CIS 120 Midterm II March 23, 2018

SOLUTIONS

1. **OCaml and Java** (8 points) Indicate whether the following statements are true or false.

a. True ☒ False ☐

The following OCaml function is tail-recursive:

```
let rec g x1 x2 =
 begin match x1 with
  | [] -> x2
  | hd::tl -> if hd < 20 then g tl (hd::x2) else g tl x2
```

This function is tail recursive because nothing needs to be saved on the workspace in a recursive call to `g`*. In otherwords, the result of the recursion is not used for additional computation.*

b. True ☒ False ☐

In OCaml, if `s` and `t` are variables of type string, such that `s = t` returns **`false`**, then `s == t` is guaranteed to return **`false`**. *This answer is true because strings that are do not contain the same characters (tested by =) cannot be stored in the same heap location (tested by ==).*

c. True ☐ False ☒

In OCaml, if `s` and `t` are variables of type string, such that `s == t` returns **`false`**, then `s = t` is guaranteed to return **`false`**. *This problem was removed from the exam.*

d. True ☐ False ☒

Records are never mutable in OCaml.

e. True ☒ False ☐

In the OCaml ASM, first-class functions are stored in the heap and may have local copies of variables that were on the stack when they were defined.

f. True ☐ False ☒

Variables stored on the stack in the OCaml Abstract Stack Machine are mutable.

g. True ☒ False ☐

Variables stored on the stack in the Java Abstract Stack Machine are mutable.

h. True ☒ False ☐

In the Java Abstract Stack Machine, the components of arrays stored on the heap are mutable, but their length is not.

2. Queue implementation (20 points)

This problem concerns the OCaml implementation of mutable queues, appearing in Appendix A.

Complete the implementation of a function `double_only_if` that, when given a function and a queue, doubles the elements in the queue for which the function returns true.

This function should have the following type:

```
val double_only_if : ('a -> bool) -> 'a queue -> unit
```

For example, if the queue `q` contains the elements `1,2` (in that order), then a call `double_only_if (fun x -> true) q` should modify `q` so that it contains the elements `1,1,2,2` (in that order). If the call was instead `double_only_if (fun x -> false) q` then `q` should be left unmodified. More test cases demonstrating the behavior of this function appear below.

The `double_only_if` function must preserve the queue invariant of its input.

Your answer **may not** use any function in Appendix A or any list library function, including `enq`, `length` or `deq`.

```
;;run_test "double all" (fun () ->
  let q = from_list [1; 2;] in
  (double_only_if (fun x -> true) q; to_list q = [1; 1; 2; 2]))

;;run_test "double none" (fun () ->
  let q = from_list [1; 2] in
  (double_only_if (fun x -> false) q; to_list q = [1;2]))

;;run_test "double first" (fun () ->
  let q = from_list [1; 2; 3] in
  (double_only_if (fun x -> x = 1) q; to_list q = [1; 1; 2; 3]))

;;run_test "double mid" (fun () ->
  let q = from_list [1; 2; 3] in
  (double_only_if (fun x -> x = 2) q; to_list q = [1; 2; 2; 3]))

;;run_test "double last" (fun () ->
  let q = from_list [1; 2; 3] in
  (double_only_if (fun x -> x = 3) q; to_list q = [1; 2; 3; 3]))
```

Hint: This problem is **much** easier if you add the `qnode` with the duplicate value **after** the original qnode in the queue.

(Use the next page for your answer.)

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

3. ASM, structural and reference equality (24 points total)

Consider the code and ASM shown in Appendix B on page 14.

a. (2 points) Does `q` satisfy the queue invariant given in class?

☒ **yes** ☐ **no**

b. (10 points) For each of the following expressions, use the ASM diagram to circle whether they evaluate to **`true`**, **`false`**, *loop forever*, or *do not type check*.

i. `qn1.next == qn2.next`

☐ **`true`** ☒ **`false`** ☐ *loops forever* ☐ *doesn't typecheck*

ii. `q.tail = qn2.next`

☒ **`true`** ☐ **`false`** ☐ *loops forever* ☐ *doesn't typecheck*

iii. `qn2 == q.head`

☐ **`true`** ☐ **`false`** ☐ *loops forever* ☒ *doesn't typecheck*

This test doesn't type check because `qn2` *has type* `int qnode` *and* `q.head` *has type* `int qnode option`.

iv. `q.head == q.tail`

☐ **`true`** ☒ **`false`** ☐ *loops forever* ☐ *doesn't typecheck*

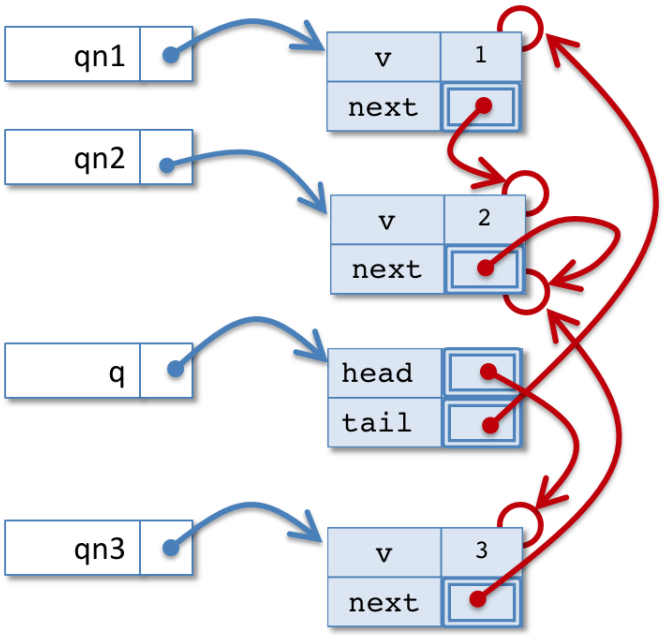
v. `q = q`

☒ **`true`** ☐ **`false`** ☐ *loops forever* ☐ *doesn't typecheck*

c. (10 points) Now, suppose the following code is placed on the workspace, starting from the configuration shown in the diagram in Appendix B on page 14.

```
let qn3 : int qnode = { v = 3; next = Some qn2 } in
qn1.next <- Some qn2;
qn2.next <- qn3.next;
q.head <- Some qn3
```

Complete the missing parts of the diagram below, showing the final state of the stack and heap after these operations have executed. You may need to draw references from the original configuration if they are unchanged. Don't forget to draw your "Some bubbles" clearly!



d. (2 points) Does `q` satisfy the queue invariant given in class after this code has executed?

☐ **yes** ☒ **no**

4. OCaml objects (20 points)

You're trying to write OCaml code that implements counters objects as we discussed in class. The type of counter objects should be

```
type counter = {
   get  : unit -> int; (* get the current value of the counter *)
   incr : unit -> unit; (* increment the counter *)
   decr : unit -> unit; (* decrement the counter *)
   reset : unit -> unit; (*reset the counter to 0 *)
}
```

You intend to write a function `new_counter () : counter` that creates a new counter, such that each counter keeps a count separate from other counters. You write the following code for `new_counter`, but it seems to be buggy.

```
1 type counter_state = { mutable count : int }
2 let ctr = {count = 0}
3 let new_counter () : counter =
4   {
5    get  = (fun () -> ctr.count) ;
6    incr = (fun () -> ctr.count <- ctr.count + 1) ;
7    decr = (fun () -> ctr.count <- ctr.count - 1) ;
8    reset = (fun () -> ctr.count <- 0) ;
9   }
```

a. If the following code appears immediately after line 9, what value does it return?

```
let count1 = new_counter() in
  count1.incr();
  count1.incr();
  count1.decr();
  count1.get ()
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ *None of the above*

b. If the following code appears immediately after line 9, what value does it return?

```
let count1 = new_counter() in
let count2 = new_counter() in
  count1.incr();
  count1.incr();
  count2.incr();
  count1.decr();
  count1.get ()
```

☐ 0 ☐ 1 ☒ 2 ☐ 3 ☐ *None of the above*

c. If the following code appears immediately after line 9, what value does it return?

```
let count1 = new_counter() in
let count2 = new_counter() in
  count1.incr();
  count2.incr();
  count1.decr();
  count2.reset();
  count1.get ()
```

☒ 0 ☐ 1 ☐ 2 ☐ 3 ☐ *None of the above*

d. Fix the bug in the code by editing the code in one of three ways. Either (a) delete a single line, (b) add a single line, or (c) delete a single line and then add a single line of code.

☐ Delete line _____.

☐ Add the following line of code after line _____:

☒ Delete line __2____ and then add the following line of code after line __3____:

New code: `let ctr = {count = 0} in`

The entire listing of the correct code is:

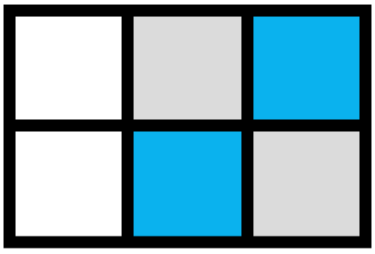
```
type counter_state = { mutable count : int }
let new_counter () : counter =
  let ctr = {count = 0} in
  {
   get  = (fun () -> ctr.count) ;
   incr = (fun () -> ctr.count <- ctr.count + 1) ;
   decr = (fun () -> ctr.count <- ctr.count - 1) ;
   reset = (fun () -> ctr.count <- 0) ;
  }
```

5. Writing and Testing Array Code (28 points)

For this part, you will write Java code and JUnit tests to implement and test a `crop` method for the Pennstagram assignment. This method takes a `PixelPicture` image and the coordinates of a rectangle within that image and returns a new `PixelPicture` containing just those pixels. For reference, documentation on the `PixelPicture` class appears in Appendix C.

```
/** Copy part of a picture specified by a rectangular region
 *
 * @param original – Picture to crop
 * @param x – The x–coordinate of the upper–left corner of the rectangle to be copied
 * @param y – The y–coordinate of the upper–left corner of the rectangle to be copied
 * @param w – Desired width of the new picture
 * @param h – Desired height of the new picture
 *
 * @throws IllegalArgumentException if original is null
 * @throws IllegalArgumentException if the upper–left coordinate is not contained in the image
 * @throws IllegalArgumentException if desired width or height is not positive
 * @throws IllegalArgumentException if the returned picture would have no pixels
 */
public static PixelPicture crop (PixelPicture original, x, y, w, h)
```

For example, if `original` is a 3 pixel by 2 pixel image that looks like this,



then the line

```
PixelPicture cropped = crop(original,0,1,3,1);
```

should produce a 3 pixel by 1 pixel image like this.

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

We can test the behavior of `crop` using the following JUnit test that refers to the two `PixelPicture`s shown on the previous page.

```
// Copy 3 by 1 image out of 3 by 2 image
@Test
public void testExampleCrop() {
  PixelPicture result = crop(original, 0, 1, 3, 1);
  assertEquals("crop", 0, PixelPicture.diff(result, cropped));
}
```

Note, the width and height of the rectangle *may* specify a region that extends past the dimensions of the original picture. In this case, the `crop` method should still try to return a result, even though the width and height of the output picture will not be as large as requested. (But note, if the width or height of the output picture would be zero, then the method should throw an `IllegalArgumentException`).

```
// Requested crop rectangle is too large, result is still 3 by 1
@Test
public void testLargeCrop() {
  PixelPicture result = crop(original, 0, 1, 100, 100);
  assertEquals("crop", 0, PixelPicture.diff(result, cropped));
}
```

Finally, we can test the exceptional behavior of this method. The following test succeeds when the `crop` method, as expected, throws an `IllegalArgumentException` on a **`null`** input.

```
// test that null argument throws correct exception
@Test(expected=IllegalArgumentException.class)
public void testNullPicture() {
  PixelPicture result = crop(null, 0, 0, 1, 1);
}
```

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

a. (9 points) Your job is to complete **three** additional tests that that cover situations when this method should throw an `IllegalArgumentException`. Each of these tests uses the `original` picture described above. 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.

3 points each. Any three of the following sorts of answers are acceptable.

```
@Test(expected=IllegalArgumentException.class)
public void testNegativeWidth () {
  PixelPicture result = crop(original, 0, 0, -1, 1);
}
@Test(expected=IllegalArgumentException.class)
public void testNegativeHeight () {
  PixelPicture result = crop(original, 0, 0, 1, -1);
}
@Test(expected=IllegalArgumentException.class)
public void testXLocationNegative () {
  PixelPicture result = crop(original, -1, 1, 1, 1);
}
@Test(expected=IllegalArgumentException.class)
public void testYLocationNegative () {
  PixelPicture result = crop(original, 1, -1, 1, 1);
}
@Test(expected=IllegalArgumentException.class)
public void testLocationOutsideImage () {
  PixelPicture result = crop(original, 4, 4, 1, 1);
}
@Test(expected=IllegalArgumentException.class)
public void testLocationAtEdge () {
  PixelPicture result = crop(original, 3, 2, 1, 1);
}
```

b. (19 points) Now finish an implementation of the `crop` method, shown in Appendix D.

NOTE: The `crop` method should *never* throw an `IndexOutOfBoundsException` or `NullPointerException`. Furthermore, your answer should not make unnecessary copies of the data in the input image.

In your answers, you may use the method declared below.

```
public static int min(int a, int b) // returns smaller of a and b
```

i. Fill in the condition for blank (a) to test for invalid inputs. Be sure to test for *all* situations described in the problem description.
5 points total.

- x and y less than zero
- x and y greater than or equal to pic width and height
- w and h less than or equal to zero

ii. What should go in blank (b) ?
1 point, no partial credit. `min (w, pw - x)`

iii. What should go in blank (c) ?
1 point, no partial credit. `min (h, ph - y)`

iv. What should go in blank (d) ? Your answer may include multiple lines of code.

```
for (int i=0; i < newData.length ; i++) {
  for (int j=0; j < newData[0].length; j++) {
    newData[i][j] = data [x+i][y+j];
  }
}
```

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: Example Abstract Stack Machine Diagram

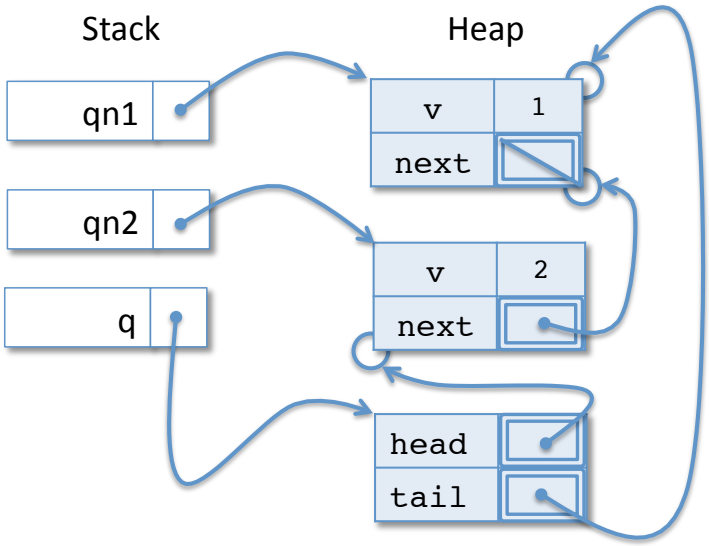
An example of the Stack and Heap components of the OCaml Abstract Stack Machine. Your diagram should use similar "graphical notation" for `Some v` and `None` values.

```
(* The types for mutable queues. *)
type 'a qnode = { v : 'a; mutable next : 'a qnode option; }

type 'a queue = {
  mutable head : 'a qnode option;
  mutable tail : 'a qnode option;
}

let qn1 : int qnode = {v = 1; next = None}
let qn2 : int qnode = {v = 2; next = Some qn1}
let q : int queue = {head = Some qn2; tail = Some qn1}
(* HERE *)
```

The OCaml program above yields the ASM Stack and Heap depicted below when the program execution reaches the point marked *(* HERE *)*.



C Appendix: PixelPicture class documentation

```
/**
 * An image represented by a 2D array of Pixels.
 *
 * PixelPictures are immutable. Although they provide access to a 2D
 * array of pixels, this array is a copy of the one stored in the NewPic.
 * The original image cannot be modified.
 */
public class PixelPicture {
  /**
     * Creates a picture given a bitmap. The bitmap should be in left-to-right,
     * top-to-bottom ordering.
     *
     * @param bmp The bitmap
     */
    public PixelPicture(Pixel[][] bmp) { ... }

   /**
     * Get the width of the image.
     */
    public int getWidth() { ... }

    /**
     * Get the height of the image.
     */
    public int getHeight() { ... }

   /**
     * Gets a bitmap (i.e., matrix of pixels) of the image. This method returns
     * a copy of the image's contents---editing the returned bitmap will not
     * affect the NewPic.
     *
     * The bitmap is in a left-to-right, top-to-bottom order. The first index is
     * the row, the second index is the column.
     *
     * @return a left-to-right, top-to-bottom array of arrays of Pixels
     */
    public Pixel[][] getBitmap() { ... }

   /**
     * Compute the difference between two images.
     *
     * This difference sums the pixel-by-pixel differences
     * between components of a pixel. It is most useful
     * for SMALL images.
     */
    public static int diff(PixelPicture p0, PixelPicture p1) { ... }

}
```

D Appendix: crop method

```
/** Copy part of a picture specified by a rectangular region
 *
 * @param original − Picture to crop
 * @param x − The x−coordinate of the upper−left corner of the rectangle to be copied
 * @param y − The y−coordinate of the upper−left corner of the rectangle to be copied
 * @param w − Desired width of the new picture
 * @param h − Desired height of the new picture
 *
 * @throws IllegalArgumentException if original is null
 * @throws IllegalArgumentException if the upper−left coordinate is not contained in the image
 * @throws IllegalArgumentException if desired width or height is not positive
 * @throws IllegalArgumentException if the returned picture would have no pixels
 */
public static PixelPicture crop(PixelPicture pic, int x, int y, int w, int h) {
     if (pic == null) {
       throw new IllegalArgumentException();
     }

     int pw = pic.getWidth();
     int ph = pic.getHeight();
     Pixel[][] data = pic.getBitmap();

     // test for invalid inputs

     if ( _______(a)________ ) {

       throw new IllegalArgumentException();
     }

     // Allocate the bitmap for the returned image

     Pixel[][] newData = new Pixel[__(b)__][__(c)__];

     // Copy into the bitmap here




     ______(d)______



     // return final image

     return new PixelPicture(newData);
}
```