OCAML PROGRAMMING

Deterministic Finite Automata

DFA

- A deterministic finite automaton M is a 5-tuple, $(Q, \Sigma, \delta, q0, F)$, consisting of
  - a finite set of states (Q)
  - a finite set of input symbols called the alphabet (Σ)
  - a transition function $(\delta : Q \times \Sigma \rightarrow Q)$
  - a start state $(q0 \in Q)$
  - a set of accepting states $(F \subseteq Q)$

States & Alphabet

- type state = int
  - (* For our symbols, we're going to just use ocaml's built in char type *)
- type symbol = char

Transition Function

- To represent a transition function, we're actually going to represent a table. The table is going to tell us where to go on a given input state q, and a given symbol s.

Current State	Input Symbol	Next State
0	“a”	1
1	“b”	2

OCAML Impl

- We represent transition funnction as a list of *tuples* in OCaml, which generalize pairs.
- Remember that a pair type is something of the form `'a * 'b` where 'a and 'b are any type
- type transition = int * symbol * int

DFA: Attempt 1

- type dfa_attempt = state list * symbol list * state * transition list * state list

Example

- let d : dfa_attempt =
  ([0;1], (* State list *)
  ['0';'1'], (* Alphabet *)
  0, (* Start state *)
  [(0,'0',0); (* transition 1 *)
   (0,'1',1); (* transition 2 *)
   (1,'0',0); (* transition 3 *)
   (1,'1',1)], (* transition 4 *)
  [1]) (* Accepting states *)

Evaluation

- The solution is all fine and well, but to access the set of states, we have to break apart the dfa.
- It will help to write some accessor functions

- let states (s:dfa_attempt) = match s with
  | (s,_,_,_,_) -> s
  - wildcards because we don't care about the other components
- let transitions ((_,_,_,t,_):dfa_attempt) = t

Second Attempt

```
type dfa =
 {
   states : state list;
   sigma : symbol list;
   start : state;
   transitions : transition list;
   accepting : state list;
}
```

Example

```
let d : dfa =
 { states = [0;1];
   sigma = ['0';'1'];
   start = 0;
   transitions =
     [(0,'0',0);
      (0,'1',1);
      (1,'0',0);
      (1,'1',1)];
   accepting = [1]
}
```

Auxiliary Functions

(* To dereference a record, use the dot notation *)

```
let states (dfa : dfa) = dfa.states
```

(* This is a function that takes in a DFA as input, and adds a transition.
*)

```
let addTransition t dfa = { dfa with transitions = t::dfa.transitions }
```

Helper Function

explode takes a string `s`, and turns it into its individual characters. This way we can run the DFA on the string "101" without explicitly writing ['1';'0';'1']

```
let explode s =
let rec expl i l =
   if i < 0 then l else
     expl (i - 1) (s.[i] :: l) in  (* s.[i] returns the ith element of s as a char *)
expl (String.length s - 1) [];; (* String.length s returns the length of s      *)
```

Helper Function

another helper function that checks whether a list contains an element

```
let rec contains e l =
  match l with
  | [] -> false
  | hd::tl -> if hd = e then true else contains e tl
```

Checking DFA Acceptance

- Attempt 1: we might keep a (mutable) variable that keeps track of what state the DFA is currently at, and then updates the state depending on that.
- Attempt 2: write a function that tells what state to go to *next* on an input

checkAccept (part 1)

```
let checkAccepts str dfa =
  (* Get the list of symbols. *)
  let symbols = explode str in
  (* If I'm at state {state}, where do I go on {symbol}? *)
  let transition state symbol =
    let rec find_state l =
      match l with
      | (s1,sym,s2)::tl ->
        if (s1 = state && symbol = sym) then
        s2 else find_state tl
      | _ -> failwith "no next state" in find_state dfa.transitions
    in find_state dfa.transitions
  in
```

checkAccept (Part 2)

```
let final_state =
    let rec h symbol_list =
      match symbol_list with
      | [hd] -> (transition dfa.start hd)
      | hd::tl -> (transition (h tl) hd)
      | _ -> failwith "empty list of symbols"
    in
    h (List.rev symbols)
in
```

Conclusion

```
if (contains final_state dfa.accepting) then
   true
  else
false
```

Alternative Solution

```
let rec search_from current_state symbol_list =
    match symbol_list with
      | [] -> current_state
      | sym::tl -> search_from (transition current_state sym) tl
  in
  let end_state = search_from dfa.start symbols in
  if (contains end_state dfa.accepting)
  then
    true
  else
   false
```