CSC173: Project 1
Finite Automata

The goal of this project is to experiment with pattern matching using finite automata, as well as get going programming in C. You will need to implement deterministic finite automata (DFAs), nondeterministic finite automata (NFAs), and a converter than turns an NFA into an equivalent DFA. The rest of this document gives you the specific requirements as well some ideas about how to proceed. Read the requirements carefully. You *must* hit the spec to get full points.

General Requirements

- You must submit code that compiles into a *single executable program* that addresses all parts of the project.
- Your program must be named `auto`.
- Your program must print to standard output (`printf`, *etc.*) and read from standard input (`scanf`, `fgets`, but *do not use* `gets`).
- You must have a data structure named `DFA` that represents a deterministic finite automaton.
- You must have a data structure named `NFA` that represents a nondeterministic finite automaton.
- For each pattern as described below, you must have a function that creates and returns an instance of an automaton (an instance of a `DFA` or `NFA` as appropriate) that recognizes the pattern.
- You must have a function or functions that test an instance of an automaton using a loop that prompts the user for input, reads an input string, runs the automaton on the input string, and prints the result, exiting the loop on some reasonable input. This is called a "read-eval-print loop," or "REPL" (pronounced REH-PULL).
- Your program must create and test automata in the order they are described below. Before testing an automaton, your program must print a short description of what pattern it is supposed to match so that the user knows what to try.

Specific Requirements

1. Implement deterministic finite automata that recognize the following languages:

   (a) Exactly the string `ab`

   (b) Any string that starts with the characters `ab`

   (c) Binary input with an even number of `1`'s

   (d) Binary input with an even number of both `0`'s and `1`'s

   (e) At least one other pattern that you think is interesting (and hopefully different from other students')

   For all automata, you may assume that the input alphabet Σ contains exactly the `char` values greater than 0 and less than 128. This range is the ASCII characters, including upper- and lowercase letters, numbers, and common punctuation. If you want to do something else, document your choice.

2. Implement non-deterministic finite automata that recognize the following languages:

   (a) Strings ending in `man`

   (b) Strings with more than one `a`, `g`, `h`, `i`, `o`, `s`, `t`, or `w`, or more than two `n`'s. That is, no string that it accepts can be the string "`washington`", although it may accept strings that are not "`washington`".

   (c) At least one other pattern that you think is interesting (and hopefully different from other students')

3. Implement a translator that receives as input an instance of an `NFA` and produces as output a (new) instance of a `DFA`. Your code should demonstrate the equivalence by running the original NFA and its translation on the three languages you used in Question 2 (the two given in the problem and the one you made up yourself). That is, you will have three functions that produce `NFA`s from Part 2 of the project. Pass the output of those functions to your converter and run the resulting `DFA` on user input as described previously.

Note that you do not need to be able to "read in" the specification of a pattern (for example as a regular expression) and create the instance of the automaton data structure. You may "hard-wire" it, meaning that your automaton-creating functions have code that creates *their specific automaton* using your data structures and APIs. Stay tuned for Unit 2 of the course if you want a crack at parsing...

Some Suggestions for How to Succeed

Here's the way I would approach this project, programming in C.

For starters, a DFA is not a complicated device. What do you need to represent, assuming the input alphabet Σ is given? We covered it in class. Now in Java you would write a class. In C, what do you use? If you don't know, go look at the "C for Java Programmers" document. Then code up the representation of a generic DFA (no specific behavior) in a data structure named `DFA`.

Next: How do you specify the behavior of a DFA (that is, what defines the pattern that it matches)? For each required pattern, you will need a function that allocates an instance of your generic `DFA` data structure, specializes it with the information required to match the pattern, and returns it (probably actually a pointer to it). You could have helper methods to do this to instances of your data structure (a form of "setter" methods).

Next: How do you "run" a DFA on an input string? That is, what does the DFA have to do to determine whether or not it matches the input? You should write a function that takes an instance of a `DFA` and an input and returns true or false.

You're almost done with part 1. Write the test function as described in the requirements. It will use your "run" function inside its REPL. Then write your main method that creates and tests each automaton described in the requirements. Voilà!

On to part 2. What's the difference between an NFA and a DFA? Hint: There can be more of something. So your `NFA` data structure will be very similar to your `DFA` data structure but some things will need to be sets. To help you out, we've provided code for a simple "set of `int`s" data structure if you want to use that, or design your own.

Then you need functions that specify the behavior of the NFA to recognize a pattern. This is like for DFAs but slightly different. Similarly, "running" an NFA on an input string is slightly different and what it means to match (accept) the string is slightly different. So different functions for NFAs, but they are generalizations of the DFA functions. Voilà!

Finally: Part 3. This part of the project is more challenging than the first two. You should know the algorithm for turning an NFA into an equivalent DFA. It is conceptually simple and easy to write in pseudocode. The challenge in implementing it is to keep track of all the states, sets of states, and transitions. You may need to revisit some of the design decisions that you made for the first parts of the project. If your changes are backwards-compatible, you won't have to change your code for parts 1 and 2.

Sample Code

We have provided the following on BlackBoard for your use if you want:

- Example header files for possible implementations of both DFA's and NFA's. These give you an idea of a possible API for your own implementation, as well as how to specify (partial) data types and (external) functions in header files.
- Full code for a simple implementation of a "set of `int`'s" using a bit-mask method (only works for `int`s between 0 and 63): `IntSet.[ch]`
- Full code for a generic implementation of a linked list that can store any reference (pointer) type: `LinkedList.[ch]`

Please report any bugs in this code ASAP. We will not be responsible for bugs reported at the last minute. We promise that all the code has been tested, but of course that doesn't mean it will work perfectly for you. Fixes will be announced on BlackBoard.

You should not need any other external code or libraries for this project. Develop your own data structures for your needs. They will be useful for future projects also.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) Any collaborators (see below)
   (b) How to build your project
   (c) How to run your project's program(s) to demonstrate that it/they meet the requirements
2. All source code for your project, including a `Makefile` or shell script that will build the program per your instructions. Do not just submit Eclipse projects without this additional information.

The TAs must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for them, the better grade your will be.**

Programming Policies

You must write your programs using the "C99" dialect of C. This means using the "`-std=c99`" option with `gcc` or `clang`. For more information, check Wikipedia.

You must also use the options "`-Wall -Werror`". These cause the compiler to report all warnings, and to make any warnings into errors that prevent your program from compiling. You should be able to write code without warnings in this course.

With these settings, your program should compile and run consistently on any platform.

Furthermore, your program must pass `valgrind` with no error messages. Of course this might differ for different executions, but it should be clean for the fixed examples requested in the project description. Any questions, ask the TAs *early*.

Projects in this course will be evaluated using `gcc` on Fedora Linux (recent version, like 25 or 26). Here are several ways you can setup this environment for yourself:

- Visit `getfedora.org` and download and install Fedora on a spare computer or separate partition of your main computer.
- Don't have a spare computer? No problem. Visit `virtualbox.org` and download VirtualBox. Then setup a new virtual machine running Fedora Linux (you may need to download the installer as above). VMWare is a commercial virtualization app if you want to buy something.
- Not confident setting up a VM or don't have space? No problem. Visit `docker.com` and download and install Docker for your computer. You can easily setup an image with Fedora Linux and `gcc`, or see below for more.
- Don't want to install anything? No problem. You need an account on the Computer Science Department's undergraduate cycle servers. Ask the TAs about it. Do not wait until the last minute. This will not happen overnight.

Docker usage: I have built a docker image based on Fedora linux and containing `gcc`, `make`, `gdb`, `valgrind`, and `emacs`. You can download it from here:

`http://www.cs.rochester.edu/u/ferguson/csc/docker/`

Once you have installed Docker, load the image to create the `fedora-gcc` image in your Docker installation:

```
docker load -i fedora-gcc-docker.tgz
```

Then you can do, for example:

```
docker run -it --rm -v "`pwd`":/home -w /home fedora-gcc bash
```

This runs `bash` (the command shell) using the `fedora-gcc` image. The `-it` option means to run interactively; `--rm` tells Docker to remove the container when the process exits; `-v` and the bit with the colon says to make your current working directory (outside docker) available on `/home` within docker; and `-w` tells docker to make `/home` the initial working directory for the `bash` process. Whew!

You can do a lot with Docker. You'll have to read the docs but it's a great option. Please report any problems with the image ASAP.

Late Policy

Don't be late. But if you are: 5% penalty for the first hour or part thereof, 10% penalty per hour or part thereof after the first.

Collaboration Policy

You will learn the most if you do the projects YOURSELF.

That said, collaboration on projects is permitted, subject to the following requirements:

- Groups of no more than 3 students, all currently taking CSC173.
- You must be able to explain anything you or your group submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the group should submit on the group's behalf and the grade will be shared with other members of the group. Other group members should submit a short comment naming the other collaborators.
- All members of a collaborative group will get the same grade on the project.