Programming Languages (Coursera / University of Washington) Assignment 2

You will write 11 SML functions (not counting local helper functions), 4 related to "name substitutions" and 7 related to a made-up solitaire card game.

Your solutions must use pattern-matching. You may not use the functions `null`, `hd`, `tl`, `isSome`, or `valOf`, nor may you use anything containing a `#` character or features not used in class (such as mutation). Note that list order does not matter unless specifically stated in the problem.

Download hw2provided.sml from the course website. The provided code defines several types for you. You will not need to add any additional `datatype` bindings or type synonyms.

The sample solution, not including challenge problems, is *roughly* 130 lines, including the provided code.

Do not miss the "Important Caveat" and "Assessment" after the "Type Summary."

1. This problem involves using first-name substitutions to come up with alternate names. For example, *Fredrick William Smith* could also be *Fred William Smith* or *Freddie William Smith*. Only part (d) is specifically about this, but the other problems are helpful.

(a) 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 identical to the argument list except the string is not in it. You may assume the string is in the list at most once. Use `same_string`, provided to you, to compare strings. Sample solution is around 8 lines.

(b) 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. Example:

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

Assume each list in *substitutions* has no repeats. The result will have repeats if *s* and another string are both in more than one list in *substitutions*. Example:

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

Use part (a) and ML's list-append (`@`) but no other helper functions. Sample solution is around 6 lines.

(c) Write a function `get_substitutions2`, which is like `get_substitutions1` except it uses a tail-recursive local helper function.

(d) Write a function `similar_names`, which takes a `string list list` of substitutions (as in parts (b) and (c)) and a *full name* of type `{first:string,middle:string,last:string}` and returns a list of full names (type `{first:string,middle:string,last:string} list`). The result is all the full names you can produce by substituting for the first name (and *only the first name*) using *substitutions* and parts (b) or (c). The answer should begin with the original name (then have 0 or more other names). Example:

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

Do not eliminate duplicates from the answer. Hint: Use a local helper function. Sample solution is around 10 lines.

2. This problem involves a solitaire card game invented just for this question. You will write a program that tracks the progress of a game; writing a game player is a challenge problem. You can do parts (a)–(e) before understanding the game if you wish.

A game is played with a *card-list* and a *goal*. The player has a list of *held-cards*, initially empty. The player makes a move by either *drawing*, which means removing the first card in the card-list from the card-list and adding it to the held-cards, or *discarding*, which means choosing one of the held-cards to remove. The game ends either when the player chooses to make no more moves or when the sum of the values of the held-cards is greater than the goal.

The objective is to end the game with a low score (0 is best). Scoring works as follows: Let *sum* be the sum of the values of the held-cards. If *sum* is greater than *goal*, the *preliminary score* is three times $(sum - goal)$, else the preliminary score is $(goal - sum)$. The score is the preliminary score unless all the held-cards are the same color, in which case the score is the preliminary score divided by 2 (and rounded down as usual with integer division; use ML's `div` operator).

(a) Write a function `card_color`, which takes a card and returns its color (spades and clubs are black, diamonds and hearts are red). Note: One case-expression is enough.

(b) 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). Note: One case-expression is enough.

(c) Write a function `remove_card`, which takes a list of cards `cs`, a card `c`, and an exception `e`. It returns a list that has all the elements of `cs` except `c`. If `c` is in the list more than once, remove only the first one. If `c` is not in the list, raise the exception `e`. You can compare cards with `=`.

(d) Write a function `all_same_color`, which takes a list of cards and returns true if all the cards in the list are the same color. Hint: An elegant solution is very similar to one of the functions using nested pattern-matching in the lectures.

(e) Write a function `sum_cards`, which takes a list of cards and returns the sum of their values. *Use a locally defined helper function that is tail recursive. (Take "calls use a constant amount of stack space" as a requirement for this problem.)*

(f) Write a function `score`, which takes a `card list` (the held-cards) and an `int` (the goal) and computes the score as described above.

(g) Write a function `officiate`, which "runs a game." It takes a `card list` (the card-list) a `move list` (what the player "does" at each point), and an `int` (the goal) and returns the score at the end of the game after processing (some or all of) the moves in the move list in order. Use a locally defined recursive helper function that takes several arguments that together represent the current state of the game. As described above:

- The game starts with the held-cards being the empty list.
- The game ends if there are no more moves. (The player chose to stop since the `move list` is empty.)
- If the player discards some card `c`, play continues (i.e., make a recursive call) with the held-cards not having `c` and the card-list unchanged. If `c` is not in the held-cards, raise the `IllegalMove` exception.
- If the player draws and the card-list is (already) empty, the game is over. Else if drawing causes the sum of the held-cards to exceed the goal, the game is over (after drawing). Else play continues with a larger held-cards and a smaller card-list.

Sample solution for (g) is under 20 lines.

3. **Challenge Problems:**

(a) Write `score_challenge` and `officiate_challenge` to be like their non-challenge counterparts except each ace can have a value of 1 or 11 and `score_challenge` should always return the least (i.e., best) possible score. (Note the game-ends-if-sum-exceeds-goal rule should apply only if there is no sum that is less than or equal to the goal.) Hint: This is easier than you might think.

(b) Write `careful_player`, which takes a card-list and a goal and returns a move-list such that calling `officiate` with the card-list, the goal, and the move-list has this behavior:

- The value of the held cards never exceeds the goal.
- A card is drawn whenever the goal is more than 10 greater than the value of the held cards. As a detail, you should (attempt to) draw, even if no cards remain in the card-list.
- If a score of 0 is reached, there must be no more moves.
- If it is possible to reach a score of 0 by discarding a card followed by drawing a card, then this must be done. Note `careful_player` will have to look ahead to the next card, which in many card games is considered "cheating." Also note that the previous requirement takes precedence: There must be no more moves after a score of 0 is reached even if there is another way to get back to 0.

Notes:

- There may be more than one result that meets the requirements above. The autograder should work for any correct strategy — it checks that the result meets the requirements.
- This problem is *not* a continuation of problem 3(a). In this problem, all aces have a value of 11.

Type Summary

Evaluating a correct homework solution should generate these 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 * {first:string, last:string, middle:string}
    -> {first:string, last:string, middle:string} 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 * move list * int -> int
```

Important Caveat

The REPL may give your functions *equivalent types* or *more general* types. This is fine. In the sample solution, the bindings for problems 1d, 2a, 2b, 2c, and 2d were all more general. For example, `card_color` had type `suit * 'a -> color` and `remove_card` had type `''a list * ''a * exn -> ''a list`. They are more general, which means there is a way to replace the *type variables* (`'a` or `''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. They are the same type, and the REPL simply chooses one way of printing the type. Also, the order of fields in records never matters.

If you write down explicit argument types for functions, you will probably not see equivalent or more-general types, but we encourage the common ML approach of omitting all explicit types.

Of course, generating these bindings does not guarantee that your solutions are correct. *Test your functions: Put your testing code in a second file. We will not grade the testing file, nor will you turn it in, but surely you want to run your functions and record your test inputs in a file.*

Assessment

We will automatically test your functions on a variety of inputs, including edge cases. We will also ask peers to evaluate your code for simplicity, conciseness, elegance, and good formatting including indentation and line breaks. Your solution will also be checked for using only features discussed so far in class, and to ensure that it does not use any of the banned features listed at the beginning of the assignment, such as `hd` and `#foo`.

Turn-in Instructions

First, follow the instructions on the course website to submit your solution file (not your testing file) for auto-grading. Do not proceed to the peer-assessment submission until you receive a high-enough grade from the auto-grader: Doing peer assessment requires instructions that include a sample solution, so these instructions will be "locked" until you receive high-enough auto-grader score. Then submit your same solution file again for peer assessment and follow the peer-assessment instructions.