Overview

For the third practical we were required to build a Java implementation of a finite state automaton interpreter. The program should be designed to take in two arguments, the first, an FSA definition and the second the data to be checked against the definition.

Design

All comments are in the extension

For this practical I decided to create three classes which include: a class that represents an FSA state, a class that handles the input definition and creates the FSA state objects and a main class.

The first of the three classes is the "FSADefinition" class, which contains two methods "createDefintion" and "setAcceptingStates". The "createDefintion" method first of creates a linkedHashMap which maps the input to a list of "FSAState" objects. The reason I have chosen to do it this way is so the program recognizes the same input and does not store multiple instances of the same input. The method then reads in the input definition file line by line and for each line creates a new FSA definition object. Next the method checks to see if the line points to an accepting state, if so the accepting state is added to a list. Finally, the method checks if the input character has already occurred in the linkedHashMap, if so the FSAState object created is added to the appropriate list in the linkedHashMap, otherwise the new input character is added to the linkedHashMap and a new list is created containg the new FSAState object. Once the method has finished reading in the file the "setAcceptingStates" method is called, here the program loops through all FSAstate objects in the linkedHashMap and if their "getTrasinitionState" attribute matches a value in the "acceptingStates" list, the "setAcceptingState" attribute is set to true.

The "Main" class is the class which is responsible for dealing with the input file. There are two methods in the main class which are: "startJob" and "getInput". The first method "startJob" creates a FSADefintion object and then generates the linkedHashMap containing the definition. Next it reads the input file in line by line and each line is split up into an array of characters. It should be noted at this point any spaces in the input file are removed. For each character in each line the "getInput" function is called. This function uses the linkedHashMap to check that the current character can proceed the previous character. If there is an invalid character the "getInput" function returns -1. The program also checks if the current character is the last character in the array and therefore makes sure that the last character is an accepting state.

For this practical I have decided to create a GUI, this is simply because it enforces validation and verification and generally makes the program more user friendly.

When the program is first run an instance of the GUI class initiated on a new thread and the graphical user interface is displayed. The GUI is relatively basic and only contains a few components. The first components are the "jButtonGetDefinitionFile" and the "jButtonGetInputFile" buttons. These buttons are used to allow the user to select the directory where the input data is located and the directory where the FSA definition is stored. To ensure valid directories are input I have used a "JFileChooser" component, which will force the user to select a file, and the directory of the selected file is stored. If a directory is not selected a warning message is displayed and the program will not continue. For ease of use I have set the default directory of the "JFileChooser" component to the directory the program is located in.
The final components are the "jButtonCheck" and the "jTextAreaOut". The "jButtonCheck" buttin calls the "startJob" method in the main class if two valid files have been selected. Once the main method is run each line of the input file is displayed in the "jTextAreaOut" component with an added message notifying the user if the line was valid or not.

For an extra extension I have included functionality which allows the user to select multiple FSA definitions and the machine will automatically change the FSA definition appropriately. To do the user must hold control and click the FSA definitions they wish to use in the program then select open. A linkedHashMap containing the definition is created for each selected definitions and the linked linkedHashMaps created are added to a list in the Main class. The file which is selected first by the user is set by default to the static attribute "currentDefintion". This means that the program will use this FSA

definition first to attempt to check the input text. When the program then encounters a character in the input file which is unrecognized or invalid for the current FSA definition the program will switch the FSA definition. This continues for as long as there are valid FSA definitions.

Extensions

Recognizing the same input

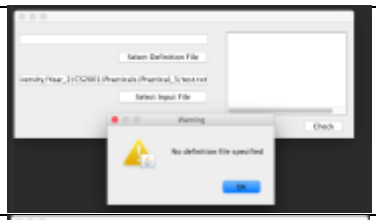
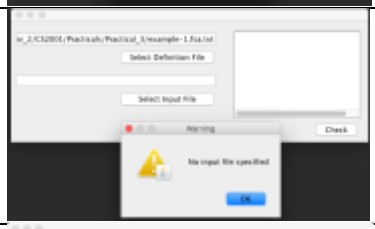
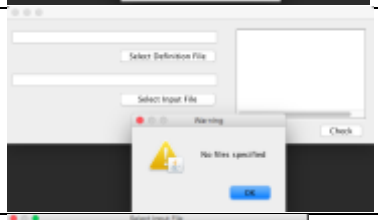
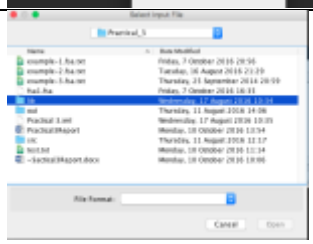
One FSA leading into another

GUI

Testing

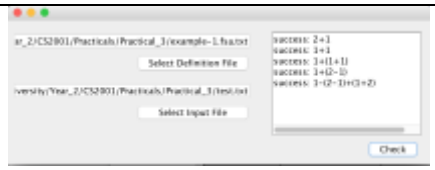
I am only going to test my extension this is because all the functionality in the basic requirement will also be encompassed in the tests

Testing interface

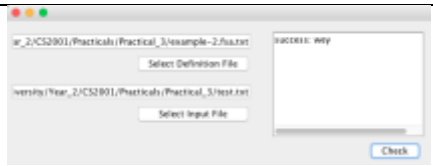
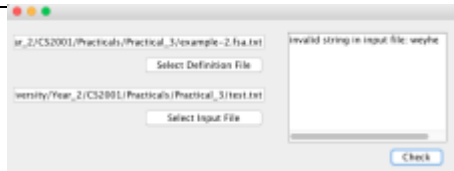
Test Description	Expected Output	Actual Output	Comment
Choose two valid files	The program should run normally	The program ran as expected	Works as expected
Choose only a valid input file	A popup warning should be displayed notifying the user a valid definition file was not chosen		Works as expected
Choose only a valid definition file	A popup warning should be displayed notifying the user a valid input file was not chosen		Works as expected
Neither input or definition file selected	A popup warning should be displayed notifying the user no valid files were chosen		Works as expected
Attempting to select a folder not a file		 The open button is greyed out	Works as expected

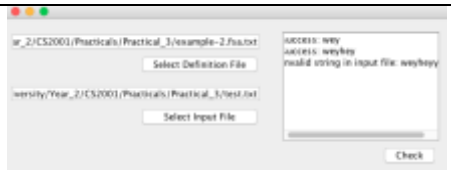
Using example-1.fsa

Test Data	Test Description	Expected Output	Actual Output	Comment
1+1	Have only one line in the test.txt file which is valid	The JText Area should display "success" and the input line		Works as expected

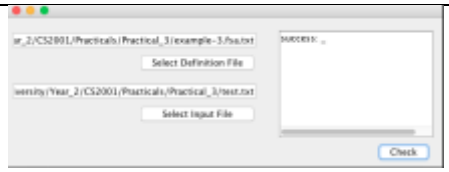
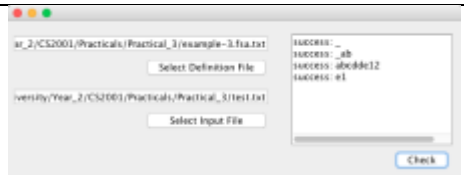
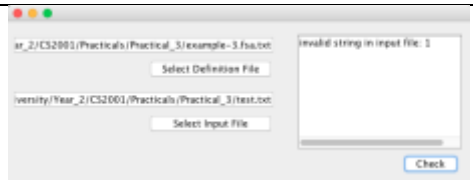
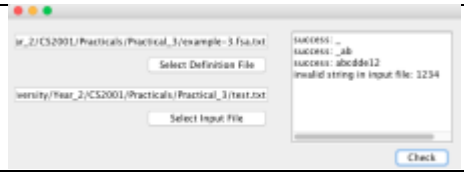
2+1 1+1 1+(1+1) 1+(2-1) 1-(2-1)+(1+2)	Have multiple lines in the test.txt file which are all valid	The JText Area should display "success" next to all the input lines		Works as expected
1-)4	Have only one line in the test.txt file which is in-valid	The JText Area should display "invalid string in input file" and the input line		Works as expected
2+1 1+1 (1+1)	Have multiple lines in the test.txt file where some are valid and some aren't	The JText Area should display "success" next to the first two input lines and "invalid string in input file" next to the last line		Works as expected
A+B	Have characters which are not in the definition	The JText Area should display "invalid string in input file" and the input line		Works as expected
Null	Leave the input file blank	No output should be shown	No output was shown	Works as expected

Using example-2.fsa

Test Data	Test Description	Expected Output	Actual Output	Comment
wey	Have only one line in the test.txt file which is valid	The JText Area should display "success" and the input line		Works as expected
wey weyhey weyheyheyhey	Have multiple lines in the test.txt file which are all valid	The JText Area should display "success" next to all the input lines		Works as expected
weyhe	Have only one line in the test.txt file which is in-valid	The JText Area should display "invalid string in input file" and the input line		Works as expected

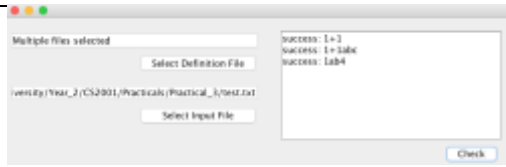
wey weyhey weyheyy	Have multiple lines in the test.txt file where some are valid and some aren't	The JText Area should display "success" next to the first two input lines and "invalid string in input file" next to the last line		Works as expected
foo	Have characters which are not in the definition	The JText Area should display "invalid string in input file" and the input line		Works as expected
Null	Leave the input file blank	No output should be shown	No output was shown	Works as expected

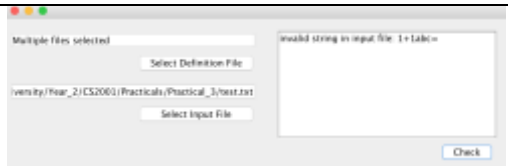
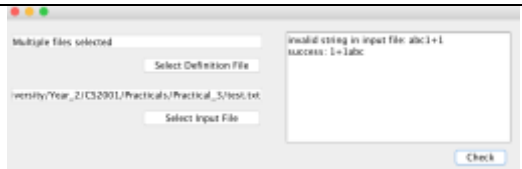
Using example-3.fsa

Test Data	Test Description	Expected Output	Actual Output	Comment
_	Have only one line in the test.txt file which is valid	The JText Area should display "success" and the input line		Works as expected
_ _ab abcdde12 e1	Have multiple lines in the test.txt file which are all valid	The JText Area should display "success" next to all the input lines		Works as expected
1	Have only one line in the test.txt file which is in-valid	The JText Area should display "invalid string in input file" and the input line		Works as expected
_ _ab abcdde12 1234	Have multiple lines in the test.txt file where some	The JText Area should display "success"		Works as expected

	are valid and some aren't	next to the first two input lines and "invalid string in input file" next to the last line		
☺	Have characters which are not in the definition	The JText Area should display "invalid string in input file" and the input line		Works as expected
	Leave the input file blank	No output should be shown	No output was shown	Works as expected

Testing multiple FSA definitions

Test Data	**Test Description**	**Expected Output**	**Actual Output**	**Comment**
Using example 1 and 3 Input: 1+1	Test with one valid input which uses only one of the FSA definitions	The program should display success next to the input		Works as expected
Using example 1 and 3 Input: 1+1 1+1abc 1ab4	Test with three valid inputs one of which uses only one of the FSA definitions the other two will require both FSA definitions	The program should display success next to all the valid		Works as expected
Using example 1 and 3 Input: abc	Test with one valid input which uses the other FSA definitions	The program should display success next to the input		Works as expected

Using example 1 and 3 Input: 1+1abc=	Test with one invalid input	The program should display invalid next to the input		Works as expected
Using example 1 and 3 Input: abc1+1 1+1abc	Test with one invalid input and one valid input	The program should display success next to the valid input and invalid next to the invalid input		Works as expected

Evaluation

After testing my program, I pleased to say it works as expected and is able to take in files and perform the correct operations on them. Moreover, I am also pleased the GUI I created is clean and user friendly.

Conclusion

In conclusion I'm satisfied with end program, it is able to carry out the required tasks with ease and is able to handle erroneous data without crashing. The added extensions massively increase the overall functionality.