Question 1: Short answers, big ideas

1. The following recursive algorithm is such that `pow2 n` calculates 2^n.

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

It is not tail-recursive. Rewrite it tail-recursively by introducing an *inner helper function* with an accumulator and calling the helper appropriately. Your solution should (conceptually) use the same approach to calculating 2^n.

```
Solution
let rec pow2 (n : int) : int =
     let go (n : int) (acc : int) : int =
          if n = 0 then acc
          else go (n-1) (2 * acc)
     in
     go n 1
```

2. Suppose we define the following functions in the OCaml toplevel:

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

2.1 What is the most general type of `app`?

Solution: the first argument is a function of type `('a -> 'b)`. This function is applied to `x`, so `x` must be of type `'a`. Since the function outputs something of type `'b`, the type of `app` must be `('a -> 'b) -> 'a -> 'b`

2.2 What would be the most general type of the function `let my_func = app (app double)`?

Solution: We know `app: ('a -> 'b) -> 'a -> 'b`, and we can see that `double : int -> int`. So when we do `app double`, we can see that the `int -> int` matches up to `('a -> 'b)`, so `'a = int, 'b = int`, so `app double: int -> int`.

Now we apply `app` to the result of `app double`, which itself is a function. `app : ('a -> 'b) -> 'a -> 'b`

So now
`my_func = app (app double) = app (fun x -> double x)`
We call (the outer) `app` on a function of type `int -> int`.
Following the logic of the previous question, this means that that
`app (app double) : int -> int` so `my_func : int -> int`

3. Consider the following session in the OCaml toplevel. What is the expected output? Give a short explanation for your answer.

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

Solution: This outputs 54 because the "x = 9" only applies to the inner x (in the parenthesis), but the second x is not a part of the let-in expression for the x=9, so the second x is 6.

4. The following function is defined with pattern matching and contains four problems: they could be syntactic mistakes, type errors, or potential runtime errors. Circle in the code each bug and write a short name for the bug.

```
let [rec] 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"]
```

Solution:

1. There is a missing `rec` keyword, needed since we make recursive calls.
2. Using `Some y` implies that `y` is an object, but `lst2` is a list.
3. This function needs to output a list, but `["Error"]` is not a list.
4. (The final catch all is not necessary?)

5. What is the most general type of `func` defined below:

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

Solution: `func` takes two arguments: `f` and `arg`.
Since we pattern match on `arg` with `[]` and return `[]` in one of the branches, we know that `arg` is a type of `'a list` and that the output of `func` is `'b list`.

When we pattern match on `x`, we can see that `x` must be `'a option` because we match on `Some x` or `None`.

In the `Some x` case, we apply `f` to `x`, so `f : 'a -> 'b`.

The result of `f x` gets added to a list with `f x :: func f xs`, so `func f xs` (in the last line) must return a `'b list` too.

Together, we can infer `f : 'a -> 'b, arg : 'a option list,`

`func f arg : 'b list`

`func : ('a -> 'b) -> 'a option list -> 'b list`

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

The higher order functions `filter : ('a bool) -> ('a list) -> 'a list` and `map : ('a -> 'b) -> 'a list -> 'b list` are extremely commonly used in the codebase that you're working on.

You notice this pattern occurs frequently: `map f (filter p l)`. This is inefficient, as a list is created for the output of filter only to be passed to map which creates yet another list.

You decide to create a function `filter_map` that performs the actions of `map` and `filter` at once, eliminating the intermediary list.

To combine the actions of `filter`'s *predicate*, of type `'a -> bool`, with the action of `map`'s *transformation*, of type `'a -> 'b`, you define a new concept called a *transforming predicate*, of type `'a -> 'b option`.

1. Give a transforming predicate that checks that its input is even, and when it is, squares it; it returns `None` when its input is not even.

```
Solution: [POSSIBLY lol]
let even_square (x : int) : int option =
     if x mod 2 = 0 then Some (x * x)
     else None
```

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 transformation returns `None` when the predicate returns `false`. Else, it uses the transformation to return a `Some` containing a value of type `'b`.

```
Solution:
let trans_pred (f: 'a -> 'b) (p: 'a -> bool) : 'a -> b option
     if p x then Some (f x) else None
```

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 tail-recursive. Do not use any other functions to implement `filter_map`.

Solution:

```
let rec filter_map (tp : 'a -> 'b option) (l : 'a list) :
'b list =
     match lst with
     |[] -> []
     | x :: xs ->
          match tp x with
          | Some y -> y :: filter_map tp xs
          | None -> filter_map tp xs
```

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.)

Solution:

```
filter_map (trans_pred f p) l
```

Question 3: Continuations? I'll pass, thanks

The following is the definition of a tree data structure, and a function `flatten : 'a tree -> 'a list` that produces a list resulting from a traversal of a tree.

`flatten` uses the built-in function `(@) : 'a list -> 'a list -> 'a list` to append two lists.

```
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
```

Translate flatten into continuation-passing style to implement a tail-recursive version of it. Your solution may use `(@)`.

Solution:

```
let flatten_cps (t : 'a tree) : 'a list =
     let rec go t cont =
          match t with
          | Leaf x -> cont [x] (* when we hit a leaf, we call
                                  the continuation with [x] *)
          | Node (l, r) ->
               go l (fun left_result ->
                    go r (fun right_result ->
                         cont (left_result @ right result)
                    )
               )
     in
     go t (fun x -> x) (* start the recursion with identity
                          continuation*)
```

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 = State of pos | Dynamic of pos * vel * acc
```

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

1. A physics engine uses a small but finite Δt to calculate change in velocity Δv and position Δx of a `Dynamic` object by

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

   x, v, and a here 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, respectively; acceleration is constant.
   A static object is unaffected by the passage of time.
   Implement this physics simulation with the function `simulate : float -> object -> object` such that simulate delta obj returns a new object with its position and velocity updated according to the rule above, using the given value for Δt.

Solution:

```
let simulate (delta : float) (obj : object) : object =
     match obj with
     | State x -> State x
     | Dynamic (x, v, a) ->
          let delta_v = a *. delta in
          let delta_x = v *. delta in
          let new_v = v +. delta v in
          let new_x = x +. delta_x in
          Dynamic (new_x, new_v, a)
```

2. Implement the function `simulate_all : float -> list object -> list object` that updates the position of all the objects in the list, by returning a lsit of new object with updated properties. You should use the function simulate from the previous part. Your solution must use the function `map : ('a -> 'b) -> 'a list -> 'b list`

Solution:

```
let simulate_all (delta : float) (objs : object list) : object
list =
     map (simulate delta) objs


Solution 2:
let simulate_all (delta : float) (objs : object list) : object
list =
     let rec map f objs
     match objs with
     | [] -> []
     | obj :: rest -> f obj :: map f rest
     in
     map (simulate delta) objs
```

3. Implement a function `collides : float -> pos -> object -> bool` such that `collides delta x obj` decides whether or not `obj` passes the point `x` after `delta` time elapses. An object is said to have crossed the point if before `delta` elapses, the object is on one side of `x` but after delta elapses it is on the other side of `x`.
   You can use the function `position : obj -> pos` which obtains the position of an object regardless of whether it's dynamic or static.
   You must use the function `simulate` defined earlier.

Solution:

```
let collides (delta : float) (x : float) (obj : object) :bool =
     let start_pos = position obj in
     let end_pos = pos (simulate delta obj) in
     (x -. start_pos) *. (x-. end_pos) < 0.0
(* if the signs of the position before and after delta are the
same, multiplying them gives a positive value, and there was no
collision. if the signs are opposite, there was a collision,
the multiplication will yield a negative result, and we return
true *)
```

4. Implement the function `any_collides : float -> pos -> obj list -> bool` that checks if any of the given objects collide with the given point during the passage of the given length of time.
   You must use the function `exists: ('a bool) -> 'a list -> bool`, which decides whether there is at least one element of a given list satisfying the given condition.

Solution:

```
let any_collides (delta : float) (point : pos) (objs : obj
list) : bool =
     exists (fun obj -> collides delta point obj) objs
```