CS1006 P2 - Eliza

190022658 & 190017146

1 Overview

In this practical, we were asked to create a chatbot program that allowed a person to talk to a program. Note that this program is not perfect as it heavily depends on the script. In addition to the practical specifications, we also implemented a main menu and a “ScriptWriter” which helps write a script which the engine understands.

Welcome to ELIZA!

Who would you like to talk to?
1. A psychotherapist
2. A tech support guy
3. Chandler Bing
4. Write your own script!
5. Exit

Example 1: Main Menu

The practical required the program to have three options/types of “people” to talk to. The first, a psychotherapist, and the second one was supposed to be a tech support guy. We had a choice for the third one. We chose Chandler Bing.

Chandler Bing is a character from the american sitcom, Friends. Portrayed by Mathew Perry, Chandler Muriel Bing is noted for his sarcastic and witty sense of humour. This sense of humour is, in a limited manner, replicated in the program.

2 Design

The program is divided into three classes - **ElizaEngine, ScriptWriter, and MainMenu**, as there are three aspects to the implementation. The purpose of each class is suggested by the name.

Nevertheless, the **MainMenu** deals with the user and their input, the **ElizaEngine** then processes the input and provides the **MainMenu** the output which is then printed onto the console. If the option to *“Write your own script!”* is selected then an instance of **ScriptWriter** is created instead of the engine. This class creates a JSON file which can be interpreted by the engine in this program.

The **MainMenu** class contains only one import statement, to import Scanner.

The first method, **askForPerson(),** returns an **int**. This method asks the user for what the user wants to do. It prints the text as in *Example 1*.

The next method, **startEngine(),** returns **void** and takes an **int** as argument. The integer is from askForPerson(). The integer specifies the user's choice. There is a switch statement inside the method that initialises **ElizaEngine** or **ScriptWriter,** and opens the specific file if initialising the engine.

The **main()** method executes both the above method.

```
[
	{
		“Initial Sentence”: “Some sentence”
	}
	{
		“Keyword1”: {
			“DecompositionRule1” : {
				“0”: “ReassemblyRule1”
				“1”: “ReassemblyRule2”
			}
			“DecompositionRule2”: {
				“0”: “ReassemblyRule1”
				“1”: “ReassemblyRule2”
			}
		}
		“Keyword2”: {...}
	}
	{
		“0”: “DefaultResponse1”
		“1”: “DefaultResponse2”
		…
	}
]
```

Example 2: Sample JSON script

When the **ElizaEngine** is initialised, the constructor accepts the file name as argument.

The constructor uses the methods, **fileIntoJsonArray()** which reads the file and converts it into a JSONArray, and **getKeywordMap()** which makes a map linking keywords to a another map

which links decomposition rules to an array list of reassembly rules. We decided to use a map instead of using the JSON Objects and Arrays because a map offers more methods.

Then there is the method **printReply()** which controls everything happening in this class. It first uses the method **getInitialSentence()** which returns a string which is supposed to be the first sentence that the chatbot is supposed to say. To get more of an idea, refer to *Example 2* which is a structure of JSON files that the program understands. There is then a loop which ends only when the user enters “bye”. The loop invokes another method called **reply()**.

The **reply()** method accepts a string as an argument which is the user's input. The string is then cleaned and split into a list of strings. Each string in the list is a word. This makes it easier to look for a keyword. Here, if there is no decomposition rule for a specific keyword for the sentence, the key word is added to a set called invalidKeywords. There is a method called **checkforKeywords()** which takes all the keywords, the words of the sentence and invalidKeywords as its argument. It uses nested loops to find a common keyword between the words of the sentence and the list of keywords (which is not in the invalidKeywords list). There is a loop which controls this until either a keyword is found or there is no keyword left. If there is no common keyword or no keyword with a proper decomposition rule, a default sentence is printed out (hilarious ones in Chandler script). If there is a proper decomposition rule for the keyword, the method **reassembler()** uses the decomposition rule and reassembles the sentence to form a sentence to print for the user.

```
“remember”: {
        “i remember *”: {
                “0”: “do you often think of 0”
        }
}
```

Example 3: An example keyword, decomposition rule, and reassembly rule

The **reassembler()** was a bit tricky. Take *Example 3* for an example. This part of the script helps if say the user inputs a sentence like “i remember how she said goodbye”. If this sentence is entered by the user, we want the program to store “how she said goodbye” so that it takes place of zero in the reassembly rule. Here zero means the first occurence of a star. The final output should be, “do you often think of how she said goodbye”.

Now to implement this by first initializing a List<List<String>> called stars. This will be quite useful later on. After this the program runs a for loop for all the decomprules. Inside this loop the program declares and initializes an integer which keeps track of the position of the word in the sentence entered by the user. There is a boolean which turns false if the decomprule is not right for the particular sentence. There is another loop inside which goes through all the words in the decomposition rule. If there is a star in the decomposition rule, and if the star is at the beginning or middle of the decomprule then a while loop runs and stores all the

words into a list until the next word is reached in the decomprule. If the star is at the end, a while loop runs and stores all the words in a list until the size of the sentence is finished. At the end of this the list is added to 'stars' which is a list of lists of strings. Here, if the list is empty, it means that the decomprule is not correct for the sentence. If the word is not a star the program simply checks if the word in the decomprule and user's sentence is equal. If it's not the decomprule is not right.

After the nested loop finishes, and if the boolean is still true, then an if statement breaks from the other loop. This means that there is a suitable decomposition rule for the statement. If there is a suitable decomprule, its time for reassembly. Since there can be multiple reassembly rules for one decomposition rule, we choose one randomly (using Random). The program then breaks the reassembly rule into an array of words. Finally there is a loop which goes through all the words, if it is a number, adds *'stars.get(number)'* to a string, otherwise just adds the word from the reassembly rule. The method finally returns the string. It returns "NSDR" if there is "No Such Decomposition Rule" for the keyword.

The last class, **ScriptWriter,** has a constructor which initializes Scanner.

A method called **createScript()** connects the methods in this class to the **MainMenu**. The program in gist, helps create a JSON file as in *Example 2*.
The first thing that the Script Writer does is to ask the user for the initial sentence. As for all inputs in the Script Writer, a scanner subclass is used, **ScriptWriter()**. The input is then checked to see if 'xx' was entered which would indicate the user would like to stop entering further values. If 'xx' is used when the program is asking for an initial sentence, it is assumed the user no longer wishes to create a script and the program stops.
All inputs from the user are stored as JSONObjects and eventually added to the JSONArray, **script**.
The next step is collecting keywords from the user. Each keyword has at least one associated decomposition rule which each has at least reassembly rule.
A while loop is used to collect keywords until 'xx' is entered. After a keyword is input, a similar while loop is used to collect decomposition rules inside this loop, a for loop to add new reassembly rules. Each reassembly rule for every decomposition rule is then added to the **rules** JSONObject which is then added to the **keyword** JSONObject.
Keyword objects are then finally added to the **script** JSONArray.
The **writeScript** method is then used after asking the user to input a filename. '.json' is automatically added to the end of the file name if not already included by the user.
The **script** is then written to the JSON file.

3 Testing

Welcome to ELIZA!

Who would you like to talk to?

1. A psychotherapist
2. A tech support guy
3. Chandler Bing
4. Write your own script!
5. Exit

3 *<return>*

Hi I'm Chandler, I make jokes when I'm uncomfortable.

hello *<return>*

Yes?

where is Henry *<return>*

Henry ate my last piece of gum so I killed them, do you think that was wrong?

i hate you *<return>*

Gee thanks

dude im tired *<return>*

You know, Donald Duck never wore pants. But when he gets out of the shower he always puts a towel around his waist. I mean, what is that about?

Testing 1: Chandler

Note: Most of the other scripts work the same way, but have different responses.

Welcome to ELIZA!

Who would you like to talk to?
1. A psychotherapist
2. A tech support guy
3. Chandler Bing
4. Write your own script!
5. Exit

4 *<return>*

Enter xx to terminate
Enter the first sentence:

Hello! How may I help you? *<return>*

Enter a keyword:

sorry *<return>*

Enter a decomposition rule:

sorry *<return>*

Enter reassembly rules:

Please don't apologise *<return>*
What feelings do you have when you apologise *<return>*
Apologies are not requires *<return>*
xx *<return>*
Enter a decomposition rule:

sorry * *<return>*

Enter reassembly rules:

Please don't apologise *<return>*
What feelings do you have when you apologise *<return>*
Apologies are not requires *<return>*
xx *<return>*

Enter a decomposition rule:

xx *<return>*

Enter a keyword:

xx *<return>*

Enter default sentence:

Please elaborate

Enter the file's name:

Test

Testing 2: Writing a script

```
[
	{
		"Initial Sentence": "Hello! How may I help you today?"
	},
	{
		"sorry":
		{
			"sorry":
			{
				"0": "Please don't apologise",
				"1":"What feelings do you have when you apologise",
				"2":"Apologies are not required"
			},
			"sorry *":
			{
				"0": "Please don't apologise",
				"1": “What feelings do you have when you apologise",
				"2":"Apologies are not required"
			}
		}
	},
	{
		“0”: “Please elaborate:”
	}
]
```

Testing 3: Test.json

4 Bugs

Even though the program works in a satisfactory manner, sometimes the responses might not make sense. For instance, it is possible that the program uses wrong pronouns/tense.

5 Conclusion

We’ve both really enjoyed working on the challenges that the ELIZA project has presented, despite the unprecedented difficulties we experienced.
As will be discussed in both of our individual reports, the Coronavirus outbreak in the UK seriously affected our work and restricted the time and resources that we had in previous projects. However, with this said, we took the challenges in our stride and continued to work on creating the Eliza engine, 3 scripts as well as going further by implementing a menu and a script writer.
The difficult thing about this practical was having no defined end point. We had to acknowledge that we simply wouldn’t be able to create a perfect English speaking and understanding chatbot but gave it our best attempt to ensure our 3 characters came across as natural as possible.

Overall, we’re incredibly proud of our work for this project, we worked well together, learned many new things and overcame the challenges of having to work from home (Glasgow and Nepal) at short notice.