CISC 4090–Theory of Computation
Spring, 2020

Programming Project # 1: A Deterministic Finite-State Automaton Emulator

Date Due: Monday 10 February 2020

Write a program that reads in a description of a deterministic finite state automaton (FSA) from a data file. It should then prompt for (and read) input strings, one per line. Using each string as input to the FSA, your program should trace the progress of the FSA, at the end of which, it should print out whether or not the FSA accepts said string.

To keep things specific (and manageable), the input alphabet Σ will be a finite set of characters, and the state set Q is a finite set of *strictly positive* integers. Moreover, we'll assume that all states are mentioned in the state transition table, which means that we don't need to know the number of states.

Input files will adhere to the following format.

- A string, whose characters are the elements of Σ. This string will not contain any blanks.
- Number of states.
- Start state.
- Number of accepting states.
- The accepting states.
- Remainder of the file: state transition table.

The elements of this file are whitespace-delimited. This means that C++ programmers can process this file by using the input stream extraction operator (i.e., >>); Python programmers can simply `read` the file and then `split` on whitespace.

For example, the data file

```
ab
3
1
1 2
2 1
3 3
2 1
```

and the data file

```
ab 3 1 1 2 2 1 3 3 2 1
```

both describe the FSA[1] $M_1 = (\{1,2,3\}, \{\mathtt{a},\mathtt{b}\}, \delta, 1, \{2\})$, whose transition function δ is given by the table

δ	a	b
1	2	1
$\underline{2}$	3	3
3	2	1

Its state transition diagram is on page 83, but with an obvious relabeling of states.

[1]However, the first version is easier for people to understand.

Let's solve this problem via a `FSA` class, having the following `public` member functions:[2]

- A one-parameter constructor, whose parameter is an `ifstream` reference to the data file. The statement

  ```
  FSA fsa{ifs};
  ```

  initializes the FSA `fas` from `ifs`, where `ifs` is a variable of type `ifstream`. For simplicity's sake, there's no other way to initialize an `FSA` object. I've made the statement "`FSA fsa;`" illegal by `delete`ing the (zero-parameter constructor in `FSA.h`.

- `describe()`, a `void` function taking no parameters. The statement "`fsa.describe();`" prints a description of `fsa`.

- `trace()`, a `void` function taking one `string` parameter. The statement "`fsa.trace(in_string);`" traces the operation of the `FSA` object `fsa` on the input string `in_string`, indicating whether or not `fsa` accepts `in_string` at the end of the trace.

The *share directory* `~agw/class/theory-comp/share/proj1` has some material that you'll need to use. At the top level of this directory are the following:

- A compiled executable `proj1-agw`.

- Data files `data1` and `data2`, which give descriptions of the machines M_1 and M_2 found on page 83 of the text. These will be the test data for your solution.

The share directory also has two subdirectories, one for C++ programmers and one for Python programmers. The `c++` subdirectory has the following files:

- `proj1.cc`: Driver code, mainly[3] consisting of the `main()` function, which opens the input file, initializes the FSA, and then runs the FSA on input strings provided by the user.

- `FSA.h`: The header file for the `FSA` class.

- `FSA.cc`: A "stub version" of the implementation file for the `FSA` class.

- `Makefile`: What you'd expect, if you have any experience with the `make` program. More precisely:

  - The command"`make`" builds the executable out of the files `proj1.cc`, `FSA.h`, and `FSA.cc`.
  - The command "`make clean`" cleans things up, i.e., it gets rid of any object files `*.o` and the `proj1` executable, as well as other detritus (core files and `emacs` leftovers).

  In particular, issuing "`make clean`" before "`make`", forces a complete recompilation.

The `python` subdirectory has the following files:

- `proj1`: Driver code, which opens the input file, initializes the FSA, and then runs the FSA on input strings provided by the user.

- `FSA.py`: A stub for the Python implementation of the `FSA` class.

[2]I'm using C++ notation to describe the `FSA` class. You are welcome to use Python, if you prefer.
[3]Heh.

Of course, this assumes that you're following my architecture for the project. If you want to approach it another way, you're on your own.

You should approach this in the usual way for students taking programming classes from me. Since some of you haven't done so, here's what's involved:

1. Create a subdirectory `theory-comp/proj1` of your `private` directory. This is to be your working directory for this project, i.e., you are to do all your work within this directory.
2. Copy the data files to your working directory, along with the contents of either the `c++` or `python` directories, depending on which language you're going to use.
3. Here's the real work:
   (a) If you're using C++, complete the implementation file `FSA.cc` for the `FSA` class.
   (b) If you're using Python, complete the implementation as outlined by the stubs in `FSA.py`.

   Don't forget to pay attention to the documentation.

Deliverables: Make a clean typescript.[4] If you're using C++, it should contain the following commands and their output:

```
cat FSA.cc
make clean
make
proj1 data1
proj1 data2
exit
```

If you're using Python, it should contain the following commands and their output:

```
rm -f *pyc
cat FSA.py
proj1 data1
proj1 data2
exit
```

For each execution of `proj1`, run the FSA on the following input data: ε, `ab`, `aabb`, `aabba`, `aabbaa`.

Good luck!

[4]If you're unfamilar with this, go to the Departmental help document `https://www.fordham.edu/cishelp`, and read up on how to submit programming assignments.