COMP 302 - Midterm Exam (Reformatted)

Question 1: Short answers, big ideas

1. The following function is not tail-recursive, make it tail recursive with the help of an inner accumulator function:

```
let rec pow2 n =
 if n = 0 then 1 else 2 * pow2 (n-1)
```

2. OCaml top-level types:

```
let double x = 2 * x
let app = fun f x -> f x
```

   (a) What is the most general type of `app`?
   (b) What is the most general type of `let my_func = app (app double)`?

3. Scoping and evaluation:

```
let x = 6 in (let x = 9 in x) * x
```

   Considering the code above, what is the expected output ?

4. Pattern matching bugs:

```
let func f lst1 lst2 =
  match (lst1, lst2) with
  | (_, []) -> []
  | ([], _) -> []
  | (x::xs, Some y) -> f x y :: func f xs lst2
  | (x::xs, y::ys) -> f x y :: func f xs ys
  | _ -> ["Error"]
```

   Identify and name the 4 bugs in the code (syntax, type, or runtime).

5. Type of partial function:

```
let rec func f arg =
  match arg with
  | [] -> []
  | x :: xs ->
    match x with
    | Some x -> f x :: func f xs
    | None -> func f xs
```

   What is the most general type of `func`?

Question 2: Higher-order functions on sale – two for the price of one

1. Write a transforming predicate of type `int -> int option` that returns `Some (x * x)` if `x` is even, and `None` otherwise.

2. Implement the function `trans_pred : ('a -> 'b) -> ('a -> bool) -> ('a -> 'b option)` which uses a transformation and a predicate to construct a transforming predicate. The resulting function returns `None` when the predicate returns false, and `Some` containing a transformed value of type `'b` when the predicate returns true.

```
trans_pred : ('a -> 'b) -> ('a -> bool) -> ('a -> 'b option)
(* your code here *)
```

3. Implement the function `filter_map : (‘a -> ‘b option) -> ‘a list -> ‘b list.` The structure of this function is similar to that of both map and filter. It traverses the list to call the transforming predicate on each element. When the transforming predicate returns None, then that element gets thrown out; but when the transforming predicate returns Some y, then the y gets included in the output list. A direct implementation of recursion is required. It does not need to be tailrecursive. Do not use any other functions to implement `filter_map`.

```
filter_map : ('a -> 'b option) -> 'a list -> 'b list
(* your code here *)
```

4. Now refactor the expression map f (filter p l) using what you have developed in this question. (That is, rewrite map f (filter p l) in terms of the functions you defined above.)

Question 3: Continuations? I'll pass, thanks

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree

let rec flatten t =
  match t with
  | Leaf x -> [x]
  | Node (l, r) -> flatten l @ flatten r
```

Rewrite `flatten` in continuation-passing style (CPS). Your version may use `(@)` and must be tail-recursive.

Question 4: Move fast, break things

In many game engines, a central concern is the implementation of physics. A physics engine tracks the **position**, **velocity**, and **acceleration** of objects in a scene and updates these variables over time.

In this question, we consider a simplified setup with two different kinds of objects, equipped with only one-dimensional physical properties.

- **Static**: These objects are equipped with only a position.
- **Dynamic**: These objects are equipped with position, velocity, and acceleration.

```
type pos = float (* position *)
type vel = float (* velocity *)
type acc = float (* acceleration *)

type object = Static of pos | Dynamic of pos * vel * acc
```

In this question, you will ultimately implement a function to decide whether any dynamic object in a scene collides with some fixed boundary. In the game, this might trigger a cutscene.

1. Physics Simulation

A physics engine uses a small but finite Δt to calculate the change in velocity Δv and position Δx of a Dynamic object:

$$\Delta v = a \cdot \Delta t \qquad \Delta x = v \cdot \Delta t$$

Here, x, v, and a correspond to `pos`, `vel`, and `acc` respectively.

A dynamic object's new velocity and position are obtained by adding these Δv and Δx to the object's current velocity and position. Acceleration remains constant. A static object is unaffected by the passage of time.

Implement this physics simulation:

```
let simulate delta obj =
    (* your code here *)
```

2. Update All Objects

Implement the function:

```
simulate_all : float -> object list -> object list
```

that updates the position of all objects in the list using the `simulate` function. Your solution must use:

```
map : ('a -> 'b) -> 'a list -> 'b list
```

```
let simulate_all delta objs =
    (* your code here *)
```

3. Collision Detection

Implement:

```
collides : float -> pos -> object -> bool
```

that decides whether `obj` passes the point `x` after `delta` time elapses.
You can use the function:

```
position : object -> pos
```

which extracts the position of an object. You must also use the previously defined `simulate` function.

```
let collides delta x obj =
    (* your code here *)
```

4. Any Object Collides

Finally, implement:

```
any_collides : float -> pos -> object list -> bool
```

that checks if any object in the list collides with the given point during the passage of `delta` time.
You must use:

```
exists : ('a -> bool) -> 'a list -> bool
```

```
let any_collides delta point objs =
    (* your code here *)
```