Instructions. Your task is to answer four written problems, and to write eleven SML functions (excluding local helper functions) as well as test cases for those functions. See the end of this document for instructions on how to submit your solution. In case you’re curious, our reference solution to the (non-karma) programming problems contains about 90 lines of code, excluding comments, blank lines, and the provided code described next.

Provided code. Download hw2template.sml from the course website. The provided code defines several types for you; you may not change those types. You will not need to write any additional datatype bindings or type synonyms to solve the homework.

Prohibitions. You may not use functions null, hd, tl, isSome, or valOf. You may not use record or tuple selectors (e.g., #1, #fieldname). You may not use mutable references or arrays. You may (in fact, must) use pattern matching.

Part I: Written Problems

Problem 1. Give the type of each of the following ML expressions. We have deliberately omitted the types of the function arguments, but you should be able to infer them.

A. (op +, true orelse false, [#"#"])  
B. fun f(x::xs) = if (x = 0) then xs else []  
C. fun g(a, b, c) = case a of  
   [ ] => (round c) = 1  
   | 1::xs => not (null xs)  
   | _ => b

Problem 2. The following function executes correctly, but it was written with poor style. On HW1, you rewrote this function to have better style, but you weren’t allowed to use pattern matching there. Now your task is to rewrite it with better style using pattern matching. You must remove all uses of hd and tl. Please consult the style guide on the course website.

fun zardoz (a:int list, b:int list) =  
  if length(a) = 0 andalso length(b) = 0 then b  
  else if (length(b) = 0) then hd(a)::tl(a)  
  else if (length(a) = 0) then [] @ b  
  else if (hd(a) < hd(b)) = true then [hd(a)]  
  @ (zardoz((tl(a)), b)) else [hd(b)] @ (zardoz(a, (tl(b))))

Problem 3. For each of the following cope snippets, rename each variable, including function parameters, by appending a distinct number to it. (We suggest you start with 1 and work your way up the natural numbers.) Then at the point where each identifier goes out of scope—i.e., is no longer visible—add a comment of the form (*end scope: variable_name*). If more than one variable goes out of scope at the same point, you can put them all in the same comment, separated by commas. For example, change

let
   val x = nil
in
   x
end
let
  val x1 = nil
in
  x1
end (*end scope: x1*)

A. fun area(r) = 
  let
    val r = r * r
  in
    Math.pi * r
  end

B. let
  val x = 3
  fun f(z) = x
in
  let
    fun g(z) = 
      let
        val x = 4
      in
        f(z)
      end
    in
      (g(), f())
  end
end

Problem 4. Complete the following comments by filling in the ellipses. At each place you fill in, give the complete, cumulative static environment created by the program. (But omit the pervasive static environment that exists at the beginning of every SML program and includes, e.g., hd, tl, etc.)

fun f (x, xs) = 
  (* Current static environment is ... *)
  case xs of
    [] => (* Current static environment is ... *) x
  | h::t => (* Current static environment is ... *) h+f(0,t)

datatype d = Foo | Bar of (int * int) list
(* Current static environment is ... *)
Part II: Name Substitutions

This part involves using first-name substitutions to create alternative names. For example, Fredrick William Flintstone could also be Fred William Flintstone or Freddie William Flintstone.

1. Write a function all_except_option, which takes a string and a string list. Return NONE if the string is not in the list, else return SOME lst where lst is like the argument list except the string is not in it. Assume the string is in the list at most once. Use same_string, provided to you, to compare strings. (There’s no reason you couldn’t use SML’s built-in = operator to compare strings for equality. But using the function we provide will help to avoid some types that might be confusing at this point in the course.) Hint: our reference solution is eight lines. Yours doesn’t have to be exactly that length, but elegant and stylish solutions won’t be much longer.

   ```ml
   all_except_option("Fred", ["Fredrick","Fred"])
   (*) result: SOME ["Fredrick"]
   ```

2. Write a function get_substitutions1, which takes a string list list (a list of list of strings, the substitutions) and a string s and returns a string list. The result has all the strings that are in some list in substitutions that also has s, but s itself should not be in the result. Assume each list in substitutions contains no duplicates. Example:

   ```ml
   get_substitutions1(["Fred","Fredrick"],["Elizabeth","Betty"],
   ["Freddie","Fred","F"],["Fred"])
   (*) result: ["Fredrick","Freddie","F"]
   ```

The result might have duplicates if s appears in more than one list in substitutions. Example:

   ```ml
   get_substitutions1(["Fred","Fredrick"],["Jeff","Jeffrey"],
   ["Geoff","Jeff","Jeffrey"],["Jeff"])
   (*) result: ["Jeffrey","Geoff","Jeffrey"]
   ```

Use problem (1) and ML’s list-append operator @ but no other helper functions. Hint: our reference solution is about five lines.

3. Write a function get_substitutions2, which is like get_substitutions1 except it uses a tail-recursive local helper function.

4. Write a function similar_names, which takes a string list list of substitutions, as in problems (2) and (3), and a full name of type name. Type name is defined in the code we provided for you as follows:

   ```ml
   type name = {first:string, middle:string, last:string}
   ```

The function you write should return a list of full names. The result list should have type name list, and it should contain all the full names that can be produced by substituting for the first name—and only the first name—using argument list substitutions. The answer should always begin with the original name, then have zero or more other names. Example:

   ```ml
   similar_names(["Fred","Fredrick"],["Elizabeth","Betty"],
   ["Freddie","Fred","F"],
   {first="Fred", middle="W", last="Flintstone"})
   (*) result: [{first="Fred", last="Flintstone", middle="W"},
   {first="Freddie", last="Flintstone", middle="W"},
   {first="Fred", last="Flintstone", middle="W"},
   {first="F", last="Flintstone", middle="W"}] *)
   ```

Hint: Use problem (2) or (3). Use a local helper function. Our reference solution is about ten lines.
Part III: Solitaire

This part involves a made-up solitaire card game. Your task is to write a program that tracks the progress of a game. You don’t need to write an AI that plays the game—doing so is left as a karma problem. You don’t even need to understand the rules of the game until you get to problem (6).

A game is played with a deck, which is a list of cards, and a goal score. The player has a hand, which is a list of cards and is initially empty. The player makes a move either by drawing, which means removing the first card from the deck and adding it to the hand, or by discarding, which means choosing one of the cards in the hand to remove. The game ends either when the player chooses to make no more moves, or when the sum of the values of the hand is greater than the goal. Card values are defined below in problem (2).

The player’s objective is to end the game with a low score. A score of zero is the best possible score. Scoring works as follows: Let sum be the sum of the values of the cards in the hand. If sum is greater than goal, the preliminary score is 3 multiplied by sum − goal; otherwise, the preliminary score is goal − sum. The game score is the preliminary score unless all the cards in the hand are the same color, in which case the game score is the preliminary score divided by 2 (and rounded down as usual with integer division; use ML’s div operator).

Your code should use the following types, which are already defined in the provided code:

```ml
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
```

1. Write a function `card_color`, which takes a card and returns its color. Spades and clubs are black; diamonds and hearts are red. Hint: a single case expression suffices.

2. Write a function `card_value`, which takes a card and returns its value. Numbered cards have their number as the value, aces are 11, everything else is 10. Hint: a single case expression suffices.

3. Write a function `remove_card`, which takes a list of cards l, a card c, and an exception e. Return a list that is the same as l except that the first occurrence of c is removed. If c is in the list more than once, remove only the first occurrence. If c is not in the list, raise exception e. Hint: compare cards with =.

4. Write a function `all_same_color`, which takes a list of cards and returns true if and only if all the cards in the list are the same color. Hint: there is a very elegant solution that is only about five lines long. It involves deep pattern matching.

5. Write a function `sum_cards`, which takes a list of cards and returns the sum of their values. You must use a local helper function that is tail recursive.

6. Write a function `score`, which takes a card list (the hand) and an int (the goal) and computes the score as described above.

7. Write a function `officiate`, which “runs a game.” It takes a card list (the deck) a move list (what the player does at each step), and an int (the goal) and returns the score at the end of the game after processing the moves in order. Use a local, recursive helper function that takes several arguments that together represent the current state of the game. A game proceeds as follows:

   • The game starts with the hands being empty.
   • The game ends if there are no more moves (i.e., the player chooses to quit, because the move list is empty).
   • If the player discards some card c, play continues (i.e., make a recursive call) with the hand not having c and the deck unchanged. If c is not in the hand, raise the IllegalMove exception.
• If the player draws and the deck is empty, the game is over. If drawing causes the sum of the
  hand to exceed the goal, the game is over. Otherwise, play continues by removing the drawn card
  from the deck and placing it in the hand.

Hint: our reference solution to problem (7) has fewer than 20 lines.

Part IV: Karma Problems

1. Write functions score_karma and officiate_karma. They should behave like their non-karma coun-
terparts, except that each ace can now have a value of either 1 or 11. Moreover, score_karma must
always return the best (i.e., least) possible score. Hint: this is easier to code than you might think.

2. Write function careful_player, which takes a deck and a goal and returns a move list, such that
calling officiate_karma with the deck, the goal, and the move list has this behavior:

• The value of the hand never exceeds the goal.
• A card is drawn whenever doing so cannot cause the hand to exceed the goal.
• If a score of 0 is reached, the player quits the game by making no further moves.

Further Instructions

Important Notes about Grading. All the “Important Notes about Grading” from HW1 apply here,
and will apply in all future homeworks, too. (So this is the last time we’ll remind you about them.) Programs
that do not compile will receive an automatic zero with no chance for a regrade; functions with improper
names or arguments, too. Style always matters.

Required bindings. Evaluating a correct homework solution must generate at least the following bindings,
in addition to the bindings from the code provided to you—but see the important caveat that follows.

val all_except_option = fn : string * string list -> string list option
val get_substitutions1 = fn : string list list * string -> string list
val get_substitutions2 = fn : string list list * string -> string list
val similar_names = fn : string list list * name -> name list
val card_color = fn : card -> color
val card_value = fn : card -> int
val remove_card = fn : card list * card * exn -> card list
val all_same_color = fn : card list -> bool
val sum_cards = fn : card list -> int
val score = fn : card list * int -> int
val officiate = fn : card list * int list * int -> int

Important Caveat: If you do not specify the types of function arguments and return values in your code,
the REPL might give your functions more general types or equivalent types. For example, depending on how
you write the code, card_value might have type suit * 'a -> color and remove_card might have type
'Hcard * 'a * exn -> 'Hcard. These are more general types, which means there is a way to replace
the type variables ('a and 'H'a) with types to get the bindings listed above. As for equivalent types, because
type card = suit*rank, types like card->int and suit*rank->int are equivalent. Also, recall that the
order of fields in records never matters.

Our grading scripts will work just fine if your functions have more general types or equivalent types to
the bindings listed above. However, your safest course of action is to write down explicit argument types
and return types for functions in your code.

Of course, generating those bindings does not guarantee that your solutions are correct...
Testing. You are required to test your functions. Put your testing code in a separate file. We will not directly grade it, but you must turn it in. Good test cases might help you get some partial credit if your solution is erroneous.

Submission Instructions

Submissions that do not adhere to these criteria will lose points:

- Put all your written solutions to part I in one file, `netid_hw2written.txt`, where `netid` is your GW NetId. This file must be plain text. We recommend using Emacs to create it.
- Put all your solution code to parts II and III (and IV, if you do it) in one file, `netid_hw2.sml`.
- Put all the tests you wrote for part II and III (and IV, if you do it) in another file, `netid_hw2_test.sml`.
- The first line of all three files should be an ML comment with your name, GW NetId, and the phrase Homework 2.
- Upload all three files to the Homework 2 assignment on BlackBoard.

Evaluation Criteria

Solutions will be evaluated on correctness with respect to the specifications in this assignment; style, including indentation and line breaks, with respect to the style guide on the course website; elegance, which is an ineffable quality that includes beauty, effectiveness, and simplicity; and adherence to using only those SML features permitted in this homework.