FPSE Fall 2025

Please read all of the following items before beginning the quiz.

- You may assume that `Core` is opened above every block of code in this quiz.
- We hope and expect that the provided type signatures and examples completely describe the expected functionality.
- For discussion questions, your answers should be only one or two sentences. Provide brief answers only. We are not looking for lengthy justification.
- If your answer signficantly overflows the provided space, you might be on the wrong track. Use the extra sheets only if necessary.
- You are not to use mutation anywhere in any coding question. Coding answers using mutation will receive zero points.
- You are allowed a two-sided, handwritten, 8.5x11in cheat sheet.

I affirm that I have completed this quiz without unauthorized assistance from any person, materials, or device.

Signed: _____

Print name: _____

(Please print your name clearly so that Gradescope can recognize it.)

Date: 19 Nov 2025

Distribution of Marks

Question	Points	Score
1	15	
2	9	
3	13	
4	13	
Total:	50	

1. Functors are a characteristic feature of OCaml: they allow parametrized modules and dependent typing of modules. Further, monads are characteristic of functional programming: they are a way to encode effects functionally and get types on those encoded effects. Let's cover both of these topics at once!

   (a) (5 points) We'll start with a concrete, non-functor example to get our footing. Here is an example usage of each of `map`, `join`, and `bind` from the `List` library:

   ```
   List.map [1; 2; 3] ~f:(fun x -> x + 1)
   - : int list = [2; 3; 4]
   List.join [[4; 5]; [6; 7]]
   - : int list = [4; 5; 6; 7]
   List.bind [8; 9] ~f:(fun x -> [x; x])
   - : int list = [8; 8; 9; 9]
   ```

   You will use `map` and `join` to implement `bind` for the `List` library.

   ```
   val bind : 'a list -> f:('a -> 'b list) -> 'b list
   ```

   Below, write an implementation of `List`'s `bind` function using only `List.map` and `List.join`.

   (b) (10 points) What you did for `List` can in fact be done for any monad. We describe this with the module types `MONAD` and `DERIVE_BIND`, which generalize what you did above.

   ```
   module type MONAD = sig
     type 'a m
     val return : 'a -> 'a m
     val map : 'a m -> f:('a -> 'b) -> 'b m
     val join : 'a m m -> 'a m
   end

   module type DERIVE_BIND = functor (M : MONAD) -> sig
     include MONAD with type 'a m = 'a M.m
     val bind : 'a m -> f:('a -> 'b m) -> 'b m
   end
   ```

   Now, implement a functor below that has the signature `DERIVE_BIND`.

2. We don't have operator overloading in OCaml, but we *can* define custom infix operators for cleaner code. Some common infix operators are `@@`, `|>`, and `@`, for example.

   In this question, you will write some infix operators from scratch.

   Before you get started, see this example for the syntax of an infix operator called `++`, as well as one usage:

```
let (++) a b = List.append a b
let one_two_three = [ 1 ; 2 ] ++ [ 3 ]
```

   (a) (3 points) Let's make an infix operator that looks like a placeholder argument for a two-argument function. It will be called `><`. Write the infix operator `><` with the following signature:

```
val (><) : ('a -> 'b -> 'c) -> 'b -> ('a -> 'c)

(* EXAMPLE USAGE *)
let f = List.append >< [4; 5; 6] in
f [1; 2; 3]
- : int list = [1; 2; 3; 4; 5; 6]
```

   (b) (3 points) The OCaml infix operator `@@` is the same as using a space to indicate function application but with looser operator precedence. It is also a lot like piping with `|>`, but in the reverse direction. See the following example behavior if you've forgotten how it works.

```
(* EXAMPLE *)
List.append [1; 2; 3] @@ List.rev [4; 5; 6]
- : int list = [1; 2; 3; 6; 5; 4]
```

   What is the type signature of `@@`?

   (c) (3 points) Now implement the `@@` operator from scratch with that signature.

3. It's always important to think about the time and space complexity of the data structures you're using, as well as what they actually look like. Functional data structures are often quite different than the corresponding mutable data structure, so let's talk about that.

   (a) (3 points) Illustrate with a tree diagram how a functional list in OCaml is structured (using the cons `(::)` constructor).

   We suggest that you use this example, `[1; 2; 3; ...; n]`, with some abstract, finite `n`.

   (b) (4 points) What is the time complexity of `List.append a b` where `a` has length n and `b` has length m? Explain your answer briefly.

   (c) (3 points) Explain how functions such as `Map.add` and `Map.set` in a functional map data structure (implemented as a balanced tree) can be $O(\log n)$ time.

   (d) (3 points) When is the amortized time complexity of the functional data structure `Map` better than the mutable data structure `Hashtbl`?

4. In class, we said the default implementation of a `fold` function is typically `fold_left`, and the reason for that was something called "tail recursion". A tail recursive function is a function where the recursive call is the last operation in the function. That is, the result of the recursive call is not ever used in the body of the function to do more computation.

   (a) (4 points) What is one reason we might prefer that `fold` is tail recursive instead of non-tail-recursive?

   (b) (9 points) It's actually possible to make *any* function tail recursive. Let's do it to `fold_right`! Make the `fold_right` function tail recursive by writing a new function `tl_rec_fold_right` that has the same behavior, is tail recursive, and has the same asymptotic time complexity.

   Hint: you may make `tl_rec_fold_right` non-recursive and define a tail recursive helper function.

   Hint2: or a combination of several tail-recursive `List` library functions can do the trick.

   Here is the original `fold_right` implementation (which is NOT tail recursive):

```
let rec fold_right (ls : 'a list) ~(init : 'acc) ~(f : 'acc -> 'a -> 'acc) : 'acc =
  match ls with
  | [] -> init
  | hd :: tl -> f (fold_right tl ~init ~f) hd
```

   Now write your answer below.

Use this page for scratch work or for your continued answers if you ran out of space. Clearly indicate if your work is an answer to a question in the quiz.