CSC173: Project 1
Finite Automata

Finite automata are simple computing "devices" that recognize patterns, as seen in class and in the textbook.

For this project, you must implement a **framework** that you (or any programmer) can use to define and execute automata in order to recognize patterns. **The framework comes directly from the formal model of automata.**

Using your framework, you should be able to define and execute **any** finite automaton for **any** pattern that can be recognized by a finite automaton.

Note: The project is **not** about writing programs that recognize specific patterns. Yes, you will use your framework in a program that creates and executes a number of different automata (details below). But the specific patterns are **not** the point of the project. In fact, I recommend that you not even think about the specific patterns as you design and implement the framework.

Process not product.

Once you have the framework, for each pattern that you need to recognize, you must:

- Design the automaton for the pattern, probably by drawing its graph on your white-board or on paper.
- Translate the graph into the elements of the formal model of automata, in particular the transition function/table that comes from the edges of its graph.
- Write a function that uses your framework to create and return an instance of an automaton with that definition.
- Demonstrate your framework by running such an instance interactively, asking the user for input strings, running the automaton on the strings, and printing the results.

The rest of this document gives you the specific requirements as well some suggestions for how to get started. Read the requirements and instructions carefully. **You must meet the requirements to get the points.**

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

Part 1: Deterministic Finite Automata (50%)

Implement a framework for defining and executing deterministic finite automata (DFAs), as follows:

- You **must** define a `DFA` data structure and whatever functions you need to create, inspect and configure instances (constructor, accessors, and mutators).
- For each required DFA (see below), you **must** have a function that creates and returns a properly-configured `DFA`. For example, for a DFA that accepts strings containing "`abc`":

  ```
  DFA* DFA_for_contains_abc()
  ```

- You **must** have a function that can execute **any** `DFA` on a given input string and return true or false (accept or reject). For example:

  ```
  bool DFA_run(DFA* dfa, char* input)
  ```

- You **must** have a function that can run **any** `DFA` in a "Read-Eval-Print Loop" (REPL): prompting for user input, running the DFA on the input, and printing the result. For example:

  ```
  void DFA_repl(DFA* dfa)
  ```

- Your `main` function **must** run instances of each of the required `DFA`s in a REPL one after the other, printing informative messages.

You **must** demonstrate your `DFA` framework on all of the following languages:

(a) Exactly the string "`dfa`" (without the quotes)

(b) Strings that start with the sequence "`cat`" (without the quotes).

(c) Strings containing exactly two `2`'s (and perhaps other characters also).

(d) Binary input (input consisting of only `0`'s and `1`'s) where there are an even number of 0's and an odd number of 1's. (If you are not sure whether zero is even or odd, please got to study session.)

Part 2: Nondeterministic Finite Automata (30%)

Implement a framework (or extend your previous framework) to allow the definition and execution of non-deterministic finite automata (NFAs), as follows:

- You **must** define an `NFA` data structure and whatever functions you need to create, inspect and configure instances (constructor, accessors, and mutators).
- For each required NFA (see below), you **must** have a function that creates and returns a properly-configured `NFA`. For example, for an NFA that accepts strings ending with "`ing`":

  ```
  NFA* NFA_for_ends_with_ing()
  ```

- You **must** have a function that can execute **any** `NFA` on a given input string and return true or false (accept or reject). For example:

  ```
  bool NFA_run(NFA* nfa, char* input)
  ```

- You **must** have a function that can run **any** `NFA` in a REPL, prompting for user input, running the NFA on the input, and printing the result. For example:

  ```
  void NFA_repl(NFA* nfa)
  ```

- Your `main` function **must** run each of the required `NFA`s in a REPL one after the other, printing informative messages.

You **must** demonstrate your `NFA` framework on all of the following languages:

(a) Strings ending in `ked`, such as "`liked`" and "`sacked`", but not, for example, "`shakedown`".

(b) Strings containing `ath`, such as "`athletic`", "`bather`", and "`path`".

(c) Strings that have more than one `o`, `f`, or `r`, or more than two `c`'s or `n`'s, or more than three `e`'s. In this automaton, acceptance means that the input string is **definitely not** a partial anagram of `conference`, because it has too many of some letter.

Note that this automaton does *not* accept *all* strings that are not anagrams of `conference`. See FOCS Ex. 10.6 and Fig. 10.14 for a similar automaton and the story behind it. 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]

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

Part 3: Equivalence of DFAs and NFAs (20%)

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

```
DFA* NFA_to_DFA(NFA* nfa)
```

Demonstrate your NFA to DFA translator 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 three functions that produce `NFA`s for 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` using your DFA REPL function.

You should also try to convert 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).

FWIW: My implementation using `IntHashSet` on an M1 MacBook Pro with 32GB of RAM took about ten minutes to convert the "Washington"-based NFA from FOCS Ex. 10.6.

General Requirements

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.

You must submit code that compiles into a **single executable program** that addresses all the parts of the project that you attempt. Your program **must** compile with the required compiler options and run without crashing.

Your program **must** print to standard output with `printf` and read from standard input with `fgets` on `stdin`. **Do not use** `gets` and be very careful with `scanf` since it skips

whitepsace including newlines. Remember that the empty string is a valid input for an automaton.

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 framework. In Unit 2 of the course We will look at a model that can be used to read ("parse") regular expressions, from which you could create automata.

Please see below regarding programming requirements, compiler settings, and IDEs. If this is new to you, and perhaps you didn't do Homework 0.0, your TAs are here to help! Come to a study session **well before the deadline**, not at the last minute.

Sample Execution

The following is an example of 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 George Ferguson

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
...
```

Where do I start???

A DFA is not a complicated thing:

> The behavior of an automaton is conceptually simple. [...] It begins in the start state, about to read the first character of the input sequence. Depending on that character, it makes a transition, perhaps to the same state, or perhaps to another state. The transition is dictated by the graph of the automaton. The automaton then reads the second character and makes the proper transition, and so on. [...] When it finishes reading the input string, it decides whether to accept or reject the input. (FOCS Sec. 10.2, pp. 533–535)

How do we describe a DFA formally? Hint: 5-tuple.

So for our framework, we need a data structure that allows us to specify the five components of the definition of a DFA (**any** DFA).

The first component of the formal definition of a DFA is... its set of states S. We could have something called a "state" and then use a set of them in our DFA. It would work. But perhaps there's an alternative.

Following the textbook and as seen in class, let's number the states of a DFA with consecutive integers. So state 0, state 1, and so on up to state $n-1$, where n is however many states there are in the automaton. In that case, all we need to keep track of is how many states we have (n). Easy. Note that this also allows us to use states (integers) as indexes into arrays or lists. That might be useful.

The second component of the formal definition of a DFA is... input alphabet Σ. We could represent a set of characters, but the project description tells us what it is for this project, so we really don't need to store it explicitly (for this project).

The remaining three components of a DFA describe its behavior. You should know what they are and how they define the behavior. One of them tells you what state the DFA starts in, one of them tells you how it behaves, and one of them tells you what it does when it finishes reading the input.

The initial state is a state, or in our framework...? An integer. Easy.

How about the transition function? Well, it's a function. You're welcome.

You could write C code for it, sort of like what they do in FOCS Section 10.3 (Figure 10.7), although I don't recommend writing code like that these days.[2] And anyway, the project requires that you be able to create *instances* of DFAs and run them on various inputs. To do that in C, you would need to isolate the decision-making part and use function pointers (C for Java Programmers, Section 8.8) to refer to it in your instances. Not impossible, but kind of a power move.

Maybe there's another way to specify the behavior, the transition function, of a DFA... something that we saw in class that is not in FOCS...

Any discrete, finite function of two arguments, for example, a state s and a symbol x, can be represented as a **table** with one entry for each combination of s and x. You can implement a table as a sequence (array or list) of rows, each of which is a sequence of entries. Or you can implement it as a two-dimensional array of entries. For a transition function, the values of the function (the entries in the table) are... states, or in our framework... integers. Nice.

Finally, the set of accepting states. You could code up a set of integers. If you know how many you have, it could just be an array of them. There are other possibilities.

Last design question: What does it mean for a DFA to "run" on an input? We did it in class and the description from FOCS is repeated at the start of this section...

The DFA is always in some current state. Prior to reading an input string, it initializes its current state to the initial state specified in its definition. It then reads the next symbol from the input, uses the current state and current symbol to call its transition function or look up in its transition table to get the next state (if any). If there is a next state, make it the current state and repeat. If there is no next state, then by definition the DFA rejects the input. If the DFA reaches the end of the input (or the input was the empty string), then it accepts the input if and only if the current state is an accepting state.

Our design is done. Whew. But that's the most important part. The rest is just the challenge of translating our design into some programming language. You might find it helpful to do it in Java first, since you are probably pretty fluent in Java. But eventually you need to do it in C.

[2] See Dijkstra, E. W. (1968) Letters to the editor: go to statement considered harmful. *Communications of the ACM* 11(3). doi:10.1145/362929.362947

Here are a couple of things to think about:

- The project requires that you be able to create *instances* of DFAs that do various things, and then run those instances. How would you do that in Java? ... Define a class. In C, what do you use? If you don't know, go look at Chapter 7 of the "C for Java Programmers" guide and do Homework 0.0 (Tutorial Lesson 3).
- Allocating arrays in C is very similar to allocating them in Java. Allocating multi-dimensional arrays requres allocating the array for the first dimension, and then allocating the arrays for the second dimension. See Section 8.6 of the "C for Java Programmers" guide. Or instead of a two-dimensional array, use an array of instances of some data structure, perhaps an `Entry` or a `Row`.
- For each required pattern, you **must** have a separate function that allocates an instance of your generic `DFA` data structure, specializes it with a transition function or table and other properties as needed for matching the pattern, and returns the new `DFA` instance (probably as a pointer to it). For "programming" the DFA, I suggest using helper functions that change a `DFA`'s transition table (a form of "setter" or "mutator" methods). That would probably be much clearer than a bunch of assigment statements. And it's the approach used in the example header file included in the project's code bundle.
- You **must** have a **single** function that takes an instance of a DFA (**any** DFA) and an input string, runs the DFA on the string, and prints something informative. We did the design previously. You just need to write the code.
- You **must** have a **single** function that takes an instance of a DFA (**any** DFA) and runs it in a REPL (read-eval-print loop). It will use your "run" function inside a loop. This should not be hard to write. The output must make it clear what your program is doing.
- Then write your main method that creates and tests each automaton described in the requirements using the REPL. If you have designed the code properly, this will be short and easy to understand.

On to part 2.

What's the main difference between an NFA and a DFA? Hint: There can be more than one 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 (see next section), or you can design and build your own.

Then you need functions that specify the behavior of the NFA. This is like for DFAs but slightly different. Similarly, "running" an NFA on an input string is slightly different (as

seen in class) and what it means to accept a string is slightly different. So different functions for NFAs, but they are generalizations of the DFA functions, as seen and discussed in class.

Finally: Part 3. This part of the project is more challenging than the first two. And you can get a good grade on the project even without doing this part.

You should know the algorithm for turning an NFA into an equivalent DFA. It is conceptually simple and straightforward to write in pseudocode (as seen in class). 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 code in a bundle 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. You are welcome to use them, change them, ignore them, whatever.
- `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. I **STRONGLY** recommend that you **NOT** use this.
- `LinkedList`: Full code for a generic implementation of a linked list that can store any reference (pointer) type. Might be useful now or in the future. Use at your own risk.

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.

To use our code, be sure that you know how to use code from multiple files in one program. **Do NOT use** `#include` with "`.c`" files. If this is new to you, see Chapter 10 of the "C for Java Programmers" guide (also Lesson 5 of the tutorial) and get yourself to a study session ASAP.

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.

Additional Requirements and Policies

The short version:

- You **must** use the following C compiler options:
  `-std=c99 -Wall -Werror`
- If you are using an IDE, you **must** configure it to use those options (but I suggest that you take this opportunity to learn how to use the command-line).
- You **must** submit a ZIP including your source code and a README by the deadline.
- You **must** tell us how to build your project in your README.
- You **must** tell us how to run your project in your README.
- Projects that do not compile will receive a grade of **0**.
- Projects that do not run or that crash will receive a grade of **0** for whatever parts did not work.
- Late projects will receive a grade of **0** (see below regarding extenuating circumstances).
- You will learn the most if you do the project yourself, but collaboration is permitted in teams of up to three (3) students.
- Do not copy code from other students or from the Internet.

Detailed information follows. . .

Programming Requirements

C programs **must** be written 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 **must** 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 platform-specific discrepancies as they arise.

If you are using an IDE (Eclipse, XCode, VSCode, CLion, *etc.*), you **must** ensure that it will also build as described above. The easiest way to do that is to setup the IDE with the required compiler options. There are some notes about this in the C Programming Resources (for CSC173 and beyond) area.

You may **NOT** use `#pragma`'s in your programs. This includes `#pragma`'s added by an IDE or any other way that they might get into your code.

You **SHOULD** test your program with the memory checking program `valgrind`. If you don't know what `valgrind` is or why it is A Good Thing, read C for Java Programmers Chapter 11: Debugging a C Program.

Programs that do not receive a clean report from valgrind have problems that **should be fixed** whether or not the program sometimes runs properly on some platforms. The only exception is unfreed memory errors (so-called "memory leaks"). We will not penalize you for those in CSC173.

If your program does not work for us, the first thing we're going to do is run valgrind on it. If valgrind reports errors: you have errors in your program. Period.

It is easy to run valgrind in a virtual machine or Docker container if you cannot install and run it natively. Mac and Windows users can install Docker Desktop. See How to run Linux (including via Docker) for more information about Linux and Docker.

For help with `valgrind`, please go to study session **well before** the project deadline.

Submission Requirements

You **must** submit your project as a ZIP archive of a folder (directory) containing the following items:

1. A file named `README.txt` or `README.pdf` (see below)
2. The source code for your project (do not include object files or executables in your submission)
3. A completed copy of the submission form posted with the project description.

The name of the folder in ZIP **must** include "CSC173", "Project 1" (or whatever), and the NetID(s) of the submitters. For example: "`CSC173_Project_1_aturing`"

Your README **must** include the following information:

1. The course: "CSC173"
2. The assignment or project (*e.g.*, "Project 1")
3. Your name and email address
4. The names and email addresses of any collaborators (per the course policy on collaboration)
5. Instructions for building your project (with the required compiler options)
6. Instructions for running your project

The purpose of the submission form is so that we know which parts of the project you attempted and where we can find the code for some of the key required features.

- **Projects without a submission form or whose submission form does not accurately describe the project 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.

Project Evaluation

You **must** tell us in your README how to build your project and how to run it.

Note that we will NOT load projects into Eclipse or any other IDE. We **must** be able to build and run your programs from the command-line. If you have questions about that, go to a study session.

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 your grade will be.** It is **your** job to make the building of your project easy and the running of its program(s) easy and informative.

For C projects, the most common command for building a program from all the C source files in the directory (folder) is:

```
gcc -std=c99 -Wall -Werror -o EXECUTABLE *.c
```

where `EXECUTABLE` is the name of the executable program that we will run to execute your project.

You may also tell use to build your project using `make`. In that case, be sure to include your `Makefile` with your submission. You **must** ensure that your `Makefile` sets the compiler options appropriately.

If you expect us to do something else, you **must** describe what we need to do in your `README` file. This is unlikely to be the case for most of the projects in CSC173.

Please note that we will **NOT** under any circumstances edit your source files. That is your job.

Projects that do not compile will receive a grade of 0. There is no way to know if your program is correct solely by looking at its source code (although we can sometimes tell that is incorrect). This is actually an aspect of a very deep result in Computer Science that we cover in CSC173.

We will then run your program by running the executable, or as described in the project description. If something else is required, you **must** describe what is needed in your `README` file.

Projects that do not run or that crash will receive a grade of 0 for whatever parts did not work. You earn credit for your project by meeting the project requirements. Projects that do not run **do not** meet the requirements.

Any questions about these requirements: go to study session **BEFORE** the project is due.

Late Policy

Late projects will receive a grade of 0. You **must** 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

I assume that you are in this course to learn. You will learn the most if you do the projects **yourself**.

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

- Teams of no more than 3 students, all currently taking CSC173.
- You **must** be able to explain anything you or your team submit, IN PERSON AT ANY TIME, at the instructor's or TA's discretion.
- One member of the team should submit code on the team's behalf in addition to their writeup. Other team members **must** submit a README (only) indicating who their collaborators are.
- All members of a collaborative team will get the same grade on the project.

Working in a team only works if you actually do all parts of the project together. If you only do one part of, say, three, then you only learn one third of the material. If one member of a team doesn't do their part or does it incorrectly (or dishonestly), all members pay the price.

Academic Honesty

I assume that you are in this course to learn. You will learn nothing if you do not do the projects **yourself**.

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

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

The use of generative AI tools is **NOT** permitted in this course.

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.