CS337 Project 3
Creating a Finite State Machine

TA in charge: Lara Schmidt

Due TBD

1 Overview

Finite State Machines are found throughout computer science. Compilers, grammars, or any kind of program where users can be in different positions, all make use of high-level states and state transitions. For this project you will create some simple finite state machines, then implement a finite state machine interpreter to test your machines on user input.

The goal is to learn how to think in terms of breaking up complex operations into moving from one defined state to another. When you undertake major programming projects you will be surprised at how often state diagrams resurface.

This project will take awhile to do properly. Begin early so you will not run out of time. Also it will be good preparation for the test.

There are three parts: In part 1, you will create seven finite state machines to match given criteria. In part 2, you will translate your diagrams into a form that can be read by a computer program. Then in part 3, you will implement a program that reads in the files from part 2. Your program will then read in strings and display if your finite state machines accepts or rejects the input.

2 Part 1

The alphabet for this project is:

`abcdefghijklmnopqrstuvwxyz0123456789~!@#%^&*()-+{}.,`

Use the notation described in class to create a unique finite state diagram for each of the following seven string descriptions. Refer to Figure 1 for an example of a finite state diagram. You will not be turning in these diagrams but they will be used in part 2.

1. Accept only integers, where integers are defined as either a single 0, or a nonzero digit followed by any number of digits.
2. Accept only comma-separated integers (use integer as defined above.) An example of a comma-separated integer would be $2,000$ or $6,500,000$. But 3000 and $3,00,0$ must be rejected. Notice that the commas must be in the thousandth, millionth, billionth, etc. positions. If the number is 999 or less it should be accepted.

Figure 1: An FSM that accepts strings with an odd number of 1s.

3. Accept signed and unsigned integers (use integer as defined above.) −5 is valid, as is +45, 45, and 0. But +0, −45.5, and −0.3 must be rejected.
4. Find the substring "laurel" in any string that uses the defined alphabet. So "erikjeremylaurel" would return true, but "laurals" would fail. Note: spaces are not part of the alphabet.
5. Accept unsigned floating point numbers which MUST contain a decimal point in the number. With floating point numbers, a 0 can be the leading digit but only if it is the only digit before the decimal. 0.003, 0.20, 0.000, 3.54 are all correct, but 0..03, 1., 55, and 01.33 are not.
6. Accept a string with up to 3 levels of parentheses, ignoring the characters between, before, or after the parentheses. Reject the string if the parentheses are unbalanced, if there are no parentheses in the string, or if the nesting is deeper than three. You should accept "compute((1+2)+((5+2*8+2)))", and "((()))" but reject "((((4))))", "(()()))", and "4+3".
7. Accept a string of digits if they are strictly increasing. 345, 4, and 069 are accepted, but 32 and 22 are rejected.

Warning: The empty string must be rejected by all Finite State Machines.

3 Part 2

The next step is to convert your finite state machine diagrams into a text file that can be read by a computer program. Each finite state machine diagram from part 1 needs to have its own file according to the following format.

Each line in the file represents a single state in the diagram. Below is the file `machine0.fsm` which is the machine-readable form of the diagram in Figure 1:

```
1 1:0 2:1
2 1:1 2:0 X
```

Each line has the following format:

```
[State Name] [Child Name]:[Transitions] [Accepting State?]
```

The first token in each line is the state name. For your implementation you can expect to see only unsigned numbers for state names. In the above example, the first line describes state 1, the second line describes state 2. NOTE: there is no upper limit on the size of the state number. 4 is as good as 4000000000.

The Finite State Machine Interpreter will always look for, and begin in, the state labeled 1. The states do not have to be in order, and there can be gaps in the numbering. A state can only be listed once per file.

After each state name, there can be an arbitrary number of additional tokens. There are two types of tokens, Accepting State Flags, and Next Step Pairs.

An Accepting State Flag (ASF) is the capital letter 'X'. When an ASF is on the line, it means the current state is an accepting state.

Note: The flag can appear before, after, or anywhere in between other Next Step Pair (NSP) tokens.

In the example file, state 2 has an ASF, so if the input string terminates in state 2, the string is accepted. State 1 does not have an ASF, so if the string ends there, it will be rejected.

A Next Step Pair consists of the state name, followed by a colon (":") and then the valid characters that allow the transition to take place. There must not be any spaces between the name, colon, and transition characters. In the example, state 1 transitions back to state 1 if a '0' is encountered–a loop. State 1 transitions to state 2 if a '1' is found.

Remember the shorthand discussed in class: If the next character in the string is not mentioned in the current state, then the string is immediately rejected.

There are four uppercase letters we will use as shorthand for different sets of the alphabet which you should use in your files. You must implement these shortcuts in part 3.

('A') Alphabetic: `abcdefghijklmnopqrstuvwxyz`

('D') Digit: `0123456789`

('N') Non-Zero: `123456789`

('S') Symbolic: `~!@#%^&*()-+{}.,`

To create an FSM that accepts strings which have a non-zero digit as the first character, and zero or more characters after it, you would create a .fsm file with two lines:

```
1 2:N
2 X 2:ADS
```

4 Part 3

You now need to write a Java program to read .fsm files in the format just described. It must be able to read arbitrary FSMs, (I will test your code on some FSMs that you have not seen.)

After your program has read in all the files (they must be called machine1.fsm, ..., machine7.fsm) created in part 2, you need to read in a file (ex. "strings.txt") which contains a single text string on each line. You will need to create your own test strings. Assume that my "strings.txt" will have blank lines, and end of line spaces that may need to be trimmed.

Your program should take the name of the strings file as well as the names of all the finite state machines you wish to run it on as command line parameters. If no machines are given, you should default to running it on machine1.fsm, . . . ,machine7.fsm in order. You may assume a file similar to strings.txt will always be given. Please call your java program FSMachine.java.

You need to evaluate each text string with each FSM you create and display which machines accept the input. The program output should be displayed on the screen, and you should include your test cases in your "readme.txt".

If you implemented both of the .fsm files above and created a "strings.txt" that contained:

```
11001101
11111
0thisshouldfailmachine0
```

then the output of

```
java FSMachine strings.txt machine0.fsm machine1.fsm
```

should look like (where the order of machines is the same as the input order):

```
machine0.fsm Accepted: 11001101
machine1.fsm Accepted: 11001101
machine0.fsm Accepted: 11111
machine1.fsm Accepted: 11111
machine1.fsm Accepted: 0thisshouldfailmachine0
```

If the string is rejected by a machine, do not print anything.

There are four shortcut characters you need to implement. The uppercase letter 'A' should cover all alphabetic characters, 'D' for 0 to 9, 'N' for 1 to 9, and 'S' for symbols. See the list in part 2 for the exact characters. You should not implement techniques such as A-m or other such shortcuts.

5 Turning in your project

Please follow the project protocol for files that should be turned in. You must also include the 7 machineX.fsm files you create in part 2 and your strings.txt.

Use the Linux turn-in software to submit your project. The command will look like: "`turnin -submit lschmidt FSMachine.java machine0.fsm machine1.fms machine2.fms ...readme.txt`" All students will be turning in the project to `lschmidt`.

6 Questions and Project Clarifications

Questions and Projecct Clarifications will be posted to Piazza. Please browse through other questions before asking one!