CIS 1200 Midterm II   November 15, 2024

Benjamin C. Pierce and Swapneel Sheth, instructors

SOLUTIONS

1. **Tail Recursion** (18 points total)

   Which of the following functions are tail recursive?

1.1 (2 points)

```
let rec mystery (l: int list) (count: int) : int =
  begin match l with
    | [] -> count
    | hd::tl -> mystery tl (count + 1)
  end
```

[✓ Tail Recursive]   ☐ Not Tail Recursive

1.2 (2 points)

```
let rec mystery (l: bool list) : bool =
  begin match l with
    | [] -> false
    | hd::tl -> not (mystery tl)
  end
```

☐ Tail Recursive   [✓ Not Tail Recursive]

1.3 (2 points)

```
let rec mystery (l: bool list) : bool =
  begin match l with
    | [] -> true
    | hd::tl -> hd && mystery tl
  end
```

[✓ Tail Recursive]   ☐ Not Tail Recursive

1.4 (2 points)

```
let rec mystery (l: 'a list) (f: 'a -> 'a) : 'a =
  begin match l with
    | [] -> failwith "error"
    | hd::[] -> f hd
    | hd::tl -> f (mystery tl f)
  end
```

☐ Tail Recursive   [✓ Not Tail Recursive]

$\boxed{1.5}$ (10 points)

This function is *not* tail recursive:

```
let rec mystery (f : int -> int) (x : int) : int =
  if x = 0 then 0
  else (f x) + (mystery f (x - 1))
```

In the space below, define a tail-recursive function that behaves the same as `mystery` on all inputs (except that it might loop where the version above would overflow the stack).

```
let rec mystery (f : int -> int) (x : int) : int =
  let rec loop (y : int) (acc : int) : int =
    if y = 0 then acc
    else loop (x - 1) ((f y) + acc)
  in loop x 0
```

2. **Deques** (29 points total)

Recall the type definitions for the `deque` data structure in Homework 4.

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

type 'a deque = {
  mutable head: 'a dqnode option;
  mutable tail: 'a dqnode option;
```

The *deque invariant* is as follows:

(1) `head` and `tail` are both tagged `None` or both tagged `Some`, and

(2) if `head = Some h` and `tail = Some t`, then

- (2a) `t` is reachable from `h` by following `next` pointers
- (2b) `t.next = None`
- (2c) `h` is reachable from `t` by following `prev` pointers
- (2d) `h.prev = None`

and, for every node `n` in the queue,

- (2e) if `n.next = Some m`, then `m.prev = Some n`
- (2f) if `n.prev = Some m`, then `m.next = Some n`.

Your job in this problem will be to help define a function `split_deque` that, given a deque `d` and a `'a` element `x`, splits `d` into two deques at the first occurrence of `x`, putting `x` at the head of the second deque.

For instance, if `d` is a deque with three elements such that `to_list d = [1; 2; 3]`, then `split_deque d 3` will return deques `(d1, d2)` such that `to_list d1 = [1; 2]` and `to_list d2 = [3]`.

If `x` is not equal to any element of `d`, then `d1` should be the same as `d` and `d2` should be empty. If `d` is empty, then two empty deques should be returned.

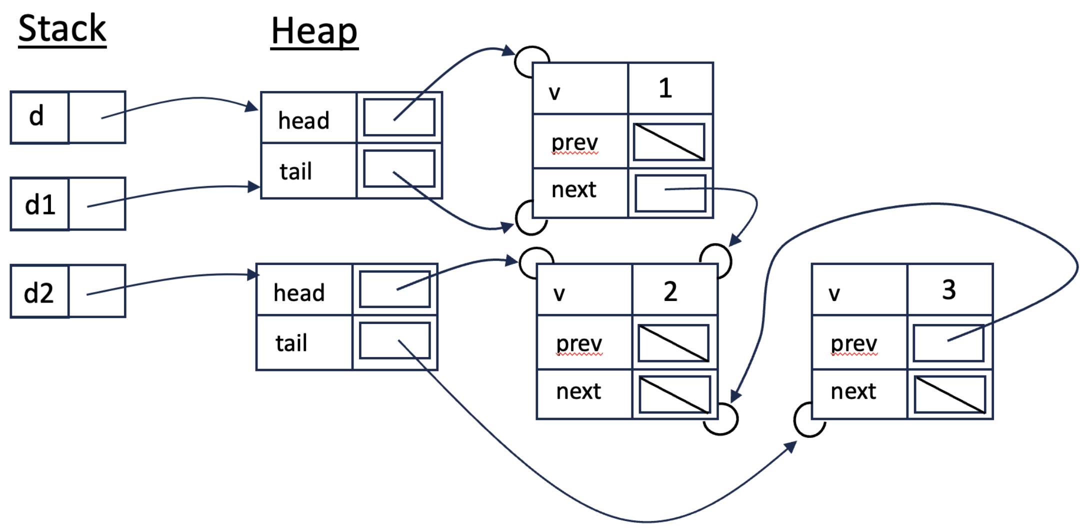
Note that the `split_deque` operation does not allocate any new `dqnode`s—it reuses the ones from the original deque `d`.

Nothing for you to answer on this page

$\boxed{2.1}$ (6 points) To begin, let's check our understanding of the deque invariant. In this part of the question (and the next), we'll show you a picture representing the state of the ASM after running a candidate implementation of `split_deque`. The implementation splits the deque in the right place, but the resulting two deques may or may not be valid—i.e., they might not maintain the deque invariants.

The input `d` is a valid deque with `to_list d = [1; 2; 3]` before a call to `(split_deque d 2)`.

Indicate which, if any, of the deque invariants are *broken* in the picture. Check all boxes that apply, or "Valid" if no invariants are broken in `d1` or `d2`.



(1) ☐ (2a) ☒ (2b) ☒ (2c) ☐ (2d) ☐ (2e) ☒ (2f) ☒ Valid ☐

The deque invariant again, for quick reference:

(1) `head` and `tail` are both tagged `None` or both tagged `Some`, and

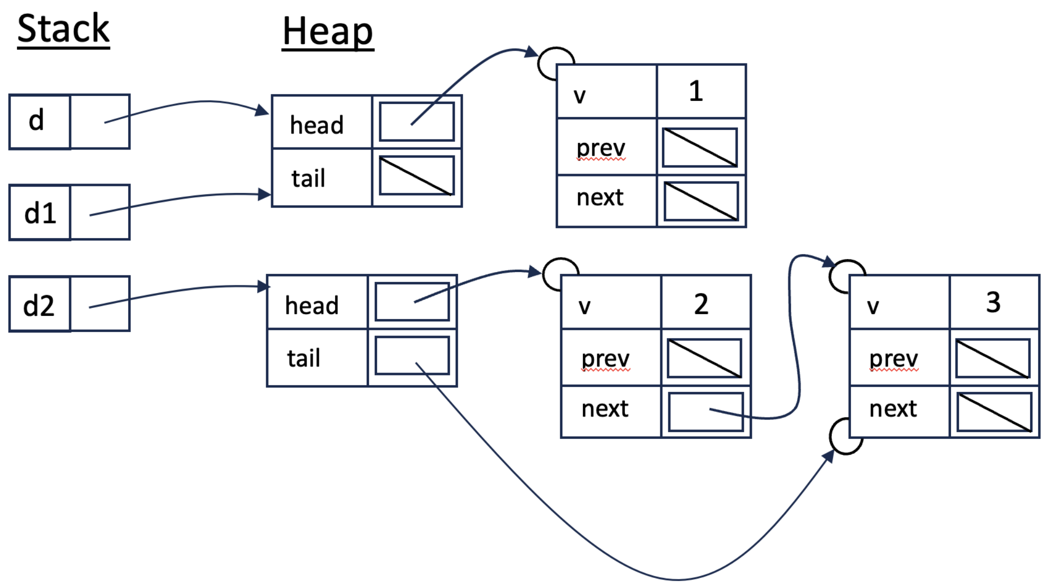
(2) if `head = Some h` and `tail = Some t`, then

- (2a) `t` is reachable from `h` by following `next` pointers
- (2b) `t.next = None`
- (2c) `h` is reachable from `t` by following `prev` pointers
- (2d) `h.prev = None`

and, for every node `n` in the queue,

- (2e) if `n.next = Some m`, then `m.prev = Some n`
- (2f) if `n.prev = Some m`, then `m.next = Some n`.

$\boxed{2.2}$ (6 points) Again, indicate which, if any, of the deque invariants are *broken* in the picture.



(1) ☒ (2a) ☐ (2b) ☐ (2c) ☒ (2d) ☐ (2e) ☒ (2f) ☐ Valid ☐

The deque invariant again, for quick reference:

(1) `head` and `tail` are both tagged `None` or both tagged `Some`, and

(2) if `head = Some h` and `tail = Some t`, then

(2a) `t` is reachable from `h` by following `next` pointers

(2b) `t.next = None`

(2c) `h` is reachable from `t` by following `prev` pointers

(2d) `h.prev = None`

and, for every node `n` in the queue,

(2e) if `n.next = Some m`, then `m.prev = Some n`

(2f) if `n.prev = Some m`, then `m.next = Some n`.

2.3 (17 points) Complete the code below for `split_deque`.

```
let split_deque (d : 'a deque) (x: 'a) : 'a deque * 'a deque =
  let rec loop (dno: 'a dqnode option) : 'a deque * 'a deque =
    begin match dno with
      | None -> (* consider which two cases can lead here... *)
                (d, create ())
      | Some n ->
          if n.v = x then
            (* n becomes head of second deque *)
            begin match n.prev with
              | None ->
                 (* consider which edge case this is... *)
                 (create (), d)
              | Some p ->
                 (* create new deque... *)
                 let d2 = {head = dno; tail = d.tail} in
                 (* consider which pointers you must update... *)
                 d.tail <- n.prev;
                 p.next <- None;
                 n.prev <- None;
                 (* fill in appropriate return value... *)
                 (d, d2)
            end
         else loop n.next
    end
  in loop d.head
```

3. **OCaml ASM** (23 points total)

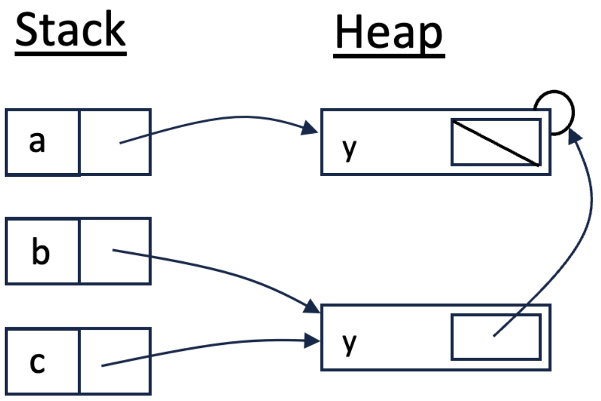
Suppose we've defined the following OCaml type:

```
type t = {mutable y : t option}
```

Then executing the following code...

```
let a : t = {y = None}
let b : t = {y = Some a}
let c : t = b
```

... will yield this state of the OCaml ASM:



3.1 (7 points) Does evaluating the following expressions now yield **true** or **false**? (Recall that == in OCaml is reference equality and = is structural equality.)

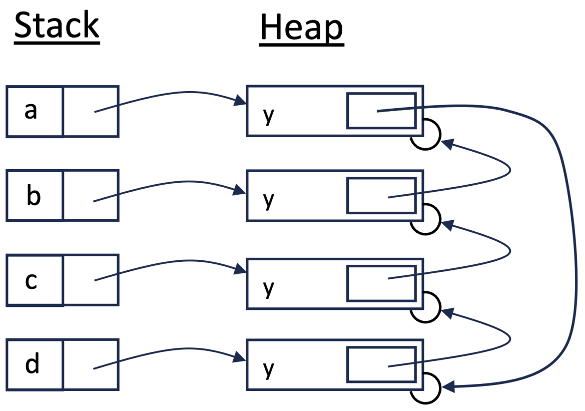
`a == b`	True ☐	False ☒
`a = b`	True ☐	False ☒
`b == c`	True ☒	False ☐
`b = c`	True ☒	False ☐
`c.y == Some a`	True ☐	False ☒
`c.y = Some a`	True ☒	False ☐

`(begin match c.y with`
`| None -> None`
`| Some d -> d.y`
`end) = None`
	True ☒	False ☐

$\boxed{3.2}$ (8 points) Suppose we start from an empty stack and heap and execute the following code.

```
let a : t = {y = None}
let b : t = {y = Some a}
let c : t = {y = Some b}
let d : t = {y = Some c}
;; a.y <- Some d
```

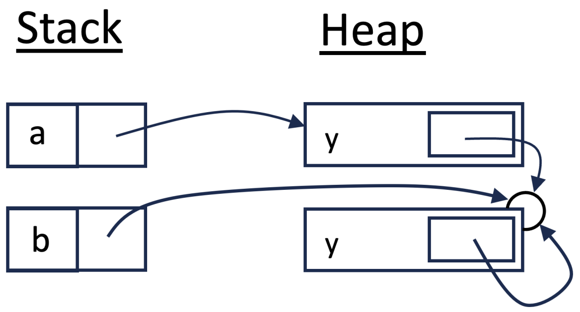
Complete the following ASM diagram to show the state after this code is executed.



$\boxed{3.3}$ (8 points) Suppose, instead, that we start from an empty stack and heap and execute this code.

```
let a : t = {y = Some {y = None}}
;; begin match a.y with
   | None -> ()
   | Some b -> b.y <- a.y
   end
let b : t option = a.y
```

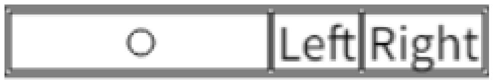
Complete the following ASM diagram.



4. **GUI Programming in OCaml** (30 points total)

Key definitions for our OCaml GUI library can be found on page 1 of the Appendix.

We can use the GUI library to create a rudimentary gaming console where we can control the position of a ball (the circle in the leftmost box below) on a 1D board, using two buttons labeled Left and Right.



$\boxed{4.1}$ (21 points) Below is incomplete code to build this widget. Fill in the blanks to create a widget where pressing "Left" will shift the ball 1 pixel to the left. (Pressing "Right" would shift the ball 1 pixel to the right, but we have omitted that part of the code for brevity.) Select functions from gctx.mli and widget.mli are provided in the Appendix.

See the description for each blank in the comments above the blank.

```
 1  let make_game (init_pos : int) : widget =
 2     let pos = {contents = init_pos} in
 3     let circle_drawing (g: Gctx.gctx) : unit =
 4      Gctx.draw_ellipse g (pos.contents, 10) 5 5 in
 5     let board, canvas_nc = canvas (100,20) circle_drawing in
 6     let left_w, left_lc, left_nc = button "Left" in
 7     let left () =
 8       pos.contents <- max (pos.contents - 1) 5 in
 9     left_nc.add_event_listener (mouseclick_listener left);
10a    let right_w, right_lc, right_nc = button "Right" in
10b    let right () =
10c      pos.contents <- min (pos.contents + 1) 95 in
10d    right_nc.add_event_listener (mouseclick_listener right);
11     let full_game = hlist [board; border left_w; border right_w] in
12     border full_game
```

$\boxed{4.2}$ For each of the following edits, decide whether they would (1) continue to compile, (2) maintain the appearance of the `make_game` widget on the screen, and (3) maintain the functionality / playability of the game.

(3 points)

Replace line 12 with `border board`

Still compiles
[✓ Yes] No

Maintains appearance
Yes [✓ No]

Maintains function
Yes [✓ No]

(3 points)
Move lines 3–5 below line 10

Still compiles
[✓ Yes] No

Maintains appearance
[✓ Yes] No

Maintains function
[✓ Yes] No

(3 points)
Replace line 11 with

```
let full_game = hpair (border left_w) (hpair (border right_w) board) in
```

Still compiles
[✓ Yes] No

Maintains appearance
Yes [✓ No]

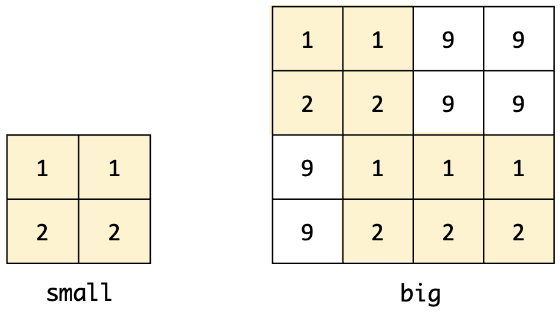
Maintains function
[✓ Yes] No

5. **Java Arrays** (20 points)

The goal in this problem will be to write a function `numberOfAppearances` that takes two square 2D arrays as parameters:

- array `small` of size `m` $\times$ `m`, where `m` > 0
- array `big` of size `n` $\times$ `n`, where `m` $\leq$ `n`

The `numberOfAppearances` function should return an integer indicating the number of times `small` appears as a *subarray* of `big`. For example, if `small` and `big` look like this...



... then `numberOfAppearances(small, big)` should return 3. Note that different occurrences of `small` inside `big` may overlap.

You may assume the arrays are non-null.

On the next page, we've given you the start of a helper method called `isSubArray`, which checks whether one array is a subarray of another *at a given position*. First, implement this method; then use it to complete the implementation of `numberOfAppearances`.

Nothing for you to answer on this page

```
private static boolean isSubArray(int[][] small, int[][] big,
                                                    int row, int col) {
    int m = small.length;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < m; j++) {
            if (small[i][j] != big[row + i][col + j]) {
                return false;
            }
        }
    }
    return true;
}

public static int numberOfAppearances(int[][] small, int[][] big) {
    int m = small.length;
    int n = big.length;
    int count = 0;

    for (int i = 0; i <= n - m; i++) {
        for (int j = 0; j <= n - m; j++) {
            if (isSubArray(small, big, i, j)) {
                count++;
            }
        }
    }
    return count;
}
```

Scratch Space

Use this page for work that you do not want us to grade. If you run out of space elsewhere in the exam and you **do** *want to put something here that we should grade, make sure to put a clear note in the normal answer space for the problem in question.*

Appendix: GUI Gctx and Widget Interface

```
(* Gctx *)

(** Draw an ellipse, centered at position with given x and y radii. *)
val draw_ellipse : gctx -> position -> int -> int -> unit

(* Widget *)

type widget = {
  repaint : Gctx.gctx -> unit;
  handle : Gctx.gctx -> Gctx.event -> unit;
  size : unit -> Gctx.dimension;
}

(** A horizontal group of widgets *)
val hlist : widget list -> widget

(*****************************************)
(** {1 Label Widgets }                  *)
(*****************************************)

(** A record of functions that allows us to read and write the string
associated with a label. *)
type label_controller = { get_label : unit -> string;
  set_label : string -> unit }

(*****************************************)
(** {1 Event Listeners }                *)
(*****************************************)

(** An event listener processes events as they "flow" through
the widget hierarchy. *)
type event_listener = Gctx.gctx -> Gctx.event -> unit

(** Performs an action upon receiving a mouse click. *)
val mouseclick_listener : (unit -> unit) -> event_listener

(*****************************************)
(** {2 Notifier }                       *)
(*****************************************)

(** A notifier widget is a widget "wrapper" that doesn't take up any
extra screen space -- it extends an existing widget with the
ability to react to events. It maintains a list of of "listeners"
that eavesdrop on the events propagated through the notifier
widget.

When an event comes in to the notifier, it is passed to each
event_listener in turn, and then pass to the child widget.*)
```

```
(** A notifier_controller is associated with a notifier widget.
It allows the program to add event listeners to the notifier.*)

type notifier_controller = { add_event_listener : event_listener -> unit; }

val notifier : widget -> widget * notifier_controller

(****************************************)
(** {3 Canvas }                         *)
(****************************************)

(** A widget that allows drawing to a graphics context *)

(** Canvases are just widgets with an associated notifier_controller.
   The repaint method of a canvas is a parameter of the widget
   constructor.*)
val bare_canvas : Gctx.dimension -> (Gctx.gctx -> unit) -> widget
val canvas      : Gctx.dimension -> (Gctx.gctx -> unit) ->
                                          widget * notifier_controller

(****************************************)
(** {4 Button }                         *)
(****************************************)

(** A button has a string, which can be controlled by the
corresponding label_controller, and an event listener, which can be
controlled by the notifier_controller to add listeners (e.g. a
mouseclick_listener) that will perform an action when the button is
pressed. *)

val button : string -> widget * label_controller * notifier_controller

(****************************************)
(** {5 Game }                           *)
(****************************************)

(** make_game takes in an int representing the initial position of the
    ball and returns a widget of the playable game. *)

val make_game : int -> widget
```