CSC173: Project 1
Finite Automata

The goal of this project is to experiment with pattern matching using finite automata. 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 and the instructions under "Project Submission" carefully. You **must** hit the spec to get full points.

If you are new to C and haven't yet done Homework 0.0, you should do that now.

Requirements

You must implement the following functionality, subject to the general requirements given in the next section.

1. Deterministic finite automata (DFAs) that, individually, recognize the following languages:
   (a) Exactly the string `happy`
   (b) Any string that starts with three `9`'s
   (c) Binary input with an even number of `1`'s
   (d) Binary input with an odd number of `0`'s and an even number of `1`'s
   (e) At least one other pattern that you think is interesting (and hopefully different from other students)
2. Non-deterministic finite automata (NFAs) that individually recognize the following languages:
   (a) Strings ending in `ing`
   (b) Strings containing `ing`
   (c) Strings with more than one `a`, `g`, `h`, `i`, `o`, `s`, `t`, or `w`, or more than two `n`'s (FOCS Figure 10.14). This automaton is described in FOCS as part of a project to find words that can be made from the letters of the word "Washington" (so-called "partial anagrams"). In this automaton, acceptance means that the

input string is **not** a partial anagram of `washington`, because it has too many of some letter. Note that the automaton in the book accepts when the criterion is met, but you will probably need to do something slightly different to accept an entire string meeting the criterion.[1]

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

3. Implement a translator function that takes an instance of an `NFA` as its only parameter and returns a new instance of a `DFA` that is equivalent to the original `NFA` (accepts the same language). Use the standard algorithm as seen in class and in the textbook.

   Your program should demonstrate that the translator works by using the first two NFAs from Part 2, translating them to DFAs, printing out how many states are in the resulting DFA, and running it on user input as described below. That is, you will have four functions that produce `NFA`s from Part 2 of the project. For the first two of those, pass the NFA that they return to your converter and run the resulting `DFA` on user input as described below.

   You should also try to convert the NFA from FOCS Figure 10.14 (the third NFA from part 2). You might want to think about (a) the complexity of the Subset Construction itself, and (b) the implementation of the data structures used by your translator, such as lists and sets. You can't beat (a), see FOCS pp. 550–552, but you could profile your code and try to do better on (b).

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.

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 each create **their specific automaton** using your data structures and APIs. Stay tuned for Unit 2 of the course if you want a crack at reading in a pattern specification, which is called "parsing."

[1] Please also note: On page 552, the textbook says that $768 + 8 \times 384 + 256 = 4864$, which is incorrect.

General Requirements

- You must submit code that compiles into a **single executable program** that addresses all parts of the project.
- If you are not using an IDE, your program (executable file) must be named `auto`. If you are using an IDE, your program must run when the project is run.
- 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 required pattern, 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. That is, you will have at least nine separate functions that create and return specific automata.
- You must have a function that takes an instance of an automaton as parameter and tests it 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). An example REPL session is given inthe next section. You may have two different versions of this function for the two different types of automata, if necessary.
- Your program must create and test automata in the order they are described above. 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.
- Note: If you are using an IDE, you must configure your project to use the compiler settings described under Programming Policies, below. In particular, you must use the compiler options "`-std=c99 -Wall -Werror`" or their equivalents. If this is new to you, seek help in a study session well before the deadline.

Sample Execution

The following is an example of roughly what your program should look like when it runs. Note that it is **your responsibility** to ensure that we can understand what your program is doing and whether that's correct. You should always prompt for input. You should always print back the user's input as confirmation. You should make sure the results are clearly visible.

```
CSC173 Project 1 by Ada Lovelace

Testing DFA that recognizes exactly "csc173"...
  Enter an input ("quit" to quit): csc
  Result for input "csc": false
  Enter an input ("quit" to quit): csc173
  Result for input "csc173": true
  Enter an input ("quit" to quit): quit

Testing DFA that recognizes an even number of 9's...
  Enter an input ("quit" to quit): 987123
  Result for input "987123": false
  Enter an input ("quit" to quit): 9999
  Result for input "9999": true
  Enter an input ("quit" to quit): quit
...
```

Suggestions for Success

Rule #1: Always start by designing your data structures and their APIs ("application programmer interfaces").

A DFA is not a complicated device. What do you need to represent? We covered it in class. Hint: 5-tuple. And the input alphabet Σ is fixed by the problem description, so you don't have to represent that explicitly.

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 and do Homework 0.0. Then code up the representation of a DFA (the 5-tuple, ignoring Σ) 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)? We covered it in class. Hint: some kind of table.

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). That would probably be helpful.

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? We covered it in class. You should write a function that takes an instance of a `DFA` and an input string as parameters and returns true or false as they are represented in C.

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.

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.

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.

Code Bundle

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

- `DFA.h`, `NFA.h`: 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.
- `IntHashSet`: Full code for an implementation of a set of `int`s using a hashtable with linked-list buckets, as described in FOCS pp. 360–363.
- `BitSet`: Full code for an implementation of a set of `int`s using a bit-mask method. This is faster than a hashtable but only works for `int`s between 0 and either 31 or 63, depending on your platform. USE AT YOUR OWN RISK!
- `LinkedList`: Full code for a generic implementation of a linked list that can store any reference (pointer) type.

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 may not use 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 `README.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. Eclipse projects must include the project settings from the project folder (`.project`, `.cproject`, and `.settings`). Non-Eclipse projects must include a `Makefile` or shell script that will build the program per your instructions, or at least have those instructions in your README.txt.

3. A completed copy of the submission form posted with the project description. Projects without this will receive a grade of 0. If you cannot complete and save a PDF form, submit a text file containing the questions and your (brief) answers.

We must be able to cut-and-paste from your documentation in order to build and run your code. **The easier you make this for us, the better grade your will be.** It is your job to make both the building and the running of programs easy and informative for your users.

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, see 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. We will deal with any platform-specific discrepancies as they arise.

If you submit an Eclipse project, it must have these settings associated with the project. Projects with that compile with warnings will be considered incomplete.

Furthermore, your program should pass `valgrind` with no error messages. If you don't know what this means or why it is A Good Thing, look at the C for Java Programmers document which has a short section about it. Programs that do not receive a clean report from `valgrind` have problems that **should be fixed** whether or not they run properly. If you are developing on Windows, you will need to look for alternative memory-checking tools.

Late Policy

Late projects will **not** be accepted. Submit what you have by the deadline. If there are extenuating circumstances, submit what you have before the deadline and then explain yourself via email.

If you have a medical excuse (see the course syllabus), submit what you have and explain yourself as soon as you are able.

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.

Academic Honesty

Do not copy code from other students or from the Internet.

Avoid Github and StackOverflow completely for the duration of this course.

There is code out there for all these projects. You know it. We know it.

Posting homework and project solutions to public repositories on sites like GitHub is a violation of the University's Academic Honesty Policy, Section V.B.2 "Giving Unauthorized Aid." Honestly, no prospective employer wants to see your coursework. Make a great project outside of class and share that instead to show off your chops.