CS2001 week 5 practical:
Finite state automata

Due 21:00 Tuesday Week 5
25% of practical mark for the module

Goal

To build a Java implementation of a finite state automaton interpreter.

Background

A Finite State Automaton (FSA) accepts input, typically strings, and decides whether it "accepts" the string as valid. Essentially the FSA encodes some pattern rules and checks whether the string complies with those rules or not.

We can describe an FSA using a simple transition table format, where we specify a sequence of triples consisting of:

- The input state of the automaton
- A character read from the input stream in that state
- The corresponding output state the machine transitions to as a result

We also need to know the initial state and the identities of any accepting states.

It's possible to build FSAs by hand every time we need them, but it's a lot easier to build a tool that, presented with a *description* of an FSA (in some format), then *interprets* that FSA to provide its behaviour: a single program that can display the behaviour of *any* FSA we can describe to it.

Specification

Your program will be tested using the School's automatic testing infrastructure. It is therefore very important that it complies exactly with the instructions below:

1. Place you code in a directory called `src/`
2. Your program should have a class `fsainterpreter` that includes a `main()` method
3. Your program should take as an argument the name of the fsa description file, and should read the input to the machine from standard input
4. It should print either `Accepted` or `Not accepted` on standard output as a result

You might therefore set up an FSA description in a file `fsa1.fsa` and a sequence of strings in a file `test.txt`, and then run your program to read the text file on its standard input using a command like:

```
java fsainterpreter fsa1.fsa <test.txt
```

The FSA description language

An FSA description consists of a sequence of triples, one per line, such as the following:

```
1 d 2
2 o 3
3 g 4 *
```

These triples state that:

- In state `1`, see `d`, transition to state `2`
- In state `2`, see `o`, transition to state `3`
- In state `3`, see `g`, transition to state `4` which is an accepting state (indicated by the `*`)

States are represented by numbers, with the start state of the first triple being the machine's initial state. In all cases, any inputs not mentioned in the description should lead the machine to reject the string by landing the machine in a non-accepting state. A state will typically appear on multiple lines of a description: it is an accepting state if *any line* indicates that it is accepting.

Requirements

You should submit three elements to MMS:

1. Your FSA interpreter program
2. Appropriate evidence of testing, for example FSA descriptions, test strings, and their results, providing files or screenshots (or both) to show that you've tested your system thoroughly
3. A short (approximately 500—1000 words) report describing how you designed the program and how your tests demonstrate that it is correct

Your interpreter should be able to process a sequence of strings on standard input according to the FSA description above.

For additional credit you might provide a way of composing FSA descriptions, for example to have one machine “lead into” another, or recognise the same input repeated several times. Explain briefly the mechanisms you use for your extensions, and demonstrate that they behave correctly.

Marking

The practical will be graded according to the guidelines in the student handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors

Hints for completing the practical

(Use or ignore as you see fit.)

Think first about the logic of the program you're writing. It has to represent the set of states of the FSA and, for each state, the characters that can be seen and the state the machine moves to as a result. Get this behaviour working first.

Then add the accepting/non accepting behaviour: under what conditions is a string accepted? How is it not accepted?

Then – and only then – worry about how to read the description of the automaton. If you've been able to build an automaton for your tests above, you can use the same methods to build it driven by what you read from the file.

Since you can represent an FSA as a diagram – a network with states as nodes and transitions as edges (labelled with a letter), you could implement your automaton as a collection of objects with links between them. Alternatively, since you can also represent an FSA as a transition table, you might decide to implement it using some sort of array. Pick whichever you find easiest to think about. There might be small performance differences between different implementations – but do these really matter for this practical?