CS 496: Assignment 1
Due: 4 June, 11:55pm

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

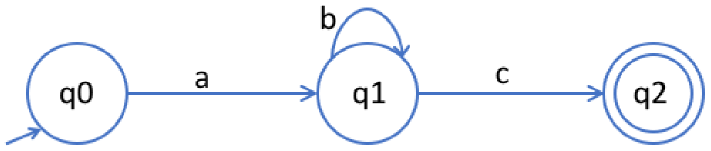
Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

$\boxed{\text{This assignment is about coding with lists, tuples and records.}}$

A *finite automaton* (FA) is a simple machine that recognizes sequences of symbols from a given alphabet. We assume that the sequence of symbols is written on a tape, the tape is a sequence of cells and each cell holds one symbol from the alphabet. A FA starts scanning these cells, one by one, from left to right. We present two examples next.

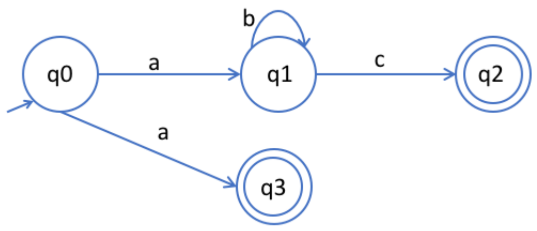
Example 2.1 *Suppose we have an alphabet of three symbols: 'a', 'b' and 'c'. Below is a pictorial representation of a FA that recognizes all the sequences of symbols of that alphabet that start with an 'a', have zero or more 'b's in the middle, and then end with a 'c':*



The designated start state *is q_0; this is indicated with a short arrow. Given the current state of a FA and the current cell being scanned, the arrow indicates the next state that the FA is to transition to. The state q_2 is a* final state *and is indicated with two concentric circles. A FA that scans all the tape and ends in a final state is said to* accept *the sequence of symbols.*

A FA is *non-deterministic* if it has a state that has two or more successor states with the same symbol. Example 2.1 is deterministic. The next example is that of a non-deterministic FA:

Example 2.2 *This FA is non-deterministic since, at state $q0$ the FA could either transition to $q1$ or to $q3$ while scanning an 'a':*



A more precise definition of Finite Automata follows. A *finite automaton* is a tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- Σ is a finite set of input symbols called the alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function;
- $q_0 \in Q$ is an initial or start state; and
- $F \subseteq Q$ is a set of accept states.

In our example, $Q = \{q_0, q_1, q_2\}$. The set of final states is $F = \{q_2\}$.

How an automaton operates. Given an input sequence of symbols, the automaton attempts to recognize it. Initially the automaton is in the start state and an input pointer points to the first symbol in the input sequence. It then follows the transition function (*cf.* the "arrows" in the figure above), changing states according to the next symbol in the sequence to be processes. For example, the arrow from q0 to q1 labeled "a" indicates that our automaton, if it is in state q0 and the next symbol in the input is an "a", then it "moves" to state q1 and passes on to process the next symbol in the input. If there is more than one possible next state, we can choose any of them.

3 Simulating Automata

3.1 Encoding Automata

We will use the following user defined datatypes.

```
type symbol = char
type input = char list

type state = string

(* transition function *)
type tf = (state * symbol * state) list

(* start state * transition function * end state *)
type fa = { states: state list; start:state; tf: tf; final: state list}
```

Below is how we encode the automaton from the examples above. We give them names `a1` and `a2`, so that can refer to it later. Both are coded as records:

```
let a1 = {states = ["q0";"q1";"q2"];
          start = "q0";
          tf = [("q0",'a',"q1"); ("q1",'b',"q1"); ("q1",'c',"q2")];
          final = ["q2"]}

let a2 = {states = ["q0";"q1";"q2";"q3"];
           start = "q0";
           tf = [("q0",'a',"q1"); ("q0",'a',"q3"); ("q1",'b',"q1")
                ; ("q1",'c',"q2")];
           final= ["q2";"q3"]}
```

3.2 Your Task

Implement the following functions. You should work on the file `fa_stub.ml`. First rename it to `fa.ml` and then start completing the exercises given below. Unless otherwise stated, you may assume that any function that has a FA as input, that FA is valid (in the sense explained below).

1. `apply_transition_function : tf -> symbol -> state -> state list`

   The expression `apply_transition_function f sym st` applies the transition function `f` to the symbol `sym` assuming that the current state is `st`. For example:

```
# apply_transition_function a1.tf 'a' "q0";;
- : state option = ["q1"]
# apply_transition_function a2.tf 'a' "q0";;
- : state option = ["q1";"q3"]
# apply_transition_function a1.tf 'b' "q0";;
- : state option = []
# apply_transition_function a1.tf f 'c' "q0";;
- : state option = []
# apply_transition_function a1.tf f 'c' "this_state_does_not_exist";;
- : state option = []
```

2. `accept : fa -> input -> bool`

   Determine whether a word is accepted by a finite automaton. Here are some examples:

```
# accept a1 (input_of_string "abbc");;
- : bool = true
# accept a1 (input_of_string "ac");;
- : bool = true
```

```
# accept a1 (input_of_string "a");;
- : bool = false
# accept a1 (input_of_string "bb");;
- : bool = false
```

The `input_of_string` function is just a helper function that translates strings to lists of symbols. It is provided in the stub so you need not implement it.

3. `is_deterministic : fa -> bool`

   This function checks whether the given automaton is deterministic or not.

4. `is_valid : fa -> bool`

   Implement `is_valid` that checks for validity. A FA is said to be *valid* if

   (a) The list of states has no duplicates;

   (b) The start state belongs to set of states;

   (c) The final states; and belong to set of states.

5. `next : tf -> state -> state list`

   This functions returns the list of all the states that are successors of some given state. For example,

```
# next a1.tf "q1";;
- : state list = ["q1"; "q2"]
# next a1.tf "q0";;
- : state list = ["q1"]
# next a2.tf "q0";;
- : state list = ["q1"; "q3"]
```

6. `reachable : fa -> state list`

   Reports list of states that are reachable from the start state, in any order. Note that the start state is stored in the `start` field of the FA. Here is an example:

```
# reachable a1;;
- : state list = ["q0"; "q1"; "q2"]
```

7. `non_empty : fa -> bool`

   Determines whether a FA accepts at least one word. Hint: make sure that at least one final state is reachable from the start state.

8. `remove_dead_states : fa -> fa`

   Removes all dead (i.e. unreachable) states from a valid FA. This includes removing them from the set of states, removing the transitions in δ that involve dead states and also removing them from the set of final states.

4 Submission instructions

Submit a single file named `fa.ml` through Canvas. No report is required. Your grade will be determined as follows:

- You will get 0 points if your code does not compile.
- Partial credit may be given for style, comments and readability.