CIS 1200 Midterm 2   March 24, 2023

Steve Zdancewic, instructor

Name: ____

PennKey (penn login id, e.g. `stevez`): ____

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

Signature: _____________________________________   Date: _________________________

- There are 120 total points. The exam length is 60 minutes.
- There are 9 pages in the exam and an Appendix for your reference.
- Please begin by writing your PennKey at the bottom of all the odd-numbered pages in the rest of the exam.
- Do not spend too much time on any one question. Be sure to recheck all of your answers.
- We will ignore anything you write on the Appendix.
- For coding problems: aim for accurate syntax, but we will not grade your code style for indentation, spacing, etc.
- If you need extra space for an answer, you may use the scratch page at the end of the exam; make sure to clearly indicate that you have done this in the normal answer space for the problem.
- Good luck!

1. **Deques and the OCaml ASM** (21 points)

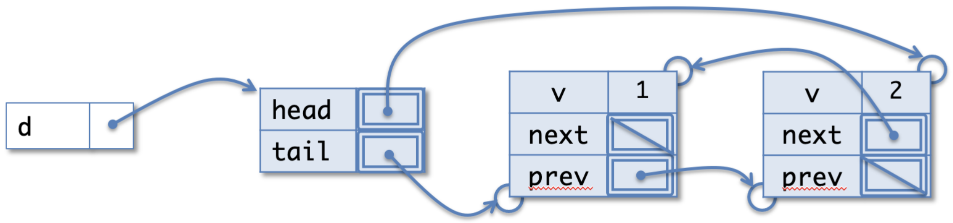
   Recall the definitions of `deque` and `dqnode` from homework 4:

```
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;
}
```

   In Appendix A you will find the invariant for deques, with each clause annotated with a letter between (a) and (f). You will also find the code for a correct implementation of the `to_list` operation. For each of the following stack and heap diagrams, indicate whether the value `d` of type `int deque` satisfies the deque invariant or mark *all* of the properties that are broken. Then choose one option for the result of running `to_list d`.

   a.



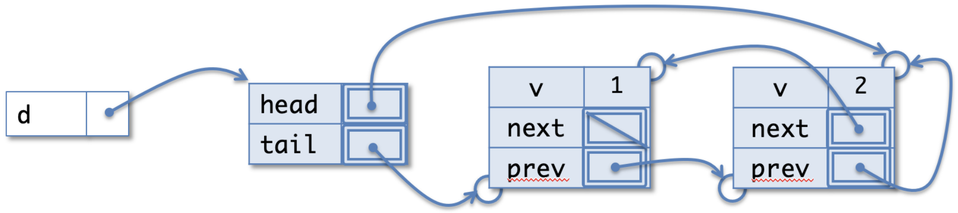
☐ Satisfies invariant *or*

☐ Clause (a) fails (“tail reachable from head via next”)
☐ Clause (b) fails (“nothing next after tail”)
☐ Clause (c) fails (“head reachable from tail via prev”)
☐ Clause (d) fails (“nothing previous before head”)
☐ Clause (e) fails (“next then prev”)
☐ Clause (f) fails (“prev then next”)

What will be the result from running **`let`** `ans = to_list d`?

☐ `ans = []`
☐ `ans = [2]`
☐ `ans = [1;2]`
☐ `ans = [2;1]`
☐ the program will go into an infinite loop

b.

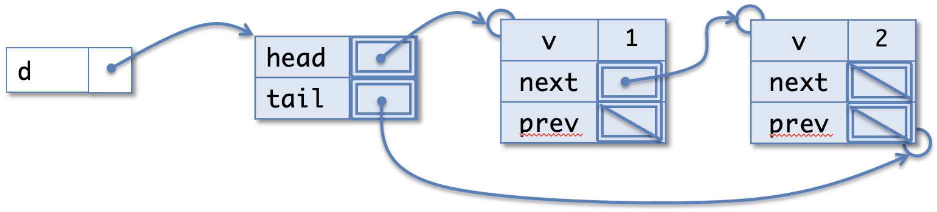


☐ Satisfies invariant *or* ☐ Clause (a) fails ("tail reachable from head via next")
☐ Clause (b) fails ("nothing next after tail")
☐ Clause (c) fails ("head reachable from tail via prev")
☐ Clause (d) fails ("nothing previous before head")
☐ Clause (e) fails ("next then prev")
☐ Clause (f) fails ("prev then next")

What will be the result from running `let ans = to_list d`?

☐ `ans = []` ☐ the program will go into an infinite loop
☐ `ans = [2]`
☐ `ans = [1;2]`
☐ `ans = [2;1]`

c.



☐ Satisfies invariant *or* ☐ Clause (a) fails ("tail reachable from head via next")
☐ Clause (b) fails ("nothing next after tail")
☐ Clause (c) fails ("head reachable from tail via prev")
☐ Clause (d) fails ("nothing previous before head")
☐ Clause (e) fails ("next then prev")
☐ Clause (f) fails ("prev then next")

What will be the result from running `let ans = to_list d`?

☐ `ans = []` ☐ the program will go into an infinite loop
☐ `ans = [2]`
☐ `ans = [1;2]`
☐ `ans = [2;1]`

2. **Programming with Deques** (24 points)

   Suppose we want to define a function move_tail_to_head that, given a deque d, rearranges the references to move the tail node (if any) to be the head, but otherwise leaves the elements in the same order. If d is empty, or if it has only one element, then it is unchanged. Assuming that d satisfies the deque invariant before move_tail_to_head is called, it should again satisfy it afterward.

   For instance, if we have a deque d with four elements such that to_list d = [1;2;3;4], then after running move_tail_to_head d, we would have to_list d = [4;1;2;3].

   Complete the code below for move_tail_to_head. Note that because you are simply updating the nodes in place, there is no need to allocate any new 'a dqnode values.

```
let move_tail_to_head (d : 'a deque) : unit =
  begin match d.tail with
    | None -> () (* no elements, so the deque remains the same *)
    | Some n ->
      begin match n.prev with
        | None -> () (* only one element, so the deque remains the same *)
        | Some p ->  (* there are at least two elements *)
          (* adjust the tail *)

          p.______________ <- _________________

          d.tail <- ___________________________

          (* move n to the head *)

          n.next <- ___________________________ ;

          n.prev <- ___________________________ ;

          d.head <- ___________________________ ;

          (* patch up the new second element *)

          begin match _________________ with

            | None -> failwith "impossible: there were two nodes"

            | Some m -> m.prev <- _________________
          end
      end
  end
```

3. **OCaml Concepts** (12 points)

(a) Consider the following (nonsensical) OCaml function `foo` that processes an `int list`:

```
let foo (l:int list) : int =
  let rec loop (m:int list) (acc:int) : int =
    begin match m with
      | [] -> acc
      | x::xs ->
        if x > acc then
          loop xs (acc + x)   (* A *)
        else
          5 + (loop xs acc)   (* B *)
    end
  in
  loop l (* C *) (loop l 0) (* D *)
```

Recall that a function call is in *tail position* if it will be evaluated in an otherwise empty workspace in the abstract stack machine. There are four calls to the inner `loop` function labeled `A`–`D`. Indicate which of them are in *tail call position* (mark all that apply):

☐ call `A` is in tail position ☐ call `B` is in tail position
☐ call `C` is in tail position ☐ call `D` is in tail position

(b) Consider the following OCaml program that constructs an object `o:obj` that supports two methods, `method1` and `method2`:

```
type obj = {method1 : int -> int ; method2 : unit -> int}

let mk_object (x:int) (y:int) : obj =
  let field = {contents = x + y} in
  {
    method1 = (fun (a:int) -> a + field.contents) ;
    method2 = (fun () -> field.contents + x)
  }

let o : obj = mk_object 1200 42
```

i. Which of the following identifiers' values must (necessarily) be stored as part of the closure constructed in the heap for `o.method1`? (Mark all that apply.)

☐ `x` ☐ `y` ☐ `field` ☐ `a`

ii. Which of the following identifiers' values must (necessarily) be stored as part of the closure constructed in the heap for `o.method2`? (Mark all that apply.)

☐ `x` ☐ `y` ☐ `field` ☐ `a`

4. **GUI Programming** (23 points)

This problem refers to the widget module of the GUI library from HW05. Parts of its interface are shown in Appendix B.

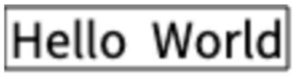
An *Easter egg* is a message hidden in a game or other application—it remains secret until the user performs some action that reveals the secret message. In terms of our GUI design, an Easter egg is just a kind of `label` widget with the string set to `""` until the user triggers the hidden message it by clicking on some *other* widget. We also let the user hide the Easter egg again by clicking on the revealed secret message. The Easter egg and the widget that triggers it might be placed in different locations in the user interface.

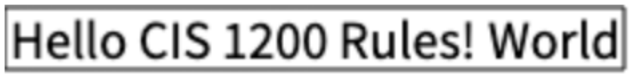
The type for the new `easter_egg` constructor is shown in Appendix B: `easter_egg w msg` wraps the provided widget `w` to make it into a "trigger" such that clicking it reveals the message in addition to whatever `w` usually does; it also returns the Easter egg itself.

For example, the following program demonstrates the use of an Easter egg by hiding the secret message `"CIS1200 Rules!"` between the labels `"Hello "` and `" World"` (which is wrapped as the trigger).

```
1  ;; open Widget
2
3  let hello_label, _ = label "Hello "
4  let world_label, _ = label " World"
5  let trigger, egg = easter_egg world_label "CIS 1200 Rules!"
6  let group = border (hpair hello_label (hpair egg trigger))
7  ;; Eventloop.run group
```

The resulting GUI displayed before and after the user clicks on "World" is shown below:

before: 
Hello World

after: 
Hello CIS 1200 Rules! World

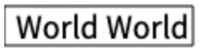
(a) Suppose we want to change the code so that the hidden message appears to the right of `" World"` in the GUI. Which of the following changes to the code will do that? Assume that each change is applied separately from the others. (Mark all that apply)

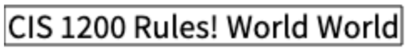
☐ swap the order of lines 3 and 4
☐ change line 5 to be: `let egg, trigger = easter_egg world_label "CIS 1200 Rules!"`
☐ change line 6 to be: `let group = border (hpair hello_label (hpair trigger egg))`
☐ change line 6 to be: `let group = border (hpair egg (hpair hello_label trigger))`

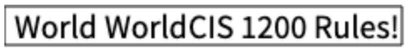
(b) Suppose we change the `group` defined in line 6 as shown below:

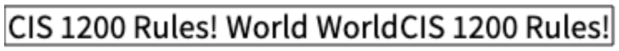
```
let group = border (hpair egg (hpair trigger (hpair trigger egg)))
```

One of the images below shows the "before" state (i.e. before the egg is revealed), another image shows the "after" state after the user has clicked on the *left* "World" label, and some of the images are not produced at all by this code. Check the boxes to indicate which picture is which.

☐ before ☐ after ☐ not produced 
World World

☐ before ☐ after ☐ not produced 
CIS 1200 Rules! World World

☐ before ☐ after ☐ not produced 
World WorldCIS 1200 Rules!

☐ before ☐ after ☐ not produced 
CIS 1200 Rules! World WorldCIS 1200 Rules!

(c) It is possible to implement the Easter egg widgets entirely in terms of the existing widget components whose interfaces appear in Appendix B. This means that the code does not need to explicitly make use of any of the "low-level" operations that are provided by the `Gctx` module. Fill in the blanks below that composes the Easter egg implementation from the existing `label` and `notifier` widgets. We have used suggestive names to guide you. You will have to make appropriate use of the `label_controller lc` but *do not* have to introduce any additional state. Note that the Easter egg should start out as hidden (i.e., with a label set to the empty string `""`).

```
let easter_egg (w:widget) (msg:string) : widget * widget =

  let egg_label, lc = label ________________________ in

  let egg =
        { repaint = ____________________________________________;

          handle = mouseclick_listener (fun () -> _________________________________);

          size = egg_label.size
        }
  in
  let trigger, nc = ____________________________________________ in

  nc.add_event_listener
     (mouseclick_listener (fun () -> _________________________________);
  (trigger, egg)
```

5. **Java Concepts** (16 points)

(a) What is the result of running the following code? (Choose one)

```
String s1 = "CIS 1200";
String s2 = "CIS ";
boolean ans = (s1 == s2 + "1200");
```

☐ `ans` contains **`true`** ☐ `ans` contains **`false`** ☐ this code raises and exception

(b) What is the result of running the following code? (Choose one)

```
int[] a = {0,1,2,3};
boolean ans = (a[4] == 3);
```

☐ `ans` contains **`true`** ☐ `ans` contains **`false`** ☐ this code raises and exception

(c) What is the result of running the following code? (Choose one)

```
String s;
boolean ans = (s.equals("CIS 1200"));
```

☐ `ans` contains **`true`** ☐ `ans` contains **`false`** ☐ this code raises and exception

(d) What is the result of running the following code? (Choose one)

```
String[] strs1 = {"a", "b", "c"};
String[] strs2 = {strs1[0], strs1[1]};
boolean ans = (strs1[0] == strs2[0])
```

☐ `ans` contains **`true`** ☐ `ans` contains **`false`** ☐ this code raises and exception

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

Write a function `hPair`, that takes in two *rectangular* arrays of type `int[][]` (i.e., every inner array has the same length) and returns a new array that places the arrays horizontally adjacent to each other. Any "open" space left by the difference in array sizes should be filled with `0`'s. Pictorially, if `a` and `b` are as shown below, then the results of using `hPair` in the two orders are as shown.

```
   a        b       hPair(a,b)   hPair(b,a)
 1 1 1     3 3      1 1 1 3 3    3 3 1 1 1
 2 2 2     4 4      2 2 2 4 4    4 4 2 2 2
           5 5      0 0 0 5 5    5 5 0 0 0
```

You may assume that the input arrays `a` and `b` are not null, that they are rectangular, and that they contains no null sub-arrays. Note that `a[i]` refers to the row `i` in `a`. If `a` has no rows, then `hPair` should return a copy of `b` (and vice versa). You may find it useful to use the static methods `Math.max` and `Math.min`, or the array `clone` method which are given in Appendix C.

```
public int[][] hPair(int[][] a, int[][] b) {










}
```

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.*

A Deques

A.1 Deque Invariant

Either (1) the deque is empty and the head and tail are both `None`, or (2) the deque is non-empty and

- `head = Some n1` and `tail = Some n2`, where

  (a) `n2` is reachable from `n1` by following `next` pointers
  ("tail reachable from head via next")

  (b) `n2.next = None` ("nothing next after tail")

  (c) `n1` is reachable from `n2` by following `prev` pointers
  ("head reachable from tail via prev")

  (d) `n1.prev = None` ("nothing previous before head"); and

- for every node `n` in the deque,

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

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

A.2 Deque functions

Code for the `'a deque` version of `to_list`:

```
let to_list (d : 'a deque) : 'a list =
  let rec loop (curr: 'a dqnode option) (l:'a list) : 'a list =
    begin match curr with
      | None -> l
      | Some n -> loop n.prev (n.v::l)
    end
  in
  loop d.tail []
```

B GUI library: `Widget.mli` excerpt

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

val hpair : widget -> widget -> widget
val vpair : widget -> widget -> 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

(** NEW! *)
val easter_egg : widget -> string -> widget * widget
```

C Javadocs excerpt

static int	`Math.max(`**int** `a,` **int** `b)` Returns the greater of two int values.
static int	`Math.min(`**int** `a,` **int** `b)` Returns the smaller of two int values.
int`[]`	`a.clone()` Returns a new copy of the array `a` (assuming `a` is of type **int**`[]`).