CIS 120 Midterm II November 8, 2019

Name (printed): ____________________________

Pennkey (login id): ____________________________

My signature below certifies 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.
- There are 100 total points. The exam lasts 50 minutes. There are 8 pages in the exam.
- Make sure your name and Pennkey (e.g., `swapneel`) are on the bottom of *every* page.
- There is a separate Appendix for your reference. Answers written in the Appendix will not be graded.
- Good luck!

1. **Mutable Queues (abstract types, invariants, mutable state, ASM)** (30 points total)

Appendix A contains the implementation of the singly-linked queue data structure from Homework 4, including the `create`, `enq`, and `deq` operations. It also summarizes the (singly-linked) queue invariants. Section A.1 depicts four queue heap structures, labeled (a) through (d).

(a) (10 points) Match each of the following code snippets to the heap structures that `q` will refer to after running the code, or choose "error" if the code leads to a static (e.g. typechecking) or dynamic error. The possible heap structures are the ones pictured in the appendix.

i.
```
let q : int queue = create ()
;; enq 1 q
```

a ☐ b ☐ c ☐ d ☐ error ☐

ii.
```
let q : int queue = create ()
;; enq 1 q
;; enq 2 q
;; begin match q.tail with
   | None -> ()
   | Some n -> n.next <- q.head
   end
```

a ☐ b ☐ c ☐ d ☐ error ☐

iii.
```
let qno : int qnode option = Some { v = 1; next = None }
let q = { head = qno; tail = qno }
```

a ☐ b ☐ c ☐ d ☐ error ☐

iv.
```
let q : int queue = create ()
;; enq 1 q
;; begin match q.tail with
   | None -> ()
   | Some n -> n.next <- Some n
   end
```

a ☐ b ☐ c ☐ d ☐ error ☐

v.
```
let q : int queue = create ()
;; enq 2 q
;; enq 1 q
;; deq q
```

a ☐ b ☐ c ☐ d ☐ error ☐

(b) (4 points) Which of the depicted heap values satisfy the queue invariants? (Mark all that do.)

a ☐ b ☐ c ☐ d ☐

(c) (16 points) Complete the following code that moves the last node of a queue (i.e. the one pointed to by the tail) to the head of the queue. Your implementation should reuse the existing nodes of the queue, so that no additional nodes need to be allocated. If the queue is empty or contains exactly one node, then nothing needs to be done; we have finished those cases for you. The resulting modified heap structures should satisfy the queue invariants.

The skeleton code we provide includes a helper function `loop`, which is intended to iterate through the queue to find the next-to-last and last nodes. Be sure to fill in its return type, and use `loop` as appropriate in the main function body.

```
let move_tail_to_head (q : 'a queue) : unit =

  let rec loop (prev : 'a qnode) (curr : 'a qnode) : ____________ =
    begin match curr.next with
    | None ->








    | Some n ->


    end
  in

  begin match q.head with
    | None -> () (* q is empty *)
    | Some n ->
      begin match n.next with
        | None -> () (* q is singleton *)
        | Some m ->










      end
  end
```

2. **OCaml Programming: Mutable State, Closures, Tail Calls** (21 points total)

(a) (4 points) Recall the definition of OCaml's generic reference type:

```
type 'a ref = {mutable contents : 'a}
```

For each of the following test cases, fill in the blank *with an integer constant* so that the test case will pass.

i.
```
let test () =
  let x : int ref = {contents = 3} in

  let y : int ref = {contents = ____________} in
  (x.contents + 1) = y.contents
;; run_test "test1" test
```

ii.
```
let test () =
  let x : int ref = {contents = 3} in
  let y : int ref = x in
    y.contents <- 17;

    x.contents = ____________
;; run_test "test2" test
```

(b) Consider the following program:

```
let make_find (elt : 'a) : ('a list -> bool) =
  let rec loop (lst : 'a list) : bool =
    begin match lst with
      | [] -> false
      | x::xs -> if x = elt then true else loop xs
    end
  in fun (lst2 : 'a list) -> loop lst2

;; let find3 = make_find 3
```

i. (2 points) True or False: The function `loop` is tail recursive.

☐ True ☐ False

ii. (3 points) Which of the following identifiers' values will be saved as bindings in the closure named by `find3`? (Mark all that apply.)

☐ `make_find`
☐ `elt`
☐ `loop`
☐ `lst`
☐ `lst2`
☐ `find3`

(c) (12 points) In our GUI library for Paint, we defined a `notifier` that was a container widget and kept a list of event listeners to notify when an event happened. We want to augment this to additionally keep a count of the number of events it has handled so far and a function that returns this count.

You need to update the `notifier` code shown below. Only four additional lines of code are needed to support this new functionality. What lines need to be added and where? (You can assume that the code below compiles and works correctly.)

```
1  type notifier_controller = {
2    add_event_listener: event_listener -> unit;
3  }
4
5  let notifier (w: widget) : widget * notifier_controller =
6    let listeners = { contents = [] } in
7    { repaint = w.repaint;
8      handle =
9        (fun (g: Gctx.gctx) (e: Gctx.event) ->
10          List.iter (fun h -> h g e) listeners.contents;
11          w.handle g e);
12     size = w.size
13   },
14   { add_event_listener =
15       (fun (newl: event_listener) ->
16         listeners.contents <- newl :: listeners.contents);
17   }
```

i. Changes need to keep a count of the number of events handled so far

Add Line after Line Number: ___________
New line to be added:

Add Line after Line Number: ___________
New line to be added:

ii. Changes need to get the count of the number of events

Add Line after Line Number: ___________
New line to be added:

Add Line after Line Number: ___________
New line to be added:

3. **Objects in OCaml and Java** (25 points total)

(a) (9 points) Given the OCaml code below, answer the following multiple choice questions. (Each has one correct answer.)

```
1  type 'a ref = {mutable contents: 'a}
2
3  type counter = {
4    get : unit -> int;
5    set : int -> unit;
6    decr : unit -> int;
7  }
8
9  let mk_counter () : counter =
10   let cnt : int ref = {contents = 0} in
11   let ctr : counter = {
12     get = (fun () -> cnt.contents);
13     set = (fun x -> cnt.contents <- x);
14     decr = (fun () -> cnt.contents <- cnt.contents - 1;
15                 cnt.contents);
16   }
17   in ctr
18
19 let ctr1 = mk_counter ()
```

i. The closest analog in Java to the code on lines 3–7 would define:

☐ a constructor

☐ a class

☐ an interface

☐ a set of fields

ii. The closest analog in Java to the code on line 10 would correspond to a field declared as:

☐ `public static int cnt;`

☐ `private static int cnt;`

☐ `public int cnt;`

☐ `private int cnt;`

iii. The closest analog in Java to the code on line 19 would correspond to:

☐ creating an instance by invoking a constructor

☐ invoking a static method

☐ invoking a method via dynamic dispatch

☐ creating an instance via inheritance

(b) (6 points) Given the Java code below, which of the following statements are true? (Mark all that apply.)

```
1   A a = new B();
2   C c = a.m1();
3   a.____();
```

- ☐ For line 1 to compile successfully, `B` has to be a supertype of `A`.
- ☐ For line 1 to compile successfully, `B` cannot be an interface type.
- ☐ For line 2 to compile successfully, `C` has to be a supertype of the return type of `m1()`.
- ☐ For line 2 to compile successfully, `C` cannot be an interface type.
- ☐ For lines 1 and 2 to compile successfully, the method `m1()` has to be available via type `A`.
- ☐ For line 3 to compile successfully, the blank can be filled by methods defined only in type `B`.

(c) (10 points) Indicate whether the following statements are true or false.

a. True ☐ False ☐
In OCaml, a single interface can define more than one abstract type whose implementations might be related.

b. True ☐ False ☐
In Java, a class may implement more than one type.

c. True ☐ False ☐
In OCaml, if we have a record with three fields, it is possible for one of the fields to be `null`.

d. True ☐ False ☐
In Java, it is possible for a class to have no public methods.

e. True ☐ False ☐
In Java, it is possible for a class to extend two classes, neither of which is a subtype of the other.

4. **Java Array Programming** (24 points)

In Java, a two dimensional array can be ragged, which means that it is not "rectangular" in shape. More precisely, a ragged 2D array `a` has an index `i` such that `a[0].length` is not equal to `a[i].length`.

Write a function `pad`, that takes a potentially ragged 2D array of integers and returns a "padded" copy `p`, which is the smallest rectangular array such that if `a[i][j]` is defined (i.e., it doesn't lead to an `ArrayIndexOutOfBoundsException`), then `p[i][j] = a[i][j]` and otherwise, `p[i][j] = 0`.

Pictorially, if `a` is as shown below, then `pad(a)` will be the same as `a` but with 0's filling out the rectangle:

```
    a                   pad(a)
0 1 2 3 0             0 1 2 3 0
4 5                   4 5 0 0 0
6 7 8                 6 7 8 0 0
9                     9 0 0 0 0
```

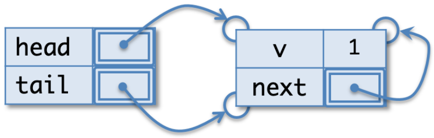
You may assume that the input array `a` is not null and that it contains no null sub-arrays. Note that `a[i]` refers to the row `i` in `a`.

```
public int[][] pad(int[][] a) {








}
```

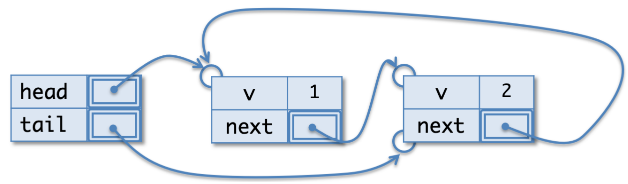
A Appendix: Queue Implementation

```
type 'a qnode = { v: 'a;
                  mutable next: 'a qnode option }

type 'a queue = { mutable head: 'a qnode option;
                  mutable tail: 'a qnode option }

(* INVARIANT:
 - q.head and q.tail are either both None, or
 - q.head and q.tail both point to Some nodes, and
   - q.tail is reachable by following 'next' pointers
     from q.head
   - q.tail's next pointer is None
*)

let create () : 'a queue =
  { head = None; tail = None }

(* Add an element to the tail of a queue *)
let enq (elt: 'a) (q: 'a queue) : unit =
  let newnode = { v = elt; next = None } in
  begin match q.tail with
  | None ->
     (* Note that the invariant tells us that q.head is also None *)
     q.head <- Some newnode;
     q.tail <- Some newnode
  | Some n ->
     n.next <- Some newnode;
     q.tail <- Some newnode
  end

(* Remove an element from the head of the queue *)
let deq (q: 'a queue) : 'a =
  begin match q.head with
  | None ->
     failwith "deq called on empty queue"
  | Some n ->
     q.head <- n.next;
     if n.next = None then q.tail <- None;
     n.v
  end
```

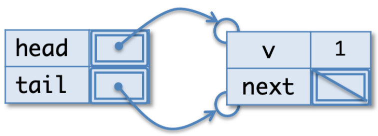
A.1 Example Queue Values



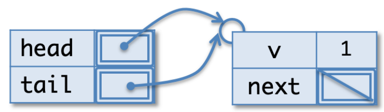
(a)



(b)



(c)



(d)