CS 415: Programming Languages

Homework 2: Ocaml

Due Friday, 23 September by 10 a.m.

For this homework, you will need to write three Ocaml functions that deal with DFAs and NFAs:

- dfasimulate: given a DFA and input, will return the state transition of the progress through the DFA, and true or false if the DFA ended in a final state after reading in all the input
- nfasimulate: same as dfasimulate, but for an NFA (this function can't use nfa2dfa, however)
- nfa2dfa: given an NFA, this function will convert it to a DFA

The only requirements are:

- The types of the functions must be as specified below, to standardize the specification for the FAs, and thus enable us to test your code
- You may NOT use any iterative programming, objects, or functions that have side-effects (this includes print statements) even though Ocaml allows this – the homework must use "pure" functions and recursion only (you are welcome to have these aspects in there while you are developing – just not in the final submission)
- You may only use the functions in the List and String modules as well as the standard OCaml functions

FA specification

All states and transitions in the FAs will have string names. You can assume that the states we use in our test FAs will only have state names made of digits and/or letters (although the transitions may include punctuation). The input for both types of FAs will be a `string list`, where the various elements of the list are the transitions the FA should take.

The type for both FAs must be `string * (string * char * string) list * string list`. The first part of the tuple, the `string`, is the initial (or current) state of the FA. The second part of the tuple, the `(string * string * string) list`, contains a list of all the transitions of the FA. A transition consists of an initial state, a symbol, and the destination state. The final part of the tuple, the `string list`, is a list of the final states. Note that the FA format is the same for both DFAs and NFAs. Also note that both state names and transitions can be strings of length greater than 1. You can assume there will be no common names between the states and transitions.

Sample OCaml code for a DFA and an NFA are given below, and more will be available on the website.

Function requirements

The two simulate functions must take in two parameters each: the FA (described above), and a list of transitions to take (as a `string list`). You are welcome to have as many other functions in your submission file as needed. Both simulate functions return a tuple of type `string list * bool`. The `string list` is the list of all the states visited. The `bool` is whether the FA is in a final state after reading the input. Thus, both simulate functions must have the following type: `string * (string *`

`string * string) list * string list -> string list -> string list * bool`. Note that you may want to pass more or less parameters into the recursive function calls for this simulation – in that case, have the dfasimulate or nfasimulate function be a wrapper function that has the above type.

The nfa2dfa function must take in an NFA and return a DFA. Thus, the type is `'a * (string * 'b * string) list * string list -> 'a * (string * 'b * string) list * string list`. Your function can have the type list `string` for `'a` and `'b`, though. The format of the FAs is as described above.

Example FAs

The DFA described in the textbook (page 603) is specified as follows. This DFA will accept any input that has an even number of 0's and an even number of 1's. The first input is a valid input for this DFA; the second is not.

```
let dfa1 = ("0",
   [("0", "0", "2"); ("0", "1", "1"); ("1", "0", "3"); ("1", "1", "0");
    ("2", "0", "0"); ("2", "1", "3"); ("3", "0", "1"); ("3", "1", "2")],
   ["0"]);;
let dfainput1 = ["0";"1";"0";"1";"0";"0"];;
let dfainput2 = ["0";"1";"0";"1";"0";"0";"1"];;
```

The code below describes a NFA. As with the DFA input, the first input is valid, the second is not.

```
let nfa1 = ("0",
   [("0", "a", "1"); ("0", "a", "2"); ("1", "a", "1"); ("1", "a", "2");
    ("2", "b", "1"); ("2", "b", "3"); ("3", "a", "1"); ("3", "a", "2")],
   ["0"; "1"]);;
let nfainput1 = ["a";"b";"a";"b";"a";"b";"a";"a";"a";"b";"a";"a"];
let nfainput2 = ["a";"a";"b";"a";"b";"b"];;
```

More examples of NFAs and DFAs will appear on the course website.

How to start

There are three parts to this assignment

The first part, dfasimulate, is (relatively) easy, as you are converting the Scheme code in the book (page 603) to Ocaml. The difficulty here will be learning Ocaml syntax. You may want to look at the user's manual for Ocaml at http://caml.inria.fr/pub/docs/manual-ocaml/, as well as the lecture slides.

The second part, nfasimulate, is moderately difficult. You will have to implement a depth-first search through the NFA.

The last part, nfa2dfa, is more difficult. This will require a bit of thought as to how to design the algorithm. In particular, you will need to implement each of the steps for converting NFAs to DFAs. If you are unsure about the algorithm, you can look in your course textbook for CS 302, do a Google search for "convert nfa to dfa" (the first search result is where the above NFA came from), or ask for help. You may find it easier to manipulate the state names during the conversion as lists instead of strings – you can use String.concat function for this.