```
(* hw3solutions.sml sample solutions *)
(* Coursera Programming Languages, Homework 3, Provided Code *)

exception NoAnswer

datatype pattern = Wildcard
         | Variable of string
         | UnitP
         | ConstP of int
         | TupleP of pattern list
         | ConstructorP of string * pattern

datatype valu = Const of int
          | Unit
          | Tuple of valu list
          | Constructor of string * valu

fun g f1 f2 p =
    let
    val r = g f1 f2
    in
    case p of
        Wildcard          => f1 ()
      | Variable x        => f2 x
      | TupleP ps         => List.foldl (fn (p,i) => (r p) + i) 0 ps
      | ConstructorP(_,p) => r p
      | _                 => 0
    end

(**** for the challenge problem only ****)

datatype typ = Anything
         | UnitT
         | IntT
         | TupleT of typ list
         | Datatype of string

(**** you can put all your code here ****);
(**** 1 : assuming all strings had at least one character ****)
val only_capitals = List.filter (fn s => Char.isUpper (String.sub(s,0)));

fun only_capitalsb xs = List.filter (fn s => Char.isUpper (String.sub(s,0))) xs;

fun only_capitalsc xs =
  case xs of
      [] => []
    | _ => List.filter (fn s => Char.isUpper (String.sub(s,0))) xs;

(**** 2 ****)
val longest_string1 =
```

```
    List.foldl (fn (s, init) =>
           if String.size s > String.size init
           then s
           else init) "";

fun longest_string1b xs = List.foldl (fn (s, init) =>
                     if String.size s > String.size init
                     then s
                     else init
                     ) "" xs;
(**** 3 ****)
val longest_string2 =
    List.foldl (fn (s, acc) =>
           if String.size s >= String.size acc
           then s
           else acc) "";
(**** 4 ****)
fun longest_string_helper f =
  List.foldl (fn (s, sofar) =>
         if f(String.size s, String.size sofar)
         then s
         else sofar) "";

val longest_string3 = longest_string_helper (fn (x, y) => x > y);

val longest_string4 = longest_string_helper (fn (x, y) => x >= y);

val longest_string3b = longest_string_helper op> ;

val longest_string4b = longest_string_helper op>= ;

(**** 5 ****)
val longest_capitalized = longest_string1 o only_capitals;

fun longest_capitalizedb xs = (longest_string1 o only_capitals) xs;

(**** 6 ****)
val rev_string = String.implode o rev o String.explode;

fun rev_string_b s = (String.implode o rev o String.explode) s;

(**** 7 ****)
(*
 * Note the pattern match of NONE and SOME
 *
 *)
fun first_answer f xs =
  case xs of
      [] => raise NoAnswer
    | x::xs' => case f x of
           NONE => first_answer f xs'
```

```
          | SOME y => y;

(**** 8 ****)
(*
 * Note the pattern match of NONE and SOME
 *
 *)
fun all_answers f xs =
  let fun loop (xs, acc) =
    case xs of
        [] => SOME acc
      | x::xs' => case f x of
                NONE => NONE
                      | SOME y => loop(xs', (y @ acc))
  in
      loop (xs, [])
  end;

(**** 9 ****)
val count_wildcards = g (fn () => 1) (fn _ => 0);

val count_wild_and_variable_lengths = g (fn () => 1) String.size;

fun count_some_var (x,p) =
  g (fn () => 0) (fn s => if s = x then 1 else 0) p;

(**** 10 ****)
fun check_pat pat =
  let fun get_vars pat =
    case pat of
            Variable s => [s]
      | TupleP ps => List.foldl (fn (p,vs) => get_vars p @ vs) [] ps
      | ConstructorP(_,p) => get_vars p
      | _ => []
      fun unique xs =
    case xs of
        [] => true
      | x::xs' => (not (List.exists (fn y => y=x) xs'))
              andalso unique xs'
  in
      unique (get_vars pat)
  end;

(*
Instead of putting all variables in a list and then looking for uniqueness,
we can have an accumulator of variables-seen-so-far and
then on each variable compare it against all those in the accumulator.

A more efficient way to check for uniqueness is to sort the list of variables
and then make sure no two adjacent variables are the same.
```

*A solution could use* `count_some_var` *as a helper function to check that each variable found in the pattern has a count of 1.*

*The treatment of* `TupleP` *is a particular place to focus. While the sample's approach of using* `List.foldl` *and recursion is particularly concise, it is not necessary to use fold here to get a 5 -- an explicit recursive helper function is okay.*
*)

```
(**** 11 ****)
fun match (v, p) =
  case (v, p) of
      (_, Wildcard) => SOME []
    | (_, Variable(s)) => SOME [(s, v)]
    | (Unit, UnitP) => SOME []
    | (Const i, ConstP j) => if i=j then SOME [] else NONE
    | (Tuple(vs), TupleP(ps)) =>
      let val lv = length vs
      val lp = length ps
      val vplist = ListPair.zip(vs, ps)
      val ans = all_answers match vplist
      in
      if lv = lp
      then ans
      else NONE
      end
    | (Constructor(s1,v2), ConstructorP(s2,p2)) =>
      if s1=s2 then match(v2,p2) else NONE
    | _ => NONE;

(**** 12 ****)
fun first_match valu patlst =
  SOME (first_answer (fn pat => match (valu,pat)) patlst)
  handle NoAnswer => NONE;
```