CS 361: Project 1 (Deterministic Finite Automata)

Due date: March 2, 2020

Spring 2020

1 Objectives

In this project you will implement a Java program that models a deterministic finite automaton.

- Familiarize with the concept of packages in Java, which allows us to organize classes into a set (package) of related classes.
- Familiarize with the java Collections API available in Java's `java.util` package. This provides us with commonly used data structures such as sets, maps, lists (sequences) that represent a different collections of objects.
- Practicing implementing interfaces. You will have to write `fa.dfa.DFA` class that implements `fa.dfa.DFAInterface` interfaces. In its turn `fa.dfa.DFAInterface` extends `fa.dfa.FAInterface`, which effectively forces `fa.dfa.DFA` to implement methods in that interface too.
- Practicing extending abstract classes. You will have to write `fa.dfa.DFAState` class that extends `fa.State` abstract class.
- Use an existing class to execute and test implementations. You will use `fa.dfa.DFADriver`, i.e., the driver class, to execute and test your own implementation.

2 The Concept of packages in Java

Have you ever thought what the import statement in the beginning of a java file means?

```
import java.io.File;
```

You're telling to the compiler where to look for the `File` class that your program is using. The compiler will look for *java* folder and inside it search for *io* folder and inside it will find *File.java* file. We say that `File` class is in `java.io` package. Java uses packages to organize classes into a bundle of related/similar classes. When you open *File.java* file you will see the following package declaration on the top:

```
package java.io;
```

This package statement declares the package name to which the class belongs to. In fact, the fully qualifying name, that is the full name of `File` class, is `java.io.File`. Please read more on Java packages and how to create and work with them in Eclipse (in case you're planning on developing your code in this IDE).

In this assignment you will be working with two packages: `fa` that holds an interface and an abstract class for any finite automaton and `fa.dfa` (that is *dfa* folder inside *fa* folder) that contains classes for a deterministic finite automaton.

Below is the directory structure of the provided files:

```
|-- fa
|    |-- FAInterface.java
|    |-- State.java
|    |-- dfa
|          |-- DFADriver.java
|          |-- DFAInterface.java
|-- tests
       |-- p1tc1.txt
       |-- p1tc2.txt
       |-- p1tc3.txt
```

To compile `fa.dfa.DFADriver` from the top directory of these files:

```
[you@onyx]$ javac fa/dfa/DFADriver.java
```

To run `fa.dfa.DFADriver`:

```
[you@onyx]$ java fa.dfa.DFADriver ./tests/p1tc1.txt
```

3 Specifications

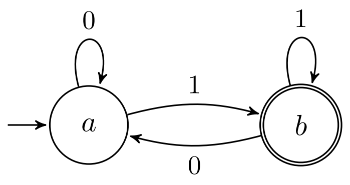
- Existing classes that you will use: `fa.dfa.DFADriver`, `fa.dfa.DFAInterface`, `fa.FAInterface`, `fa.State`.
- Classes that you will implement: `fa.dfa.DFA` and `fa.dfa.DFAState`.

3.1 Input file

The input file to the driver class, i.e., `fa.dfa.DFADriver` has the following format:

- The 1st line contains the names of the final states, i.e., elements of F. The names are separated by the white space. It can be empty if there are no final states.
- The 2nd line contains the name of the start state, i.e., q_0.
- The 3rd line contains the rest of the DFA's states, i.e., those states that are not F or q_0. It can be empty if all states have been specified in the previous two lines.
- The 4th line lists the transitions. Transitions are separated by the white space. Three symbols compose a transition $s_0 s_1 s_2$, where
  - The first symbol s_0 is the name of the "from" state.
  - The second symbol s_1 is the symbol from the alphabet, i.e., $s_1 \in \Sigma$.
  - The third symbol s_2 is the name of the "to" state.
- Starting from line 5, each line contains a string for which we need to determine whether it is in the language of the DFA. The strings are over the DFA's alphabet and we use 'e' symbol to represent the empty string ε.

For example this DFA



has the following encoding and the set of strings to test:

```
b
a

a0a a1b b0a b1b
```

```
0
1
00
101
e
```

Note: The 3rd line is empty because both states are already specified on the 1st and 2nd lines.

We provide you with three input files, but we encourage you to create several of your own to further test your implementation.

3.2 `DFADriver` (provided class in `fa.dfa` package, the class with `main` method)

Note: You don't need to modify the class. You will use it to test your DFA implementation.

You are given `DFADriver`[1] class that reads the input file, instantiates a corresponding DFA and simulates that DFA instance on each input string. `DFADriver.java` is adequately documented.This class takes an input file with the described above DFA encoding and does the following:

1. Prints out the string description of the DFA (by calling `toString()` of `DFA` class).
2. For each input strings prints out "yes" if an input string is in the language of the machine, i.e., if the DFA accepts that string, or "no" otherwise.
3. Prints out whether the `complement` of the DFA is correct.

Refer to Sample Input/Output section for examples.

3.3 `DFA` (class you need to implement in `fa.dfa` package)

`DFA` class *must* implement `DFAInterface` interface. Make sure to implement all methods inherited from this interfaces and one that `DFAInterface` extends. Make sure that the output of `toSring()` method matches exactly to the outputs provided in Sample Input/Output Section. You will add instance variables to represent at least some elements of the DFA 5-tuple, i.e., $(Q, \Sigma, \delta, q_0, F)$. You might also add additional methods, which must be private, i.e., helper methods. Below are additional requirements:

1. DFA elements that represent sets, e.g., set of states Q, must be implemented using one of the concrete classes that implements `java.util.Set` interface. Please browse through the Set documentation to determine an appropriate concrete class.
2. The transition function should be implemented using one of the concrete classes that implements `java.util.Map` interface. Once again go over the Map documentation to determine a right concrete class.
3. `toString()` method
   Even though Q and Σ are sets, we require the following ordering on them to ensure the consistent output of `toString()` method. The states and the symbols *must* appear in the same order they are read from the file. Thus, first all final states are displayed as

[1]We omit the package in the class name for brevity.

they appear on the first line. Next in the order will be the start state (if it was not in the set of final states F already). Next, the order contains the rest of the states in the order they've appeared on the third line.

4. `complement()` method
   This method must return the complement of this DFA. That is a new DFA, such as its language is the complement of `this` DFA's language. Recall from the slides, that in order to do so, we need to make all final states of this DFA non-final (i.e., regular) states and vice versa.

3.4 `DFAState` (class you need to implement in `fa.dfa` package)

`DFAState` class *must* extend `State` abstract class. You might add additional instance variables and methods to your `DFAState` class to represent the rest of elements of DFA's 5-tuple that are not in `DFA` class. For example, each row of the transition function can be stored in the corresponding state.

Note: Use object-oriented design principles! Store states as a set of `DFAState` objects and not as a set of `String`, i.e., set of state's names.

4 Sample Input/Output

Below are samples of inputs/outputs for the three provided test cases.

```
[you@onyx ]$ cat ./tests/p1tc1.txt
b
a

a0a a1b b0a b1b
0
1
00
101
e
[you@onyx ]$ java fa.dfa.DFADriver ./tests/p1tc1.txt
Q = { b a }
Sigma = { 0 1 }
delta =
                   0         1
         b         a         b
         a         a         b
q0 = a
F = { b }

no
yes
no
yes
```

```
no

Is dfaC correct? : yes
```

```
[you@onyx ]$ cat ./tests/p1tc2.txt
3
0
1 2
001 010 103 112 201 211 303 313
010
00
101
111011111111110
1110111111111010
[you@onyx ]$ java fa.dfa.DFADriver ./tests/p1tc2.txt
Q = { 3 0 1 2 }
Sigma = { 0 1 }
delta =
                   0         1
         3         3         3
         0         1         0
         1         3         2
         2         1         1
q0 = 0
F = { 3 }

no
yes
no
yes
no

Is dfaC correct? : yes
```

```
[you@onyx ]$cat ./tests/p1tc3.txt
G D
A
B C E F
A1B A2C B1D B2E C1F C2G D1D D2E E1D E2E F1F F2G G1F G2G
121212121
12221212121
12
2
1212
[you@onyx ]$ java fa.dfa.DFADriver ./tests/p1tc3.txt
Q = { G D A B C E F }
```

```
Sigma = { 1 2 }
delta =
                   1         2
         G         F         G
         D         D         E
         A         B         C
         B         D         E
         C         F         G
         E         D         E
         F         F         G
q0 = A
F = { G D }

yes
yes
no
no
no

Is dfaC correct? : yes
```

5 Grading Rubrics

1. **5** points for the properly commented (Javadocs and inline comments) code and the properly formatted README.
2. **4** points for code compiling and running on onyx.
3. **4** points for using a class that implements `java.util.Set` to represent sets and a class that implements `java.util.Map` to represent functions..
4. **3** points for using object-oriented design principles.
5. **3** points for the correct implementation of `toString()` method for `DFA` class.
6. **75** points for program running correctly. We will have 15 test files (3 of which are provided to you) each containing 5 test input strings. For each correctly accepted/rejected string you will get 1 point. So, if all test files pass you will get $15\times\ 5 = 75$.
7. **6** point for the correct implementation of `complement` method for `DFA` class (0.4 points for each passed test file).

6 Submitting Project 1

Documentation:

If you haven't done it already, add **Javadoc comments** to your program. It should be located immediately before the class header and before each method that was not inherited.

- Have a class javadoc comment before the class.
- Your class comment must include the *@author* tag at the end of the comment. This will list you as the author of your software when you create your documentation.

- Use *@param* and *@return* tags. Use inline comments to describe how you've implemented methods and to describe all your instance variables.

Include a plain-text file called **README** that describes the your program and how to use it. Remember to include also in the README who are the authors of the code, and what is the section of CS 361 you are attending. TAs need this information for grading purposes. Expected formatting and content are described in README_TEMPLATE. An example is available in README_EXAMPLE.
You will follow the same process for submitting each project.

1. Open a console and navigate to the project directory containing your source files,
2. Remove all the `.class` files using the command:
```
rm *.class
```
3. In the same directory, execute the submit command :
```
submit cs361 cs361 p1_1
```
4. Look for the success message and timestamp. If you don't see a success message and timestamp, make sure the submit command you used is EXACTLY as shown

Required Source Files:
Make sure the names match what is here **exactly** and are submitted in the proper folders/-packages

- `DFA.java` in `fa.dfa` package.
- `DFAState.java` in `fa.dfa` package.
- `README`.

After submitting, you may check your submission using the "check" command as in the example below:
```
submit -check cs361 cs361 p1_1
```