Name:

PennKey (penn login id):

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

- There are 100 total points. The exam length is intended to be 50 minutes. The exam is asynchronous and you have 24 hours to complete the exam. You can take as much or as little time as you need as long as you submit by the deadline. We encourage you to block out a one-hour period of time without distractions in which to complete the exam.
- This exam is open notes, open computer, and open internet.
- Do not collaborate with anyone else when completing this exam.
- Submit your exam via Gradescope (see link on Piazza) before the deadline.
- This is an editable PDF document. Please fill out the entries as indicated. (Use an "X" for the checkbox answers.)
- Different PDF viewers provide different editing capabilities. On OSX, Preview is a good choice. On Windows, Adobe Reader or Adobe Acrobat should work. We recommend *not* using a web-browser embedded PDF viewer (e.g., in Chrome or Safari) since refreshing the browser tab will lose your work. In all cases, save your work periodically!
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc.
- The code in the exam is available on Codio, and you're welcome to use that rather than trying to type in this code yourself. *However, using Codio for the exam is completely optional and you can do well in the exam even if you decide not to use it.*
- If you have a question or need clarification, please put a private (optionally anonymous) post on Piazza.
- There are 11 pages in the exam.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- There is a separate Appendix for your reference. Please do not submit the Appendix.
- Good luck!

1. Types (16 points)

For each OCaml expressions below, fill in the blank so that the types are consistent for each line. If type says "ill typed", complete the type so that there is a type error on that line.

Some of these expressions refer to the functions transform and fold, to the constructors of the type 'a tree, or to the ListQ module that implements the Q interface. These are all defined in the Appendix. Note that all of the code appears after the module ListQ has been opened.

For most of these questions, there are multiple correct answers and you only need to provide one. Leaving the box blank will not count as a correct answer.

We have done the first one (z) for you. (2 points each)

```
;; open ListQ
let z : int list =
 1 :: 2 :: [3; 4]
let a : (string list * bool list) list = (* note the brackets below... *)
                                                       ]
 [
let b : ill-typed =
                                          (* z is defined above *)
 begin match z with
 | [] -> true
 | _ ->
 end
let c : ('a -> int) list =
                                          (* note the brackets below... *)
 ſ
                                                      1
let d : ill-typed =
 fun x -> x
let e : int -> int tree =
let f : int list queue =
 enq
let g : 'a queue list -> bool list =
 transform
let h : 'a list -> 'a queue =
 fold
```

2. Binary Trees (14 points)

Consider a generic binary tree 'a tree, which is defined as shown in Appendix B. We can define a *balanced* binary tree as a binary tree with the following additional invariants:

- *Empty* is a balanced binary tree
- *Node(lt, x, rt)* is a balanced binary tree if
 - the difference in heights between 1t and rt is at most 1
 - It and rt are both themselves balanced binary trees

The trees below show some examples of balanced and unbalanced binary trees, where, as usual, Empty constructors are not shown, to avoid clutter.

```
Balanced Binary Tree: Balanced Binary Tree:
t1:int tree = 7
                                      t2 : int tree =
                                                               7
                       / \setminus
                                                              / \setminus
                     4 9
                                                            4 9
                                                              \backslash
                                                               8
                               Unbalanced Binary Tree:
7 t4 : int tree = 7
Unbalanced Binary Tree:
t3 : int tree =
                                                               / \setminus
                                \backslash
                                                             8
                                 9
                                                                   9
                                  \setminus
                                                            /
                                                                    \backslash
                                                           5
                                                                     6
                                    8
                                                          /
                                                         4
                                                                       1
```

(Nothing to do on this page.)

Write a function that given a 'a tree returns whether it is a balanced binary tree or not.

(* Hint 1: Do you need a helper function? If so, you can write that below. *)

(* Hint 2: You can use the **abs** : **int** -> **int** function to find the absolute value of an integer. *)

let rec is_balanced (t : 'a tree) : bool =

3. List Processing and Higher-Order Functions (28 points total)

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

For these problems *do not* use any list library functions other than @ (list append). Constructors, such as :: and [], are fine.

(a) (6 points) Use transform or fold, along with suitable anonymous function(s), to implement a function reverse_transform that takes in a higher-order function of the type f: 'a -> 'b and a 'a list and transforms the list into a 'b list, but one that is in the reverse order of the original list. For example, the call reverse_transform (fun x -> x * x) [1; 2; 3; 4] evaluates to the list [16; 9; 4; 1].

let reverse_transform (f: 'a -> 'b) (l: 'a list) : 'b list =

(b) (6 points) Use transform or fold, along with suitable anonymous function(s), to implement a function split that takes in a ('a * 'b) list and returns a tuple of lists 'a list * 'b list. For example, the call split [(1, ``a''); (2, ``b''); (3, ``c''); (4, ``d'')] evaluates to ([1; 2; 3; 4], [``a''; ``b''; ``c''; ``d'']).

let split (p: ('a * 'b) list) : 'a list * 'b list =

(c) (6 points) Use transform or fold, along with suitable anonymous function(s), to implement a function find_first that takes in a predicate function of the type int -> bool and an int list and returns the first element of the list that satisfies the predicate and returns 0 if none of the elements in the list satisfy the predicate. For example, the call find_first (fun x -> x > 2) [1; 10; 2; 3; 4; 5] evaluates to 10.

let find_first (pred: int -> bool) (l: int list) : int =

(d) (10 points) Consider a variant of fold called fold2 that works on two lists and has the types as shown below. It combines each element of the two lists pair-wise, i.e., it combines the heads of each list with the result of processing their tails. Assume that the two lists have the same length (use failwith when they aren't).

First, implement fold2 using either transform or fold or pattern matching.

Next, use fold2, along with suitable anonymous function(s), to implement a function greater_values that takes in two 'a lists (assume they are the same length) and returns a 'a list such that each element in the output list is the greater of the two elements (or their value if they're equal) in the corresponding input lists. For example, the call greater_values [1; 5; 3; 7] [2; 2; 4; 9] evaluates to [2; 5; 4; 9].

```
let greater_values (l1: 'a list) (l2: 'a list) : 'a list =
```

4. Modules and Abstract Types (42 points total)

There is nothing to do on this page except read the problem statement.

In this problem we will implement an abstract type of *queues*, which are another collection datatype, similar to the 'a set from Homework 3 and to OCaml's built-in 'a list.

Step 1: Understand the Problem A 'a queue, like a list, behaves as an *ordered* sequence of values of type 'a. Unlike a list, a queue provides the "first-in, first-out" behavior that is familiar to you from waiting in line at a movie theater or for CIS 120 office hours. Its two main operations allow us to *enqueue* an element, which corresponds to adding it to the end of the line, and to *dequeue* an element, which corresponds to retrieving the element at the front of the line and removing it from the queue (this operation might fail if the queue is empty). In both cases, these operations return the updated queue value.

Step 2: Design the Interface The signature below defines an abstract type ' a queue and operations on it. The value empty is a queue with no elements in it.

```
module type Q = sig
type 'a queue
val empty : 'a queue
val is_empty : 'a queue -> bool
val enq : 'a queue -> 'a -> 'a queue
val deq : 'a queue -> ('a queue * 'a) (* fails if queue is empty *)
end
```

Step 3: Define Test Cases The main property defining a queue is its "first-in, first-out" behavior. The test case below demonstrates the desired behavior when the string "x" is enqueued into the empty queue and then dequeued.¹

```
let test () =
    let q0 = enq empty "x" in
    let (q1, a) = deq q0 in
    a = "x" && is_empty q1
;; run_test "deq enq" test
```

¹Recall that the notation let (x, y) = p in ..., where p is a pair, names p's first element x and its second element y.

The following test cases are missing several values, indicated by ??.

a. (4 points) What values should replace the ??'s so that this is a correct test?

let test () = let q0 = enq empty "x" in let q1 = enq q0 "y" in let q2 = enq q1 "z" in **let** (q3, a) = deq q2 **in** let (q4, b) = deq q3 in **let** (q5, c) = deq q4 **in** a = ?? && b = ?? && c = ?? && is_empty ?? In each case, **choose one**. Mark the box with "X". • For a = ??, the missing value ?? must be: "v" "×" " " q3 q4 q5 • For b = ??, the missing value ?? must be: "x" "v" "z" q5 q3 q4 • For c = ??, the missing value ?? must be: "x" "v" "z" q3 q4 q5 • For is_empty ??, the missing value ?? could be: "v" "z" "x" q3 q4 q5

b. (4 points) What values should replace the ??'s so that this is a correct test?

1 **let** test () = 2 **let** q0 = enq empty ?? in 3 let q1 = enq ?? "b" in 4 **let** (q2, a) = deq ?? **in** 5 **let** q3 = enq q2 ?? in 6 let (q4, b) = deq q3 in a = "a" && b = "b" && not (is empty q4)7

In each case, **choose one**. Mark the box with "X".

"a"

a0

"a"

q0

• For the code on line 2 the missing value ?? must be:

"b" any value of type string α2

q1

• For the code on line 3, the missing value ?? must be:

"b" any value of type string q1 q2

• For the code on line 4, the missing value ?? must be:

"a" "b" any value of type string a0 q1 q2

• For the code on line 5, the missing value ?? must be:

"a" "b" any value of type string a0 q1 α2

Step 4: Implement the Code (12 points)

Recall that we implement an abstract type by choosing a suitable representation type, typically including representation invariants—e.g., for a 'a set implementation we could use sorted lists or binary search trees. There are several possible ways to implement the Q interface.

For your reference, Appendix C gives one complete implementation of 'a queue using 'a list as the representation type and maintaining the invariant that the element to be dequeued is at the head of the list.

Complete the following implementation of ListRevQ, a second version of list-based queues that maintains the invariant that **the element (if any) to dequeue is the** *last* **element of the list**. You need to add only code for the enq and deq operations.

```
module ListRevQ : Q = struct
  (* INVARIANT: queue elements are stored in the reverse order
      in which they will be dequeued: the last element of the list
      (if any) will be the element returned by deq.
   *)
  type 'a queue = 'a list
  let empty : 'a queue = []
  let is_empty (q : 'a queue) : bool =
    q = []
  (* (a) Complete this function. *)
  let rec enq (q : 'a queue) (x : 'a) : 'a queue =
```

(* (b) Complete this function. (which fails if the queue is empty) *)
let rec deq (q : 'a queue) : 'a queue * 'a =

end (* of the ListRevQ struct *)

Step 5: Revisit / Refactor

a. (6 points) The interface explained in Step 2 does not directly provide an interface function to calculate the number of elements in a queue. A *client* of any Q module (including ListQ) can implement such a function itself. Complete the function below, which returns the number of elements in the given queue. This function should never fail, and it should respect the Q interface. Note that this code *is not* part of the ListQ module.

open ListQ

let rec length (q : 'a queue) : int =

b. (16 points) Calculating the queue's length is possible using this interface (as demonstrated above), but inefficient for long queues—the code has to traverse the whole queue to count the elements. A more efficient way to provide this behavior would be to add the length operation to the Q interface so that the implementation of the queue can take advantage of representation invariants that speed up the length calculation.

One very straightforward way to achieve that is to change the 'a queue representation and representation invariant such that a queue is implemented by a pair of a list of elements and its length.

On the following page, complete the implementation of a new version of Q that provides the length function as part of the implementation. Your code should follow the invariant described there and exploit it to give an efficient implementation of length. Note that we have given you some of the code from ListQ (in Appendix C) as helper functions that you should use as appropriate.

```
module ListLenQ = struct
  (* INVARIANT: (n, l) is a 'a queue when
    - l : 'a list
    contains the queue elements the order in which they will be
    dequeued: the head of the list (if any) will be the element
    returned by deq.
    - n : int is the length of 1 *)
  type 'a queue = (int * 'a list)
 let empty : 'a queue =
 let is_empty (q : 'a queue) : bool =
  let rec enq_helper (l : 'a list) (x : 'a) : 'a list =
   begin match 1 with
   | [] -> [x]
    y::ys -> y::(enq_helper ys x)
    end
  let enq (q : 'a queue) (x : 'a) : 'a queue =
  let deq_helper (l : 'a list) : 'a list * 'a =
   begin match 1 with
   | [] -> failwith "empty queue"
   | x::xs -> (xs, x)
    end
  let deq (q : 'a queue) : 'a queue * 'a =
  let length (q : 'a queue) : int =
```

```
end (* of the ListLenQ struct *)
```