CIS 120 Midterm II March 29, 2013

SOLUTIONS

1. Java True/False (20 points)

Circle T or F.

a. [T] F In the Java ASM, object values are stored in the heap.

b. T [F] In the Java ASM, method definitions are stored in the heap.

c. T [F] In the Java ASM, a (nonstatic) method dispatch `o.m()` uses the static type of `o` to determine which version of the method `m` to invoke.

d. [T] F In the Java ASM, a (nonstatic) method dispatch `o.m()` implicitly pushes the `this` reference onto the stack.

e. [T] F Once an array is allocated, its length cannot be changed.

f. [T] F If `x` is an array of length 7, the expression `x[7]` will trigger an `ArrayIndexOutOfBoundsException`.

g. T [F] The expression **`new`** `C()` invokes the `main` method of the class `C`.

h. T [F] Instance variables (aka fields) can never be `null`.

i. [T] F A class may implement multiple interfaces.

j. [T] F `Object` is a superclass of all classes in Java.

2. Java types (20 points)

Consider the following Java class and interface definitions inspired by Homework 7:

```
public interface Transformer {
  public Pixel transformPixel (Pixel p);
}
public class GrayScaleAverage implements Transformer {
  public Pixel transformPixel(Pixel p) { ... }
}
public class Manipulate {
  public static NewPic transform(NewPic p, Transformer t) { ... }
  public static Transformer asTransformer(Transformer t) {
    return t;
  }
}
public class Pixel {
  public Pixel (int r, int g, int b) { ... }
}
public class NewPic {
  public NewPic(String file) { ... }
}
```

Write down a type for each of the following Java variable definitions. Due to subtyping, there may be more than one correct answer. Any correct answer will be accepted. Write **ill-typed** if the compiler would flag a type error for that line (i.e. Eclipse would underline something in red). The first one has been done for you as a sample and is used in the remaining the definitions.

a. `___NewPic__________ p = new NewPic("italy.png");` *// sample*

b. `_____int___________ x1 = 1 / 2;`

c. `_____double________ x2 = 1.0 / 2.0;`

d. `_____int___________ x3 = (int)Math.round(1.0 / 2.0);`

e. `____ill-typed_____ x4 = new Transformer();`

f. `___GrayScaleAverage/Transformer___ x5 = new GrayScaleAverage();`

g. `__Pixel____________ x6 = new Pixel(256,300,300);`

h. `___NewPic_______ x7 = Manipulate.transform(p,new GrayScaleAverage());`

i. `__ill-typed________ x8 = new GrayScaleAverage().transform(p);`

j. `__ill-typed________ x9 = GrayScaleAverage.transformPixel(new Pixel(0,0,0));`

k. `__Transformer_____ x10 = Manipulate.asTransformer(new GrayScaleAverage());`

Grading Scheme: Note: All answers that are not "ill-typed" could also read `Object`.

3. Encapsulation (16 points)

Which of these Java classes completely encapsulate their local state? Circle T if the state can only be modified using the methods of the class or F otherwise. For each class that does not encapsulate its state, give a short code snippet (at most three lines) that demonstrates how its state could be mutated.

a.
```
class A {
  public int x;
  public A() { x = 3; }
  public int get() { return x; }
}
```

T [F]

```
A a = new A()
a.x = 5;
```

b.
```
class B {
  private int[] x;
  public B(int [] x0) { this.x = x0; }
  public int get(int i) { return x[i]; }
}
```

T [F]

```
int[] x = new { 1 };
B b = new B(x);
x[0] = 5;
```

c.
```
class C {
  private A a;
  public C() { a = new A(); }
  public int get() { return a.get(); }
}
```

[T] F

d.
```
class D {
  private A a;
  public D() { a = new A(); }
  public A get() { return a; }
}
```

T [F]

```
D d = new D()
A a = d.get();
a.x = 5;
```

Grading Scheme: Four points each. For F answers, 2 points for the answer, 2 points for the code snippet.

4. OCaml ASM (16 points)

Recall the following definitions of queues in OCaml:

```
(* Mutable queues, as defined in class. *)
type 'a qnode = { v: 'a;
                  mutable next: 'a qnode option}

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

Suppose the OCaml ASM executes starting with the configuration shown on the next page. Modify the stack and heap diagram to show what it will look like at the point in the computation marked *(* HERE *)*.

Note:

- you might need to allocate new heap objects,
- you may need to add "`Some` bubbles" in the appropriate places,
- you might have to move pointers (if so, mark the erased pointers clearly with an X). If your work is not clear, it will not get credit.
- you should show only the final state!
- the Appendix of the exam contains the complete implementation of queues and an example of the stack and heap diagram for an OCaml program.

(The problem continues on the next page.)

The code on the workspace below refers to the following function.

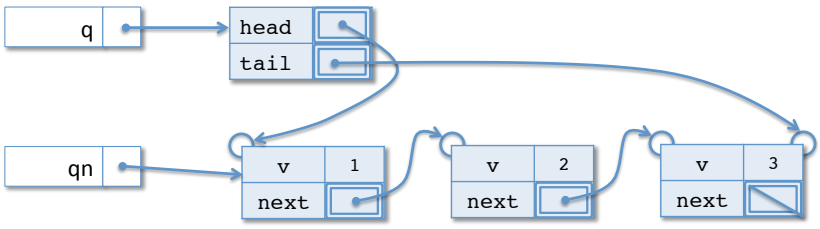
```
let rec g (qno : 'a qnode option) (m : int) : 'a qnode =
  begin match qno with
  | Some qn -> if m = 0 then qn else g qn.next (m-1)
  | None -> failwith "invalid input"
  end
```

Workspace

```
enq q 7;
let qn1 = g q.head 1 in
qn.next <- Some qn;
let qn2 = g q.head 1 in
(* HERE *)
```

Stack

Heap



Final heap:

queueASM-answer-cropped.pdf

Grading Scheme: Points as below. May also deduct up points for extraneous definitions that are not clearly mistakes for these.

a. *3 - Allocate new qnode with v = 7 and next null.*

b. *2 - Change next for qnode 3 to point to Some newnode.*

c. *2 - Change tail to point to Some newnode.*

d. *3 - stack variable qn1 points to qnode 2 (no Some bubble)*

e. *3 - qnode 1's tail points to Some qnode 1 (new Some bubble)*

f. *3 - stack variable qn2 points to qnode 1 (no Some bubble)*

5. Program Design (28 points total)

Implement a function, called `dedup`, which removes all adjacent repeated elements from a queue. In other words, this function should remove any value from the queue that is equal to the value that occurs immediately before it.

For example, if a queue `q` contains the values `1,2,2,3,3,2` (in that order) then after an execution of `dedup q`, the queue `q` should contain `1,2,3,2` (in that order), where the second `2` and the second `3` have been removed.

a. (0 points) The first step is to *understand the problem*. There is nothing to write here—your answers to the other parts will show how well you have accomplished this step.

b. (3 points) The next step is to *define the interface*. Write the type of the operation as you might see it in a .mli file.

```
val dedup : __________ 'a queue -> unit _______________
```

Grading Scheme: -1 for incorrect argument, -1 for more than one argument, -1 for return anything other than `unit`.

c. (8 points) The next step is to *write test cases*. For example, one possible test case is derived from the example in the program description. (This test case uses the queue operations `create`, `enq`, and `to_list`, defined in the appendix.)

```
let test () : bool =
  let q = create () in
  enq q 1;
  enq q 2;
  enq q 2;
  enq q 3;
  enq q 3;
  enq q 2;
  dedup q;
  to_list q = [1;2;3;2]
;; run_test "Given from problem description" test
```

Now, complete **two** more test cases for this method, accompanied by a short explanation of why they are interesting. **You will LOSE points on this problem if you don't include the explanation.**

i. Interesting test cases include

- empty queue (queue should be unchanged)
- no duplicates (queue should be unchanged)
- multiple duplicates in a row 1,1,1 to 1
- duplicate at the beginning 1,1,2 to 1
- duplicated last element 1,2,2 to 1,2
- multiple instances of duplicates 1,1,2,1,1 to 1,2,1

Grading Scheme: 4 points per test case: 1 point for creating a queue (could be blank if intentionally empty), 1 points for result of `to_list`*, 2 points for explanation. If the test case is not "interesting" then -2 points.*

d. (17 points) The final step is to *implement the function*. You implementation should not allocate any new `qnode`s, it should only remove the appropriate nodes from the queue. Therefore, you **cannot use `enq`**. We've given you a hint by providing the interface to a `helper` function that you should use to traverse the queue. The arguments to this helper function should be the current node of the traversal and a reference to the node after it in the queue (if there is one).

```
let dedup (q : 'a queue) : unit =
      let rec helper (curr : 'a qnode) (nxt : 'a qnode option) : unit =
       begin match nxt with
            | Some n -> if curr.v = n.v then
                     (curr.next <- n.next;
                          if curr.next = None then
                                    q.tail <- Some curr
                               else
                          helper curr curr.next)
                  else
                          helper n n.next
            | None -> ()
            end
      in begin match q.head with
      | Some qn -> helper qn qn.next
      | None -> ()
      end
```

Grading Scheme:

- *1 - pattern match for nxt*
- *2 - checking that curr.v = n.v (using either = or ==)*
- *2 - checking whether to update the tail (using either = or ==)*
- *2 - updating the tail to the correct node (-1 if forgot Some)*
- *2 - recursive call in the deletion case*
- *2 - recursive call in the nondeletion case*
- *2 - pattern match q.head*
- *2 - initial call to helper*
- *2 - empty queue should do nothing (-1 for failwith)*
- *2 - set curr.next to n.next*
- *other errors at discrection*

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 ->
        begin match q.tail with
        | Some qn2 ->
          if qn == qn2 then
            (* deq from 1-element queue *)
            (q.head <- None;
            q.tail <- None;
            qn2.v)
          else
            (q.head <- qn.next;
            qn.v) (* Make sure to use parens around ; expressions. *)
        | None -> failwith "invariant violation"
        end
     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 []
```

Appendix: 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 of 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 *)*.