CSC173: Project 1
Finite Automata

1. Implement deterministic finite automata (DFAs) 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) Any other examples that you think are interesting

   A DFA is not a complicated device. Think about how it is specified and code that in C. Include easy ways to test or demonstrate your implementation.

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

   (a) Strings ending in `man`

   (b) Strings that contain more than one each of the letters in `washington` other than `n`, or more than two `n`'s.

   (c) Any other examples that you think are interesting

   An NFA differs from a DFA only in that the finite control specifies a set of possible next states rather than a single state. This implies that (a) an NFA can be "in" any of a set of possible states, and (b) an NFA accepts if any execution path ends in an accepting state.

3. Implement a translator that receives as input an NFA and produces as output a DFA. Your code should demonstrate the equivalence by running the original NFA and its translation on examples from:

   (a) Both languages given in question (2) above

   (b) At least one other language defined by a non-trivial NFA

   The "Subset Construction" algorithm is conceptually simple (see FOCS pp. 548-549). The challenge in implementing it is to keep track of all the states, sets of states, and transitions. You may have made some design decisions in implementing DFAs and/or NFAs that worked well for their original purposes but aren't as well-suited to supporting the translation algorithm. Feel free to use alternative implementations for this part of the project if you like.

You may make any reasonable assumptions about the input vocabulary, such as assuming alphanumeric characters, ASCII characters, the digits 0 and 1, *etc.*.

If your demonstration programs require user interaction, be sure that prompts are clear and be sure to document *everything* about how to run and use your programs in your submission's documentation. Your job is to make your user's job easy.

It is possible to write these each as separate programs, but it would involve massive code duplication. Better to define the core DFA and NFA data structures and functions in code that can be used by other programs. For this, you need to put the declarations in a "header" (`.h`) file that is included by both your implementation (`.c`) file and the `.c` files that use DFAs or NFAs in programs. Then compile all the files and link them as appropriate into one or more executable programs.

If this seems complicated, start by doing the DFA as a standalone program (a single `.c` file). Then break into into the DFA "library" (`.c` and `.h`) and a program that demonstrates a specific DFA (a different `.c` file). Link the compiled (`.o`) files together to make an executable program.

Sample Code

We have provided the following on BlackBoard:

- 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 could do something different, but then you might want to explain why in your documentation...
- Full code for a simple implementation of a "set of `int`'s" as a linked list (you should know from CSC172 that using a linked list for a set datatype is not the greatest idea): `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.

Project Submission

Your project submission MUST include the following:

1. A README.txt file or PDF document describing:
   (a) How to build your project
   (b) How to run your project's program(s) to demonstrate that it/they meet the requirements
   (c) Any collaborators (see below)
   (d) Acknowledgements for anything you did not code yourself (you should avoid this, other than the code we've given you, which you don't have to acknowledge)
2. All source code for your project, including a `Makefile` or shell script that will build the program per your instructions. TAs may accept Eclipse projects, but you should try not to count on it.

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.

Please watch BlackBoard for updates as we may need to make adjustments to these requirements.

Course Collaboration Policy

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.