FPSE Fall 2023

This version contains solutions and staff notes/discussion. Comment out \answertrue to get the student version.

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

- 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.

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: 15 Nov 2023

Distribution of Marks

Question	Points	Score
1	4	
2	4	
3	4	
4	3	
5	4	
6	3	
Total:	22	

1. (a) (2 points) Write an OCaml function that removes duplicates from a list and meets the following signature.

```
val remove_duplicates : 'a list -> compare:('a -> 'a -> int) -> 'a list
```

You can assume that the provided `compare` function returns `0` if and only if the elements are equal, i.e. they count as a duplicate. Keep the first occurance of each item in the list. Do not change the order of the elements in the list, except for removing duplicates. You may use `let rec` if you want. added this to avoid Q about whether let rec is allowed - Scott accepted change - Brandon

```
(* EXAMPLE *)
remove_duplicates [1; 3; 1; 2; 1; 2] ~compare:Int.compare
- : int list = [1; 3; 2]
```

Your implementation goes below. I suggest removing the header, they should be able to infer it from the example. Question is too easy otherwise. - Scott Done - Brandon

```
(* SOLUTION *)
let remove_duplicates (ls : 'a list) ~(compare:'a -> 'a -> int) : 'a list =
  let rec loop acc = function
  | [] -> acc
  | hd :: tl when List.mem acc hd ~equal:(fun x y -> compare x y = 0) -> loop acc tl
  | hd :: tl -> loop (hd :: acc) tl
  in
  loop [] ls |> List.rev

Grading notes:
  * They can mess up some syntax as long as it's clear what they're trying to do
  * Consider if their list will be reversed
```

(b) (1 point) Now, write one invariant test for `remove_duplicates` that always works on any given list. Use `assert_equal` from `OUnit2`, and assume `OUnit2` has been opened already. May want to mention to use assert_equal from OUnit - ? Or just say use assert. It will provoke questions about this as written. - Scott Added as requirement. Thoughts on keeping this header? - Brandon

```
let test_remove_duplicates_invariant (ls : 'a list) : unit =
  (* Your test logic goes below this comment *)
```

We had to note during the quiz that they have a polymorphic compare: 'a -¿ 'a -¿ int because the invariant test is polymorphic. - Brandon

```
(* SOLUTION *)
(* assume we have polymorphic [compare : 'a -> 'a -> int] *)
let remove_duplicates = remove_duplicates ~compare in
assert_equal
  (remove_duplicates ls)
  (remove_duplicates @@ remove_duplicates ls)

Grading notes:
  * As long as the test always passes and uses remove_duplicates, it's okay
```

(c) (1 point) What is `Quickcheck`? Briefly describe how `Quickcheck` might be used in your invariant test above.

SOLUTION:

Quickcheck helps to run many random inputs through the given function.

It could be used to check that every randomly generated list passes the invariant test.

Grading notes: there's not much wiggle room here. As long as they know it runs tests on random inputs, and the rest of their answer is logically correct, then they have probably arrived at this exact answer.

2. (a) (2 points) Make an empty `Core.Map` where the keys are records of type `{ first : int ; second : float }`. You might like to use preprocessor extensions from `ppx_jane` to help.

```
(* SOLUTION *)
module T : Map.Key =
  struct
    type t =
      { first  : int
      ; second : float }
      [@@deriving sexp, compare]
  end

module M_map = Map.Make (T)

let empty = M_map.empty
```

(b) (2 points) Now, add both of the following mappings into the empty map, and call the new map `m`:

- `{first = 5 ; second = 10.}` maps to `Some "hello world"`, and
- `{first = 6 ; second = 11.}` maps to `None`.

Pay careful attention to the return type, and think about whether an option (or something similar) is returned.

```
(* SOLUTION *)
let m =
  empty
  |> Map.add_exn ~key:T.first = 5; second = 10. ~data:(Some "hello world")
  |> Map.add_exn ~key:T.first = 6; second = 11. ~data:None

Grading notes:
  * Some syntax mistakes are okay
  * A mistake like not using add_exn is more critical, and they will lose some points
    * Alternatively, they can pattern match with Or_duplicate if they remember that
```

3. In this question, we'll discuss the complexity of various data structures in OCaml.

   (a) (2 points) Name the (amortized) time complexity in big-O notation to look up a key or index in each of the following OCaml data structures. Assume comparisons between elements (e.g. between keys) are $O(1)$. If you're not sure, then take a guess and provide a very brief justification for that guess.

      - Look up the n'th element in a `List` : SOLUTION: O(n)
      - Add a key-value pair into a `Map` : SOLUTION: O(log n)Note I would either add or change to the case of adding a key-value pair to a map as that is more subtle. - Scott changed to add key-value pair. Used to be lookup in Map - Brandon
      - Look up a key in a `Hashtbl` : SOLUTION: O(1)
      - Check if an element exists in a `Set` : SOLUTION: O(log n)

   (b) (1 point) When might the time complexity of a functional data structure be better than a mutable data structure?

   SOLUTION: copying.

   Grading notes: I don't think that resizing a table affects time complexity of a mutable data structure because it would still be amortized constant, so copying is the only valid answer I know of.

   (c) (1 point) Besides the one case of improved time complexity in part (b), give one other reason we might choose to use `Map` over `Hashtbl`. There is an issue that their (b) answer could be that it is faster in some cases which we give them for free in (c) - need to rule out that pattern of answer somehow. - Scott Swapped b and c (so your b and c now refer to the wrong one) and added exception that they can't reuse answer to part b. - Brandon

   SOLUTION: because Map has no side effects, and it's therefore easier to tell what happens just from a function signature.

   Grading notes: anything that makes sense can earn a point here.

4. (3 points) In this question, you'll write code to curry and uncurry a function.

```
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

(* EXAMPLES *)
let f (x, y) = x + y
in
curry f 1 2
- : int = 3

let f x y = x + y
in
uncurry f (3, 4)
- : int = 7
```

Now implement `curry` and `uncurry` in the space below.

```
(* SOLUTION *)
let curry = fun f -> fun x -> fun y -> f (x, y)
let uncurry = fun f -> fun (x, y) -> f x y
```

5. (4 points) Write implementations for the bind (`>>=`) and map (`>>|`) infix operators for an option-like monad according to the signatures and example usages below. Maybe just make this for the bind only, not map. Map is just a variation on bind. - Scott No changes here yet. I kind of like keeping map. Maybe we have them write map using bind? I added an example to the solution below, so take a peek at that. - Brandon

```
(* EXAMPLES *)
Nothing >>= fun x -> Something (x + 1)
- : int MyOption.t = MyOption.Nothing

Something 5 >>= fun x -> Something (x + 1)
- : int MyOption.t = MyOption.Something 6

Nothing >>| fun x -> x + 1
- : int MyOption.t = MyOption.Nothing

Something 5 >>| fun x -> x + 1
- : int MyOption.t = MyOption.Something 6
```

Your implementation goes below.

```
module MyOption :
  sig
    type 'a t =
      | Nothing
      | Something of 'a
    val (>>=) : 'a t -> ('a -> 'b t) -> 'b t
    val (>>|) : 'a t -> ('a -> 'b) -> 'b t
  end
  =
  struct
    type 'a t =
      | Nothing
      | Something of 'a
    (* Complete the rest of the module under this comment *)

    let (>>=) (x : 'a t) (f : 'a -> 'b t) : 'b t =

      (* SOLUTION *)
      match x with
      | Nothing -> Nothing
      | Something y -> f y


    let (>>|) (x : 'a t) (f : 'a -> 'b) : 'b t =

      (* SOLUTION *)
      match x with
```

```
    | Nothing -> Nothing
    | Something y -> Something (f y)
```

agree re note to staff - Scott

The ‘let‘ statements to help with infix syntax have been added above. The note to staff has been removed. - Brandon

```
end
```

6. (3 points) In this question, you will write an `.ml` file to accompany an `.mli` file. You can provide any functionality you want, as long as it satisfies the `.mli` file. This question is only to demonstrate that you understand signatures.

```
(* begin my_module.mli *)
module type S =
  sig
    type t
    val x : t
  end

module type S2 =
  sig
    include S
    val y : t -> t
  end

module M (_ : S) : S2
(* end my_module.mli *)
```

Fill in the space below as if it is `my_module.ml`.

```
(* begin my_module.ml *)

  (* SOLUTION *)
  module type S =
    sig
      type t
      val x : t
    end

  module type S2 =
    sig
      include S
      val y : t -> t
    end

  module M (S : S) : S2 =
    struct
      include S
      let y = fun _ -> S.x
    end


  Grading notes:
    * Any extra code is okay as long as the signature is satisfied.


(* end my_module.ml *)
```

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.

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.