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 proceed. 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 DFAs 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 “CSC” (capitalized, without the quotes)
(b) Strings that contain the sequence “end” (without the quotes).
(c) Strings starting with a vowel (a,e,i,o, or u).
(d) Binary input with an even number of both 0’s and 1’s. (Note: zero is an even number, and you should know why.)

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 NFAs 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 at, such as “at” and “cat”, but not, for example, “attack”.

(b) Strings containing got, such as “got”, “forgot”, and “forgotten”.
(c) Strings that have more than one a,e,h,i, or g, or more than two n’s or p’s. In this automaton, acceptance means that the input string is definitely not a partial anagram of happening, because it has too many of some letter.
Note that this automaton does not accept all strings that are not anagrams of happening. 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.

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 NFAs 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 Σcontains 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
...
```