# CIS 120 Midterm II

March 25, 2015

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: ________________________

| | |
|---|---|
| 1 | /22 |
| 2 | /15 |
| 3 | /22 |
| 4 | /23 |
| 5 | /18 |
| Total | /100 |

- Do not begin the exam until you are told to do so.
- You have 50 minutes to complete the exam.
- There are 100 total points.
- There are 9 pages in this exam, plus 2 pages of Appendix.
- Make sure your name and Pennkey (a.k.a. username) is on the top of this page.
- Be sure to allow enough time for all the problems—skim the entire exam first to get a sense of what there is to do.

**1. OCaml and Java: True/False** (22 points)

Circle T or F.

**a.** T F Variables stored on the stack in the OCaml Abstract Stack Machine are not mutable, although they may refer to mutable records in the heap.

**b.** T F All records are mutable in OCaml.

**c.** T F A well-typed OCaml program can fail with an error because it tried to use a `null` pointer.

**d.** T F In both Java and OCaml, if `v1` and `v2` are references, then the expression `v1 == v2` evaluates to true exactly when they refer to the same location in the heap.

**e.** T F In OCaml, if `s` and `t` are variables of type `string` such that `s == t`, then `s = t` is guaranteed to return **`true`**.

**f.** T F In Java, if `s` and `t` are variables of type `String` such that `s.equals(t)`, then `s == t` is guaranteed to return **`true`**.

**g.** T F The following OCaml function will terminate by exhausting stack space if called as `loop 10`.

```
let rec loop (x:int) : int list =
  if x = 0 then [] else x :: loop x
```

**h.** T F There is a type that can be filled in for the blank in the following OCaml program to make it well typed.

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

let x : ____________________ = { contents = [] }
;; x.contents <- ((fun () -> ()) :: x.contents)
```

**i.** T F In our GUI library, an `event_listener` is a first-class function stored in the hidden state of a `notifier` widget. When an event occurs in the widget, the `notifier` invokes all of the registered `event_listeners`.

**j.** T F Applications built using our OCaml GUI library should put all code for drawing in event listeners.

**k.** T F In the OCaml ASM, first-class functions are stored in the heap and may have local copies of variables that were on the stack when they were defined.

**2. Higher-order functions: fold** (15 points)

The following functions below are written without using `fold`. Circle the correct redefinition in terms of this higher-order function.

```
let rec fold (combine: 'a -> 'b -> 'b) (base : 'b) (x: 'a list) : 'b =
  begin match x with
  | [] -> base
  | h :: t -> combine h (fold combine base t)
  end
```

**a.**
```
let rec reverse (x : int list) : int list =
  begin match x with
  | [] -> []
  | h :: t -> reverse t @ [h]
  end
```

How would you rewrite this function using `fold`?

**i.**
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> h :: y) 0 x
```

**ii.**
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> y @ [h]) 0 x
```

**iii.**
```
let reverse (x : int list) : int list =
      fold (fun (h:int) (y:int list) -> y @ [h]) [] x
```

**iv.** `reverse` can't be written with fold.

**b.**
```
let rec g (x: 'a) (m: 'a list) : bool =
  begin match m with
  | [] -> false
  | hd :: tl -> x == hd || g x tl
  end
```

How would you rewrite this function using fold?

**i.**
```
let g (x:'a) (m: 'a list) : bool =
    fold (fun (y:'a) (z:bool) -> (x == y) || z) false m
```

**ii.**
```
let g (x:'a) (m: 'a list) : bool =
    fold (fun (y:'a) (z:bool) -> (x == hd) || g x tl) false m
```

**iii.**
```
let g (x:'a) (m: 'a list) : bool =
    fold (fun (y:'a) (z:bool) -> (x == z)) false m
```

**iv.** `g` can't be written with fold.

**c.**
```
let rec f (x : 'a option list) : 'a list =
  begin match x with
  | [] -> []
  | None :: t -> f t
  | Some y :: t -> y :: f t
  end
```

How would you rewrite this function using fold?

**i.**
```
let f (x : 'a option list) : 'a list =
  fold (fun (h : 'a) (r : 'a list) -> h :: r) [] x
```

**ii.**
```
let f (x : 'a option list) : 'a list =
  fold (fun (h : 'a option) (r : 'a list) ->
    begin match h with
    | None -> r
    | Some y -> y :: r
    end) [] x
```

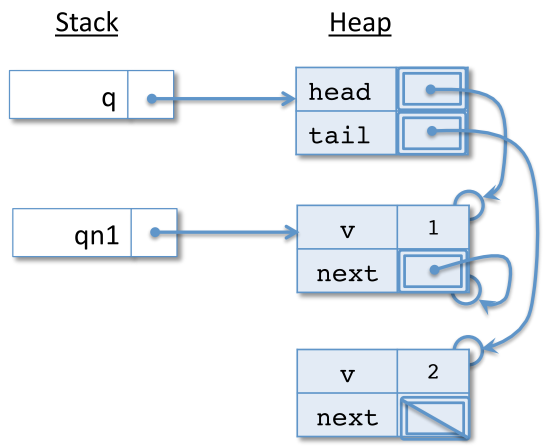
**iii.**
```
let f (x : 'a option list) : 'a list =
  fold (fun (h : 'a option) (r : 'a list) ->
    begin match r with
    | [] -> None
    | y :: tl -> Some y
    end) None x
```

**iv.** `f` can't be written with fold.

3. **OCaml Abstract Stack Machine** (22 points)

Recall the mutable queue implementation from class and homework and shown in Appendix A.

Suppose that we have the OCaml ASM shown below.



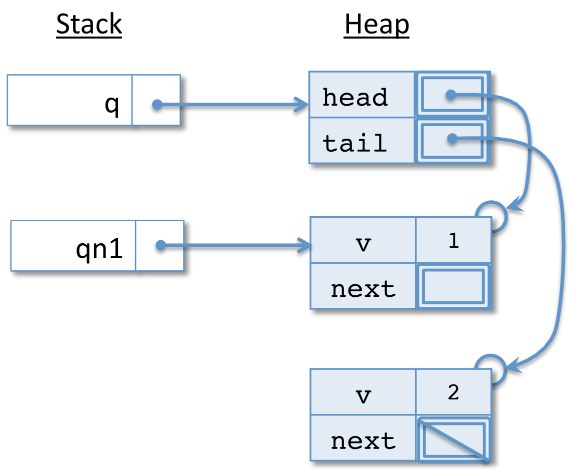

**a.** Does `q` satisfy the queue invariant? Circle **true** or **false** .

**b.** Write a short piece of code that can be loaded onto the workspace to get the ASM to this configuration. Feel free to call any of the functions in Appendix A, if it helps. However, note that the stack and heap after your code executes must look *exactly* as in the drawing: it must put variables on the stack in the correct order and it cannot include any extra stack variables or qnodes in the heap.

**c.** Suppose we start executing from the ASM configuration shown on the previous page. Modify the template stack and heap diagram below to show what it will look like after the following code executes on the workspace.

```
let qn0 = { v = 0; next = q.tail } in
qn1.next <- Some qn0
```

Note that you may need to allocate new stack variables, heap nodes or add `Some` bubbles in appropriate places. If you need to erase a line, mark it clearly with an "X".




**d.** Does `q` satisfy the queue invariant after the code in part (c) has executed? Circle **true** or **false**.

## 4. Mutable Queues Implementation (23 points)

Implement a function, called `intersperse`, that inserts a value *between* every value in a queue.

For example, if `q` contains the values `1,2,3` (in that order) then, after an execution of `intersperse 0 q`, the queue `q` should contain the values `1,0,2,0,3` (in that order). On the other hand, if `q` is empty or contains a single element, then a call to `intersperse` should not modify the queue. All calls to `intersperse` should leave `q` in a valid state.

Your implementation may define a single recursive helper function to traverse the queue. However it may not call any other functions, such as `from_list`, `to_list`, `deq` or `enq`.

```
let intersperse (x : 'a) (q : 'a queue) : unit =
```

5. **Listeners** (18 points)

Recall the `value_controller` type from Homework 6.

```
type 'a value_controller = {
  get_value : unit -> 'a;
  set_value : 'a -> unit;
  add_change_listener : ('a -> unit) -> unit;
}
```

A value controller is an "object" that encapsulates a particular value. The methods of this object allow applications to access the value, change the value, and register change listeners. The listeners are called with the new value whenever the value is changed.

For example, if `vc` is an `int value_controller`, then the following code

```
vc.add_change_listener (fun (v:int) ->
       print_endline ("Value changed to: " ^ (string_of_int v)));
vc.set_value 10;
vc.set_value 20
```

prints the following text on the console when executed

```
Value changed to: 10
Value changed to: 20
```

On the next page, implement `make_value_controller`. This function should return a new value controller when given an initial value.

You are allowed to use the following helper function in your answer, but no other library functions.

```
let rec iter (f : 'a -> unit) (x : 'a list) : unit =
  begin match x with
  | [] -> ()
  | hd :: tl -> f hd ; iter f tl
  end
```

```
let make_value_controller (v : 'a) : 'a value_controller =
```

## Appendix A: OCaml Linked 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;
}

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

let is_empty (q:'a queue) : bool =
  q.head = None

let enq (x:'a) (q:'a queue) : unit =
  let newnode_opt = Some { v = x; next = None} in
  begin match q.tail with
    | None -> q.head <- newnode_opt;
      q.tail <- newnode_opt
    | Some qn2 ->
      qn2.next <- newnode_opt;
      q.tail <- newnode_opt
  end

let deq (q:'a queue) : 'a =
  begin match q.head with
    | None -> failwith "error: empty queue"
    | Some qn ->
      begin match q.tail with
        | Some qn2 ->
          if qn == qn2 then
            (* deq from 1-element queue *)
            (q.head <- None;
             q.tail <- None;
             qn2.v)
          else
            (q.head <- qn.next;
             qn.v) (* Make sure to use parens around ; expressions. *)
        | None -> failwith "invariant violation"
      end
  end


let to_list (q : 'a queue) : 'a list =
  let rec loop (qn : 'a qnode option) (acc : 'a list) : 'a list =
    begin match qn with
      | None -> List.rev acc
      | Some qn1 -> loop qn1.next (qn1.v :: acc)
    end in
  loop q.head []
```

## Appendix B: Example Abstract Stack Machine Diagram

An example of the Stack and Heap components of the OCaml Abstract Stack Machine. Your diagram should use similar "graphical notation" for `Some v` and `None` values.

```
(* The types for mutable queues. *)
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }

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

let qn1 : int qnode = {v = 1; next = None}
let qn2 : int qnode = {v = 2; next = Some qn1}
let q : int queue = {head = Some qn2; tail = Some qn1}
(* HERE *)
```

The OCaml program above yields the ASM Stack and Heap depicted below when the program execution reaches the point marked *(* HERE *)*.