CIS 1200 Midterm I September 29, 2023

Steve Zdancewic and Swapneel Sheth, instructors

Name: _____

PennKey (penn login id, e.g., `stevez`): _____

I certify that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

- Please wait to begin the exam until you are told it is time for everyone to start.
- When you begin, please start by writing your PennKey at the bottom of all the odd-numbered pages in the rest of the exam.
- There are 120 total points. The exam length is 60 minutes.
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc.
- There are 10 pages in the exam and an Appendix for your reference. Please do not submit the Appendix.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- Good luck!

1. **Binary (Search) Trees** (26 points total)

   (a) (14 points) Write a function `convert_tree` that given a binary tree `t` and an integer `n`:

   - First, adds that `n` to the value of the root node, if it exists.
   - Second, adds the old node value to the right child node and subtracts the old node value from the left child node.
   - Applies the second step recursively to the entire tree.

   For example, given the `original` tree below and `n = 4`, after applying the function, the `converted` tree would look like this.

```
let original : int tree =            3
                                    / \
                                   4   1
                                  /   / \
                                 5   8   12

let converted : int tree =           7
                                    / \
                                   1   4
                                  /   / \
                                 1   7   13
```

   Now, implement the function:

```
let rec convert_tree (t: int tree) (n: int) : int tree =
```

   _____

   _____

   _____

   _____

   _____

   (b) (4 points) Does `convert_tree` preserve the BST invariants? (I.e., if the input tree `t` were a BST, would the output tree *always* be a BST?)

   ☐ Yes ☐ No

   If yes, explain why. If no, provide a counter-example.

You will be given a pair of Binary Search Trees (`original` and `updated`). Start with the `original` tree and using a series of `insert`s and `delete`s create the `updated` tree.

For each part:

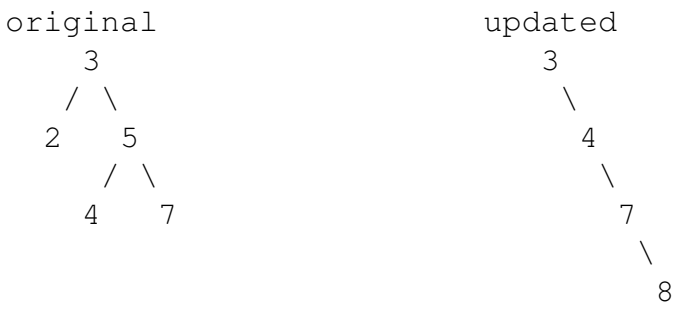
- Start with the innermost parenthesis.
- You can call `insert` and `delete` a maximum of 2 times each.
- You may not need to use all the blanks provided.

The code for `insert` and `delete` is available in Appendix A.

We've done the first one for you.

Answer: _____ (_____ (insert (delete original 8) 5))

(c) (4 points)



Answer: _____ (_____ (_____ (_____ original ____) ____) ____) ____

(d) (4 points)

Answer: _____ (_____ (_____ (_____ original ____) ____) ____) ____

2. **List Processing and Higher Order Functions** (26 points total)

   Recall the higher-order list processing functions shown in Appendix B.

   For these problems *do not* use any list library functions. Constructors, such as `::` and `[]`, are fine.

   (a) (7 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `divided_by_5` that takes in an `int list` and returns a tuple where the first element is the number of elements that can be divided by 5 and the second element is a list of those numbers.

   For example, the call `divided_by_5 [1; 15; 20; 4]` evaluates to `(2, [15; 20])`.

```
let divided_by_5 (l: int list) : int * int list =
```

   _____

   _____

   _____

   _____

   (b) (7 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `dupl_predicate` that takes in a predicate of the type `'a -> bool` and a `'a list` and returns a new list where elements that match the predicate are duplicated and elements that don't remain as singular elements.

   For example, the call `dupl_predicate (fun x -> x > 5) [1; 7; 6; 2; 8]` evaluates to `[1; 7; 7; 6; 6; 2; 8; 8]`

```
let dupl_predicate (pred: 'a -> bool) (l: 'a list) : 'a list =
```

   _____

   _____

   _____

   _____

(c) (12 points) Use `transform` and/or `fold`, along with suitable anonymous function(s), to implement a function `multiplier` that takes in a list whose elements are of type `int * (int list)` where the first element of the tuple is a multiplier and the second element of the tuple is a list of integers. `multiplier` will return a list of lists where each number in a list is multiplied by its corresponding multiplier.

For example, the call

```
multiplier [(2, [1; 2; 3]); (-3, [2; 3; 4]); (0, [3; 2; 1])]
```

evaluates to `[[2; 4; 6]; [-6; -9; -12]; [0; 0; 0]]`.

```
let multiplier (l: (int * (int list)) list) : int list list =
```

_____

_____

_____

_____

_____

3. **Types** (24 points total)

   For each OCaml value below, fill in the missing type annotations or else write "ill typed" if there is no way to fill in the annotation that does not cause a type error.

   Your answer should be the *most generic* type that OCaml would infer for the value——*i.e.*, if `int list` and `bool list` are both possible types of an expression, you should write `'a list`.

   Some of these expressions refer to the types and functions defined in Appendix A, B, and C.

   We've done the first one for you.

```
let example: _____int list_____ = [3; 1]
```

   (4 points each)

   (a)
```
let a: _____________________________ = "h"::"e"::"l"::"l"::["o"]
```

   (b)
```
let b: _____________________________ = (5, [1; 2; 3; 4])
```

   (c)
```
let c: _____________________________ = [(fun x -> x + 5); (fun x -> 120)]
```

   (d)
```
let d: _____________________________ = transform (fun y -> y + 3)
```

   (e)
```
let e: _____________________________ = begin match [true; false; true] with
                                       | [] -> true
                                       | hd::tl -> hd || hd + 5
                                       end
```

   (f)
```
let f: _____________________________ = fun x -> Node (Empty, x, Empty)
```

4. **Abstract Data Types & The Design Process** (44 points total)

Disney is looking to make their queue process more automated. To do so, they decide to create a *queue system* that prioritizes fast-pass customers over regular customers.

To model this situation in code, we create an *abstract data type* with operations `enq`, which adds a value to the queue, and `deq`, which returns the next value (if any) along with an updated queue. The elements of the queue are dequeued in the order in which they are enqueued except that fast-pass values always come *before* regular ones. Whether a value is to be considered "fast-pass" or "regular" is indicated by a `bool` argument to `enq` (**`true`** means "fast"). The data type will also need an `empty` value and support a `to_list` operation, which returns the list of values *in the order they will be dequeued*.

a. We now consider how to define the signature of this `DISNEYQUEUE` abstract type. We can characterize the possible designs as:

- **Unimplementable**: no well-typed **`struct`** *implementation* could satisfy the interface: any implementation would have to raise an error (e.g., `failwith`) or infinitely loop
- **Unusable**: implementable, but lacking functionality: no *client* code could usefully call functions of the interface to achieve a non-trivial result
- **Unsafe**: implementable and usable, but that doesn't ensure implementation invariants are preserved: the client can provide inputs that break implementation invariants
- **Good**: implementable, usable, and able to enforce invariants

For each of the following signatures, mark the box next to the characterization that best describes it. Additionally, if it is *not* "Good", briefly describe why you chose that choice. ***Use each characterization exactly once!***

(a) (4 points)

```
module type DISNEYQUEUE = sig
  type 'a disney_queue = ('a * bool) list    (* see Note *)
  val empty : 'a disney_queue
  val deq : 'a disney_queue -> 'a * 'a disney_queue
  val enq : 'a disney_queue -> 'a -> bool -> 'a disney_queue
  val to_list : 'a disney_queue -> 'a list
end
```

☐ Unimplementable ☐ Unusable ☐ Unsafe ☐ Good

Note: the presence of **`type`** `'a disney_queue = ('a * bool) list` in the signature *reveals* the definition of the type to client code.

Explanation: _____

_____

_____

(b) (4 points)

```
module type DISNEYQUEUE = sig
  type 'a disney_queue
  val empty : 'a disney_queue
  val deq : 'a disney_queue -> 'a * 'a disney_queue
  val enq : 'a disney_queue -> 'a -> bool -> 'a disney_queue
  val to_list : 'a disney_queue -> 'a list
end
```

☐ Unimplementable ☐ Unusable ☐ Unsafe ☐ Good

Explanation: _____

_____

_____

(c) (4 points)

```
module type DISNEYQUEUE = sig
  type 'a disney_queue
  val deq : 'a disney_queue -> 'a * 'a disney_queue
  val enq : 'a disney_queue -> 'a -> bool -> 'a disney_queue
  val to_list : 'a disney_queue -> 'a list
end
```

☐ Unimplementable ☐ Unusable ☐ Unsafe ☐ Good

Explanation: _____

_____

_____

(d) (4 points)

```
module type DISNEYQUEUE = sig
  type 'a disney_queue
  val empty : 'a disney_queue
  val deq : 'a disney_queue -> 'b * 'b disney_queue
  val enq : 'a disney_queue -> 'a -> bool -> 'b disney_queue
  val to_list : 'a disney_queue -> 'b list
end
```

☐ Unimplementable ☐ Unusable ☐ Unsafe ☐ Good

Explanation: _____

_____

_____

b. (4 points) Now we can write test cases that check the desired properties of our DISNEYQUEUE abstract type. One (correct) example test is shown below:

```
let test () =
  let q1 = enq empty 1 false in
  let q2 = enq q1 2 true in
  let (r1, q3) = deq q2 in
  let (r2, q4) = deq q3 in
  r1 = 2 && r2 = 1 && is_empty q4
;; run_test "1 slow then 2 fast" test
```

Now, fill in the blank below such that the following code is another good test:

```
let test () =
  let q1 = enq empty 1 false in
  let q2 = enq q1 2 true in
  let q3 = enq q2 3 true in
  let q4 = enq q3 4 false in
  to_list q4 =
                  ____________________________
;; run_test "to_list test" test
```

c. There are many possible ways to implement the DISNEYQUEUE interface.

(2 points) For each possible *representation type* labeled A–D below, mark the box if it can be used to provide a **Good** implementation. (At least one is good, but more than one might be.)

```
module DQ_Impl : DISNEYQUEUE = struct
  type 'a disneyqueue =                        (* GOOD? *)
     (* A *)           'a list * bool            [ ]
     (* B *)           'a list * 'a list         [ ]
     (* C *)           'a list * int             [ ]
     (* D *)           'a set                    [ ]

  (* ... other definitions omitted will use that representation ... *)
end
```

(6 points) Now, pick *one* of the **Good** representations above and explain (briefly) what *representation invariant* your code could use to implement DISNEYQUEUE for that type.

Type: _____
Invariant:

_____

_____

_____

d. (16 points) One way to represent a `DISNEYQUEUE` (different from above) is through a list, where each element in the list is a tuple of the value (i.e., customer) and a `bool` flag that indicates whether they are a fast-pass (**`true`**) or regular customer (**`false`**). The **`struct`** for such an implementation is provided in **Appendix C**.

For such an implementation, we establish the invariant that for any `disney_queue`, the fast-pass customers are at the front of the list, while the regular customers at the back of the list. Additionally, the relative ordering of fast-pass and regular customers should be maintained.

(a) Now implement the `enq` function yourself. Recall that the `enq` function seeks to add the item `v` to the queue while maintaining invariants and relative ordering.

*Note: for full credit, your solution **must** leverage the invariants and short-circuit (terminate earlier than the end of the list) if possible. Solutions that do not will receive partial credit.*

```
let rec enq (q: 'a disney_queue) (v: 'a) (fast: bool) : 'a disney_queue =
```