Summary

Our program is able to perform simple conversations with a user. It takes in an input from the user, breaks it down using keywords and decomposition rules, and uses this to construct a response. This allows the program to simulate a conversation with the user by generating a response to each of their inputs.

When the program is run it reads and stores information on how it should generate responses to different inputs from an external text file. It then repeatedly allows the user to enter text inputs, and will respond to each input from the user, until a quit command is given by the user, at which point the program ends.

We made 3 scripts, a psychotherapist, a politician and for our own personality, we chose a motivational speaker, who will attempt to give positive encouragement to the user.

Design

When designing the program, we chose to split the functionality into 4 classes, as we felt this would make the program easier to understand. These were the Script Reader, Decomposition Rule, Keyword and Eliza classes.

- The Eliza class contained a main method from which the program would be run.
- The Script Reader's purpose was to provide methods to read and store the information from the script files, as well as providing methods to make use of it.
- The keyword class was used to store information on a keyword, including the keyword and its decomposition rules.
- The Decomposition rule class was used to store information on a decomposition rule, including the rule itself, and its recomposition rules.

Eliza class

The Eliza class only contained the main method that would be used to run the program. The method took in a single argument, the file location of the script file to be read.
The method first creates a Script Reader object with the file location to read. It then enters a loop which allows the user to enter an input and generates and prints a response to that input. This loop is broken if the user enters one of the quit commands in the script file. After this loop breaks it displays the goodbye message and exits.
This method works at a very high level by almost entirely uses methods from the Script Reader object.

Script Reader class

Probably the most significant class was the Script Reader class, which handled reading, storing and using the information within the script files.

We decided to store the information from the script file as attributes of this class:

- The welcome and goodbye messages were stored as strings, since there was only one of each message
- The quit commands were stored as a set, since the number of commands could vary in size, and we needed to efficiently check if the user input was contained within the quit commands.

- Pre and post substitution rules were contained in maps, with the key being the word to replace, and the value being the replacement word(s). We chose this as it allowed a varying number of rules, and we could easily check if a word should be replaced or not.
- The keywords, decomp and reassembly rules were stored as list of keyword objects, as it allowed a varying number of keywords

We decided to create separate classes to be able to store keywords, decomposition rules and recomposition rules within the Script Reader class.
Another option we considered for this was using ArrayLists of ArrayLists to store the information, but this seemed horrible and would make reading/writing/modifying the code very difficult. This is because each keyword can have multiple decomposition rules, which can in turn have multiple recomposition rules, leading to very complex lists.
It seemed easier to create an object for each keyword (which would contain its decomp and reassembly rules) and store a list of these objects instead.

When a Script Reader is initialised, the file location of the script is passed into the constructor, which then reads the information from the file and stores it in the attributes above.
This method was designed very closely with the script file, as the syntax of the scripts was very important to how the constructor would function.

The file format that we settled on was that the first character of each line would be a 'denoter character'. Which would denote what information the line contained, for example '&' represented a keyword and '@' a quit command. The only exception to this was the first 2 lines were always reserved for welcome and goodbye messages.
This meant that the position or order of information did not matter. The only ordering that did matter was the order of keywords, decomp and reassembly rules. As when the constructor encounters a decomposition rule, it assumes it relates to the most recent keyword, and that a recomposition rule relates to the most recent decomp rule.

Additionally, it is assumed that all the keywords and decomposition rules are ordered by priority, with the highest first.

The constructor also ignored excess spaces, as well as empty lines and lines not beginning with one of the denoter characters (allowing us to add comments).

This format meant that the script files were easily human readable and writable, which was important because we were writing them manually.

We briefly considered using a standard format such as JSON or XML, however we decided it would be more code for the program to read the information in this format. Also, as we would be manually typing the script files, using a stricter format would have made the files much more error prone when being typed up, and much harder to be read by a human.

As well as the attributes and constructor, the Script Reader class contained various getter and setter methods, as well as other simple methods you would expect (like addQuitCommand() to add a quit command to the set).

We decided to put the method that generates a response to the users input in the Script Reader class. This made sense as the method's primary tasks involved matching the users' input to the correct keyword and decomposition rule, both of which were stored in this class. The main steps of the method were:

- Apply pre-substitution
- Find the right keyword
- Find the right decomposition rule
- Pick a random recomposition rule
- Assemble output
- Apply post-substitution

To keep the code neat and modular, all these steps were put into their own methods, except for assemble output. This made what was otherwise quite a large and complex method much smaller and more readable.

Keyword class

We decided to make this class in order to keep the program simpler.

Originally this class stored a string of the keyword it represented, as well as a List of all the decomposition rule objects that the keyword had. However, it was later updated to store an array of keyword strings instead of a single string. This was to allow the storing of slight variations of the keyword (e.g. "believe", "believes").

The idea behind this class is that instead of having to store a list of keywords and a 2d list of decomposition rules, each object of this class represents a keyword, and stores as an attribute the list of decomposition rules. This allows the Script Reader class to store just a list of these objects instead.

This class also contains several getter and setter methods, as well as a method to find the most appropriate decomposition rule to the user's input.

Decomposition Rule class

We decided to make this class in order to keep the program simpler.

This class has attributes to store the string decomposition rule that it represents, as well as a string list of recomposition rules that can apply to this decomposition rule.

This class works very similarly to the Keyword class and contains a method to select a random recomposition rule from the list.
As any recomposition rule can be used from the list to reassemble the input, we decided one effective way to ensure varied responses from Eliza was to have the method select a random recomposition rule. An alternative to this would be to have it cycle through all the different recomposition rules, but there are no major advantages to either as they are very similar.

Sample Dialogues

Here are some sample dialogues of the Eliza bot

Eliza therapist script:

```
Hello there!
I feel sad
Eliza: hmm, why do you feel sad?
i need to complete all of these maths problems
Eliza: why do you need to complete all of these maths problems?
so i can get a good grade
Eliza: i think that is good.
but i hate spending all night doing maths problems
Eliza: hating can be quite a damaging thing in life, is there anyway you could let it go?
but i find maths really hard
Eliza: i find maths really hard too.
```

Eliza politician script:

```
[owen@localhost src]$ java Eliza politicianScript.txt
Hello there Citizen!
Sir what are your opinions on brexit?
Eliza: i represent freedom, justice and all the things citizens like you need.
will you raise taxes if elected
Eliza: that's a good question, anyway
who are you and why are you running for office?
Eliza: i am but an average person like you, looking for change, to how things are run.
why should we vote for you?
Eliza: because i can help in these unprecedented times.
but these times aren't unprecedented, we are doing better than we have ever done
Eliza: that's a good question, anyway
what are you going to do about climate change?
Eliza: i represnt the needs of the people.
will you help the children in our society get an education
Eliza: what do you need help with?
eduction
Eliza: can you elaborate on that?
what do you promise to do about educational policy if elected?
Eliza: i promise to do everything in my power to help with educational policy if elected?.
Jorris Bohnson, what are your thoughts on tax?
Eliza: i represnt the needs of the people.
quit
Goodbye, have a nice day and vote for Joris Bohnson!
[owen@localhost src]$
```

Eliza motivational speaker script:

```
[owen@localhost src]$ java Eliza motivationalSpeaker.txt
Greetings friend! I hope you are having a good day!
i feel sad
Eliza: if you don't like feeling sad, try being happy!
i haven't had a good day
Eliza: woah! its so cool that you have not had a good day!
i hate doing maths problems
Eliza: you should try letting go of your hatred of doing maths problems, it could make you much happier
i think you might be right
Eliza: hmm thats a very interesting thought!
i don't know how to solve this quadratic equation though
Eliza: i'm not quite sure i understand, i believe in you!
it's very difficult
Eliza: i hope everyone feels positive!
i don't think i can do it
Eliza: i do like to think about things, i am sure you are very good at thinking
quit
Have a happy fun smile time!!!
[owen@localhost src]$
```

Bugs/defects

We do not believe there are any major bugs or defects in the program.
Some issues we know about are:

- If the script is formatted incorrectly, e.g. a keyword has no decomposition rules, the program will crash
- For a decomposition rule that is * or empty, putting 0 or 1 in the recomposition rule will display the 0th or 1st character, rather than the text before/after the decomposition rule as intended
- Eliza will throw an exception if the location is incorrect

Mercurial Repository link: https://orsm1.hg.cs.st-andrews.ac.uk/Eliza