CIS 1200 Midterm 2   November 10, 2023

Steve Zdancewic and Swapneel Sheth, instructors

SOLUTIONS

1. **OCaml Concepts — Recursion** (15 points total)

   (a) For each of the following functions, determine whether the program may stack overflow on some input, go into an infinite loop on some input, or terminate on **all** inputs.

   For each function, choose one answer.

   i. (3 points)

```
let rec contains (l: 'a list) (x: 'a) : bool =
    begin match l with
    | hd::tl -> if hd = x then true else contains tl x
    | _ -> false
    end
```

   □ Stack overflow   □ Infinite loop   ☒ Terminates on **all** inputs

   ii. (3 points)

```
let rec is_empty (l: 'a list) : bool =
    (l = []) || (is_empty l)
```

   □ Stack overflow   ☒ Infinite loop   □ Terminates on **all** inputs

   iii. (3 points)

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

   ☒ Stack overflow   □ Infinite loop   □ Terminates on **all** inputs

   iv. (3 points)

```
let rec list_max (l: 'a list) : 'a =
    begin match l with
    | [] -> failwith "empty list"
    | [x] -> x
    | x::y::z -> list_max ((max x y)::z)
    end
```

   □ Stack overflow   □ Infinite loop   ☒ Terminates on **all** inputs

   (b) (3 points)

   It is possible to write an function that stack overflows on one input and goes into an infinite loop on another input.

   True ☒   False □

2. **Queues** (46 points total)

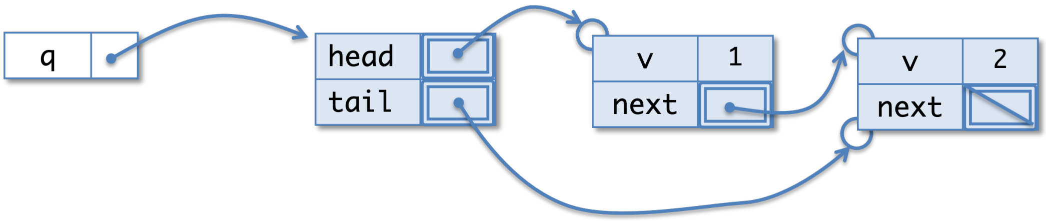
Recall the definitions and invariants of the (singly-linked) queue from Homework 4:

```
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                    (Invariant A)
 - q.head and q.tail both point to Some nodes, and
   - q.tail is reachable by following 'next' pointers            (Invariant B)
     from q.head
   - q.tail's next pointer is None                               (Invariant C)
*)
```

Consider the following (potentially buggy) `mystery` function:

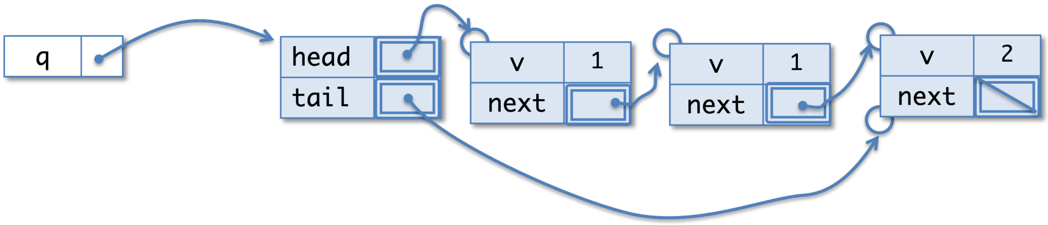
```
let mystery (v: 'a) (q: 'a queue) : unit =
  let rec loop (qno: 'a qnode option) : unit =
    begin match qno with
    | None -> ()
    | Some qn -> if qn.v = v then (
        let node = {v = v; next = qn.next} in
        qn.next <- Some node
    )
    else loop qn.next
  end in
  loop q.head
```

(a) (14 points) Consider a queue `q` as shown below:



We call the function as follows: `mystery 1 q`.

Draw the updated ASM diagram below after the `mystery` function is done executing. (You can ignore temporary stack bindings and other things on the heap, if any.)



(b) (3 points) For the function call above (`mystery 1 q`), are the queue invariants preserved?

☒ Yes ☐ No

(c) (9 points) Does the `mystery` function *always* preserve queue invariants? If yes, explain why. If no, provide an example of a valid `q` and an `int v` that will result in an invalid queue and also specify which invariant is being violated (A, B, or C from above).

☐ Invariants Always Preserved **OR** Invariant Violated is: ☐ A ☐ B ☒ C.

Reason/Example: The call `mystery 2 q` will violate queue invariant C since the queue's tail's next will not point to `None`.

(d) (20 points) Write a function `is_sorted` that takes in a (valid) queue and returns **`true`** if all the `qnode`'s `v` values are sorted in ascending order, and **`false`** otherwise.

To understand the behavior of `is_sorted`, the following tests are provided.

```
let test() : bool =
  let q = create () in
  enq 1 q;
  enq 2 q;
  enq 2 q;
  enq 3 q;
  is_sorted q

;; run_test "is_sorted true" test

let test() : bool =
  let q = create () in
  enq 3 q;
  enq 2 q;
  enq 1 q;
  not (is_sorted q)

;; run_test "is_sorted false" test
```

```
let is_sorted (q: 'a queue) : bool =
  let rec loop (prev: 'a qnode option) (curr: 'a qnode option) : bool =
    begin match (prev, curr) with
    | None, Some c -> loop curr c.next
    | Some p, Some c -> p.v <= c.v && loop curr c.next
    | _ -> true
  end in
  loop None q.head
```

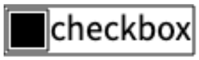
3. **GUI Programming** (19 points total)

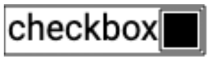
Consider the following (incomplete) implementation of the `checkbox` widget from the GUI homework. It builds a checkbox compositionally using a `bool value_controller`, a `label`, and a (bordered) `simple_canvas`, but is currently missing a `notifier` (that is used on line 13). The appendices include the interface information for the `widget` constructors.

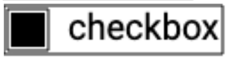
```
1  let checkbox (init:bool) (s : string) : widget * bool value_controller =
2    let vc = make_controller init in
3    let listener = mouseclick_listener
4                     (fun () -> vc.change_value (not (vc.get_value ()))) in
5    let paint_box (g:Gctx.gctx) =
6      if vc.get_value () then Gctx.fill_rect g (0,0) (19,19) else ()    in
7    let (widget1, _) = label s in
8    (* A *)
9    let widget2 = border (bare_canvas (20,20) paint_box) in
10   (* B *)
11   let widget3 = hpair widget1 widget2 in
12   (* C *)
13   nc.add_event_listener listener;
14   (widget3, vc)
```

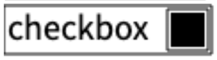
(a) (4 points) Assuming that the code is completed to add a notifier (see below), which of the following shows the GUI that will result from defining the checkbox with the following code? (Choose one.)

```
let (cb,_) = checkbox true "checkbox"
;; Eventloop.run (border cb)
```

☐ 

☒ 

☐ 

☐ 

(b) (3 points) Suppose we add the following line at the location marked `A` on line 8:

```
let (widget1, nc) = notifier widget1 in
```

Which parts of the checkbox widget can the user click to toggle its state?

☒ the label ☐ the canvas ☐ both the label and the canvas

☐ neither the label nor the canvas

(c) (3 points) Suppose we add the following line at the location marked `B` on line 10:

```
let (widget2, nc) = notifier widget2 in
```

Which parts of the checkbox widget can the user click to toggle its state?

☐ the label ☒ the canvas ☐ both the label and the canvas

☐ neither the label nor the canvas

(d) (3 points) Suppose we add the following line at the location marked `C` on line 12:

```
let (widget3, nc) = notifier widget3 in
```

Which parts of the checkbox widget can the user click to toggle its state?

□ the label □ the canvas ☒ both the label and the canvas

□ neither the label nor the canvas

(e) (3 points) Consider named function `paint_box` defined on line 5 of the code above. Which of the following variable bindings *must* be stored with the closure's saved stack created by the OCaml ASM when it puts that function in the heap? (Mark all that apply)

□ `init` □ `s` ☒ `vc` □ `listener` □ `g`

(f) (3 points)

Assuming a correct implementation of `make_controller`, the two methods `vc.get_value` and `vc.change_value` in the code above will refer to distinct closures in the heap, and those closures will *not* share any common saved stack variable bindings.

True □ False ☒

4. **Java Array Programming: MagicSquare** (40 points)

Write a function `isMagicSquare`, that takes in a *square* array of type **`int`**`[][]` and returns a boolean indicating if the given array is a magic square.

A magic square is defined as a matrix in which the sums of the numbers in each row, each column, and both main diagonals are the same. Empty squares are also considered magic squares.

(Contrary to “actual” magic squares, we do not care about numbers being distinct).

Pictorially, given two square matrices `a` and `b` as shown below:

```
     a                b
   8 1 6            1 2 3
   3 5 7            4 5 6
   4 9 2            7 8 9
```

The result of calling `isMagicSquare(a)` should be **`true`** because the sum of each row, each column, and both main diagonals is 15. However, calling `isMagicSquare(b)` should be **`false`**.

For the implementation below, you may assume the following:

- The input array `a` is not null.
- The input array `a` contains no null sub-arrays.
- The input array `a` is not ragged (i.e., all sub-arrays have equal length).

For full credit, your implementation must:

- Check if the square is empty and return **`true`** in that case.
- Check if the input array `a` is not square and return **`false`** in that case.

Note that `a[i]` refers to the row `i` in `a`.

We’ve given you an outline for `isMagicSquare` on the next two pages. Use the outline and complete the implementation.

```
public static boolean isMagicSquare(int[][] a) {

    // Check if array is empty
    if (a.length == 0) {
        return true;
    }

    // Check if array is square
    if (a.length != a[0].length) {
        return false;
    }

    // Calculate a reference value
    int magicSum = 0;
    for (int j = 0; j < a[0].length; j++) {
        magicSum += a[0][j];
    }

    // Check the sum of each row
    for (int i = 1; i < a.length; i++) {
        int rowSum = 0;
        for (int j = 0; j < a[i].length; j++) {
            rowSum += a[i][j];
        }
        if (rowSum != magicSum) {
            return false;
        }
    }

    // Check the sum of each column
    for (int j = 0; j < a[0].length; j++) {
        int colSum = 0;
        for (int i = 0; i < a.length; i++) {
            colSum += a[i][j];
        }
        if (colSum != magicSum) {
            return false;
        }
    }

    // Check the sum of the main diagonal
    int mainDiagonalSum = 0;
    for (int i = 0; i < a.length; i++) {
        mainDiagonalSum += a[i][i];
    }
    if (mainDiagonalSum != magicSum) {
        return false;
    }

    // Check the sum of the other diagonal
    int otherDiagonalSum = 0;
    for (int i = 0; i < a.length; i++) {
        otherDiagonalSum += a[i][a.length - 1 - i];
    }
    if (otherDiagonalSum != magicSum) {
        return false;
    }

    // Return the final answer (if needed)
    return true;
}
```

APPENDIX: GUI Widget Interface

```
(** A widget is an object that provides three services:
   - it can repaint itself (given an appropriate graphics context)
   - it can handle events
   - it knows its dimensions (relative to a graphics context)  *)
type widget = {
  repaint : Gctx.gctx -> unit;
  handle  : Gctx.gctx -> Gctx.event -> unit;
  size    : unit -> Gctx.dimension;
}

(** A widget that does nothing but take up space *)
val space : Gctx.dimension -> widget

(** Adds a border around another widget *)
val border : widget -> widget

(** A pair of horizontally adjacent widgets *)
val hpair : widget -> widget -> widget

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

(** 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 }

(** Construct a label widget and its controller. *)
val label : string -> widget * label_controller

(** 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

(** 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; }

(** Construct a notifier widget and its controller *)
val notifier : widget -> widget * notifier_controller

val bare_canvas : Gctx.dimension -> (Gctx.gctx -> unit) -> widget
```

```
(** A controller for a value associated with a widget.

    This controller can read and write the value. It also allows
    change listeners to be registered by the application. These listeners are
    run whenever this value is set. *)
type 'a value_controller = {
  add_change_listener : ('a -> unit) -> unit;
  get_value           : unit -> 'a;
  change_value        : 'a -> unit
  }

(** A utility function for creating a value_controller. *)
val make_controller : 'a -> 'a value_controller

(* end widget *)
```