CIS 120 Midterm II November 9, 2018

SOLUTIONS

1. **OCaml and Java** (20 points) Indicate whether the following statements are true or false.

a. True ☐ False ☒

The following OCaml function is tail-recursive:

```
let rec loop lst =
 begin match lst with
  | [] -> false
  | hd::tl -> if hd = false then ((loop tl) || false) else true
 end
```

b. True ☐ False ☒

The above `loop` function will terminate by exhausting stack space when the input is `[false; false]`.

c. True ☒ False ☐

In the OCaml ASM, stack bindings can be stored as part of a function on the heap.

d. True ☐ False ☒

In OCaml, the job of a `listener` is to help route an event to the appropriate component (or widget) for subsequent handling.

e. True ☒ False ☐

In OCaml, a record with some functions and mutable state is similar to an "object" in Java.

f. True ☐ False ☒

Records are always mutable in OCaml.

g. True ☐ False ☒

In Java, the following will only typecheck if `A` is a subtype of `B`.

```
A a = new B ();
```

h. True ☐ False ☒

In Java, if we have the following line: `a.m1()`, where `a` is defined as above, the code will compile without errors if `m1()` is only defined in the type `B`.

i. True ☐ False ☒

In Java, if we have the following line: `c.m1()`, where `c` is defined to have static type `C`, it is always possible to know statically (i.e., at compile time), the *exact* method that would be executed at run time.

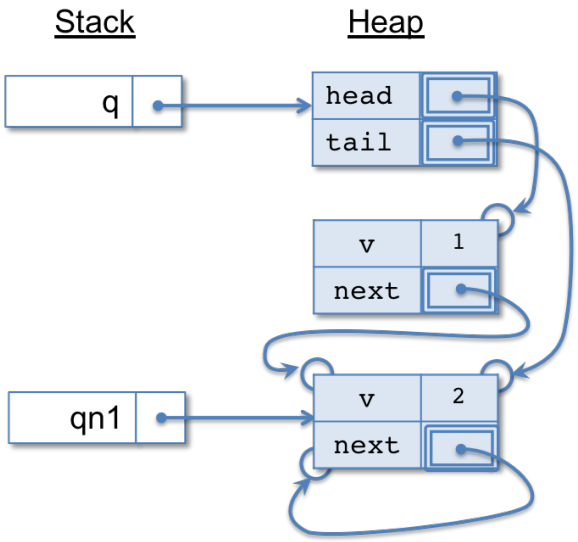
j. True ☒ False ☐

In Java, a static method can create objects dynamically (at runtime) and call non-static methods on those objects.

2. OCaml ASM and Queues (24 points)

Recall the mutable queue implementation from class and homework. The relevant code is also shown in Appendix A.

Suppose that we have the OCaml ASM shown below.



a. Does `q` satisfy the queue invariant? Circle **Yes** or **No**.

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.

```
let q = create () in
enq 1 q;
enq 2 q;
let qn1 = begin match q.tail with
  | Some x -> x
  | None -> failwith "impossible"
  end in
qn1.next <- Some qn1
```

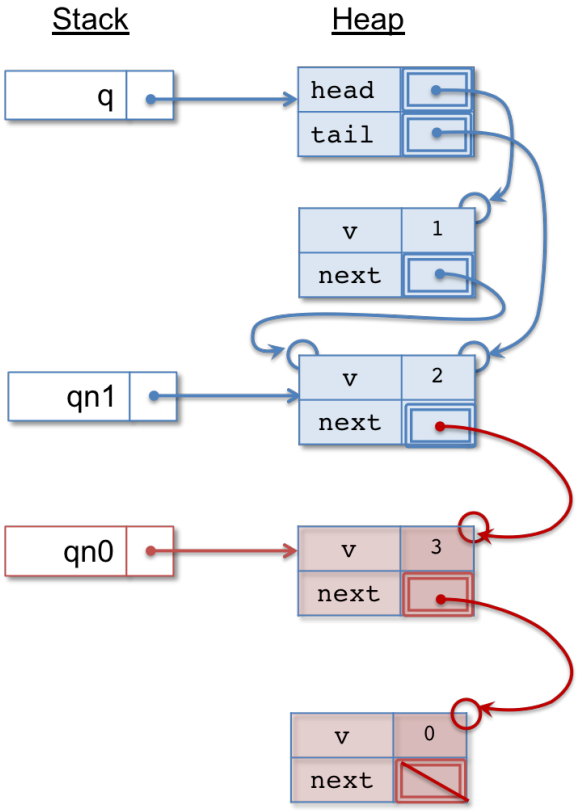
or

```
let q = { head = None; tail = None } in
let qn1 = { v = 2 ; next = None} in
  q.tail <- Some qn1;
  q.head <- Some { v = 1; next = Some qn1 };
  qn1.next <- Some qn1
```

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 = 3; next = Some { v = 0; next = None} } 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 in our provided diagram, mark it clearly with an "X".



d. Does `q` satisfy the queue invariant after the code in part (c) has executed?
Circle **Yes** or **No**.

3. OCaml: Objects and Encapsulation (12 points)

Consider the following Ocaml code, which use the `counter` object type.

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

type counter = {
  set : int -> unit;
  incr : unit -> int;
  reset : unit -> unit;
}

let mk_counter () : counter =
  let cnt : int ref = {contents = 0} in
  let ctr : counter = {
    set = (fun x -> cnt.contents <- x);
    incr = (fun () -> cnt.contents <- cnt.contents + 1;
                      cnt.contents);
    reset = (fun () -> cnt.contents <- 0);
  }
  in ctr

let mk_counters () : counter * counter * counter =
  let ctr1 = mk_counter () in
  let ctr2 = mk_counter () in
  (ctr1, ctr2, ctr2)

let (ctr1, ctr2, ctr3) = mk_counters ()

;; ctr1.set 3
;; print_int (ctr1.incr ())
;; print_int (ctr2.incr ())
;; ctr3.set 5
;; print_int (ctr2.incr ())
;; ctr2.reset ()
;; print_int (ctr3.incr ())
```

a. What will be printed to the terminal when the above program is run?

4 1 6 1

b. What would the following evaluate to (after all the code above has been executed)?

i. `ctr3.cnt` — `cnt` does not belong to type counter

ii. `ctr1 == ctr2` — **`false`**

iii. `ctr2 == ctr3` — **`true`**

iv. `ctr1 == ctr3` — **`false`**

v. `ctr2 = ctr3` — Cannot structurally compare function values

4. Writing and Testing Array Code (29 points)

If you are unfamiliar with minesweeper, it is a single-player puzzle video game. The objective of the game is to clear a rectangular board containing hidden mines or bombs without detonating any of them, with help from clues about the number of neighboring mines in each field.

A `MineSweeperBoard` class holds a representation of the contents of the playing field for a Mine Sweeper game. Internally, the playing field is represented using a 2-dimensional array of String values. A cell can have one of the following states:

- "O" = covered cell
- "F" = flag
- "M" = flagged mine
- "+" = covered mine
- "*" = uncovered mine (about to explode!)
- "1"..."8" = number of adjacent mines

A partial documentation of the `MineSweeperBoard` class is given in Appendix B.

In this section, we will test the behavior of some of `MineSweeperBoard` methods.

```
@Test
public void testGetCellMine() {
   // create a 4 by 4 MineSweeperBoard with 2 mines
   MineSweeperBoard board = new MineSweeperBoard(4, 4, 2);
   // put a mine at (1, 1)
   board.setCell(1, 1, "M");
   // assert that a Mine was returned
   assertEquals("Mine", "M", board.getCell(1, 1));
}
```

Note that MineSweeperBoard constructor will throw an `IllegalArgumentException` for certain input values. To test exceptions use the following template:

```
// test that null argument throws correct exception
@Test(expected=IllegalArgumentException.class)
public void testExceptionBoard() {
   // here's the code that will throw the exception
}
```

(Problem continues. No answers required on this page.)

a. (12 points) Your job is to complete **four** additional tests. Be sure to give your tests descriptive names. We will be grading the quality of your tests and how well they cover the invalid inputs of this method. Each test should cover a different situation.

```
@Test(expected=IllegalArgumentException.class)
public void testNegativeWidth () {
  MineSweeperBoard board = new MineSweeperBoard(-4, 4, 2);
}

@Test(expected=IllegalArgumentException.class)
public void testNegativeHeight () {
  MineSweeperBoard board = new MineSweeperBoard(4, -4, 2);
}

@Test
public void testGetCellInvalid () {
  MineSweeperBoard board = new MineSweeperBoard(4, 4, 2);
  assertEquals("INVALID_CELL", board.getCell(0, -1));
}

@Test
public void testnumberOfAdjacentMines () {
  MineSweeperBoard board = new MineSweeperBoard(4, 4, 2);
  board.setCell(0, 1, "M");
  board.setCell(1, 1, "+");
  assertEquals(2, board.numberOfAdjacentMines(0, 0));
}
```

b. (17 points) Now finish an implementation of the `numberOfAdjacentMines` and `uncoverCell` methods, shown in Appendix B. You can use other methods from the `MineSweeperBoard` class even though they are not yet implemented.

NOTE: The methods should *never* throw an `IndexOutOfBoundsException`.

i. (12 points)

```
/**
* Count the number of mines (flagged mine, covered mine, and uncovered mine)
* that appear in cells that are adjacent to the specified cell.
* Adjacent cells include north, south, east, west and diagonal cells.
*
* @param x      the column of the cell.
* @param y      the row of the cell.
**/
public int numberOfAdjacentMines(int x, int y) {
   int numMines = 0;

   for (int dx = -1; dx <= 1; dx++) {
     for (int dy = -1; dy <= 1; dy++) {
         if ((dx != 0 || dy != 0)
            && (getCell(x + dx, y + dy).equals("+")
            || getCell(x + dx, y + dy).equals("M")
            || getCell(x + dx, y + dy).equals("*"))) {
               numMines++;
         }
      }
   }

   return numMines;
}
```

ii. (5 points)

```
/**
 * Uncover the specified cell.
 * If the cell already contains a flag it should not be uncovered.
 * If there is not a mine under the specified cell then the value
 * in that cell is changed to the number of mines that appear in adjacent cells.
 * If there is a mine under the specified cell, that cell is changed to uncovered mine.
 * If the specified cell is already uncovered or is invalid, no change is made to the board.
 *
 * @param x      the column of the cell to be uncovered.
 * @param y      the row of the cell to be uncovered.
 **/
public void uncoverCell(int x, int y) {
   if (getCell(x, y).equals("+")) {
      setCell(x, y, "*");
   } else if (getCell(x, y).equals("O")) {
      int adjMines = numberOfAdjacentMines(x, y);
      setCell(x, y, adjMines));
   }
}
```

5. Tracing Array Code (15 points)

Consider the following algorithm:

```
1  public void f(int[] array) {
2    for (int i = array.length / 2; i > 0; i = i / 2) {
3      for (int j = 0; j < i; j++){
4        g(array, j, i);
5      }
6      // values? – part b.
7    }
8
9    g(array, 0, 1);
10
11 }
12
13 public void g(int[] array, int start, int incr) {
14   for (int i= start + incr; i < array.length; i = i + incr) {
15     for (int j = i; (j >= incr) && (array[j] < (array[j-incr])); j = j - incr){
16       int tmp = array[j];
17       array[j] = array[j-incr];
18       array[j-incr] = tmp;
19     }
20   }
21 }
```

We assume that the array is initialized as follows:

```
int[] array = {10, 100, 3, 4, 120, 1};
```

a. (3 points) List all the possible values of `incr` in `g()` given f is called with the array initialized above:

_3, 1_

b. (12 points) Below, provide the contents of `array` after each iteration of the outer for loop in `f` (i.e., at line number 6).

Iteration 1: `4, 100, 1, 10, 120, 3`

Iteration 2: `1, 3, 4, 10, 100, 120`

A Appendix: 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 ->
      q.head <- qn.next;
      (if qn.next = None then q.tail <- None);
      qn.v
 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 []

let from_list (xs : 'a list) =
  let q = create () in
  List.iter (fun x -> enq x q) xs;
  q
```

B Appendix: MineSweeperBoard class documentation

```
/**
 * A MineSweeper board represented by a 2D array of Strings.
 *
 * A cell can have one of the following state
 * ''O'' = covered cell
 * ''F'' = flag
 * ''M'' = flagged mine
 * ''+'' = covered mine
 * ''*'' = uncovered mine (about to explode!)
 *  ''1''..''8''    = number of adjacent mines
 *
 */
public class MineSweeperBoard {


    /**
    * Creates new Mine Sweeper Board
    * @param width - The width of the board
    * @param height - The height of the board
    * @param numberOfMines - The number of mines in the board
    *
    * @throws IllegalArgumentException if width is negative or less than 2
    * @throws IllegalArgumentException if height is negative or less than 2
    * @throws IllegalArgumentException if numberOfMines is negative
    *
    */
    public MineSweeperBoard(int width, int height, int numberOfMines) { ... }

    /**
    * Get the number of columns in this MineSweeperBoard.
    *
    * @return the number of columns in this MineSweeperBoard.
    */
    public int width() { ... }


    /**
    * Get the number of rows in this MineSweeperBoard.
    *
    * @return the number of rows in this MineSweeperBoard
    */
    public int height() { ... }

    /**
    * Get the contents of the specified cell on this MineSweeperBoard.
    *The value returned from this method must be one of the values listed above.
    *
    * @param x - The column containing the cell.
    * @param y - The row containing the cell.
```

```
    * @return the value contained in the cell specified by x and y, or
    * "INVALID_CELL" if the specified cell does not exist.
    */
    public String getCell(int x, int y) { ... }


    /**
    * Set the contents of the specified cell on this MineSweeperBoard.
    *The value passed in should be one of the values listed above
    *
    *@param x − The column containing the cell
    *@param y − The row containing the cell
    *@param value − The value to place in the cell
    */
    public void setCell(int x, int y, String value) { ... }

    /**
    * Uncover the specified cell.
    * If the cell already contains a flag it should not be uncovered.
    * If there is not a mine under the specified cell then the value
    * in that cell is changed to the number of mines that appear in adjacent cells.
    * If there is a mine under the specified cell, that cell is changed to uncovered mine.
    * If the specified cell is already uncovered or is invalid, no change is made to the board.
    *
    * @param x the column of the cell to be uncovered.
    * @param y the row of the cell to be uncovered.
    */
    public void uncoverCell (int x, int y) { ... }

    /**
    * Count the number of mines (flagged mine, covered mine, and uncovered mine)
    * that appear in cells that are adjacent to the specified cell.
    * Adjacent cells include north, south, east, west and diagonal cells.
    *
    * @param x the column of the cell.
    * @param y the row of the cell.
    */
    public int numberOfAdjacentMines (int x, int y) { ... }


}
```