CS2001/CS2101 week 3 practical: Finite state machines

Due 21:00 Wednesday Week 3
25% of practical mark for the module

Goal

To build a Java interpreter for finite state machines.

Background

A Finite State Machine (FSM) takes an input, typically a string of symbols or actions, and outputs a corresponding string of output symbols. Essentially the FSM is a simple processor that responds to inputs in a way that can be sensitive to the previous inputs it has received.

We can describe an FSM using a transition table, where we specify a sequence of 4-tuples consisting of:

1. The input state of the automaton
2. A character read from the input stream in that state
3. The character that the machine outputs in response
4. The state the machine moves into ready for the next input

We also need to know the initial state, which by convention is the input state of the first line of the transition table.

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

Specification

Your program will be tested using `stacscheck`. 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 `fsminterpreter` that includes a `main()` method
3. Your program should take as an argument the name of the FSM description file (format described below), and should read the input to the machine from standard input, printing the resulting output characters on standard output in a single line

If the description of the machine is not well-formed the interpreter should print `Bad description` and exit without trying to interpret the input. If the interpreter receives an input character that isn't in the set of accepted input symbols, it should print `Bad input` and exit immediately.

Your program should take as input the name of the FSM description as an argument and a sequence of input characters on its standard input. You might therefore test your program using a line of input characters in a file `test.txt` (so you don't have to repeatedly type in the input yourself), and then run your program to read the text file on its standard input using a command like:

```
java fsminterpreter 1.fsm <test.txt
```

(There are others approaches to testing, of course.)

Describing the FSM transition table

An FSM description consists of a transition table such as the following:

```
1 a 1 2
1 b 0 1
2 a 2 3
2 b 0 3
3 a 3 1
3 b 0 3
```

These tuples state that:

- In state `1`, see `a`, output 1, transition to state `2`
- In state `1`, see `b`, output `0`, remain in state `1`
- In state `2`, see `a`, output `2`, transition to state `3`

...and so forth. States are represented by numbers, with the start state of the first tuple being the machine's initial state. There can in principle be a *lot* of states, so don't assume there will only be some small number of them. You can however assume that all input, and all outputs, will consist of single characters.

You will find some example files on StudRes to test against. You should run `stacscheck` from your practical directory as usual, for example:

```
stacscheck /cs/studres/CS2001/Practicals/W03-FSM/Tests/
```

Requirements

You should submit three elements to MMS:

1. Your FSM interpreter program
2. Appropriate evidence of testing, for example FSM 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 presented to it on its standard input according to the FSM description above.

Marking

The practical will be graded according to the grade descriptors used for CS21001/CS2101, which can be found at:

https://studres.cs.st-andrews.ac.uk/CS2001/Assessment/descriptors.pdf

A good solution worthy of a top grade would be a well-designed, -tested, and -explained interpreter able to read and interpret FSM descriptions according to the above specification, including rejecting ill-formed machines and illegal inputs, written with consistent coding style, good comments, and clear logic.

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 FSM and, for each state, the characters that can be seen, the output, and the state the machine moves to as a result. Get this behaviour working first, testing it extensively. Make sure you check the corner cases. The easiest way of doing this is by writing some tests you can perform automatically, so that you execute the *same* tests every time.

Then – and only then – worry about how to read the description of the machine. If you've been able to build a machine for your tests above, you then just have to populate the machine from the input file description.

What rules makes a machine description legal, according to the description above? How can you check the legality of inputs? What will you test?

Think carefully about the design of the interpreter. You might, for example, have methods for creating the machine (adding states and transitions), a method for running the machine, and methods for getting input and generating output. Breaking things down like this makes the logic much clearer, and also makes it easier to build sub-classes that override only specific parts of the functionality. Comment everything properly and make sure it's well-laid-out.

Given a choice between two implementations, the correct choice is almost always the one that's simpler to explain to someone else.