CS 361: Project 1 (Deterministic Finite Automata)

Fall 2024

Due Date: Sept 18, 2024

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 package provides us with commonly used data structures such as sets, maps, lists (sequences) that represent 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 transitively 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.
- Apply test-based development using JUnit test cases.

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 the compiler where to look for the `File` class that your program is using. The compiler will look for *java* folder, and inside it will 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.

3 JUnit test cases

To determine whether your implementation is correct, you will run `DFATest.java` located in `test.dfa` package. It contains methods annotated with `@Test` keywords. Unlike regular java programs, JUnit file requires additional arguments after `javac` and `java`. When a JUnit is executed, it invokes each of the annotated test methods separately. A test method uses

assertion checks, e.g, `assertTrue`, `assertFalse`, `assertEquals`, `assertNull` and so on, to check the expected behavior of your implementation, i.e., whether methods return correct answers.
For example, `assertTrue(dfa.addState("a"))`, expects that method `addState` returns true when it adds a new state with name `a` to a DFA object.
Below is the directory structure of the provided files:

```
|-- fa
|    |-- FAInterface.java
|    |-- State.java
|    |-- dfa
|          |-- DFAInterface.java
|-- test
       |-- dfa
            |-- DFATest.java
```

To compile `test.dfa.DFATest` on **onyx** from the top directory of these files:

```
[you@onyx]$ javac -cp .:/usr/share/java/junit.jar ./test/dfa/DFATest.java
```

To run `test.dfa.DFATest` on **onyx** type in one single line:

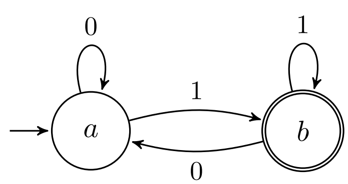
```
[you@onyx]$ java -cp .:/usr/share/java/junit.jar:/usr/share/java/hamcrest/core.jar
org.junit.runner.JUnitCore test.dfa.DFATest
```

4 Specifications

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

4.1 Creating a DFA

JUnit uses a series of invocations to create a DFA object. For example, to create the following DFA



It uses these a series of method invocations (with assertions) on a created DFA object:

```
DFA dfa = new DFA();
dfa.addSigma('0');
dfa.addSigma('1');

assertTrue(dfa.addState("a"));
assertTrue(dfa.addState("b"));
assertTrue(dfa.setStart("a"));
assertTrue(dfa.setFinal("b"));
```

```
assertTrue(dfa.addTransition("a", "a", '0'));
assertTrue(dfa.addTransition("a", "b", '1'));
assertTrue(dfa.addTransition("b", "a", '0'));
assertTrue(dfa.addTransition("b", "b", '1'));
```

We provide you with tests for three DFA encodings, but we encourage you to create several of your own to further test your implementation.

4.2 `DFATest` given class in `test.dfa` package, this is the driver class.

Note: You should not modify the existing methods. You will use them to test your DFA implementation.

You are given `DFATest`[1] class that instantiates a DFA and checks for expected behavior. For each DFA instance, it has series of 6 tests (X is a DFA id, e.g., 1, 2 and so on):

- `test1_X` invokes dfa instantiation method and checks whether adding, setting states and transition methods of DFA work as expected.
- `test2_X` checks the correctness of a successfully instantiated DFA, e.g., states have correct names and types.
- `test3_X` passes strings to dfa and checks whether it accepts strings in the dfa's language and rejects string not in the dfa's language.
- `test4_X` invokes toString of dfa, and checks whether it is the same (up to white spaces) as the expected output, which is hard coded in the test case.
- `test5_X` invokes swap method of dfa (see explanation below) and checks for the correct instantiation of a swapped DFA.
- `test6_X` passes strings to the swapped DFA and checks whether it correctly accepts and rejects them.

Your implementation should pass all test cases of `DFATest`. During grading, we add 11 more similar test cases to further test your implementation. Thus, we encourage you to create additional test cases too.

4.3 `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 interface and one that `DFAInterface` extends. Make sure that the output of `toSring()` method matches exactly to the outputs in the tests. 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]We omit the package in the class name for brevity.

1. DFA elements that are 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 as they added to a DFA in a test case. For example, the invocation sequence is

   ```
   DFA dfa = new DFA();
   dfa.addSigma('0');
   dfa.addSigma('1');
   ```

   then for Σ `toString` should print

   ```
   Sigma = { 0 1 }
   ```

   However, for this sequence

   ```
   DFA dfa = new DFA();
   dfa.addSigma('1');
   dfa.addSigma('0');
   ```

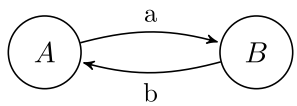
   `toString` should print

   ```
   Sigma = { 1 0 }
   ```

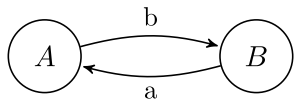
   The same ordering applies to the states. Once again, refer to `test4_X` test cases for the expected outputs. For the above DFA diagram (dfa1 in the JUnit), the expected output is

   ```
   Q = { a b }
   Sigma = { 0 1 }
   delta =
                   0       1
           a       a       b
           b       a       b
   q0 = a
   F = { b }
   ```

4. `swap(char symb1, char symb2)` method
   This method must return a copy of `this` DFA, with the transitions labels swapped between `symb1` and `symb2`. For example, if `this` DFA has a transition from state `A` to state `B` on symbol `a` and a transition from state `B` to state `A` on symbol `b`, i.e.,



then the swapped version will have a transition from `A` to state `B` on symbol `b` and a transition from state `B` to state `A` on symbol `a`, i.e.,



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

`DFAState` class *must* extend `State` abstract class, which already has a default constructor to set the name for the state. You would invoke it through `super(name)` statement. You might add additional instance variables and methods to your `DFAState` class to represent the rest of DFA's 5-tuple elements that are not in `DFA` class. For example, each row of the transition function table can be stored in the corresponding state as a map/dictionary (for example as `java.util.HashMap`, which also part of Java's collection framework).

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.

5 Grading Rubrics

1. **6** points for the properly commented (Javadocs and inline comments) code and the properly formatted README.
2. **3** points for the code compiling and running on onyx.
3. **3** 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. **4** points for using object-oriented design principles.
5. **84** for passing tests cases. We will have 14 test DFAs (3 of which are provided to you) each containing 6 test cases. For each correctly passed test, you will get 1 point. So, if all test cases pass you will get $14\times 6 = 84$.

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 the members of your team as the authors of the software when you create its 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 program and how to compile and run it. Remember to include also in the README who are the authors of the code. As you are allowed to have partners with up to one person, please be sure to name all the authors in README and inside the *@author* tag. 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. Only one person in your team should submit this assignment

1. Open a console and navigate into 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 :
   Section 1:
   ```
   submit cs361 cs361 p1_1
   ```
   Section 2:
   ```
   submit cs361 cs361 p1_2
   ```
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:
where X is the section you submitted to:
```
submit -check cs361 cs361 p1_X
```