CSC 330 —Programming Languages
Spring 2015
Assignment No. 2

Note 1 **This assignment is to be done individually**

Note 2 You can discuss the assignment with others, but copying code is prohibited.

- Due date: Jan 23, 2015, at the beginning of the class.
- This assignment is worth 1% of your total course mark.
- Submit electronically via Connex a single zip file with your solutions (two files)
- At the beginning of the class submit a paper copy of your program.

Objectives

After completing this assignment, you will have experience:

- Pattern matching.
- Tail recursion.

Your task, should you choose to accept it

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 **should 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.

From connex download `hw2.sml`. 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.

Section 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 4 is specifically about this, but the other problems are helpful.

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 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.

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. 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 1 and ML's list-append (@) **but no other helper functions**. Sample solution is around 6 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 Parts 2 and 3) 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 2 or 3. 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.

Section 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 5 to 9 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 two 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).

5. 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.

6. 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.

7. 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 =.

8. 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. The color of empty list is true. Hint: An elegant solution is very similar to one of the functions using nested pattern-matching in the lectures.

9. 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**.

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

11. 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 but 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 is under 20 lines.

Section 3: Your Tests

I have provided a file with test bindings `hw2-test.sml`. Make sure that everyone of the test bindings evaluates to true. Add at least one new test for each function. Your test should be named `testN_0` where N is the Part number. For example, your test for the first function `all_except_option` should be called `test1_0`, and your test for `sum_cards` should be called `test9_0`

Hints

1. Implement one function at a time.
2. Remember that the REPL will keep all the bindings you define, and keep adding new ones every time you load the current file. I recommend you restart the REPL every time you change the datatype of a binding or its name.
3. These are the bindings your program should generate:

```
val same_string = fn : string * string -> bool
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
datatype suit = Clubs | Diamonds | Hearts | Spades
datatype rank = Ace | Jack | King | Num of int | Queen
type card = suit * rank
datatype color = Black | Red
datatype move = Discard of suit * rank | Draw
exception IllegalMove
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
```

Evaluation

Solutions should be:

1. Correct. We might use more tests than the ones provided. It will be run using SML/NJ.
2. In good style, including indentation and line breaks