Principles of Programming Languages, Midterm Exam (4/23, 18:00 ~ 19:30)
Instructor: Jiwon Seo

Name: _______________ Student ID: __________________

Instruction: read questions carefully and write your answers. Please explain your answers when necessary, and do not just write down answers only. Good Luck!

Problems	Score
1. Lexical Scope	**/15**
2. Module	**/10**
3. Higher Order Functions	**/10**
4. Abstract Data Type	**/15**
5. ML Programming	**/20**
Total	**/70**

Problem 1. [Lexical Scope] For each of the following programs, give the value that *ans* is bound to after evaluation.

(a) (5 points)

```
val x = 1
fun f y =
    let val x=x
    in
        if y > 0
        then fn z => x + z
        else fn z => x - z
end

val x = 2
val g = f 3
val x = 7
val ans = g 2
```

3

(b) (5 points)

```
fun fold1 (f: 'a*'b->'b) (acc: 'b) (l: 'a list): 'b =
  case l of
    [] => acc
  | x::xs => fold1 f (f(x,acc)) xs

val ans = fold1 (fn (x,y) => x-y) 0 [1, 4, 9, 16, 25, 36]
```

21

(c) (5 points)

```
fun fold2 (f: 'a*'b->'b) (acc: 'b) (xs: 'a list): 'b =
   case xs of
     [] => acc
   | x::[] => f(x, acc)
   | x::xs' => f(x, (fold2 f acc xs'))

val ans = fold2 (fn (x,y) => x-y) 0 [1, 4, 9, 16, 25, 36]
```

-21

Problem 2. [Module] In this problem, suppose we have an ML structure M and signature S in this standard usage:

```
signature S =
sig
...
end

structure M :> S =
struct
...
end
```

Assume everything type-checks initially, meaning M matches S. For each of the following statements, answer "always," "sometimes," or "never" [10 pt, 2 pt each]

(a) If S originally contains **val mylist: int list** and we replace it with **val mylist: 'a list**, then M will still match S.

never

…

(b) If S originally contains an abstract type **type t** and we replace this with **datatype t=Int of int**, then M will still match S

sometimes

(c) If S originally contains an abstract type **type t** and we replace this with **datatype t=Int of int**, then the client of M will still type-check. (assume that M matches S)

always

(d) If S originally does not have **exception MyException** and we add it to S, then M will still match S.

sometimes

(e) If S originally contains **exception MyException** and we remove it from S, then the client of M will still type-check. (assume that M matches S)

sometimes

Problem 3. [Higher Order Functions] Write a function `map_composed` **that takes two functions f and g and a list and returns a list of values produced by applying f and g to each element; i.e., for each element in the list, we apply g first and then apply f to the result of applying g. For example,** `map_composed f g [1,2,3]` **returns** `[f(g(1)), f(g(2)), f(g(3))]`.

Note that

- **Function map_composed takes its argument in <u>a curried form</u>.**
- **You should not use any ML built-in functions. For example, <u>do not use List.map</u>.**

(a) Implement map_composed in the following. What is the type of map_composed ? [5 pt]

```
fun map_composed f g mylist =
    case my list of
       [] => []
      |x::[] => [f(g(x))]
      |x::xs => f(g(x)):: (map_composed f g xs)


혹은
    case my list of
       [] => []
      |x::xs => f(g(x)):: (map_composed f g xs)




(* write down the type of map_composed here *)

map_composed: ('a -> 'b) -> ('c -> 'a) -> 'c list -> 'b list


`a, `b, `c 는 이름이 달라도 되지만 같은 위치에 같은 이름이 와야 함.
```

(b) Implement `map_square` **function using** `map_composed` **function. <u>Do not</u> implement any helper function; implement it using anonymous function(s). The function takes a function and a list; it first squares the element and applies the function to return the result list. For example,** `map_square g [1,2,3]` **returns** `[g(1),g(4),g(9)]` **[5 pt]**

```
val map_square = fn g => ((map_composed g (fn x => x*x)));
```

Problem 4. [Abstract Data Type] Recall the set abstract data type (ADT) described in the class. It is implemented with record type with functions as its fields. Now you want to add a member function "map", which is similar to the map function we learned in the class. The function takes a function as its argument and applies the (argument) function to the members in the set; then it returns a set of the return values. Note that the map function should return a set. [15 pt]

(a) Fill in the blanks in the following code. You can use List.map function [10pt].

```
datatype set = S of { insert : int -> set,
                      member : int -> bool,
                      size   : unit -> int,


                      map : __(int->int)->set________}

val empty_set =
let
    fun make_set xs =
    let fun contains i = List.exists (fn j => i=j) xs
    in
      S { insert = fn i => if contains i
                           then make_set xs
                           else make_set (i::xs),
          member = contains,
          size   = fn () => length xs,


          map = fn (f:int->int) => make_set (List.map f xs)



                ____________________________________________
       }
    end
in
  make_set []
end
```

(b) Fill in the following client code, such that s5 is bound to a set derived from s4, but its members are incremented by 1; that is, if s4 has {3, 4}, s5 must have {4,5}. [5pt]

```
(* example client *)
val S s1 = empty_set;
val S s2 = (#insert s1) 34;
val S s3 = (#insert s2) 34;
val S s4 = #insert s3 19;



val S s5 = (#map s4) (fn (x) => x+1)
```

What numbers are in set s5 in the above code?

{ 20, 35 }

Problem 5. [ML Programming] Assume that you want to buy a toy. You have a number of coins in your pocket. You want to find out if you can exactly pay the price of the toy with your coins. [20 pt]

(a) First you want to find out if you can pay the price with the coins you have. Implement the function `canPay: (int list * int) → bool`**. The two arguments are the coins (int list) and the price (int); the function returns true if you can exactly pay the price, and false otherwise. For example,** `canPay([10, 15, 15], 11)` **returns false, because any subset of the coins cannot add to 11. For another example,** `canPay([10, 15, 15], 25)` **returns true, because 10+15 is 25. You can assume that all your coins have positive values. [10pt]**

```
fun canPay(coins: int list, price: int) =
  case coins of

       [] => __price = 0_______________

     | c::coins' => ___price=c________________________

                    orelse __canPay(coins', price-c)_

                    orelse __canPay(coins', price)__
```

Example function calls:

the following should return true

```
canPay([5, 15, 10], 25)
canPay([5, 5, 10], 20)
```

the followings should return false

```
canPay([5, 15, 10], 50)
canPay([20, 10, 10], 15)
```

(b) Now you want to find the number of possible ways to pay the price – that is, the number of combinations of all possible payment that matches the price. Implement the function `countPossiblePay: (int list * int) → int`**. The two arguments are the coins (int list) and the price (int) – the same as** `canPay`**. The function returns the number of possible way to pay the price with the coins. For example,** `countPossiblePay([10a, 10b, 50a, 50b], 60)` **returns 4, because there are four possible ways to pay 60 with the coins: (10_a+50_a), (10_b+50_a), (10_a+50_b), (10_b+50_b). The subscripts a, b are for the description only. Again, assume that all the coins have positive values. [10pt]**

```
fun countPossiblePay(coins: int list, price: int) =
  case coins of
    [] => if price = 0

           then __1_________________________

           else __0_________________________
  | c::coins' => if price=c

                  then _countPossiblePay(coins', price)+1__

                       _____________________________________

                  else __countPossiblePay(coins', price-c)+_

                       __countPossiblePay(coins', price)__
```

Example function calls:

```
countPossiblePay([5, 15, 25, 5, 10], 20)   (* this should return 3 *)
countPossiblePay([10, 15, 25, 20, 5, 5], 25)(*this should return 5 *)
```