Sean Mullan

For this project I used the Carnegie Mellon University Pronouncing Dictionary. The dictionary is made up of over 130,000 words and their phonemes. The phonemes come from a set of 39 possible sounds based on the ARPAbet symbol set, and each one also can be labeled as either 0,1, or 2 to show the lexical stress given to that sound.

In addition to the Carnegie Mellon dictionary, I also used two sets of 39 audio files that I created with speech from words that I said as well as words that my roommate, Adron Mason, said. The final version of the project uses my phonemes, but his are included in the project file. All of the phonemes are saved in .wav format to make it easier to use with Java, and the files were created using Audacity.

The overarching goal of my project was to take in text as input and then output an audio version of that text. To do this, there were several steps that I needed to complete. First, it must parse any input into a set of phonemes that give a representation of what is going to be spoken. Second, it must combine a preexisting library of phoneme sounds in order to create the final output. Lastly, it will output both the text representation of the phonemes as well as the audio.

Parsing the initial input into individual words was the very first step. I am currently using a basic user interface, so they are to input text into a large, labeled text box, and that is what gets input to the rest of the program. The input, once the user clicks the start button, is stripped of all punctuation. The new text then gets split up into individual words. The words are each processed through a program that goes through a dictionary that was made from the CMU Pronouncing Dictionary. The dictionary is simply broken up into the single string that represents the word being searched for and the array of strings that represents the list of phonemes making up the sounds of the word.

If any of the words are not in the CMU dictionary, they then get passed to a spell check program. The spellchecker is the same one that I made with Matt Cotter in Python for an earlier assignment; I just had to translate it into Java. The spellchecker starts by taking every word in the CMU dictionary that starts with the same letter as the word being checked, then it shortens the list to words that within five characters of the length of the checked word. Next it finds the edit distance between each remaining word using a substitution cost of five and an insertion and deletion cost of three. From here, the remaining words are sorted into list by their edit distance, and up to the best six are picked. If there are less than six, the program just takes whatever it can get. The original word is then added to the list with the tag "<No Change>" added to the end. If there are no other possibilities then the original word is added to the empty list with the tag "<No Alternatives>". This is presented to the user, and they then pick the correct spelling. If they pick the original word, the word then gets passed to a best guess parser.

The best guess parser starts by parsing off the tag that was added onto the word. After this, the word is passed through a best guess formula that looks at each letter individually. The first letter is treated as a special case and is looked at individually, but the phoneme associated with it can be changed depending on the next letters. Each letter after the first is looked at in context with the previous and next letters, so if the first letter was C and the next letter is H, the "K" phoneme that was added will be removed in exchange for a "CH" phoneme. Additionally, there are some cases where the position of the word in the sentence gets looked at. This is for cases like GH, where it takes the "G" sound at the beginning of words (ghost) and the "F" sound anywhere else (enough). After the unknown word gets parsed, it is put back with the rest of the other words, now all broken into phonemes.

After an array has been built of arrays of phonemes for each word, this is passed into a class that handles the speech. This class holds a dictionary that is made up of each phoneme paired with the file that holds the related audio. Once the array of arrays is input to this class, the class pulls out each file needed on a word-by-word basis and concatenates them into a single AudioInputStream that gets played through a Clip.

The order in which this process occurs is of surprisingly importance, since this was the single biggest issue I had in the development of this project. Originally, I had been changing each file into an AudioInputStream first and putting these into the dictionary. This led to the issue of each phoneme only being played once. For instance, the word attack would sound like atck with the second AH sound being lost. This was due to an issue involving internal pointers within the AudioInputStream class. Creating each AudioInputStream only as I needed solved all of these problems, so the current implementation doesn't have any issues on this side.

After each word gets played, the next one is loaded after a 100ms delay to simulate the pause between words in a sentence. Once the final word gets played, the array of phoneme arrays gets converted into a single string. The string is passed back to the user interface that originally took the input, and it prints it into a second text box that is right below the first.

Other than the issues with audio output that I had, the only other issues that I had were with the phonemes. The phoneme sounds that I had gotten from my roommate were very choppy sounding in the final outputs. I rerecorded the phonemes, taking more care to get longer files for each phoneme, and this helped a bit, but I think that the problem stems mostly from how I was handling the phonemes.

The success rate of the program I would say is pretty good. The success for the parsing is rather high, since the CMU corpus is massive. With unknown words it is usually correct, but there are many ways that letters can affect the sounds around them, so I definitely cover all of them in my rudimentary rule set. The success for the audio is pretty good as well. It is not any where close to being clear, but I would say that it is comprehensible a fair bit of the time.

If I were to go about improving this project further I would probably improve flow between phonemes and more rules for the unknown word parser. The current audio is only made up of unigram phonemes, so the flow from word to word is not very clear. Unfortunately, to get every possible combination of the 39 phonemes with a following phoneme would be very time consuming. So, I am considering playing with the volume at the edges of the audio files, or make it so that the AudioInputStreams overlap, in order to give a smoother feel to the audio. The additional rules for the parser go without saying. There are far more rules than I have thought of already, so I will be adding them as I encounter them in the future.