsg" % path)
    folder_list = get_directory(p)
    #print("FOLDER_LIST", folder_list)

    write_list_to_file(
        folder_list, "%s/scripts/folders_out" % path)

    # use web service to create folder dictionary
    get_dictionary("%s/scripts/folders_out.txt" % path,
                  "%s/scripts/folders.dict" % path)
    dict_list = get_words_from_dictionary()

    folder_list = compare_lists(folder_list, dict_list)
    write_list_to_file(
        folder_list, "%s/scripts/folders_out" % path)

    add_to_grammar(
        "%s/grammars/folders.gram" % path, "%s/scripts/folders_out.txt" % path, "folders")
    master_path = "%s/languages/cmd2/master.dict" % path

    combine_files(
        master_path, "%s/scripts/folders.dict" % path)
    os.remove("%s/scripts/folders.dict" % path)
    os.remove("%s/scripts/folders_out.txt" % path)
```

Figure 4.9 - Code excerpt showing the `update_folder_grammar_dictionary()` function

Frontend

main()

This is the main function of the frontend part of the program, it handles the creation and display of the command prompt and is where the 'main loop' of the program is located, where the command input, formatting and execution all occur. In the case that the user types a command such that the variable 'input_string' is not empty (they have not pressed enter without typing anything), `get_command()` is called to get the speech input. Otherwise, the program will attempt to execute what was typed as a command. In both cases, the command is sent to `words_to_character()`.

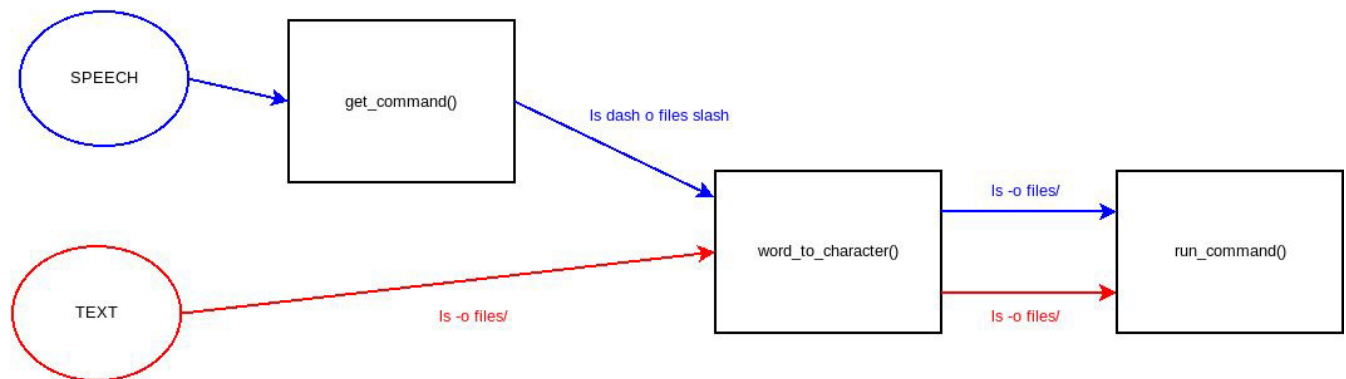


Figure 4.10 - Diagram showing the processing that happens for each of the two input types, speech and text that occurs in `main()`

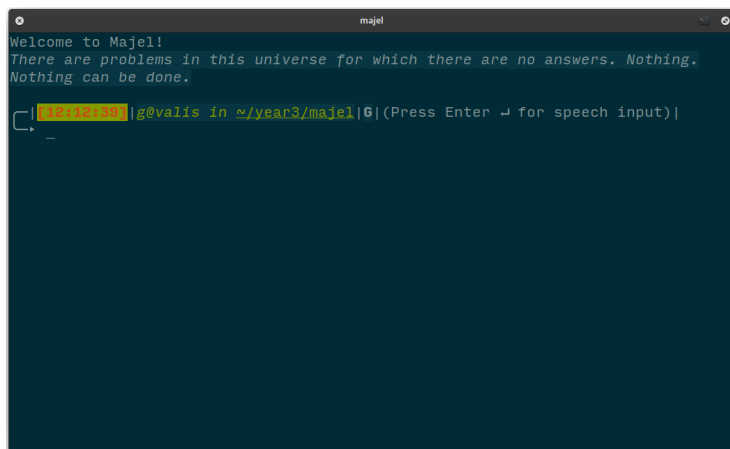


Figure 4.11 - Screenshot of Majel running in a terminal window, ready for command

```
if __name__ == "__main__":
    #language setup - could be changed later
    l = ("/home/g/year3/majel/languages/acoustic-model",
        "/home/g/year3/majel/languages/cmd1.lm",
        "/home/g/year3/majel/languages/cmd2/master.dict")
    gram = "/home/g/year3/majel/grammars/command.gram"
    #display setup
    command_words = str()
    username = getpass.getuser()
    hostname = socket.gethostname()
    test_recognition()
    #welcome message
    print("Welcome to Majel!\n" + "\x1b[3;39;40m" + get_quote() + "\x1b[0m")
    #prompt start
    while True:
        if is_repo(os.getcwd()):
            git_string = "\x1b[1;41;40m" + "G" + "\x1b[0m"
        else:
            git_string = ""
        current_time = "\x1b[1;31;42m[" + datetime.now().strftime("%H:%M:%S") + "]" + "\x1b[0m"
        path = "\x1b[3;32;40m" + username + "@" + hostname + \
            " in \x1b[0m" + "\x1b[4;32;40m" + os.getcwd() + \
            "\x1b[0m" + git_string
        path = path.replace("/home/%s"%username, "~")

        prompt = "\n└─|s|s|(Press Enter ↵ for speech input)|\n↳ " % (current_time, path)
        input_string = input(prompt)
        command_words = shlex.split(input_string)
        s = False
        if command_words == []: # use voice
            s = True
            command_words = get_command(l, gram, selective_mode)
            #print(command_words)

        if len(command_words) != 0 and command_words != [""]: # if some words were returned -
            command_words = word_to_character(command_words, s)
            run_command(command_words)
        else:
            print("I'm sorry %s, I'm afraid I can't do that." % username.capitalize())
    exit_func()
```

Figure 4.12 - Code screenshot showing the main() function, showing the creation and formatting of the prompt string and the command input (get_command()), processing (words_to_character()) and execution(run_command()).

get_command()

This function handles the input and transcription of a voice command. It returns the command that was spoken as an ordered list, with each word spoken as an element in the list. It uses a Microphone object from the SpeechRecognition module to capture the input from the default microphone set in the OS. The number of seconds that the program waits for input is set by the 'timeout' variable which can be set via a command. This is then passed into the 'recognise_sphinx()' function of a Recogniser object, which also takes in a tuple of language parameters 'lang' and the path to the main grammar file. The final structure of this grammar file is discussed below. This then returns as a string the transcribed text, 'out' which is returned by the function as a lowercase list. If the recognition fails and an 'UnknownValueError' is raised, an empty list is returned, which is detected by the next step of the program in `main()`. This function also supports the experimental 'selective mode' where instead of the single 'best' transcription being returned, instead the user can select from the top 10 possible transcriptions.

```
def get_command(lang, gram, selective_mode=False):
    recog = sr.Recognizer() # this is the object that recognises the speech
    mic = sr.Microphone(chunk_size=1024, sample_rate=44100)
    with mic as source:
        #recog.adjust_for_ambient_noise(source)
        #time.sleep(0.5)
        print("listening...")
        audio = recog.listen(source, phrase_time_limit=timeout)
    try:
        print("working...")

        if selective_mode:
            out2 = recog.recognize_sphinx(audio, language=lang, show_all=True)
            out = show_possible(out2)
        else:
            out = recog.recognize_sphinx(audio, language=lang, grammar=gram)
            #os.system('clear')
    except sr.UnknownValueError:
        #out2 = recog.recognize_sphinx(audio, language=lang, show_all=True)
        print("didn't quite get that")
        out = ""
    #return out.hyp().hypstr.lower().split()
    return out.lower().split()
```

Figure 4.13 - Code excerpt showing the `get_command()` function.

word_to_character()

This function takes in an ordered list of words and converts it into an executable command. It does this in a number of ways, including converting words such as 'slash', 'dash' etc into characters, combining paths into single strings separated by slashes and changing the case of words or characters.

```
#print("phrase before replace step: ",phrase)
phrase = replace_in_list(phrase,"dash","-")
phrase = replace_in_list(phrase,"slash","/")
phrase = replace_in_list(phrase,"dot",".")
#print("phrase after replace step: ",phrase)
for word in phrase:
    #print(word)
    if word == "<s>":
        phrase.remove(word)
```

Figure 4.14 - Code excerpt from the first stage of words_to_character()

The first part of this function uses the support function 'replace_in_list()' which takes a list, and two strings and returns the list with all instances of the first string replaced with the second. It then loops through the phrase and removes all instances of '<s>' which are sometimes introduced during the transcription step and represent silence in the input speech.

```
prev_word = "BLANK"
count = 0
while count < len(phrase):
    for index,word in enumerate(phrase):
        #print(index,word)
        if word == "upper":
            phrase[index+1] = phrase[index+1].upper()
            phrase.pop(index)
        if word == "lower":
            phrase[index+1] = phrase[index+1].lower()
            phrase.pop(index)
        if word == "-":
            #print("dash found!")
            phrase[index+1] = "-" + phrase[index+1]
            phrase.pop(index)
        if word == "capital":
            phrase[index+1] = phrase[index+1].capitalize()
            phrase.pop(index)
        if word == "-capital":
            phrase[index+1] = "-" + phrase[index+1].capitalize()
            phrase.pop(index)

        if word == "execute":
            phrase[index+1] = "./" + phrase[index+1]
            phrase.pop(index)
        if word == "." and index < len(phrase)-1:
            if phrase[index+1] == ".":
                phrase.pop(index+1)
                phrase[index] = ".." + phrase[index+1]
                phrase.pop(index+1)
                #print("dot dot found!")
            else:
                phrase[index] = phrase[index-1] + "." + phrase[index+1]
                phrase.pop(index-1)
                phrase.pop(index)
```

Figure 4.15 - Code except showing the second step of `word_to_character()`

The next step makes use of Python's `enumerate()` function to loop through each word in the phrase with an associated index. It first applies any case modification that the user has specified ('upper', 'lower', 'capital') and appends a "-" to the next element after a "-" is found. Next it appends "." to the next element after "execute". Finally, it handles the case that there are two dots adjacent to each other, usually representing the parent to the current directory. Note that in all these cases the instruction word is removed from the phrase.

```
#print("word is ", word, "at index", index)
#print("prev_word is", prev_word)
if prev_word[0] == "/" and word[0] == "/":
    new_word = prev_word + word
    #print("new word is ", new_word)
    phrase[index] = new_word
    phrase.pop(prev_index)
    prev_word = new_word
else:
    prev_word = word
    prev_index = index
```

Figure 4.16 - Code excerpt showing the next part of the enumerate loop, which handles the combining of file paths into a single string

The final part of the enumerate loop deals with the creation of a single file path string from separate strings. It does this by checking if the first character of the current word and the previous word is '/' meaning that these words form a single file path. It then combines these two words into a single string `new_word` and places this new string in the place of the current word, `phrase[index]`. For example if the current word was `"/world"` and the previous word in the phrase was `"/hello"` the phrase afterward would replace these two with the word `"/hello/world"`.

run_command()

This function is responsible for the actual execution of commands. It takes the command in as a parameter in an ordered list. It first checks if the command is one of majel's 'built-in' commands, if it is not, it checks if it contains any aliases and then executes the command using python's 'subprocess' module. If the command contains a potentially 'dangerous' program to execute, it prompts the user if they are sure they want to execute it. The program will inform the user if the command they typed is invalid, or if they do not have permissions to run it.

```
if "exit" in command:
    exit_func()
if command[0] == "cd":
    change_directory(command)
elif command[0] == "majel-timeout" and len(command) == 2:
    global timeout
    timeout= int(command[1])
    print("set timeout to ",timeout)
elif command[0] == "majel-pronouce" and len(command) == 2:
    show_pronouciation(command[1])
elif command[0] == "majel-alias" and len(command) == 3:
    add_alias(command[1],command[2])
elif command[0] == "majel-update" and len(command) == 1:
    if is_connected("google.com"):
        setup.update_all()
    else:
        print("not connected!")
elif command[0] == "majel-hand":
    word = input("Type the word you want to override the pronunciation for: ")
    pronc = input("Type the pronunciation: ")
    setup.set_override(word.upper(),pronc.upper())
```

Figure 4.17 - Code except showing the section of run_command() that handles Majel's built-in commands.

```
else:
    try:
        command = replace_alias(command)
        # command_exec = [command[0], "".join(command[1:])]
        # print(command_exec)
        # subprocess.run(command_exec)

        if "sudo" in command or "rm" in command or "mv" in command or "cp" in command:
            p = input(
                "are you sure you want to run the command: '%s'? (Y/N)→ " % " ".join(command))
            if p in ["y", "yes", "Y", "aye aye", "sure", "ok"]:
                subprocess.run(command, check=True)
            else:
                subprocess.run(command, check=True)
    except subprocess.CalledProcessError:
        print("reversing the polarity of the neutron flow...")
        command = replace_in_list(command, " ", "_")
        subprocess.run(command)
    except FileNotFoundError:
        print("Invalid Command: %s" % " ".join(command))
    except PermissionError:
        print("User %s does not have permission to do that" % getpass.getuser())
```

Figure 4.18 - Code except showing the 'dangerous' command confirmation and the error handling section of `run_command()`

change_directory()

This function simulates a 'cd' command in BASH in that it will change the current working directory to that directory that is specified. If it is run with no specified directory, it will change to the user's home folder. It supports the same type of input as BASH with '.' representing the current directory and '..' the parent. When the directory is changed, functions from the setup script that update the dynamic grammars and phonetic dictionary with any new files and folders in the directory that is being changed to. The function uses the built-in `os.chdir()` function to change the directory. The function will inform the user if the directory does not exist or is not a directory.

```
def change_directory(command):
    try:
        if len(command) == 1:
            os.chdir("/home/%s/" % getpass.getuser())
            if is_connected("google.com"):
                setup.update_folder_grammar_dictionary("/home/%s/" % getpass.getuser())
                setup.update_file_grammar_dictionary("/home/%s/" % getpass.getuser())
                test_recognition()
            else:
                print("Not connected - new names will not be updated.")
        else:
            print("path is {}".join(command[1:]))
            if is_connected("google.com"):
                setup.update_folder_grammar_dictionary("".join(command[1:]))
                setup.update_file_grammar_dictionary("".join(command[1:]))
                test_recognition()
            else:
                print("Not connected - new names will not be updated.")
            os.chdir("".join(command[1:]))
    except FileNotFoundError:
        print("no such directory: %s" % command[1])
    except NotADirectoryError:
        print("%s is not a directory" % command[1])
```

Figure 4.19 - Code excerpt showing the `change_directory()` function.

Grammar Structure

Having discussed the core functions of the program, I will now detail the grammar structure of the 'command.gram' file that is passed as parameter to `get_command()` and then the Recogniser object. The final form of this file is as follows:

```
#JSGF V1.0;
grammar command;
import <progs.gram>;
import <folders.gram>;
import <files.gram>;
import <exts.gram>;
import <alias.gram>;
public <command> = EXIT
| ([SUDO]) (EXECUTE|<progs.root>|<alias.root>) [<parameter>*];

<parameter> = [<option>][<pathexpr>[<file>]];
<option> = <dash>[<modifier>]<chars>;
<pathexpr> = <dot><slash><pathexpr>
             |<dot><dot><slash><pathexpr>
             |<foldername><slash><pathexpr>
             |<NULL>;

<file> = <filename><dot><fileextension>
        |<filename>;

<foldername> = [<modifier>]<folders.root>;
<filename> = [<modifier>]<files.root>;
<fileextension> = <exts.root>;
<chars> = B | A | C | F | G | M | O | V | S | X | R | Q;
<modifier> = CAPITAL | UPPER | LOWER;
<slash> = SLASH;
<dash> = DASH;
<dot> = DOT;
```

At the top of the file, the dynamic grammars are imported. Next is the sole 'public' (meaning that it can be spoken) rule of the grammar, 'command'. This consists of either the keyword 'EXIT' that exits the program or of a actual command; the structure of a command is given as an optional 'SUDO' followed by either 'EXECUTE', the name of a program from the 'progs' grammar or an alias from the 'alias' grammar. The final component is any number (including zero) of 'parameter'. A parameter is defined as an 'option', a 'DASH' followed by a 'char' and/or a 'pathexpr' with an optional 'file'. The 'pathexpr' rule is right side recursive and describes a file path of infinite length. 'file' defines a file as a filename followed by a 'DOT' followed by a file extension or a filename with no extension; in both cases, the name and extension come from the 'files' grammar or the 'exts' grammar. Various parts of the grammar can also have an optional 'modifier' which can be 'CAPITAL', 'UPPER' or 'LOWER'; these are used by the program to change the case of the proceeding word to the user's specification.

5. Results and Evaluation

In this section I will detail two different types of testing of the program; I will first carry out unit tests to ensure that all the features of the program are functional. I will then carry out useability tests to test the program's ability to recognise a variety of spoken commands. Finally, I will evaluate the testing results, and give some ways that the current implementation could be improved.

Unit Testing

In this section I will conduct a series of tests of the core features of the program, first describing the test, then the expected result and finally the actual output of the program.

Unit Test 1 - Initialization

Description: Test of initial setup of the system, with the creation of the phonetic dictionary file and the dynamic grammar files, based on where the program is being initialized from. For this test, the program has been added to the system path, allowing it to be run from anywhere on the system. This will be achieved by running 'majel setup' from a BASH terminal in the folder.

Expected Result: The program will first create the phonetic dictionary using the files and folders from the current directory, and programs from the BASH history file. It will then populate the dynamic grammars using these files, folders, and programs.

Result:

I created a folder with three empty directories and three files, as shown in Figure 5.1. I then ran the command 'majel setup' in a BASH shell in this folder (Figure 5.2). The program then initialized, populating the 'folders', 'files' and 'exts' grammar files based on the directory, as well as the 'progs' grammar based on the BASH history. The phonetic dictionary file was also created, containing the pronunciation of each word in these grammars. This is displayed in Figures 5.3 to 5.6.

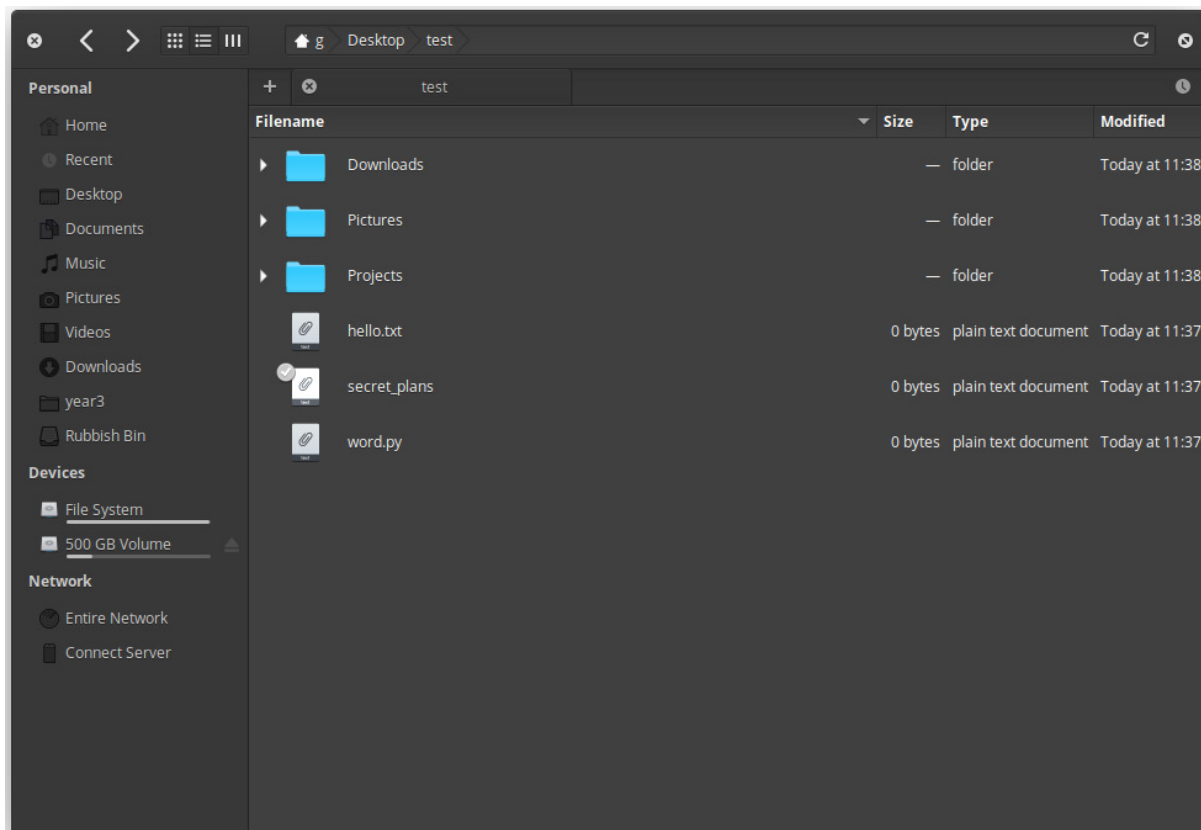


Figure 5.1- Screenshot of file manager showing the directory that majel setup will be run in.

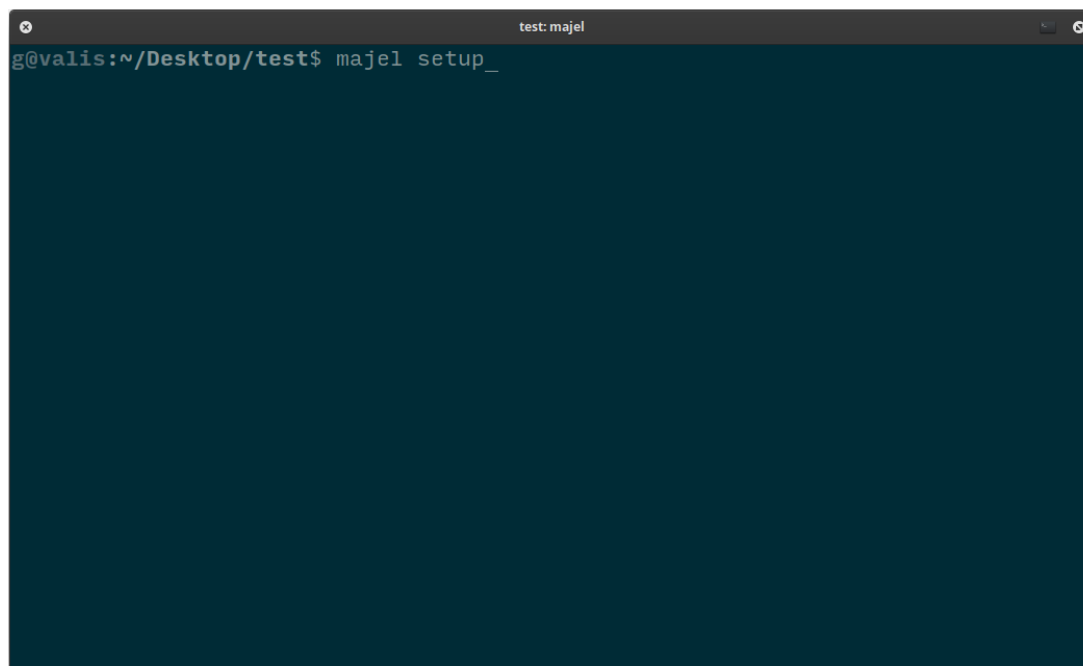


Figure 5.2 - Screenshot of a terminal window in the directory with the command ready.

```
grammars > exts.gram
1  #JSGF V1.0;
2  grammar exts;
3  public <root> = (<rule0>|<rule1>);
4  <rule0> = TXT;
5  <rule1> = PY;
6
7
```

Figure 5.3 - Screenshot showing the exts.gram file, having been updated using the file extensions in the current folder.

```
grammars > files.gram
1  #JSGF V1.0;
2  grammar files;
3  public <root> = (<rule0>|<rule1>|<rule2>);
4  <rule0> = SECRET_PLANS;
5  <rule1> = HELLO;
6  <rule2> = WORD;
```

Figure 5.4 - Screenshot showing the files.gram file, having been updated using the file names in the current folder.

```
grammars > folders.gram
1  #JSGF V1.0;
2  grammar folders;
3  public <root> = (<rule0>|<rule1>|<rule2>);
4  <rule0> = PROJECTS;
5  <rule1> = DOWNLOADS;
6  <rule2> = PICTURES;
```

Figure 5.5 - Screenshot showing the folders.gram file, having been updated using the directory names in the current folder.


```
grammars > | progs.gram
1  #JSGF V1.0;
2  grammar progs;
3  public <root> = (<rule0>|<rule1>|<rule2>|<rule3>|<rule4>|<rule5>|<rule6>|<rule7>);
4  <rule0> = LS;
5  <rule1> = PYTHON3;
6  <rule2> = CD;
7  <rule3> = MAJEL;
8  <rule4> = GIT;
9  <rule5> = APT;
10 <rule6> = EXIT;
11 <rule7> = PIP;
```

Figure 5.6 - Screenshot showing the start of the progs.gram file, having been updated based on the most commonly used programs from the BASH history.

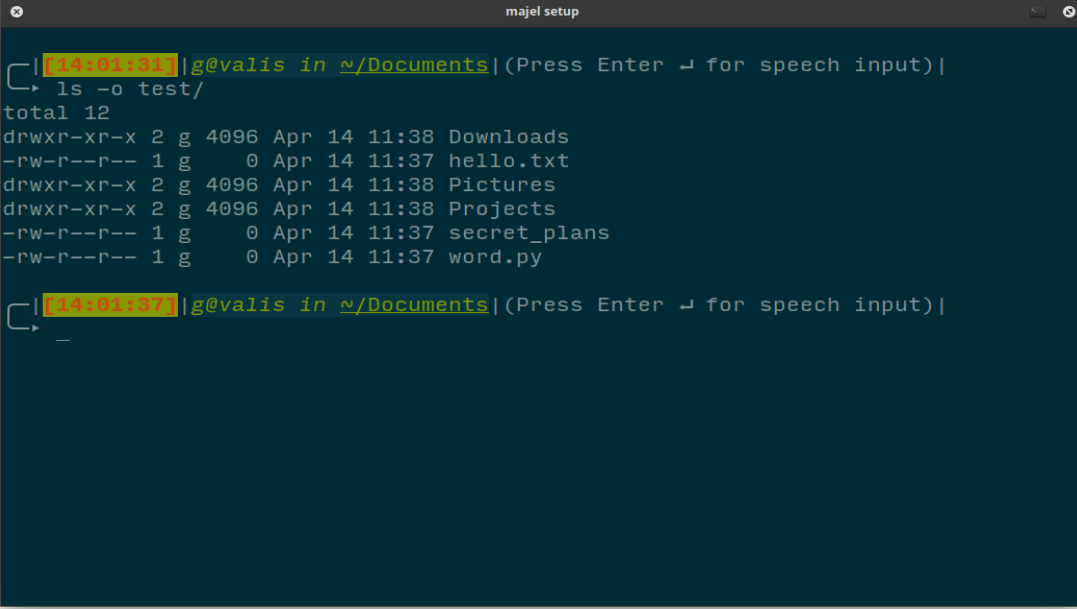
Unit Test 2 - Running Typed Commands

Description: Test displaying program's ability to run traditional typed commands, in a manner similar to existing shell programs. For this test, I will run the command 'ls -o test'.

Expected Result: I will run the command from the parent folder of the folder created for Unit Test 1. After the command is entered, I would expect to see the contents of the folder 'test' printed to the screen, along with extra information such as permissions, owner and time created as it is being run with the '-o' option.

Result

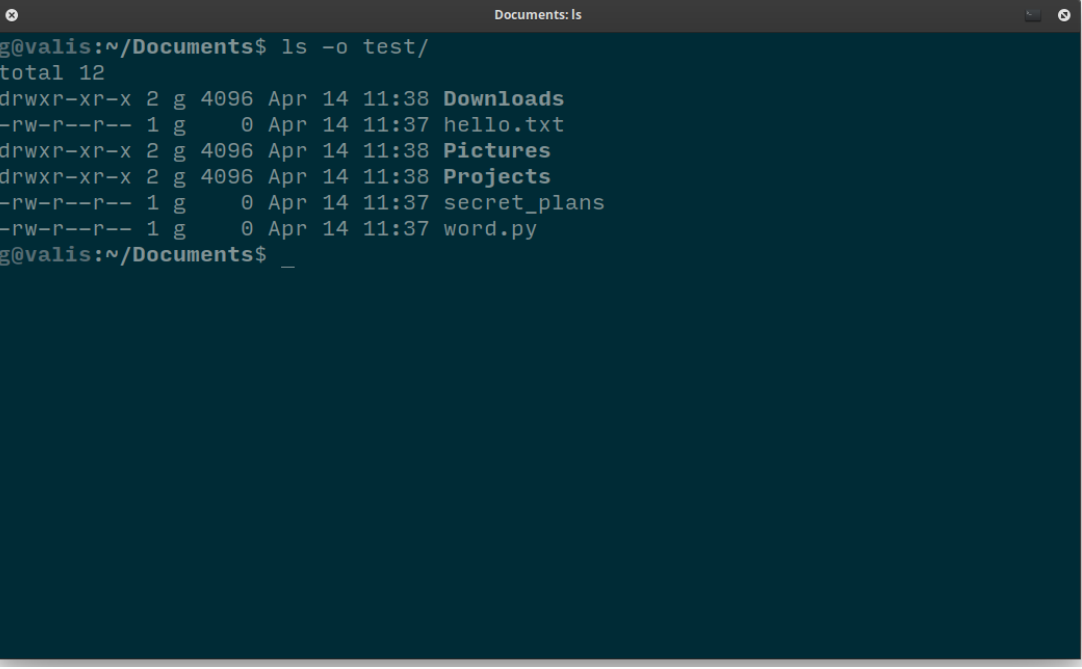
I launched the program in the parent folder of the test directory and ran the command (Figure 5.7). This was the expected result, and identical to the output of the same command in BASH (Figure 5.8).



```
majel setup
[14:03:31] g@valis in ~/Documents | (Press Enter ↵ for speech input) |
→ ls -o test/
total 12
drwxr-xr-x 2 g 4096 Apr 14 11:38 Downloads
-rw-r--r-- 1 g 0 Apr 14 11:37 hello.txt
drwxr-xr-x 2 g 4096 Apr 14 11:38 Pictures
drwxr-xr-x 2 g 4096 Apr 14 11:38 Projects
-rw-r--r-- 1 g 0 Apr 14 11:37 secret_plans
-rw-r--r-- 1 g 0 Apr 14 11:37 word.py

[14:03:37] g@valis in ~/Documents | (Press Enter ↵ for speech input) |
→ _
```

Figure 5.7- Screenshot of terminal program showing successful execution of typed command within Majel.



```
Documents: ls
g@valis:~/Documents$ ls -o test/
total 12
drwxr-xr-x 2 g 4096 Apr 14 11:38 Downloads
-rw-r--r-- 1 g 0 Apr 14 11:37 hello.txt
drwxr-xr-x 2 g 4096 Apr 14 11:38 Pictures
drwxr-xr-x 2 g 4096 Apr 14 11:38 Projects
-rw-r--r-- 1 g 0 Apr 14 11:37 secret_plans
-rw-r--r-- 1 g 0 Apr 14 11:37 word.py
g@valis:~/Documents$ _
```

Figure 5.8 - the same command as Figure 5.7, but executed in BASH.

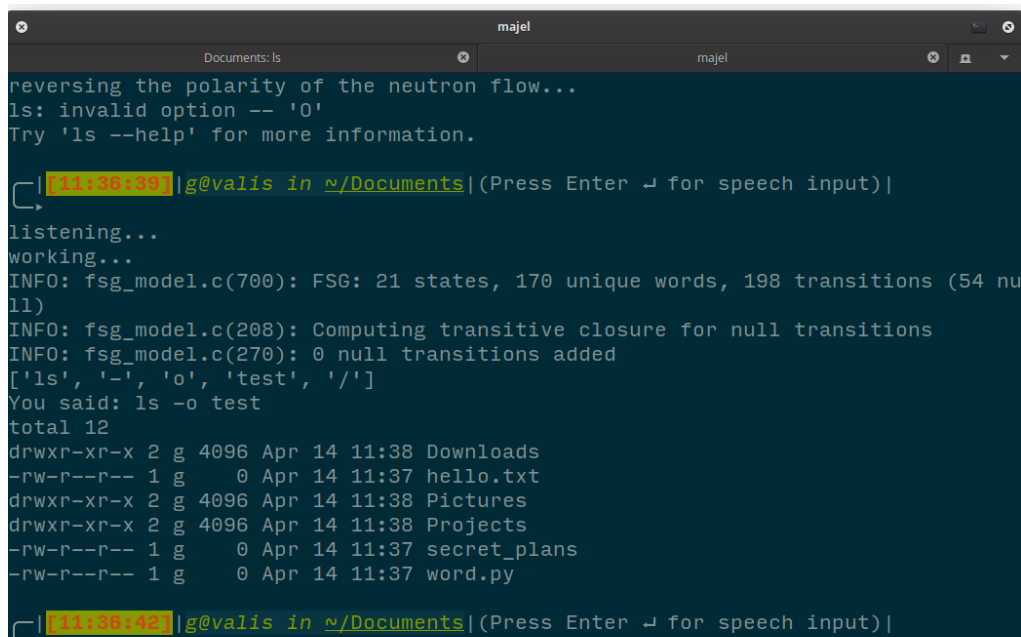
Unit Test 3 - Running Spoken Commands

Description: Test displaying the program's ability to interpret, format and execute spoken commands. For this test I will speak the same command that was typed in Unit Test 2, 'ls -o test/'. As mentioned in chapter 3, I will use a high quality headset microphone, as it's use results in the best possible transcription.

Expected Result: When the user presses enter with no command typed, the program will prompt the user to speak (via printing 'listening...' to the terminal) and will execute the spoken command. As the command is the same as the previous test, the output for the same command spoken aloud will be the same.

Result

The command was executed exactly as spoken. As shown in Figure 5.9 below, some debug information labeled 'INFO:' is printed when voice commands are executed; this is a feature from where pocketsphinx interacts with the grammar file, and cannot be disabled in the python interface.



```
majel
Documents: ls
reversing the polarity of the neutron flow...
ls: invalid option -- '0'
Try 'ls --help' for more information.

[11:36:38] [g@valis in ~/Documents] (Press Enter ↵ for speech input)
listening...
working...
INFO: fsg_model.c(700): FSG: 21 states, 170 unique words, 198 transitions (54 null)
INFO: fsg_model.c(208): Computing transitive closure for null transitions
INFO: fsg_model.c(270): 0 null transitions added
['ls', '-', 'o', 'test', '/']
You said: ls -o test
total 12
drwxr-xr-x 2 g 4096 Apr 14 11:38 Downloads
-rw-r--r-- 1 g    0 Apr 14 11:37 hello.txt
drwxr-xr-x 2 g 4096 Apr 14 11:38 Pictures
drwxr-xr-x 2 g 4096 Apr 14 11:38 Projects
-rw-r--r-- 1 g    0 Apr 14 11:37 secret_plans
-rw-r--r-- 1 g    0 Apr 14 11:37 word.py

[11:36:42] [g@valis in ~/Documents] (Press Enter ↵ for speech input)
```

Figure 5.9 - Screenshot showing output after a spoken command

Unit Test 4 - Updating Dynamic Grammar Files and Phonetic Dictionary File

Description: Test displaying the program's ability to append to the phonetic dictionary file with new words and also add to the 'dynamic' grammar files. This will be achieved by changing the directory to a directory that contains new directory names, file names and file extensions. I will use the test directory setup in the previous two tests, with a new directory that has not been processed by the program.

Expected Result: The new file names, directory names and file extensions will be added to the appropriate grammar files. All of these will be added to the phonetic dictionary.

Result

Prior to changing the directory, the dynamic grammar files were in the states shown in Figures 5.3 to 5.6. The directory that is being changed to 'Projects' has the structure shown in Figures 5.10 (below). After the command 'cd Projects/' is run, the dynamic grammars are in the states shown in Figures 5.11 to 5.13, with the new elements having been added. The phonetic dictionary file has also been updated, as shown in Figure 5.14.

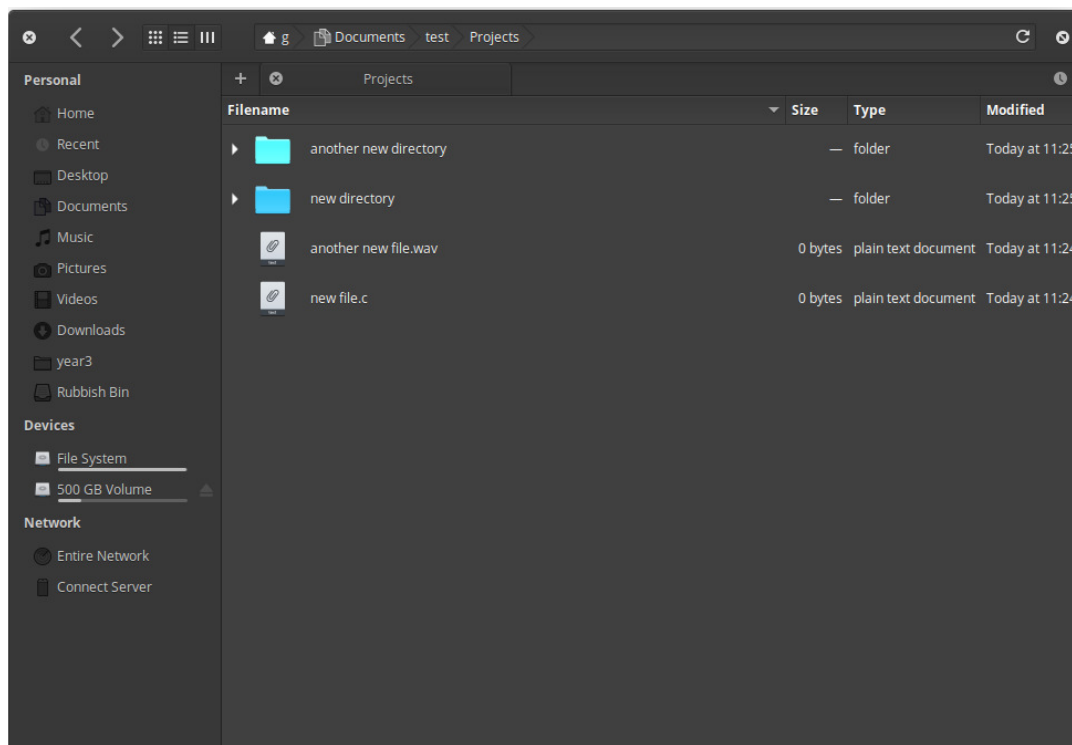


Figure 5.10 - Structure of directory that is changed into in test 4

```
grammars > ⌵ folders.gram
1  #JSGF V1.0;
2  grammar folders;
3  public <root> = (<rule0>|<rule1>|<rule2>|<rule3>|<rule4>);
4  <rule0> = PROJECTS;
5  <rule1> = DOWNLOADS;
6  <rule2> = PICTURES;
7  <rule3> = ANOTHER_NEW_DIRECTORY;
8  <rule4> = NEW_DIRECTORY;
```

Figure 5.11 - Screenshot of folders.gram after cd command, showing that the new directories have been added.

```
1  #JSGF V1.0;
2  grammar files;
3  public <root> = (<rule0>|<rule1>|<rule2>|<rule3>|<rule4>);
4  <rule0> = HELLO;
5  <rule1> = SECRET_PLANS;
6  <rule2> = WORD;
7  <rule3> = ANOTHER_NEW_FILE;
8  <rule4> = NEW_FILE;
```

Figure 5.12 - Screenshot of files.gram after the cd command, showing that the new file names have been added.

```
grammars > ⌵ exts.gram
1  #JSGF V1.0;
2  grammar exts;
3  public <root> = (<rule0>|<rule1>|<rule2>|<rule3>);
4  <rule0> = TXT;
5  <rule1> = PY;
6  <rule2> = C;
7  <rule3> = WAV;
```

Figure 5.13- Screenshot of exts.gram after the cd command, showing that the new file extensions have been added.

```
languages > cmd2 > master.dict
185  WORD  W ER D
186  TXT T IY EH K S T
187  PY  P AY
188  NEW_DIRECTORY  N UW D ER EH K T ER IY
189  ANOTHER_NEW_DIRECTORY  AH N AH DH ER N UW D ER EH K T ER IY
190  NEW_FILE  N UW F AY L
191  ANOTHER_NEW_FILE  AH N AH DH ER N UW F AY L
192  C  S IY
193  WAV W AA V
194  NEW_DIRECTORY  N UW D ER EH K T ER IY
195  ANOTHER_NEW_DIRECTORY  AH N AH DH ER N UW D ER EH K T ER IY
196  NEW_FILE  N UW F AY L
197  ANOTHER_NEW_FILE  AH N AH DH ER N UW F AY L
198  C  S IY
199  WAV W AA V
```

Figure 5.14 - Screenshot of the end of the phonetic dictionary file, with the new words having been added.

Unit Test 5 - Adding Aliases

Description: Test for the ability to add and use aliases, both in typed and spoken commands. This will be done by adding an alias for the command 'ls -o' as 'list-directory'. This will be done via the built in command 'majel-alias'. Another command, 'majel-update' will also be run, to generate the dictionary and grammar entries for the new alias.

Expected Result: After the alias has been added and the command 'majel-update' run, it will be usable in both typed and spoken commands.

Result:

After running the command 'majel-alias list-directory "ls -o"' shown in Figure 5.15, the new entry was added to alias.txt(Figure 5.16). After 'majel-update' is run, the alias is usable in typed commands (Figure 5.17) and spoken commands

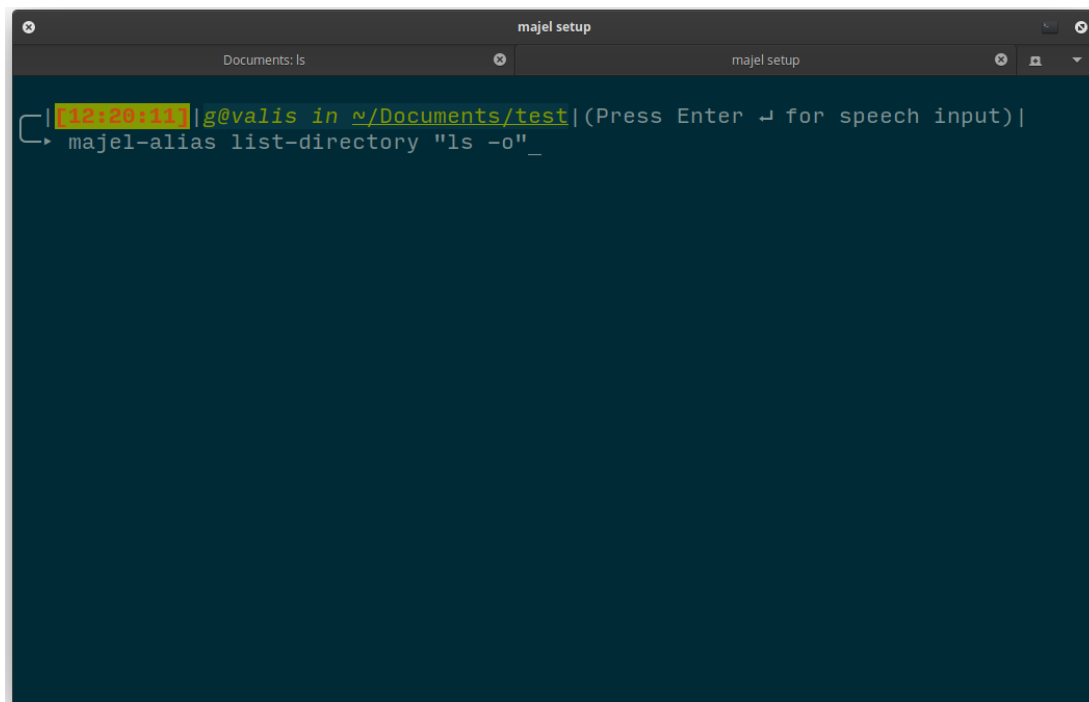


Figure 5.15 - Showing the alias adding command before it is executed.

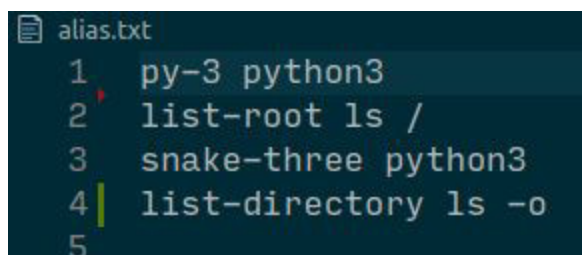
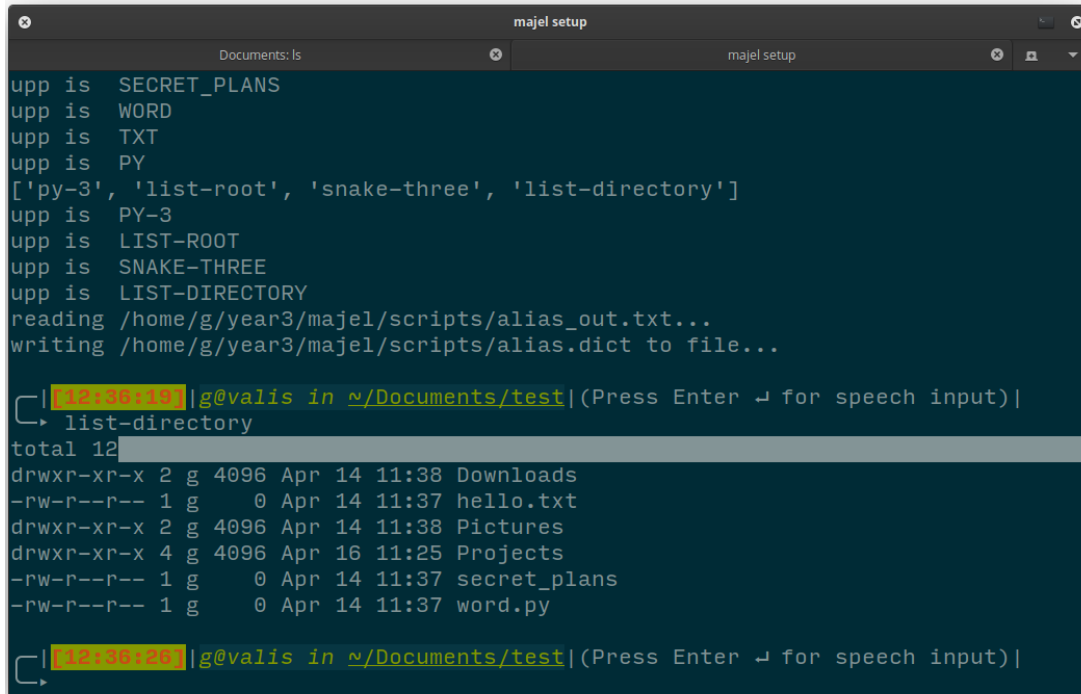


Figure 5.16 - Screenshot of alias.txt, showing the newly added alias.



```
majel setup
Documents: ls
majel setup

upp is SECRET_PLANS
upp is WORD
upp is TXT
upp is PY
upp is ['py-3', 'list-root', 'snake-three', 'list-directory']
upp is PY-3
upp is LIST-ROOT
upp is SNAKE-THREE
upp is LIST-DIRECTORY
reading /home/g/year3/majel/scripts/alias_out.txt...
writing /home/g/year3/majel/scripts/alias.dict to file...

[12:36:13] g@valis in ~/Documents/test | (Press Enter ↵ for speech input) |
list-directory
total 12
drwxr-xr-x 2 g 4096 Apr 14 11:38 Downloads
-rw-r--r-- 1 g 0 Apr 14 11:37 hello.txt
drwxr-xr-x 2 g 4096 Apr 14 11:38 Pictures
drwxr-xr-x 4 g 4096 Apr 16 11:25 Projects
-rw-r--r-- 1 g 0 Apr 14 11:37 secret_plans
-rw-r--r-- 1 g 0 Apr 14 11:37 word.py

[12:36:25] g@valis in ~/Documents/test | (Press Enter ↵ for speech input) |
```

Figure 5.17 - Screenshot showing the alias being used in a typed command.

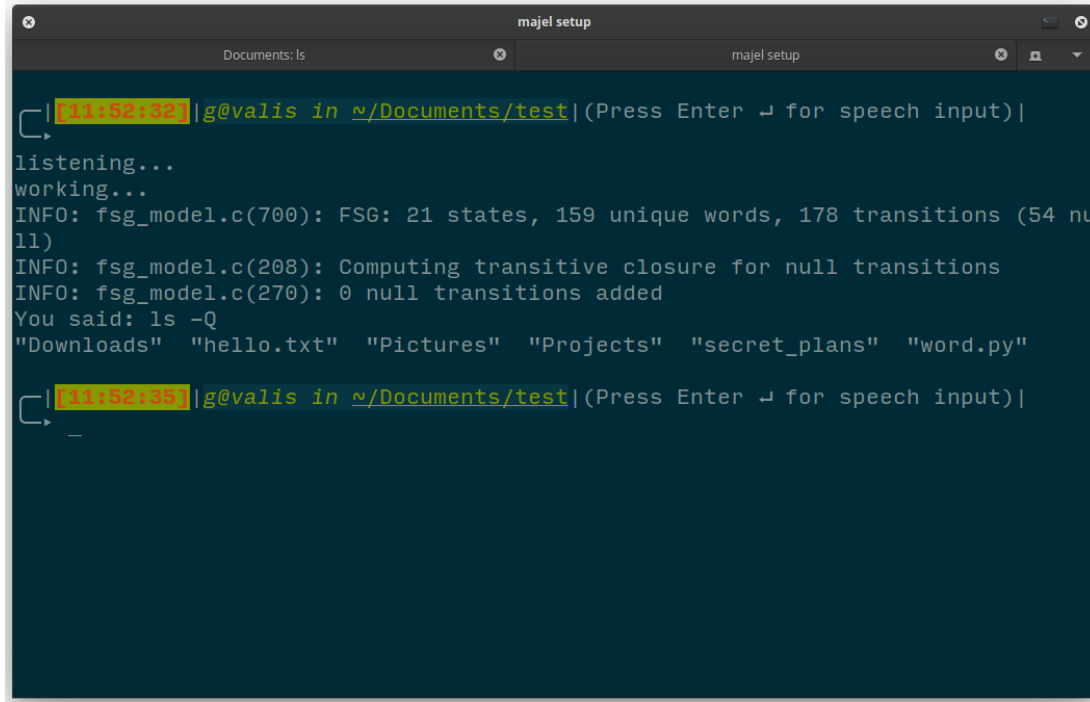
Unit Test 6 - Applying Modifiers

Description: Test for the ability for the system to detect and apply ‘modifiers’ in speech inputs; specifying the case of the next word spoken in the command. For example speaking the command ‘ls dash upper s’ will run the command ‘ls -Q’.

Expected Result: After the command ‘ls dash upper q’ is spoken as an input, the program ‘ls’ will be executed, with the -Q instructing the program to “enclose entry names in double quotes” [16].

Result:

The program prompted the user for a spoken command and ‘ls dash upper Q’ was spoken. The program then displayed the content of the current directory, with each element enclosed in double quotes.



```
lg@valis in ~/Documents/test (Press Enter ↵ for speech input)|
listening...
working...
INFO: fsg_model.c(700): FSG: 21 states, 159 unique words, 178 transitions (54 null)
INFO: fsg_model.c(208): Computing transitive closure for null transitions
INFO: fsg_model.c(270): 0 null transitions added
You said: ls -Q
"Downloads" "hello.txt" "Pictures" "Projects" "secret_plans" "word.py"
lg@valis in ~/Documents/test (Press Enter ↵ for speech input)|
_
```

Figure 5.17 - Screenshot showing the 'upper' modifier having been applied to the 'q' element in the command.

Unit Test 7 - Handling 'Dangerous' Commands

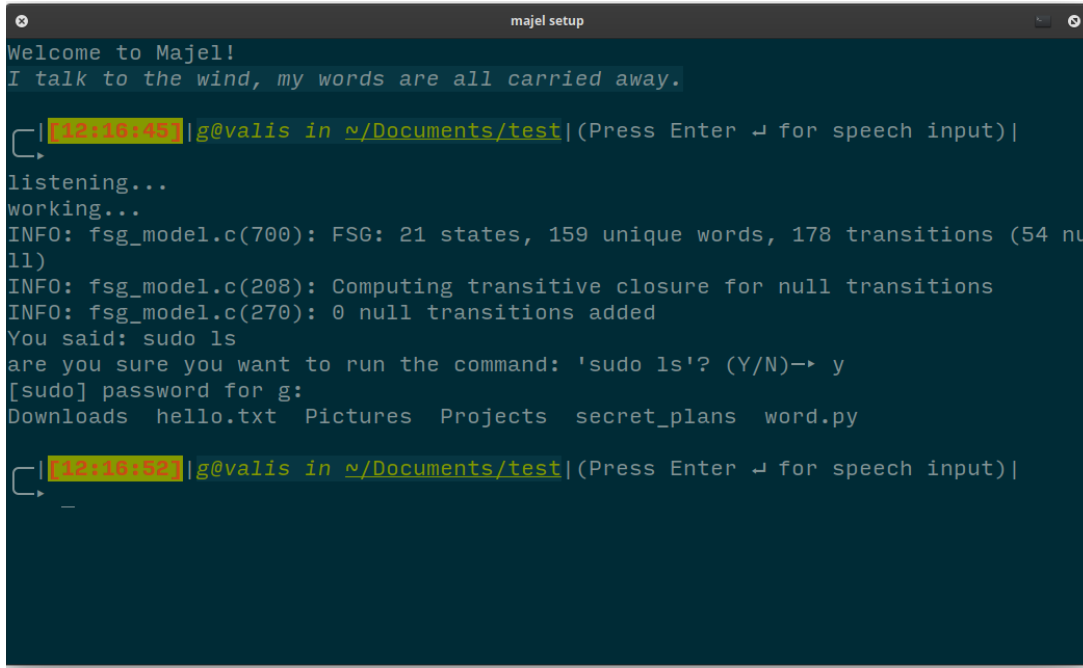
Description: Test to show that a warning prompt will appear when the user tries to run a command that could result in loss of important data or system instability. I have categorised dangerous commands as those that delete or move files and any command that is run as a superuser via 'sudo'. This is mainly to catch any incorrect transcription of a spoken command before it is executed. For this test I will execute the command 'sudo ls'.

Expected Result: After speaking the command, the program will display a yes or no prompt asking if the user is sure they want to run the command. If they input 'no' the command will not be run, if 'yes' then it will be.

Result:

The program prompted the user for a spoken command and 'sudo ls' was spoken.

The program then prompted the user if they were sure they wanted to execute the command, and 'y' was entered by the user. The command was then executed, first prompting the user for the superuser password.



```
majel setup
Welcome to Majel!
I talk to the wind, my words are all carried away.

[12:16:45] g@valis in ~/Documents/test | (Press Enter ↵ for speech input)
listening...
working...
INFO: fsg_model.c(700): FSG: 21 states, 159 unique words, 178 transitions (54 null)
INFO: fsg_model.c(208): Computing transitive closure for null transitions
INFO: fsg_model.c(270): 0 null transitions added
You said: sudo ls
are you sure you want to run the command: 'sudo ls'? (Y/N)→ y
[sudo] password for g:
Downloads hello.txt Pictures Projects secret_plans word.py

[12:16:52] g@valis in ~/Documents/test | (Press Enter ↵ for speech input)
_
```

Figure 5.18- Screenshot showing the yes/no prompt for a sudo command.

Unit Test 8 - Overriding Pronunciation

Description: Test showing the program's capacity for the user to add alternative pronunciations for words. This takes advantage of a feature of the online dictionary creation tool to include a 'hand' file that contains fixed pronunciations for words that must appear in the resultant phonetic dictionary. For this test, the pronunciation 'AA' will be added for the letter 'A'.

Expected Result: After the pronunciation has been added and majel-update invoked, the word will be recognised via the newly added pronunciation.

Result:

First the command 'majel-pronounce A' is run, listing all the pronunciations in the phonetic dictionary that contain the word 'A' (Figure 5.19). Then command 'majel-hand'

is run (Figure 5.20), adding a new entry to the hand.txt file (Figure 5.21). The new output of 'majel-pronounce A' is shown in Figure 5.22.



```
majel setup

WHOAMI  W OW M IY

RNANO   R N AE N OW

INTELLIJ-IDEA-COMMUNITY IH N T EH L AH JH AY D IY AH K AH M Y UW N AH T IY

SLASH   S L AE SH

DASH    D AE SH

CAPITAL K AE P IH T AH L

A       AH

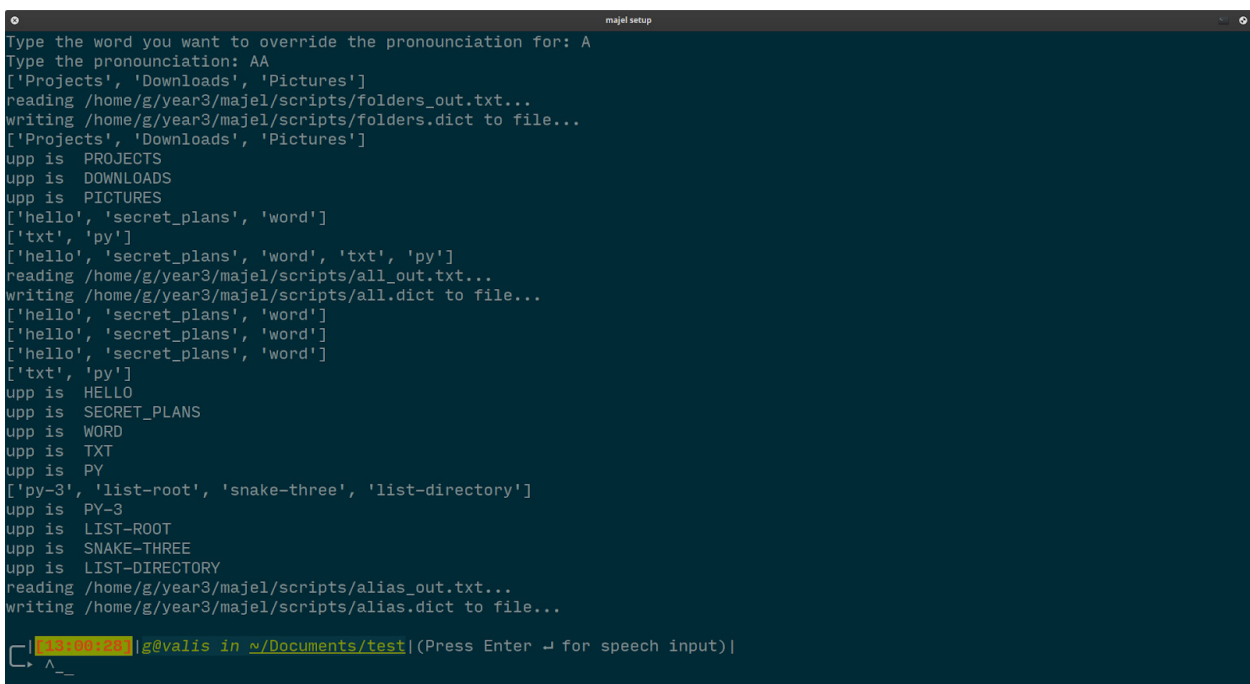
A(2)    EY

SECRET_PLANS  S IY K R AH T P L AE N Z

SNAKE-THREE   S N EY K TH R IY

18:46:27 |g@valis in ~/Documents/test|(Press Enter ↵ for speech input)|
```

Figure 5.19 - showing the result of the inbuilt command 'majel-pronounce A'

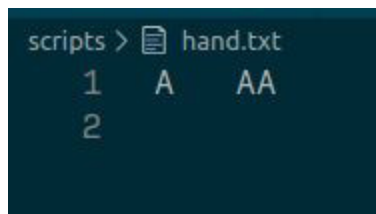


```
majel setup

Type the word you want to override the pronunciation for: A
Type the pronunciation: AA
['Projects', 'Downloads', 'Pictures']
reading /home/g/year3/majel/scripts/folders_out.txt...
writing /home/g/year3/majel/scripts/folders.dict to file...
['Projects', 'Downloads', 'Pictures']
upp is PROJECTS
upp is DOWNLOADS
upp is PICTURES
['hello', 'secret_plans', 'word']
['txt', 'py']
['hello', 'secret_plans', 'word', 'txt', 'py']
reading /home/g/year3/majel/scripts/all_out.txt...
writing /home/g/year3/majel/scripts/all.dict to file...
['hello', 'secret_plans', 'word']
['hello', 'secret_plans', 'word']
['hello', 'secret_plans', 'word']
['txt', 'py']
upp is HELLO
upp is SECRET_PLANS
upp is WORD
upp is TXT
upp is PY
['py-3', 'list-root', 'snake-three', 'list-directory']
upp is PY-3
upp is LIST-ROOT
upp is SNAKE-THREE
upp is LIST-DIRECTORY
reading /home/g/year3/majel/scripts/alias_out.txt...
writing /home/g/year3/majel/scripts/alias.dict to file...

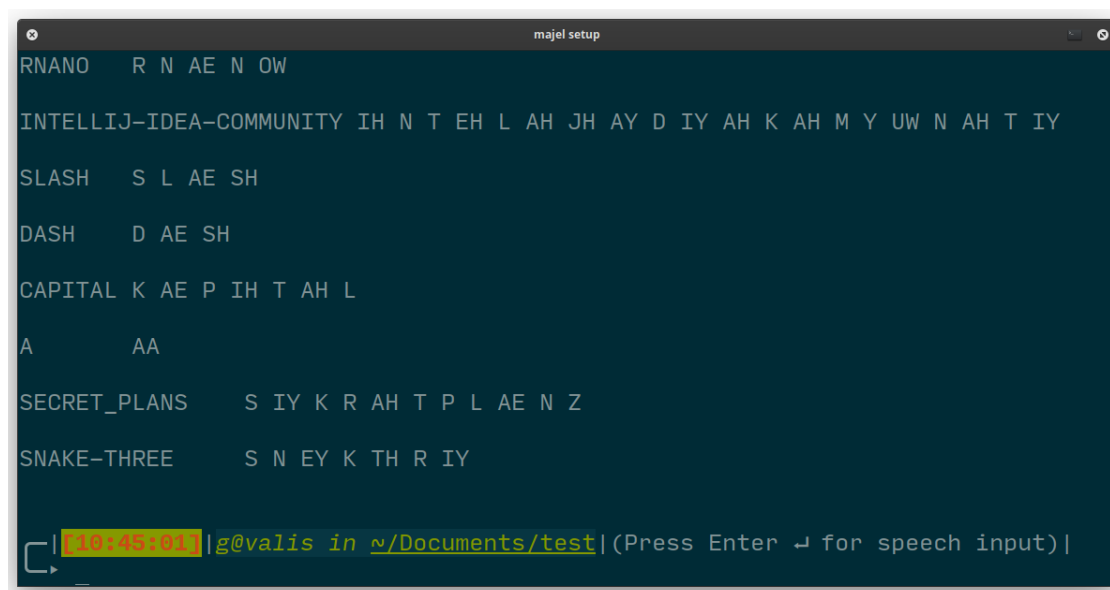
18:46:28 |g@valis in ~/Documents/test|(Press Enter ↵ for speech input)|
^_
```

Figure 5.20- showing the 'majel-hand' command and output.



```
scripts > hand.txt
1 A AA
2
```

Figure 5.21 - Screenshot of *hand.txt*, showing the newly added pronunciation.



```
majel setup
RNANO    R N AE N OW
INTELLIJ-IDEA-COMMUNITY IH N T EH L AH JH AY D IY AH K AH M Y UW N AH T IY
SLASH    S L AE SH
DASH     D AE SH
CAPITAL  K AE P IH T AH L
A        A AA
SECRET_PLANS S IY K R AH T P L AE N Z
SNAKE-THREE S N EY K TH R IY

[19:45:01] lg@valis in ~/Documents/test | (Press Enter ↵ for speech input)|
```

Figure 5.22- output of 'majel-pronounce A', showing the newly added pronunciation.

Usability Testing

In this section I will evaluate the usability of the program via a series of tests. To do this, I will run spoken commands of differing complexity and verbosity a number of times, and note the number of times that the program successfully interprets and executes the command. I will consider a test a success if more than fifty percent of the attempts are correctly processed.

I used a high quality headset microphone for these tests which were carried out in an otherwise silent room. I used the directory setup from the above section.

Ideally, I would have tried to have other people use the system and carry out these tests also, in order to get more accurate and varied data from a number of different voices, accents and audio recording systems. However, under the circumstances that this report is being written, that is currently impossible.

| Test Number | Command (<i>Spoken</i>) | Number of Successes | Test Status | Notes |
|-------------|---|------------------------|-------------|---|
| 1 | ls (" <i>ls</i> ") | 10/10 (100%) | PASS | |
| 2 | python (" <i>python</i> ") | 9/10 (90%) | PASS | In the failed test, 'update-grub' was run instead. |
| 3 | firefox -v (" <i>firefox dash v</i> ") | 10/10 (100%) | PASS | |
| 4 | nano hello.txt (" <i>nano hello dot txt</i> ") | 6/10 (100%) | PASS | Program did not produce any command at all in failed cases. |
| 5 | cd Projects/ (" <i>cd capital projects slash</i> ") | 10/10 (100%) | PASS | |

| | | | | |
|----|--|------------|------|--|
| 6 | list-root (<i>"list root"</i>) | 6/10 (60%) | PASS | 'list-root' is an alias for the command 'ls /' |
| 7 | sudo nano secret_plans (<i>"sudo nano secret plans"</i>) | 8/10 (80%) | PASS | In failed tests, the filename was mistranscribed. |
| 8 | ls -Q -a (<i>"ls dash upper q dash a"</i>) | 4/10 (40%) | FAIL | Program often fails to properly interpret the second argument, incorrectly representing the letter 'a' as another character. |
| 9 | ls ../ (<i>"ls dot dot slash"</i>) | 9/10 (90%) | PASS | In failed test, no command was processed, likely because the program did not detect the word 'slash' failing the grammar. |
| 10 | ls ../test/ (<i>"ls dot dot slash test slash"</i>) | 0/10 (0%) | FAIL | Bug in the program fails to correctly append 'test' to end of file path. |

Table 5.1 - Results of Usability Test

These results indicate that the program is largely usable. The two failed tests involve more complicated commands with several component parts, with test number 8 failing at the interpretation step(`get_command()`) and test 10 at the transformation step (`word_to_character()`). While the latter failure would be trivial to fix, requiring changes to a few lines, the former would require a lot more work on and testing of the

grammar structure. The bug in `word_to_character()` could be avoided entirely with a more sophisticated approach, which is detailed in the Conclusion to this chapter.

Conclusion

The results of the unit testing were overwhelmingly successful, with all of the core features of the program implemented and functional. However, there were some issues raised in the usability testing. There are some aspects that failed and that could be improved on a code level, which I will detail here.

The function that converts from the raw string of spoken input into a valid command, is currently somewhat messily written, with several edge case catching conditional statements, and a mix of different approaches. I have sometimes used a function that replaces each occurrence of a given element in the list, and other times I have looped through the command using conditional statements to replace elements as they are encountered; the ordering of these function calls and loops is what results in the bug that occurred in Usability Test 10. Given more development time, I would unify the method used, perhaps employing some sophisticated regular expressions to match and replace the command elements.

In the current implementation, the creation of the phonetic dictionary is done online, using a web service. While this does help offload some computation, increasing efficiency of the program, it does require that the system the program is run on be connected to the internet, and that the service remains available. Ideally, I would implement my own means of dictionary creation, that would be able to run offline.

I have used in the program a set of wrapper functions from the SpeechRecognition module to interact with PocketSphinx for Python. While these wrappers did ease development by drastically simplifying the process of getting the transcribed speech, there would be benefits of interacting with PocketSphinx directly myself. For example, it would be possible to hide the grammar transcription messages and I would be able to tailor the transcription process more finely to the context of the current directory.

While the grammar structure I have used is effective, it could be improved in a number of ways, as exemplified by the failure of Usability Test 8. Currently, any program can be run with any possible option and file or folder arguments, even if that program does not accept those options. It would be possible to fix this by implementing a dynamic grammar for each program, that would 'learn' what options that program could take via

analysis of history files and the manual pages for that program. I could also add a means for the user to add their own options ('update', '-help') outside of the commonly used ones ('-a', '-v', '-t') that are currently implemented. While I found that the language model approach was ineffective, I believe that it is ultimately a more sophisticated and effective approach; perhaps with access to more training data for the corpus file, a better result could be achieved.

6. Further Work

There are a number of ways that the project could be expanded, which I will detail here. Firstly, the program could support languages other than English. This would require obtaining an acoustic model for that language, and a means of creating the phonetic dictionary for that language. Given the online tool provided by CMU only supports the English language, some other means of creating the phonetic dictionary would have to be found. Otherwise, it would be relatively simple to implement, with the only change being the parameters passed to the `get_command()` function.

Another possible expansion to the program would be the implementation of more advanced features common to other shell programs, such as command piping and scripting. While this would not really be related to the core theme of speech recognition in the program, it would bring the program up to parity with other shells. It would likely be somewhat difficult to implement, and would mean that I would have to abandon use of the subprocess module and write my own means of running the commands from within Python, which might be beyond my abilities.

In order to make the program function more like existing 'voice assistant' programs, a 'wake word' system could be implemented. This would be a daemon process that would listen for a certain word to be spoken by the user; when this word would be spoken the program would launch, and start listening for a command. I would likely be able to implement this in a rudimentary way using the 'pocketsphinx_continuous' binary provided by pocketsphinx in a BASH script running as a background process. The BASH script would periodically monitor the output of the binary, and launch my program only when the wake word is present in the binary's output.

In the same vein as the previous improvement, some form of text to speech technology could be implemented. Currently, the user must observe the command window to see the output of their command which detracts from the 'hands free' nature of the program; synthesizing a voice to speak the output would alleviate this. The voice synthesis could also be used to show the user what the expected pronunciation of a given word is, helping to reduce errors in transcription. There is a python module 'pyttsx3' [17] that provides text to speech support, that would be useful in implementing this.

Another feature that would improve the program would be a 'smarter' system for transforming the spoken string into an executable command, that would take the context

of the current directory into account. For example if the command 'cd documents slash' was spoken in a directory that contains a subdirectory 'Documents' only, the transformation process would detect this and automatically reformat the string to match. By the same token, if the directory contains subdirectories 'Documents', 'DOCUMENTS' and 'documents', the program would prompt the user to clarify. This could also extend to the transcription process also, with file/folder names, file extensions that are present in the current directory being given more weight in the transcription than those not present in the current directory. This function is predicated on the idea that the user will be most likely to want to manipulate items within the current directory, which may not always be the case; as such, an option to disable this feature would be useful.

The concept of a formal grammar definition for a command line interface is currently unexplored. I have implemented a very simple version in my `command.gram` file, however I believe that there is potential for a much expanded and general solution. There has been some research done to employ genetic algorithms to infer grammar structure[18], rather than creating it by hand as I have here. This works via the analysis of source code (in this case this would be CLI history files) and the structure of an existing grammar (perhaps English) to infer the rules of the new grammar, effectively a much more sophisticated version of what I was attempting with the language model approach. This would require much more time and resources than I had available to me for this project, and far outside of its scope.

7. Reflections on Learning

In this section I will review what I have learned about each technology involved in the project, and how the process of development has proceeded as a whole.

When I first chose Python as the language I would use for this project, chief among the reasons was my familiarity with it, having used it in several prior projects. I found in development that it was uniquely suited for this task, with several useful modules already existing, such as the tools for interacting with pocketsphinx and for grammar creation. With that said, there were times when this was a detriment rather than an aid; for example the 'jskf' module is still very much in development and is currently poorly documented. As a result, some development time was dedicated to puzzling out its usage; in hindsight it would have been better to write my own method of creating and editing the grammar files.

When I started this project I had only a vague understanding of the mechanics of speech recognition. I now have a much greater knowledge of the component parts required such as the language model, phonetic dictionary and word search restriction structure. I also understand the limitations and challenges of the technology such as the need for good word search restriction structure, and the differences between different approaches to this. I also realised how simple and modular implementation of speech recognition could be, and that I will endeavor to include it in my future projects.

Prior to researching grammar as a means of word search restriction, I had some knowledge of language theory from the second year module 'Introduction to the Theory of Computation'. I have found it a very interesting and engaging topic, and hope to be able to work in this area in future, and hope to expand the concept of command line grammar.

I have learnt a lot about how CLI programs function, and how they interact with the host operating system. Having effectively written my own shell program for this project, I was surprised how easy it was to implement; I may endeavor in future to write my own shell program 'from the ground up' as a side project. My newfound knowledge of CLI will hopefully serve me well in the future.

I have found working on the project an exciting and engaging process. I was always motivated to develop the next part of the program, and to implement new strategies and

ideas towards solving the problem. Being able to use my own project proposal rather than select one set by a supervisor meant that I could tailor the project to my own interests and strengths. Another key source of motivation was weekly meetings with my project supervisor; this meant that I always tried to have something new to show for each meeting, allowing me to effectively roadmap what the work for the next week would be. Whenever a problem in development occurred such as with the language model approach or with interacting with BASH, these meetings were useful to discuss and resolve these problems. There were a few developmental 'deadends' chiefly the language model approach; these took up a lot of the development time that could have otherwise been used to implement some additional feature. In the future, more research and planning before starting the implementation of an idea might help reduce these issues.

References

1. "Gnu.org." Bash - GNU Project. Accessed April 27, 2020.
<https://www.gnu.org/software/bash/>
2. Bash Reference Manual. Accessed April 27, 2020.
<https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Simple-Commands>.
3. Shmyrev, Nickolay. "CMUSphinx Open Source Speech Recognition." CMUSphinx Open Source Speech Recognition. Accessed April 27, 2020.
<https://cmusphinx.github.io/>
4. Shmyrev, Nickolay. "Overview of the CMUSphinx Toolkit." CMUSphinx Open Source Speech Recognition. Accessed April 27, 2020.
<https://cmusphinx.github.io/wiki/tutorialoverview/>
5. Cmusphinx. "Cmusphinx/Pocketsphinx." GitHub, March 28, 2020.
<https://github.com/cmusphinx/pocketsphinx>
6. Shmyrev, Nickolay. "Before You Start." CMUSphinx Open Source Speech Recognition. Accessed April 27, 2020.
<https://cmusphinx.github.io/wiki/tutorialbeforestart/>
7. Shmyrev, Nickolay. "Basic Concepts of Speech Recognition." CMUSphinx Open Source Speech Recognition. Accessed April 27, 2020.
<https://cmusphinx.github.io/wiki/tutorialconcepts/>.
8. JSpeech Grammar Format. Accessed April 27, 2020.
<https://www.w3.org/TR/2000/NOTE-jsgf-20000605/>.
9. "Pocketsphinx." PyPI. Accessed April 27, 2020.
<https://pypi.org/project/pocketsphinx/>.
10. "PyAudio." PyPI. Accessed April 27, 2020.
<https://pypi.org/project/PyAudio/>.
11. "Pyjsgf." PyPI. Accessed April 27, 2020.
<https://pypi.org/project/pyjsgf/>.
12. Shmyrev, Nickolay. "Building a Language Model." CMUSphinx Open Source Speech Recognition. Accessed April 27, 2020.
<https://cmusphinx.github.io/wiki/tutoriallm/>.
13. "Sphinx Knowledge Base Tool -- VERSION 3." Sphinx Knowledge Base Tool VERSION 3. Accessed April 27, 2020.
<http://www.speech.cs.cmu.edu/tools/lmtool-new.html>.
14. Jurafsky, Dan, and James H. Martin. *Speech and Language Processing: an Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, NJ: Pearson Prentice Hall, 2009.

15. "LOGIOS Lexicon Tool." CMU Lexicon Tool. Accessed April 27, 2020.
<http://www.speech.cs.cmu.edu/tools/lextool.html>
16. ls(1) - Linux manual page. Accessed April 27, 2020.
<http://man7.org/linux/man-pages/man1/ls.1.html>
17. "pytttsx3." PyPI. Accessed April 27, 2020.
<https://pypi.org/project/pytttsx3/>
18. Penta, Massimiliano Di, Pierpaolo Lombardi, Kunal Taneja, and Luigi Troiano.
"Search-Based Inference of Dialect Grammars." *Soft Computing* 12, no. 1 (2007):
51–66.
<https://doi.org/10.1007/s00500-007-0216-5>