Replacement Coursework 2 (Automata)

This coursework is worth 10%. It is about deterministic and non-deterministic finite automata. The coursework is due on ??? March at 5pm. Make sure the files you submit can be processed by just calling `scala <<filename.scala>>`.

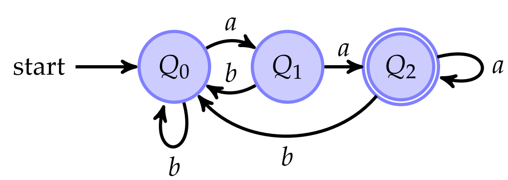
Important: Do not use any mutable data structures in your submission! They are not needed. This means you cannot use `ListBuffers`, for example. Do not use `return` in your code! It has a different meaning in Scala, than in Java. Do not use `var`! This declares a mutable variable. Make sure the functions you submit are defined on the "top-level" of Scala, not inside a class or object. Also note that the running time will be restricted to a maximum of 360 seconds on my laptop.

Disclaimer

It should be understood that the work you submit represents your own effort! You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Part 1 (Deterministic Finite Automata)

There are many uses for Deterministic Finite Automata (DFAs), for example testing whether a string should be accepted or not. The main idea is that DFAs consist of some states (circles) and transitions (edges) between states. For example consider the DFA



where there are three states (Q_0, Q_1 and Q_2). The DFA has a starting state (Q_0) and an accepting state (Q_2), the latter indicated by double lines. In general, a DFA can have any number of accepting states, but only a single starting state (in this example only a and b).

Transitions are edges between states labelled with a character. The idea is that if I am in state Q_0, say, and get an a, I can go to state Q_1. If I am in state Q_2 and get an a, I can stay in state Q_2; if I get a b in Q_2, then I have to go to state Q_0. The main point of DFAs is that if I am in a state and get a character, it is always clear which is the next state—there can only be at most one. The task of Part 1 is to implement such DFAs in Scala using partial functions for the transitions.

Tasks

(1) Write a polymorphic function, called `share`, that decides whether two sets share some elements (i.e. the intersection is not empty). [1 Mark]

(2) The transitions of DFAs are given by partial functions, with the type of (state, character)-pair to state. For example the transitions of the DFA given above can be defined as

```
val dfa_trans : PartialFunction[(State,Char), State] =
  { case (Q0, 'a') => Q1
    case (Q0, 'b') => Q0
    case (Q1, 'a') => Q2
    case (Q1, 'b') => Q0
    case (Q2, 'a') => Q2
    case (Q2, 'b') => Q0
  }
```

The main idea of partial functions (as opposed to functions) is that they do not have to be defined everywhere. For example the transitions above only mention characters *a* and *b*, but leave out any other characters. Partial functions come with a method `isDefinedAt` that can be used to check whether an input produces a result or not. For example

```
dfa_trans.isDefinedAt((Q0, 'a'))
dfa_trans.isDefinedAt((Q0, 'c'))
```

gives `true` in the first case and `false` in the second.

Write a function that takes transition and a (state, character)-pair as arguments and produces an optional state (the state specified by the partial transition function whenever it is defined; if the transition function is undefined, return None). [1 Mark]

(3) Write a function that "lifts" the function in (2) from characters to strings. That is, write a function that takes a transition, a state and a list of characters as arguments and produces the state generated by following the transitions for each character in the list. For example you are in state Q_0 in the DFA above and have the list `List(a,a,a,b,b,a)`, then you need to generate the state Q_1 (as option since there might not be such a state). [1 Mark]

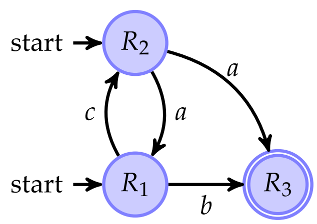
(4) DFAs are defined as a triple: (staring state, transitions, final states). Write a function `accepts` that tests whether a string is accepted by an DFA or not. For this start in the starting state of the DFA, use the function under (3) to calculate the state after following all transitions according to the characters in the string. If the state is a final state, return true; otherwise

false.

[1 Mark]

Part 2 (Non-Deterministic Finite Automata)

The main point of DFAs is that for every given state and character there is at most one next state (one if the transition is defined; none otherwise). However, this restriction to at most one state can be quite limiting for some applications.[1] Non-Deterministic Automata (NFAs) remove this restriction: there can be more than one starting state, and given a state and a character there can be more than one next state. Consider for example



where in state R_2 if you get an a, you can go to state R_1 *or* R_3. If we want to find out whether a NFA accepts a string, then we need to explore both possibilities. We will do this "exploration" in the tasks below in a breath-first manner. The possibility of having more than one next state in NFAs will be implemented by having a *set* of partial transition functions. For example the NFA shown above will be represented by the set of partial functions

```
val nfa_trans : NTrans = Set(
  { case (R1, 'c') => R2 },
  { case (R1, 'b') => R3 },
  { case (R2, 'a') => R1 },
  { case (R2, 'a') => R3 }
)
```

The point is that the 3rd element in this set states that in R_2 and given an a, I can go to state R_1; and the 4th element, in R_2, given an a, I can go to state R_3. When following transitions from a state, we have to look at all partial functions in the set and generate the set of all possible next states.

Tasks

(5) Write a function `nnext` which takes a transition set, a state and a character as arguments, and calculates all possible next states (returned as set).

[1 Mark]

[1] Though there is a curious fact that every NFA can be translated into an "equivalent" DFA, that is accepting the same set of strings. However this might increase drastically the number of states in the DFA.

(6) Write a function `nnexts` which takes a transition set, a *set* of states and a character as arguments, and calculates *all* possible next states that can be reached from any state in the set.

[1 Mark]

(7) Like in (3), write a function `nnextss` that lifts `nnexts` from (6) from single characters to lists of characters. [1 Mark]

(8) NFAs are also defined as a triple: (set of staring states, set of transitions, final states). Write a function `naccepts` that tests whether a string is accepted by a NFA or not. For this start in all starting states of the NFA, use the function under (7) to calculate the set of states following all transitions according to the characters in the string. If the set of states shares and state with the set of final states, return true; otherwise false. Use the function under (1) in order to test whether these two sets of states share any states [1 Mark]

(9) Since we explore in functions under (6) and (7) all possible next states, we decide whether a string is accepted in a breath-first manner. (Depth-first would be to choose one state, follow all next states of this single state; check whether it leads to a accepting state. If not, we backtrack and choose another state). The disadvantage of breath-first search is that at every step a non-empty set of states are "active"…that need to be followed at the same time. Write similar functions as in (7) and (8), but instead of returning states or a boolean, these functions return the number of states that need to be followed in each step. The function `max_accept` should return the maximum of all these numbers.

Consider again the NFA shown above. At the beginning the number of active states will be 2 (since there are two starting states, namely R_1 and R_2). If we get an a, there will be still 2 active states, namely R_1 and R_3 both reachable from R_2. There is no transition for a and R_1. So for a string, say, ab which is accepted by the NFA, the maximum number of active states is 2 (it is not possible that all states are active with this NFA; is it possible that no state is active?). [2 Marks]