Week 2: Finite State Automata
Due date: Friday Week 2, 21:00

130016030
University of St Andrews
September 26, 2014

Introduction

The main goal of this practical is to build a Java implementation of a finite state automaton interpreter. 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.

Structure

1 UI package

UI package contains **UI.java** file. It is self-motivated extension based on the previous practical. It was decided to build user-interface because "Merge FSA" extension was done as well. In this case, instead of writing **java fsainterpreter fsa1.fsa < test.txt**, user can use UI to control the program and choose either build one FSA or build two FSA and then merge them into one. Also, to make it robust if user accidently types incorrect value, the program would not break and will ask user to type again.

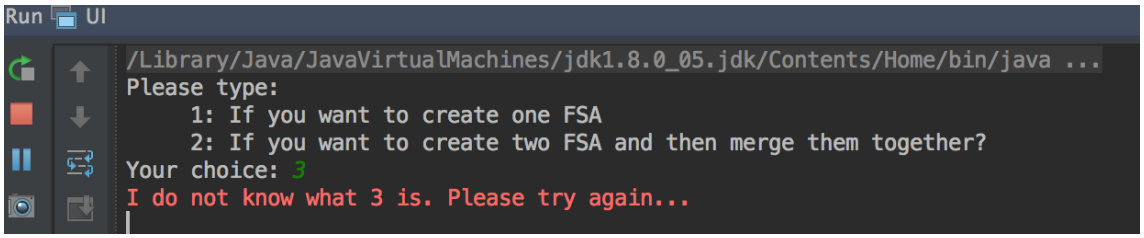

Figure 1: Error handle if user types wrong number

2 Representation package

Representation package contains three classes: **FSA.java**, **State.java and Edge.java**. These classes represents finite state automata: FSA has several states and each state has several edges that links one state to another.

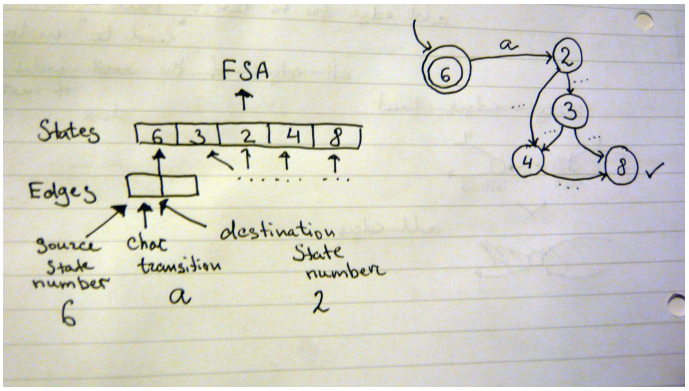

Figure 2: Java FSA representtion

FSA.java
As it can be seen in Figure 2, finite state automata contains list of States. It was decided to use Linked list rather than Array list to use memory space more effectively. Linked list keeps expanding as you add elemens, whereas Array list has initial capacity and doubles it's space. Each state has unique number. To ensure that in FSA does not contain two or more the same states, isStateExist procedure was developed. It also has methods, such as: find accepting state, find max state, get initial state. There methods are required for "Merge FSA" extension.

Edge.java
Edge is used to connect states. It contains source state number, character that transit from one state to another and destination state number.

State.java
Also, refering to the Figure 2, each state has it is several edges. It has unique number and boolean identinier that determines is state accepted or not. Linked list was choosen for exactly the same reason as in FSA.java. For "Merge FSA" extension it is a procedure that increments all number of states in State.java and Edge.java.

3 Builder package

FSAConstructor.java
It creates finite state automate by reading finite state description file and building it line by line. First of all, it parses the string by spliting it into Source State, Transition Character and Destination State. Then it checks is source state already exist. If not it creates the source state and adds corresponding edge to it. Otherwise, it will just add the edge to the exisisting state by getting it from the list. After that, it checks is destination state exist. If not, then it checks is the despination state accepted. Finally, it adds destination state to FSA.
This is how FSA is built from the file. A part of the source code is included at the next page.

```
// If there is no source state, add it.
if(!fsa.isStateExist(stateFrom)){
        state = new State(stateFrom, false);
        state.addEdge(new Edge(stateFrom, transition, stateTo));
        fsa.addState(state);
}
else{
// If state is already exist, just add edge to it.
        state = fsa.getState(stateFrom);
        state.addEdge(new Edge(stateFrom, transition, stateTo));
}

// If there is no destination state, add it.
if (!fsa.isStateExist(stateTo)){
        state = new State(stateTo, false);
        // If destination state is accepted add it by changing a boolean value
        if (input.length == 4 && input[3].equals("*")){
                state = new State(stateTo, true);
        }
        fsa.addState(state);
}
```

FSAMerger.java

This class is used to merge two finite state automates. One of the required extensions was to implement it. First of all, it constructs two FSA for a two files specified by user. Then, it increases number of states and edges for a second FSA by a maxim state number from the first FSA. After that, it links all accepting states from first FSA to the initial state of the second FSA by adding the edges with a " " (empty space) transition character. Finally, it transforms all states from second FSA to the first to get a final one.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Contents/Home/bin/java ...
Please type:
     1: If you want to create one FSA
     2: If you want to create two FSA and then merge them together?
Your choice: 2
Please type your path to the file of two FSA. For instance: fsa_descriptions/example1.fsa fsa_descriptions/example2.fsa
Your path:fsa_descriptions/example_2.fsa fsa_descriptions/example.fsa
Please type "quit" if you want to finish
Your word: weyheyheyhey dog
accepted
Your word: weyhey do
not accepted
Your word: quit

Process finished with exit code 0
```

Figure 3: User session with two finite state descriptions

In the example that Figure 3 shows it can bee seen that the space is required to move from one FSA to another. However, it is actually only one FSA. **The important note** is it was decided to keep accepted states from first FSA accepted. For this reason, words "wey", "weyhey" will be accepted. Althougth, it might be easily changed by changing the State boolean variable using *set()* method. **It is also important** that if user decides to merge to identical FSA description files, the program will recognise the same input repeated several times. As a result, all suggested extensions were completed with some self-driven ones.

4 Parser package

FSAReader.java
When the FSA is built based on FSA description, there is need to run it against each of the strings provided as input, printing for each either accepted or not accepted on standard output. On top of that, user can type the word "quit" to finish the program.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Contents/Home/bin/java ...
Please type:
     1: If you want to create one FSA
     2: If you want to create two FSA and then merge them together?
Your choice: 1
Specify the path to the file. For instance: fsa_descriptions/example.fsa
Your path:fsa_descriptions/example.fsa
Please type "quit" if you want to finish
Your word: dg
not accepted
Your word: dog
accepted
Your word: quit

Process finished with exit code 0
```

Figure 4: "Quit" word is uset to exit the program.

FSAChecker.java
This class is used to check that the word specified by user is in finite state automata. It has a method that deals with exceptions, namely Null Pointer as well as Empty Stack one. This method checks all characters in word. Firslty, it finds the edge of the given state based on character. Then, if this edge is actually exist, it goes into it. Finally, if last state where the loop finished is accepted, the method will return true.

Testing

FSACheckerTest.java
This project is written in TDD maner. This class has 22 JUnit test that proves that the program works correctly. In order to make a good program with many extensions it is also important to make a robust application.

To begin with, before each test it creates FSA with 4 states where the last state is accepted, and 3 edges that link these states. There are 4 tests that insures that the programm does not break if string is object is empty or null. There are also 2 simple tests that checks are given words accepted or not. All other tests insures that the algorithms that were already discussed above works correctly.

Conclusion

As usual, this project includes javadoc, Apache ANT build, source code and this report. There is only one main method in UI class. Images in this report can be also accessed in the /img folder in case you cannot see it. Please make sure that you have a **Java 8** to run it.