CIS 120 Midterm II March 31, 2017

SOLUTIONS

1. **OCaml and Java** (14 points) Check one box for each part.

a. The following OCaml function will terminate by exhausting stack space if called as `loop 10`:

```
let rec loop m =
  if m = 0 then 1 else loop (m+1)
```

☐ True ☒ False

b. The following OCaml function is tail-recursive:

```
let rec g x1 x2 =
  begin match x1 with
  | [] -> x2
  | hd::tl ->
     if hd > 10 then g tl (hd::x2)
                else hd :: (g tl x2)
  end
```

☐ True ☒ False

c. 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`.

☐ True ☒ False

d. In the OCaml GUI library, a `change_listener` is a first-class function stored in the hidden state of a `value_controller` widget. Every few milliseconds, the value controller invokes all of the stored `change_listeners`.

☐ True ☒ False

e. In the Java ASM, a static method dispatch `C.m()` implicitly pushes a `this` reference onto the stack.

☐ True ☒ False

f. Casting a variable to class `A` changes both its static and dynamic types to `A`.

☐ True ☒ False

g. The Java compiler can sometimes tell at compile time whether a cast will succeed.

☒ True ☐ False

2. Queues and the ASM

Page 1 in the appendix shows several possible states of the OCaml ASM where a variable `q` on the stack points to an `int queue` record in the heap. (For reference, page 2 in the appendix repeats the definitions of the basic types and operations for linked mutable queues.)

a. (12 points) In which of these ASM states does `q` satisfy the queue invariant?

☒ A satisfies queue invariant
☐ A does not satisfy queue invariant

☐ B satisfies queue invariant
☒ B does not satisfy queue invariant

☐ C satisfies queue invariant
☒ C does not satisfy queue invariant

☒ D satisfies queue invariant
☐ D does not satisfy queue invariant

☐ E satisfies queue invariant
☒ E does not satisfy queue invariant

☐ F satisfies queue invariant
☒ F does not satisfy queue invariant

b. (16 points) For each of the following code snippets, indicate which (if any) of the above ASM shapes it produces. (Note that some of the snippets bind variables besides `q`; we are omitting all of these variables from the stacks in the ASM diagrams above and focusing just on `q` and the state of the heap.)

i.
```
let q = create ()
;; enq 2 q
;; enq 1 q
let _ = deq q
```

☐ A ☐ B ☐ C ☐ D ☐ E ☐ F

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

☒ A ☐ B ☐ C ☐ D ☐ E ☐ F

iii.
```
let q = create ()
let qn1 = {v = 1; next = None}
let qn2 = {v = 2; next = Some qn1}
;; q.tail <- Some qn2
;; q.head <- Some qn1
```

☐ A ☐ B ☒ C ☐ D ☐ E ☐ F

iv.
```
let q = create ()
let qn1 = {v = 1; next = None}
let qn2 = {v = 2; next = Some qn1}
;; q.head <- Some qn1
;; q.tail <- qn2.next
```

☐ A ☐ B ☐ C ☒ D ☐ E ☐ F

3. Translating Java objects to OCaml

The two Java classes on page 5 of the appendix implement simple "tickets," each with a color and a number, and "ticket machines" that print tickets. The `get` method of a given `TicketMachine` object produces tickets of a single color, which is chosen when the `TicketMachine` is created. The numbers of the tickets produced by a `TicketMachine` always start at 0 and increase each time its `get` method is called.

a. (4 points) Fill in the blanks so that the test passes:

Answer:

```
public class TicketMachineTest {

  @Test
  public void test() {
    TicketMachine m1 = new TicketMachine();
    TicketMachine m2 = new TicketMachine();
    TicketMachine m3 = new TicketMachine();

    Ticket t1 = m1.get();
    Ticket t2 = m1.get();
    Ticket t3 = m2.get();
    Ticket t4 = m3.get();

    assertEquals(t1.getColor(), "red");
    assertEquals(t1.getNumber(), 1);
    assertEquals(t2.getColor(), "red");
    assertEquals(t2.getNumber(), 2);
    assertEquals(t3.getColor(), "blue");
    assertEquals(t3.getNumber(), 1);
    assertEquals(t4.getColor(), "red");
    assertEquals(t4.getNumber(), 1);
  }
}
```

b. (22 points)

Fill in the blanks in the following OCaml translation of the `ticketMachine` class. (We've done the `ticket` class for you.)

Answer:

```
type ticket = {getColor: unit->string; getNumber: unit->int}

let mk_Ticket (c:string) (n:int) : ticket =
  { getColor = (fun () -> c);
    getNumber = (fun () -> n) }

type ticketMachine = {get: unit->ticket}

let nextColor = { contents = "red" }

let bumpColor c = if c = "red" then "blue" else "red"

let mk_TicketMachine () : ticketMachine =
  let myColor = !nextColor in
  nextColor := bumpColor (!nextColor);
  let nextNumber = { contents = 0 } in
  { get = (fun () ->
           nextNumber := !nextNumber + 1;
           mk_Ticket myColor !nextNumber)
  }
```

4. Java: Subtyping and Inheritance

This question concerns the Java code for the "shapes" hierarchy of interfaces and classes that we saw in lecture. This code can be found on page 3 in the appendices, for your reference.

Consider the following Java (type-correct) code that adds two new classes:

```
1  public class Square extends Rectangle {
2    public Square(double x, double y, double size) { super(x,y,size,size); }
3  }
4
5  public class Midterm2 {
6    public static void main(String args[]) {
7      Circle c = new Circle(0, 0, 10);
8      Displaceable d = c;
9      Point p = new Point(10, 0);
10     Shape s = c.getBoundingBox();
11     d.move(c.getRadius(), p.getY());
12     if (d.getX() > 5.0) {
13       s = new Square(0,0,25);
14     } else {
15       s = c;
16     }
17     s.move(5, 0);
18
19   }
20 }
```

Choose *one or more* answers for each of the following:

a. (2 points) What is the static type of the variable `s` on line 17?

☐ Circle ☐ Displaceable ☐ Point
☐ Rectangle ☒ Shape ☐ Square

b. (2 points) What is the dynamic class of the variable `s` on line 17?

☐ Circle ☐ Displaceable ☐ Point
☐ Rectangle ☐ Shape ☒ Square

c. (2 points) Does the program typecheck if we insert the following code at line 18?

```
Area a = new Displaceable();
```

☐ Yes, this change is OK.

☐ No: There is a type error on line 18 because Area is not a subtype of Displaceable.

☒ No: There is a type error on line 18 because Displaceable is not a subtype of Area.

☒ No: There is an error on line 18 because Displaceable is an interface not a class.

Note: this question has two correct answers. Two points were given for choosing both, one point for just one.

d. (2 points) Does the program typecheck if we change line 9 to instead read:

```
Shape p = new Point(10, 0);
```

☐ Yes, this change is OK.
☒ No: There is a type error on line 9 because Point is not a subtype of Shape.
☐ No: There is a type error on line 9 because Shape is not a subtype of Point.
☐ No: There is a type error on line 11 when we try to use `p.getY()`.

e. (2 points) Does the program typecheck if we change line 8 to instead read:

```
Shape d = c;
```

☒ Yes, this change is OK.
☐ No: There is a type error on line 8 because Displaceable is not a subtype of Shape.
☐ No: There is a type error on line 8 because Circle is not a subtype of Shape.
☐ No: There is a type error on line 11 when we try to use `d.move(...)`.

5. **Java Array Programming** John Conway's *Game of Life* is a famous example of a *cellular automaton*—a "zero-player" game, where the evolution of the board is completely determined by its initial state. A "player" of the game simply creates an initial board configuration, from which the board evolves, step by step, following a simple set of rules.

   The game board consists of a rectangular 2d array of integers. At any given moment, each cell on the board is said to be either *alive* if its contents are 1 or *dead* if its contents are 0.

   On each *step* of the game, the new contents of each cell are calculated using its old contents together with the contents of its 8 nearest neighbors (not including the cell itself). Cells on the edge of the board will have fewer than 8 neighbors, but the rules for them are otherwise the same as for other cells.

   - If the cell is currently alive, then
     - if less than two of its neighbors are currently alive, then on the next state the cell will be dead (of loneliness);
     - if more than three of its neighbors are currently alive, then on the next state the cell will be dead (of suffocation);
     - if either two or three of its neighbors are currently alive, then on the next state the cell will be alive.
   - If the cell is currently dead, then
     - if exactly three of its neighbors are currently alive, then on the next step the cell will be alive (newly born).

   The following method encodes this logic:

```
private static int liveOrDie (int currCell, int countOfNeighbors) {
  if (currCell == 0) {
    if (countOfNeighbors == 3) return 1;
    else return 0;
  } else {
    if (countOfNeighbors < 2 || countOfNeighbors > 3) return 0;
    else if (countOfNeighbors == 3) return 1;
    else return currCell;
  }
}
```

   Your task in this problem will be to complete the definition of a static Java method `step`, which calculates the next state of a Life board from a current state.

a. (4 points) Here are two test cases that demonstrate the intended behavior of `step`. (For JUnit experts: the `assertArrayEquals2` method is something we built ourselves using JUnit's 1-d `assertArrayEquals` method; its behavior is what you'd expect.)

```
@Test
public void step1() {
  int[][] current = {
     {0, 0, 0, 0, 0},
     {0, 0, 0, 0, 0},
     {0, 1, 1, 1, 0},
     {0, 0, 0, 0, 0},
     {0, 0, 0, 0, 0},
  };
  int[][] next = {
     {0, 0, 0, 0, 0},
     {0, 0, 1, 0, 0},
     {0, 0, 1, 0, 0},
     {0, 0, 1, 0, 0},
     {0, 0, 0, 0, 0},
  };
  assertArrayEquals2
    (next, step(current));
}
```

```
@Test
public void step2() {
  int[][] current = {
     {1, 1, 1, 1, 1},
     {0, 0, 1, 0, 0},
     {0, 0, 1, 0, 0},
     {0, 1, 1, 1, 0},
     {0, 0, 0, 0, 0},
  };
  int[][] next = {
     {0, 1, 1, 1, 0},
     {0, 0, 0, 0, 0},
     {0, 0, 0, 0, 0},
     {0, 1, 1, 1, 0},
     {0, 0, 1, 0, 0},
  };
  assertArrayEquals2
    (next, step(current));
}
```

Fill in the expected result in the test case below so that the test passes.

Answer:

```
@Test
public void step2() {
  int[][] current = {
     {0, 1, 0},
     {1, 1, 0},
     {0, 0, 1},
  };
  int[][] next = {
     {1, 1, 0},
     {1, 1, 1},
     {0, 1, 0},
  };
  assertArrayEquals2
    (next, step(current));
}
```

b. (18 points) Complete the definition of `step` below. (You may find the static library methods `Math.min` and `Math.max` useful, but don't worry if you don't end up using them: there are a number of different ways to write a correct solution.) Your solution should call `liveOrDie` at some point.

Answer:

```
public static int[][] step(int[][] current) {
  int width = current.length;
  int height = current[0].length;
  int[][] next = new int[width][height];
  for (int i = 0; i < width; i++) {
    for (int j = 0; j < height; j++) {
      int count = 0;
      for (int x = Math.max(i - 1, 0); x <= Math.min(i + 1, width-1); x++) {
        for (int y = Math.max(j - 1, 0); y <= Math.min(j + 1, height-1); y++) {
          if (i != x || j != y) {
            count += current[x][y];
          }
        }
      }
      next[i][j] = liveOrDie(current[i][j], count);
    }
  }
  return next;
}
```

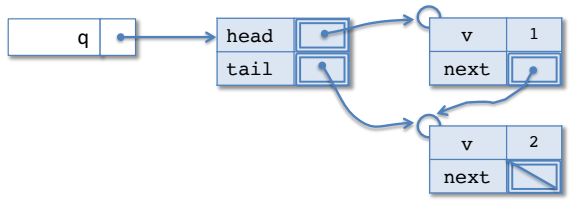
CIS 120 Midterm II Appendices

Do not write answers in this portion of the exam.

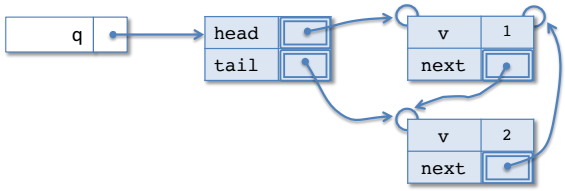
Do not open until the exam begins.

A Appendix: Some ASM Heap States

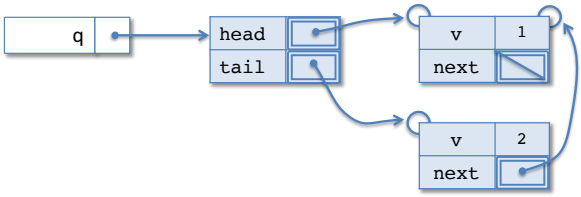
A



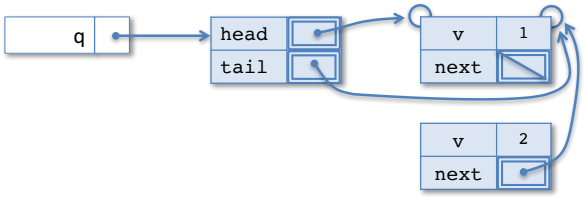
B



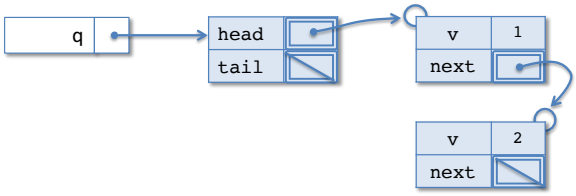
C



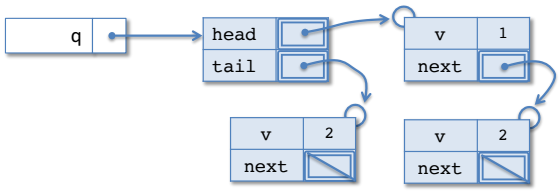
D



E



F



B 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
```

C Appendix: Java "Shapes" Interfaces and Classes

```
public interface Displaceable {
  double getX();
  double getY();
  void move(double dx, double dy);
}

public interface Area {
  double getArea();
}

public interface Shape extends Area, Displaceable {
  Rectangle getBoundingBox();
}

public class Point implements Displaceable {

  private double x;
  private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }

  public void move(double dx, double dy) {
    x = x + dx;
    y = y + dy;
  }
}

public class DisplaceableImpl implements Displaceable {

  private Point pt;

  public DisplaceableImpl(double x, double y) {
    this.pt = new Point(x,y);
  }

  public double getX() { return pt.getX(); }

  public double getY() { return pt.getY(); }

  public void move(double dx, double dy) {
    pt.move(dx, dy);
  }
}
```

```
public class Rectangle extends DisplaceableImpl implements Shape {
  private double w, h;

  public Rectangle(double x, double y, double w, double h) {
    super(x,y);
    this.w = w;
    this.h = h;
  }

  public double getArea() { return w * h; }

  public Rectangle getBoundingBox() {
    return new Rectangle(getX(), getY(), this.w, this.h);
  }

}

public class Circle extends DisplaceableImpl implements Shape {
  private double radius;

  public Circle(double x, double y, double radius) {
    super(x,y);
    this.radius = radius;
  }

  public double getRadius() { return radius; }


  public double getArea() { return Math.PI * this.radius * this.radius; }

  public Rectangle getBoundingBox() {
    return new Rectangle(getX()-radius, getY()-radius,
           2 * radius, 2 * radius);
  }
}
```

D Appendix: Ticket and TicketMachine Definitions

```
public class Ticket {
  private String color;
  private int number;

  public Ticket (String c, int n) {
    this.color = c;
    this.number = n;
  }

  public String getColor() {
    return color;
  }

  public int getNumber() {
    return number;
  }
}


public class TicketMachine {
  private static String nextColor = "red";
  private String myColor;
  private int nextNumber = 0;

  public static String bumpColor (String c) {
    if (c.equals("red")) return "blue";
    return "red";
  }

  public TicketMachine() {
    myColor = nextColor;
    nextColor = bumpColor(nextColor);
  }

  public Ticket get() {
    nextNumber++;
    return new Ticket(myColor, nextNumber);
  }
}
```