CIS 1200 Midterm 2 March 22, 2024

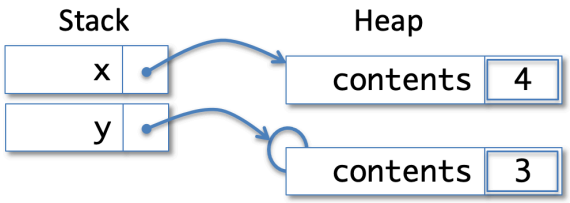
SOLUTIONS

1. **ASM** (10 points total)

   For each ASM diagram below, write a short piece of code that, when placed on the workspace with an empty stack and heap, would eventually reach the ASM configuration shown. Some questions use this definition of a mutable record type.

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

   For example, if the stack and heap look like the following:

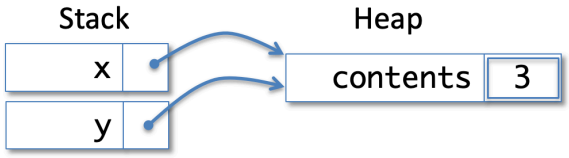


   Then your workspace code might look like this:

```
let x : int ref = { contents = 4 }
let y : int ref option = Some { contents = 3 }
```

   Above, we have included the types of `x` and `y` for clarity; you may omit type annotations in your solutions. If an ASM configuration shown is impossible, check the box marked "no code can produce this diagram."

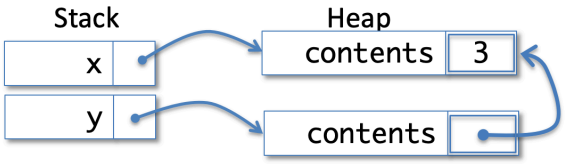
   (a)




```
let x = { contents = 3 }
let y = x
```

   ☐ *No code can produce this diagram*

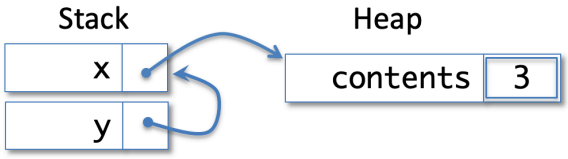
(b)




```
let x = { contents = 3 }
let y = { contents = x }
```

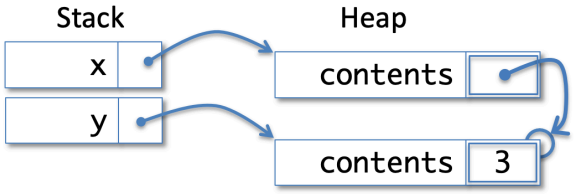
☐ *No code can produce this diagram*

(c)



☒ *No code can produce this diagram*

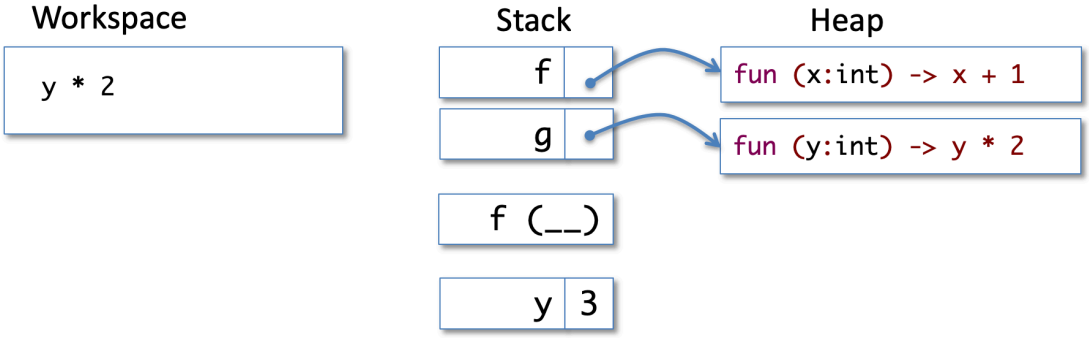
(d)




```
let f = { contents = None }
let g = { contents = 3 }
;; f.contents <- Some g
```

☐ *No code can produce this diagram*

(e) This diagram also includes the workspace as part of the ASM configuration. This diagram occurs midway during the execution—the final result has not yet been determined. However, your job is still the same: figure out what code to put on the initial workspace of an ASM that could produce this diagram.




```
let f = fun (x:int) -> x + 1
let g = fun (y:int) -> y * 2
f (g 3)
```

☐ *No code can produce this diagram*

2. **Programming with linked queues** (20 points total)

Recall the definitions and invariants of the (singly-linked) queue from Homework 4. For reference, these definitions also appear in Appendix B.

In this problem, you will use the design process to implement a generic function, called `intersperse`, that inserts a given value between each pair of values in a queue. For example, suppose `q` contains the values `1,2,3`, in that order. Then, after an execution of `intersperse 0 q`, the queue `q` should contain `1,0,2,0,3`.

The `intersperse` function should assume that its argument is a valid queue. If the queue is empty or contains only a single element, the queue should be unchanged. As the purpose of `intersperse` is to mutate its argument, it should always return `()`.

The first step of the design process is to understand the problem as described above.

(a) (2 points) The next step is to define the interface. Write the type for `intersperse` as you might find it in a `.mli` file. Your type **must** be generic.

```
val intersperse : ___'a -> 'a queue -> unit____________________
```

(b) (8 points) The next step is to write test cases. For example, one *good* test case is derived from the problem description above. (The queue operations `from_list` and `to_list` are defined in the appendix.)

```
let test () =
  let q = from_list [1;2;3] in
  intersperse 0 q ;
  to_list q = [1;0;2;0;3]
```

When testing your code, some test cases are better than others. This problem asks to to identify the *good* tests from other, less useful tests. For each test case below, classify it using one the following words:

- **wrong**: the test contradicts the problem description and describes a behavior of `intersperse` that is inconsistent with the correct behavior of the function.
- **invalid**: the test uses an invalid queue as the argument to `intersperse`. For reference, the queue invariant appears in Appendix B.
- **redundant**: the test is too similar to the case shown on the previous page. (i.e. all implementations that pass the example test will also pass this test.)
- **good**: the test is appropriate for `intersperse` and passing it will increase your confidence that the function is implemented correctly.

(NOTE: The problem continues on the next page. Make sure that you filled in the type above.)

Check one option for each test case below. If the test is *wrong* or *invalid*, add a short explanation. Leave blank otherwise.

i.
```
let test () =
  let q = { head = None; tail = None } in
  intersperse 0 q;
  to_list q = []
```

This test is

☐ wrong ☐ invalid ☐ redundant ☒ good

If wrong, the expected result of `to_list q` should be

_____

If invalid, `q` violates the invariant because

_____

ii.
```
let test () =
  let node1 = { v = 1; next = None } in
  let node2 = { v = 2; next = None } in
  let node3 = { v = 3; next = None } in
  let q = { head = Some node1; tail = Some node3 } in
  intersperse 0 q;
  to_list q = [1;0;2;0;3]
```

This test is

☐ wrong ☒ invalid ☐ redundant ☐ good

If wrong, the expected result of `to_list q` should be

_____

If invalid, `q` violates the invariant because
q.tail is not reachable by following next pointers from q.head

iii.
```
let test () =
  let q = from_list ["a";"b";"c"] in
  intersperse "0" q;
  to_list q = ["a"; "0"; "b"; "0"; "c"]
```

This test is

☐ wrong ☐ invalid ☒ redundant ☐ good

If wrong, the expected result of `to_list q` should be

_____

If invalid, `q` violates the invariant because

_____

iv.
```
let test () =
  let node1 = { v = 1; next = None } in
  let node2 = { v = 2; next = Some node1 } in
  let node3 = { v = 3; next = Some node2 } in
  let q = { head = Some node3; tail = Some node1 } in
  intersperse 0 q;
  to_list q = [1;0;2;0;3]
```

This test is

☒ wrong ☐ invalid ☐ redundant ☐ good

If wrong, the expected result of `to_list q` should be
`[3;0;2;0;1]`
If invalid, `q` violates the invariant because

_____

(c) (10 points) The next step is to write code. Unwisely, you asked ChatGPT to implement this function for you.

Here is its response:[1]

```
let intersperse (x : _) (q : _) =
  begin match q.head with
  | None -> ()
  | Some node ->
    let rec loop (curr_node : 'a qnode option) : unit =
      begin match curr_node with
      | None -> ()
      | Some n ->
        let new_node = { v = x; next = n.next } in
        n.next <- Some new_node;
        loop new_node.next
      end
    in
    loop (Some node)
  end
```

i. Unfortunately, this code is buggy! It produces a different answer than expected for a correct implementation. Fill in the blank with its result.

```
let q = from_list [1;2;3] in
intersperse 0 q ;
to_list q = [1;0;2;0;3;0]
```

ii. Does the following test case pass?

```
let test () =
  let q = from_list [1;2;3] in
  intersperse 0 q ;
  valid q
```

☐ Yes ☒ No

If no, what part of the queue invariant does not hold for `q`?
The q.tail refers to a qnode where the next reference is not None.

iii. Is this definition of `intersperse` tail recursive?

☒ Yes ☐ No

[1] This code was written by ChatGPT and modified to fix compilation errors, omit the type annotation, and match CIS 1200 style.

3. **Object Encodings** (18 points total)

Suppose you would like to create an OCaml "object" representing a *mutable set*.

You might use a mutable set to store a collection of values. The `mk_set` function should construct an empty set. The `insert` method should modify the set so that it contains the new element. The `member` method should return whether a particular value is contained in the set. The `size` method should return the total number of values contained in the set. The `to_list` method should return all elements in the set, in ascending order.

```
let test () : bool =
  let s = mk_set () in
  s.insert 1;
  s.insert 3;
  s.insert 1;
  s.member 3 && s.size () = 2 && s.to_list () = [1;3]
;; run_test "set operations" test
```

(a) (2 points)

Complete the following record type definition that you will use for your object encoding by adding the types of the `insert` and `member` components. The `mk_set` function (see the next page) should create a value of this type.

```
type 'a set = {
    insert : 'a -> unit;
    member : 'a -> bool;
    size : unit -> int;
    to_list : unit -> 'a list
}
```

You will represent this mutable set using an *ordered list*, stored in a mutable record. We've given you the implementations of the `insert` and `member` methods, which both use definitions from the list library shown in Appendix A.

```
type 'a set_state = { mutable elements : 'a list }

let mk_set () : 'a set =

  let state = { elements = [] } in
  {
    insert = (fun x ->
       state.elements <- List.insert_sorted x state.elements);

    member = (fun x -> List.member x state.elements);

    size   = ____________see part (b) below _____________ ;

    to_list = ___________see part (c) below _____________

  }
```

(b) (2 points) Select the correct implementation of the `size` method. (Check only one.)

- ☐ `fun x -> 0`
- ☐ `fun x -> List.length x`
- ☒ `fun () -> List.length state.elements`
- ☐ `List.length state.elements`
- ☐ *None of the above*

(c) (2 points) Select the correct implementation of the `to_list` method. (Check only one.)

- ☐ `fun x -> x`
- ☒ `fun () -> state.elements`
- ☐ `state.elements`
- ☐ `mk_set state.elements`
- ☐ *None of the above*

(d) (2 points) Which stack variables must be saved in the closure of the higher-order function used for `insert`. (Check all that apply, or none of the above.)

- ☐ `state.elements`
- ☒ `state`
- ☐ `elements`
- ☐ `List.insert_sorted`
- ☐ `x`
- ☐ *None of the above*

Check **`true`** or **`false`** for each statement. (10 points)

(e) These definitions ensure that the list of elements stored in the `state` record is always sorted. There is no way for a user of this code to violate this invariant.

☒ **`true`** ☐ **`false`**

(f) What is the value of this expression?

```
let s1 = mk_set () in
let s2 = mk_set () in
s1 == s2
```

☐ **`true`** ☒ **`false`**

(g) What is the value of this expression?

```
let s1 = mk_set () in
let s2 = s1 in
s1.insert 2;
s2.member 2
```

☒ **`true`** ☐ **`false`**

(h) What is the value of this expression?

```
let s1 = mk_set () in
let s2 = mk_set () in
s1.insert 1;
s2.insert 1;
s1.to_list () == s2.to_list ()
```

☐ **`true`** ☒ **`false`**

(i) What is the value of this expression?

```
let s1 = mk_set () in
s1.insert 2;
let s2 = s1 in
s1.insert 3;
s1 == s2
```

☒ **`true`** ☐ **`false`**

4. **Reactive Programming** (12 points total)

Suppose you are building an application with the GUI library from homework 5. For reference, the interface for the `Widget` module appears in Appendix C.

As part of your application, you have a list of checkboxes for users to indicate preferences. For convenience, you also include a "Select all" button at the bottom of the list.

For example, suppose the application starts and the user selects "Option B", producing an application state as shown on the left below. If the user then clicks the "Select all" button, then all all options should be selected, as shown on the right.

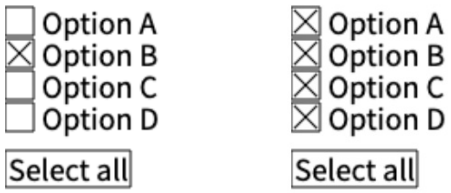


Figure 1: Application before and after the user clicks "Select all"

The first step of constructing this application is to create the four checkboxes marked "Option A" through "Option D" and the "Select all" button. The code below does so.

```
let checkboxes =
   [ checkbox false " Option A" ;
     checkbox false " Option B" ;
     checkbox false " Option C" ;
     checkbox false " Option D" ]

let ws = List.transform (fun (x,y) -> x) checkboxes

let vcs = List.transform (fun (x,y) -> y) checkboxes

let (w_all, lc_all, nc_all) = button "Select all"
```

(a) (4 points) What are the types of the variables defined above? Some have been filled in for you.

```
val ws : ___widget list__


val vcs : __bool value_controller_list___

val w_all  : widget
val lc_all : label_controller
val nc_all : notifier_controller
```

(b) (4 points)

Use the definitions on the previous page to make it so that when the "Select all" button is pressed, all of the checkboxes in the list above are checked. In your answer, you may use operations from the `List` library in Appendix A and the `Widget` library in Appendix C.

```
let on_click () : unit =
  List.iter (fun vc -> vc.change_value true) vcs

;; nc_all.add_event_listener (mouseclick_listener on_click)
```

(c) (4 points)

Now use the definitions on the previous page to create the toplevel widget that combines the checkboxes and the button together. Your widget should look like Figure 1. For a clean layout, you should use a `space (10,10)` widget between the check boxes and the button. Remember that the `button` function from the widget library does not draw a border around the clickable text; you'll need to add a `border` to make it look like the figure. In your answer, you may use operations from the `List` library in Appendix A and the `Widget` library in Appendix C.

```
let top : widget =
   vlist [ vlist ws ; space (10,10) ; border w_all ]
```

A APPENDIX: List library Interface

```
(* Count the number of elements in the list.
   For example, length [1;2;3] return 3 *)
val length : 'a list -> int

(* Add an element to a sorted list, if the element is not already present.
   For example, insert 3 [1;2;4] returns [1;2;3;4] *)
val insert_sorted : 'a -> 'a list -> 'a list

(* Test whether an element is contained in the list.
   For exmaple, member 3 [1;2;3;4] returns true   *)
val member: 'a -> 'a list -> bool


(* Apply the function to each element of the list. *)
val iter : ('a -> unit) -> 'a list -> unit

(* Apply the function to each element in the list and return results. *)
val transform : ('a -> 'b) -> 'a list -> 'b list

(* Reduce the list into a single value. *)
val fold : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b

(* Reverse the order of the list.
   For example, rev [1;2;3] returns [3;2;1]  *)
val rev : 'a list -> 'a list

(* Create a new list containing all of the elements of the first list, followed
   by those in the second.  This function is the same as the`@` operator.
   For example, append [1;2] [3;4] returns [1;2;3;4] *)
val append : 'a list -> 'a list -> 'a list
```

B APPENDIX: Queue

```
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 }

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

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

(* For brevity, the definitions of these operations are not shown. *)

let to_list (q : 'a queue) : 'a list = ...

let from_list (xs : 'a list) :'a queue = ...

let valid (q: 'a queue) : bool = ...
```

C 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 pair of vertically adjacent widgets *)
val vpair : widget -> widget -> widget

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

(** A vertical group of widgets *)
val vlist : 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 }

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

(***************************************)
(** {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
```

```
(*****************************************)
(** {1   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; }

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

(*****************************************)
(** {1   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

(*****************************************)
(** {1   Checkbox                        } *)
(*****************************************)

(** 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

(** A checkbox widget. *)
val checkbox : bool -> string -> widget * bool value_controller
```