```
(* Dan Grossman, Coursera PL, HW2 Provided Code *)

(* if you use this function to compare two strings (returns true if the same
   string), then you avoid several of the functions in problem 1 having
   polymorphic types that may be confusing *)
fun same_string(s1 : string, s2 : string) =
    s1 = s2

(* put your solutions for problem 1 here *)
(* 1a *)
fun all_except_option (s,xs) =
  case xs of
      [] => NONE
    | x::xs' => if same_string(s,x)
                then SOME xs'
                else case all_except_option(s,xs') of
              NONE => NONE
                        | SOME y => SOME(x::y);
(* 1b *)
fun get_substitutions1 (substitutions,str) =
  case substitutions of
      [] => []
    | x::xs => case all_except_option(str,x) of
           NONE => get_substitutions1(xs,str)
         | SOME y => y @ get_substitutions1(xs,str);

fun get_substitutions1_b (substitutions,str) =
  case substitutions of
      [] => []
    | x::xs =>
      let val foo = all_except_option(str,x)
      in
      case foo of
          NONE => get_substitutions1_b(xs,str)
        | SOME y => y @ get_substitutions1_b(xs,str)
      end;

(* 1c *)
fun get_substitutions2 (substitutions,str) =
  let fun loop (acc,substs_left) =
    case substs_left of
            [] => acc
      | x::xs => loop ((case all_except_option(str,x) of
                                  NONE => acc
                                | SOME y => acc @ y),
                           xs)
  in
      loop ([],substitutions)
  end;
```

```
(* 1d *)
fun similar_names (substitutions,name) =
  let val {first=f, middle=m, last=l} = name
      fun make_names xs =
    case xs of
        [] => []
      | x::xs' => {first=x, middle=m, last=l}::(make_names(xs'))
  in
      name::make_names(get_substitutions2(substitutions,f))
  end;

(* you may assume that Num is always used with values 2, 3, ..., 10
   though it will not really come up *)
datatype suit = Clubs | Diamonds | Hearts | Spades
datatype rank = Jack | Queen | King | Ace | Num of int
type card = suit * rank

datatype color = Red | Black
datatype move = Discard of card | Draw

exception IllegalMove

(* put your solutions for problem 2 here *)
(* 2a *)
fun card_color card =
  case card of
      (Clubs,_) => Black
    | (Spades, _) => Black
    | (_, _) => Red;

(* 2b *)
fun card_value card =
  case card of
      (_, Num n) => n
    | (_, Ace) => 11
    | (_, _) => 10;

(* 2c *)
fun remove_card (cs, c, e) =
  case cs of
      [] => raise e
    | x::cs' => if x = c then cs' else x :: remove_card(cs', c, e);

fun remove_card_b (cs, c, e) =
  let fun f cs =
    case cs of
        [] => raise e
      | x::cs' => if x = c then cs' else x :: f cs'
  in
      f cs
  end;
```

```
(* 2d *)
fun all_same_color cs =
  case cs of
      [] => true
    | [_] => true
    | head::neck::tail => card_color head = card_color neck
              andalso all_same_color(neck::tail);

fun all_same_color_b cs =
  case cs of
      head::neck::tail => card_color head = card_color neck
              andalso all_same_color(neck::tail)
    | _ => true;

(* 2e *)
fun sum_cards cs =
  let fun loop (acc,cs) =
    case cs of
        [] => acc
      | c::cs' => loop (acc+card_value c, cs')
  in
      loop (0,cs)
  end;

(* 2f *)
fun score (cs,goal) =
  let val sum = sum_cards cs
  in
      (if sum >= goal
       then 3 * (sum - goal)
       else goal - sum)
      div (if all_same_color cs then 2 else 1)
  end;

(* 2g *)
fun officiate (cards,plays,goal) =
  let fun loop (current_cards, cards_left, plays_left) =
        case plays_left of
        [] => score(current_cards, goal)
      | (Discard c)::tail =>
        loop (remove_card(current_cards,c,IllegalMove),cards_left,tail)
      | Draw::tail =>
        (* note: must score immediately if go over goal! *)
            case cards_left of
        [] => score(current_cards,goal)
          | c::rest => if sum_cards (c::current_cards) > goal
              then score(c::current_cards,goal)
              else loop (c::current_cards,rest,tail)
  in
      loop ([],cards,plays)
```

```
end;
```