CIS 120 Midterm II November 8, 2013

SOLUTIONS

1. Facts about OCaml and Java (15 points)

For each part, circle true or false.

a. [T] F The `.equals` method in Java is roughly similar to OCaml's = operator.

b. T [F] The == operator in both OCaml and Java tests whether two compound values have identical structure.
No: == is reference equality in both languages.

c. T [F] Both Java and OCaml provide generic types. OCaml also supports subtyping, while Java does not.
No: It's the other way round.

d. T [F] In Java, there is a class that is a subtype of any other class.
No: The class hierarchy is tree structured.

e. [T] F In Java, an interface can extend zero, one, or several other interfaces.

f. T [F] In Java, a class can extend zero, one, or several other classes.
No: Only interfaces.

g. T [F] In the Java ASM, large data structures such as object values are stored in the stack, not the heap.
No: In both ASMs, large structures live in the heap.

h. [T] F In the OCaml ASM, bindings of variables to values in the stack are immutable, while in Java they are mutable.

i. [T] F A Java variable of type `String` behaves like an OCaml variable of type `string option ref`.

j. T [F] A static method in Java may refer to an automatically bound variable `this`.
No: Static methods have no `this`.

k. T [F] In Java, the static type of a variable is always a subtype of the dynamic class of the object value that it refers to.
No: A supertype.

l. T [F] In OCaml, mutable record fields may contain either a proper value or `null`.
No: `null` *is a Java thing.*

m. T [F] A Java array can be resized by assigning a new value to its `length` field.
No: Arrays have a fixed size.

n. [T] F OCaml's anonymous functions can be approximated in Java by using a class with one method.

o. T [F] Recursive functions cannot be defined in Java.
No: They can. (They may use too much stack space on large inputs, but this is a separate issue.)

2. Java Jargon (8 points)

Briefly (two sentences max) define the phrase "dynamic dispatch" as it applies to Java.

This phrase refers to the fact that the result of invoking a method on an object depends on the object's dynamic class, not its static type.

Grading Scheme: Important ideas that we looked for in an answer:

- *method invocation*
- *behavior (or "code that gets run")*
- *dynamic class*
- *static type*

Point Breakdown:

- *-4 for no mention of the dynamic class's method being called*
- *-2 for none of the following being described (only one is required for full points):*
  - *Compile time vs. Runtime checking*
  - *Static vs. Dynamic type*
  - *The class table*
- *-2 At grader's discretion for other errors (e.g. false claims about Java that were incidental to a correct explanation of dynamic dispatch)*

3. Subtyping, Interfaces and Static Types (20 points)

On page 11 in the Appendix, you will find the code for Java interfaces `Widget`, `LabelController`, and `Gctx` and classes `Label` and `Empty`, loosely inspired by Homework 8, plus the class `Foo` defining three static methods.

a. For each variable, fill in its *static type* and the *dynamic class* of the object that it refers to in the `main` method. Note that the type declarations for `a4`, `a5` and `a6` have been intentionally omitted. There are several static types that could be used to make these declarations work - use the most specific one.

variable	static type	dynamic class
a1	Widget	Label
a2	LabelController	Label
a3	Object	Empty
a4	Widget	Label
a5	LabelController	Label
a6	Widget	Empty

b. Which of these statements/declarations would type check if added to the end of the `main` method? Circle GOOD if the line would not cause a compile-time error, and TYPE ERROR otherwise.

a3 = l;	[GOOD]	TYPE ERROR
a2 = a4;	GOOD	[TYPE ERROR]
Widget x = Foo.asWidget(a1);	[GOOD]	TYPE ERROR
Object y = Foo.asWidget(a2);	GOOD	[TYPE ERROR]
Object z = Foo.asLabelController(e);	GOOD	[TYPE ERROR]
a2.repaint(new Gctx())	GOOD	[TYPE ERROR]
a2.setLabel(“CIS 120 rocks!”);	[GOOD]	TYPE ERROR
a4.setLabel(“CIS 120 rocks!”);	GOOD	[TYPE ERROR]

4. Objects in OCaml (12 points)

Consider the following Java class:

```
class Bar {
   private int a;
   private int b;
   public Bar(int x) { a = 120; b = x }
   public int get() { return a * b; }
   public void set(int x) {a = x};
}
```

We can encode `Bar` objects as values belonging to an OCaml record type with two fields, `get` and `set`, corresponding to the methods of the same name above.

```
type m = { get : unit -> int;
           set : int -> unit }
```

Your task in this problem is to implement a "constructor" for such objects—a function `create` that builds values belonging to this record type. For example, the result of the following program should be 42.

```
let obj = create 6 in
obj.set(7);
obj.get()
```

Complete the definition of `create` below to achieve this behavior.

```
let create (x : int) : m =
```

Answer:

```
let a = {contents = 120} in
{ get = (fun () -> !a * x );
  set = (fun y -> a.contents <- y) }
```

5. OCaml Queue + ASM (20 points)

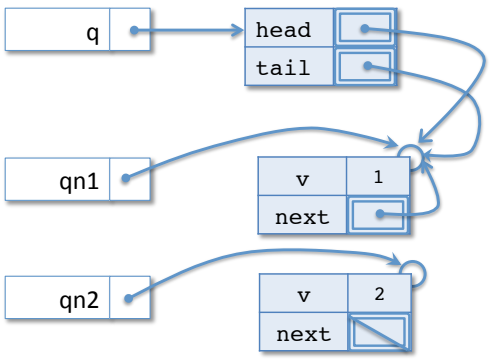
Recall the following definitions of queues in OCaml:

```
(* Data structure for 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 }
```

a. Suppose we have the OCaml ASM shown below. Write a short piece of code that can get the ASM to this state.

(For reference, we provide you with the code for the queue from homework 5 in the appendix, page 12. You are free to call these functions as part of your solution to this part, or not, as you prefer; correct solutions can be written both ways. However, note that the heap structure we've given you does *not* satisfy the queue invariant, so your solution clearly cannot consist *only* of calls to the queue functions, which always maintain the invariant.)



Answer:

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

Suppose we start executing from the ASM configuration shown on the previous page. Fill in the template stack and heap diagram below to show what it will look like at the point in the computation marked *(∗ HERE ∗)*. (Notice that this code calls a queue function even though our initial configuration does *not* satisfy the queue invariant!)

Note:

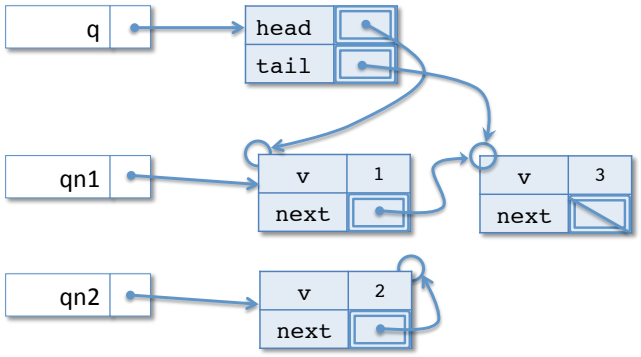
b.

- You should show only the final state!
- You might need to allocate new heap objects.
- You may need to add "`Some` bubbles" in the appropriate places,
- The Appendix of the exam contains the complete implementation of queues and an example of the stack and heap diagram for an OCaml program.
- Make sure your work is clear! If your work is not clear, it will not get credit.

Code

```
qn2.next <- Some qn2;
enq q 3;
(* HERE *)
```

Answer (extend or modify the diagram below, as appropriate):



c. Is the queue invariant satisfied for queue `q` at the point in the computation marked *(∗ HERE ∗)* of part 5b? (Just write "yes" or "no" — no justification is needed.)

Yes!

6. Program Design - Array Programming (25 points)

Use the four step design methodology to implement a static method called `isGoodSquare` that takes as input a two-dimensional array of ints and returns true if and only if the array is a square matrix where the sum of the numbers in every horizontal row and every vertical column is the same. The method should return **`false`** for ill-formed inputs (null, non-square array). On an empty (0-length) input array, it should return **`true`**.

a. Step 1 is *understanding the problem*. You don't have to write anything for this part—your answers below will demonstrate whether or not you succeeded with Step 1.

b. Step 2 is *formalizing the interface*. We have done this for you:

```
public static boolean isGoodSquare(int[][] sq) { ... }
```

c. The next step is *writing test cases*. For example, one possible testcase is a valid good square:

```
@Test
public void testValidGoodSquare() {
  int sq[][] = {{8,1,6},{3,5,7},{4,9,2}};
  assertEquals(true, isGoodSquare(sq));
}
```

The interesting parts of this test are the name, which should communicate the reason for the test ("valid good square"), plus the expected result value and the array to be tested. To avoid writing too much boilerplate, we might abbreviate this test case as follows:

Test name	Input array	Expected output
Valid good square	`{{8,1,6},{3,5,7},{4,9,2}}`	`true`

On the next page, write three more test cases for this method in the same style.

Possible answers: Unequal sums, Not square, Null matrix, empty matrix, unary matrix

d. The final step is to *implement the method*. Please do so below. Do not use any external libraries.

Answer:

```
public static boolean isGood(int[][] sq) {

  if (sq == null) return false;
  int len = sq.length;

  if (len == 0) return true;

  int candSum = 0;
  for(int i = 0; i < len; i++) {
    candSum += sq[0][i];
  }

  for(int i = 0; i < len; i++ ) {
    int sum1 = 0;
    int sum2 = 0;
    if (sq[i] == null || sq[i].length != len) return false;
```

```
  for(int j = 0; j < len; j++) {
    sum1 += sq[i][j];
    sum2 += sq[j][i];
  }
  if (sum1 != candSum || sum2 != candSum) return false;
}

return true;
```

Appendix: Widget Code

```
public interface Widget {
  public void repaint (Gctx gc);
}

public interface LabelController {
   public void setLabel(String s);
}

public class Gctx {
   public Gctx() { ... }
}

public class Label implements Widget, LabelController {
   private String s;
   public Label(String s) { this.s = s; }
   public void setLabel(String s) { this.s = s; }
   public void repaint(Gctx gc) { ... }
}

public class Empty implements Widget {
   public void repaint(Gctx gc) { }
   public Empty() {}
}

public class Foo {
   public static Widget asWidget(Widget w) {
      return w;
   }

   public static LabelController asLabelController(LabelController l){
      return l;
   }

   public static void main(String[] args) {
      Label l = new Label("CIS 120");
      Empty e = new Empty ();
      Widget a1 = l;
      LabelController a2 = l;
      Object a3 = e;

      _______ a4 = Foo.asWidget(l);
      _______ a5 = Foo.asLabelController(l);
      _______ a6 = Foo.asWidget(e);

   }
}
```

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 (q:'a queue) (x:'a) : 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: OCaml ASM Example

This is 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 *)*.