Stone Harris
CS320 - Theory HW2

Krumbl Kookies
This sensational cookie brand is working on making new flavors for its next week special! A client has proposed the following cookie flavors:

```
type cookie_details = float * float

type cookie_flavor =
        | CaramelPumpkin of cookie_details
        | LaddooLemon of cookie_details
        | Nevadito of cookie_details
```

Where cookie_details contains the cookie's diameter (cm) and price ($) respectively.

Question 1.

Your objective is to exhaustively match all patterns of the following expressions using only the match keyword. You should match until there are only base types (int, float, bool, string). A wildcard (_) should be used to match the right-hand side of the cons (_::_) pattern.

Match the following variables

1.1

n: cookie_flavor

```
        match n with
        | CaramelPumpkin(a, b) -> 1
        | LadooLemon(a, b) -> 2
        | Nevadito(a, b) -> 3;;
```

1.2

w: int list option list

```
        match w with
        | [] -> 1
        | a :: _ -> match a with
                | None -> 2
                | Some([]) -> 3
                | Some(a :: _) -> 4;;
```

1.3

```
x: (int * (bool list * string)) option
        match x with
        | None -> 1
        | Some(ii, ([], ss) ) -> 2
        | Some(ii, (bb :: _, ss)) -> 3
```

Question 2.

For this question, you may include the types cookie_flavor and cookie_details in your answer if appropriate. Consider a function with polymorphic type f : ‘a -> ‘a * ’a -> (‘a * ’a) list

2.1.

What is the type of f (CaramelPumpkin(1., 1.)) ? Justify your answer.

Type: **cookie_flavor*cookie_flavor -> (cookie_flavor*cookie_flavor) list**

- **This is because f is taking in CaramlePumpkin(1., 1.) which is of type cookie_flavor*cookie_flavor and from there f spits out a (‘a * ’a) list, so in this case it would be a (cookie_flavor*cookie_flavor) list**

2.2.

What is the type of the following statement

```
let cd: cookie_details = (0., 0.) in
        fun x -> f x cd
```

Justify your answer.

Type: **‘x -> cookie_details -> ‘y**

- **This is because we don’t know what exactly the statement is taking in or spitting out, just that it is a curried function that is operating on a cookie_details value with cd**

Question 3. For this question, you may include the types cookie_flavor and cookie_details in your answer if appropriate. Consider a function with polymorphic type f : (‘a * ‘b) -> (‘b -> ‘a) -> ‘a

3.1.

What is the type of f (Nevadito(1., 2.), true) ? Justify your answer.

Type: **(bool -> cookie_flavor) -> cookie_flavor**

- **This is because f is taking in a boolean and a cookie_flavor and returning just a cookie_flavor value**

3.2.

What is the type of fun coo kie -> f (coo, kie) ? Does it have the same type as f? Justify both answers.

Type: **(a*b) -> c**

- **This has the same type as f. This is because the two functions do the same thing and thus are equivalent to each other, sharing the same type.**

Question 4.

Consider a function with polymorphic type g : (‘a -> ‘b –> ‘c) -> ‘d

4.1.

What is the type of g (List.fold_right) ? Justify your answer. Hint: this is the type signature:
List.fold_right: (’a -> ’b -> ’b) -> ’a list -> ’b -> ’b

Type: **((a -> b -> b) -> a list -> b -> b) -> b**

- **This is because g takes in a function (which would be fold_right) so I added that into the full typing for g to represent the currying that goes on.**

4.2.

What is the type of g (fun acc (e: cookie_flavor) -> List.cons e acc)? Briefly explain your reasoning.

Type: **(cookie_flavor list) * cookie_flavor) -> cookie_flavor list**

- **This is because g is taking in a function that spits out a cookie_flavor list * cookie_flavor and then utilizes a cons to produce a final product of a cookie_flavor list**

Question 5.

Is the following function well-typed? Briefly explain your reasoning.

```
let mystery (x : cookie_flavor list) (y: cookie_details * bool): cookie_flavor list =
        match x, y with
        | [], ((diam, price), b) -> List.fold_right (List.cons) [] x
        | h::t, (_, true) -> []
        | _::_, (_, false) -> [Nevadito(y)]
```

- **The function, mystery, is not well-typed because when y is fed to Nevadito(), y is of type (cookie_details*bool) instead of just type cookie_details, which it is expecting. This type checking is done incorrectly, thus the function mystery is not well-typed.**

Question 6.

Krumbl actually finds these flavors quite lovely and wants to implement them in a factory! To do so, they need to implement the following functions:

reverse: cookie_flavor list -> cookie_flavor list

append: cookie_flavor list -> cookie_flavor list -> cookie_flavor list

concat : (cookie_flavor list list) -> cookie_flavor list

filter: (cookie_flavor -> bool) -> cookie_flavor list -> cookie_flavor list

fold_right : (cookie_flavor -> ’b -> ’b) -> cookie_flavor list -> ’b -> ’b

List.fold_left: (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a

Implement the following standard list functions. When given the same input, they should have the same output as their standard library counterparts.

6.1.

reverse : cookie_flavor list -> cookie_flavor list

```
let rec reverse (flavors: cookie_flavor list): cookie_flavor list =
  let rec do_rev (a: cookie_flavor list) (b: cookie_flavor list): cookie_flavor list =
    match a, b with
      | [], otherbs  -> otherbs
      | aa :: otheras, otherbs -> do_rev(otheras, aa::otherbs);;
  do_rev ([], flavors)
```

6.2.

append : cookie_flavor list -> cookie_flavor list -> cookie_flavor list

```
let rec append (a: cookie_flavor list) (b: cookie_flavor list): cookie_flavor list =
    let rec do_append(a: cookie_flavor list) (b: cookie_flavor list): cookie_flavor list =
            match a, b with
             | [], otherbs  -> otherbs
             | aa :: otheras, otherbs -> do_append(otheras, aa::otherbs);;
    do_append (reverse a) b
```

6.3.

```
concat : (cookie_flavor list list) -> cookie_flavor list

let rec concat (flavorlists: cookie_flavor list list): cookie_flavor list =
  match flavorlists with
 | [] -> []
 | fl :: otherlists -> append fl (concat otherlists);;
```

6.4.

```
filter : (‘a -> bool) -> ‘a list -> ‘a list

let rec filter (check: (cookie_flavor->bool)) (flavors: cookie_flavor list): cookie_flavor list =
  match cf, flvs with
   | cf, [] -> []
   | cf, flv :: otherfs -> match (cf flv) with
                            | false -> filter cf otherfs
                            | true -> flv :: (filter cf otherfs);;
```

6.5.

```
fold_right : (cookie_flavor -> ’b -> ’b) -> cookie_flavor list -> ’b -> ’b

let rec fold_right  (do_fold: cookie_flavor -> ’b -> ’b) (flavors: cookie_flavor list) (acc: ’b): ’b =
  match do_fold, flavors, acc =
   | _, [], value -> value
   | df, flv :: others, value -> fold_right df others (df flv value);;
```

Question 7.

Krumbl now wants you to make a create_cookie_boxes function that will take three parameters: a cookie box, the diameter of the cookie (cm) and the price of a cookie in dollars (Each cookie will have the same diameter and price)

```
fix_cookie_box: cookie_flavor list -> float -> float -> cookie_flavor list list
```

For each element in the cookie_flavor list, create a new list of 2 cookies consisting of the same flavor, but using the new diameter and new price provided.

For example,

fix_cookie_box [LaddooLemon (55.4, 66.9); CaramelPumpkin (77.3, 88.3)] 1.2 3.4 =
[[LaddooLemon (1.2, 3.4); LaddooLemon (1.2, 3.4)];[CaramelPumpkin (1.2, 3.4);
CaramelPumpkin (1.2, 3.4)]]

You can use the functions declared above in 6.1 - 6.5 and still have access to List.fold_left

7.1

Create the function

Question 8.

8.a

Given the following mystery1 function, what is the type of the mystery1 function?. You must also explain your answer. Failure to explain your answer or giving incorrect explanation will result in zero credits

```
let rec mystery1 (f, l1, l2) =
        match (l1, l2) with
        | ([], []) -> []
        | (_, []) -> []
        | ([], _) -> []
        | (h1::t1, h2::t2) -> [f (Some (LaddooLemon(h1, h2)))] :: mystery1 (f, t1, t2)
```

- mystery1 is of type: **((cookie_flavor option -> 'a) * float list * float list)) -> 'a list list**
- **This is because mystery1 takes in a tuple with three parts: function f of type cookie_flavor option -> 'a, as well as two float lists. After mystery1 computes, it spits out a type 'a list list.**

8.b

Given the following mystery2 function, what is the type of the mystery2 function? You must also explain your answer. Failure to explain your answer or giving incorrect explanation will result in zero credits

```
let rec mystery2 f l1 l2 =
        match l1, l2 with
        | [], [] -> []
        | _, [] -> []
        | [], _ -> []
        | Some h1::t1, LaddooLemon h2::t2 ->
        (
        Some (h1 (Some (LaddooLemon(4.1, 5.3)))) :: mystery2 f t1 t2
        )
    | _, _ -> []
```

- mystery2 is of type: **(f: 'b) (l1 : (cookie_flavor option -> 'a) option list) (l2 : cookie_flavor list):'a option list**
- **This is because mystery2 takes in a function f (of which its typing is unknown and somewhat obsolete), as well as l1 which is of type (cookie_flavor option -> ‘a) option list. (super confusing). Then l2 which is different than l1 because it is just a cookie_flavor list. After all of the computations a value of type 'a option list is then spit out by mystery2.**