CIS 120 Final Exam 5 May 2017

**SOLUTIONS**

## 1: OCaml, Higher Order Functions

Recall the higher-order list processing functions `transform` and `fold` (reproduced on page 1 of the Appendix).

Each part of this problem below begins with a sample function written using simple recursion over lists, followed by several alternative versions written using `fold` and/or `transform`. In each part, first indicate what the function returns for the sample input shown. Then mark *all* of the alternatives that implement the same behavior as the recursive sample. *There may be zero, one, or more than one such function.* Some of the alternatives do not typecheck—do not mark these.

### 1.1: (5 points)

```
let rec func1 (x: 'a) (lst: 'a list) : bool =
  begin match lst with
   | [] -> false
   | hd :: tl -> x = hd || func1 x tl
   end

let ans1 = func1 3 [0; 1; 2; 3]
```

What is the value computed for `ans1` in the code above?

*Answer:* `ans1 =`

*ANSWER:* `func1` *is* `member`. *ans1* = `true`.

Which of the following functions behave the same as `func1` on all inputs? (Check all that apply.)

☒
```
let func1 (x: 'a) (lst: 'a list) : bool =
  fold (fun item acc -> (item = x) || acc) false lst
```

☐
```
let func1 (x: 'a) (lst: 'a list) : bool =
  fold (fun item acc -> x || acc) false lst
```

☒
```
let func1 (x: 'a) (lst: 'a list) : bool =
  fold (fun item acc -> item || acc)
       false
       (transform (fun y -> y = x) lst)
```

**1.2: (5 points)**

```
let rec func2 (lst: int list): int list =
  begin match lst with
  | [] -> []
  | hd :: tl ->
        (if hd > 1 then (hd :: func2 tl) else func2 tl)
  end

let ans2 = func2 [0; 1; 2; 3]
```

What is the value computed for `ans2` in the code above?

*ANSWER:* `func2` *removes integers less than 2 from the input list. ans2 =* `[2; 3]`.

Which of the following functions behave the same as `func2` on all inputs? (Check all that apply.)

☒
```
let func2 (lst: int list): int list =
  fold (fun x acc -> if x > 1 then x :: acc else acc) [] lst
```

☐
```
let func2 (lst: int list): int list =
  fold (fun x acc -> (if x > 1 then x else []) :: acc) [] lst
```

☐
```
let func2 (lst: int list) : int =
  fold (fun x acc -> x + acc) 0
       (transform (fun x -> if x > 1 then 1 else 0) lst)
```

**1.3: (5 points)**

```
let func3 (lst: 'a list): 'a list =
  let rec loop (res: 'a list) (acc: 'a list) =
     begin match res with
     | [] -> acc
     | hd :: tl -> loop tl (hd :: acc)
     end in
   loop lst []

let ans3 = func3 [0; 1; 2; 3]
```

What is the value computed for `ans3` in the code above?

*ANSWER:* `func3` *reverses a list. ans3 =* `[3;2;1;0]`.

Which of the following functions behave the same as `func3` on all inputs? (Check all that apply.)

☐
```
let func3 (lst: 'a list): 'a list =
  fold (fun x acc -> x :: acc) [] lst
```

☒
```
let func3 (lst: 'a list) : 'a list =
  fold (fun x acc -> acc @ [x]) [] lst
```

☐
```
let func3 (lst: 'a list): 'a list = fold @ [] lst
```

## 2: OCaml: Binary Search Trees

Recall the definition of generic binary trees and the binary search tree insert function:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree

let rec insert (t:'a tree) (n:'a) : 'a tree =
  begin match t with
    | Empty -> Node(Empty, n, Empty)
    | Node(lt, x, rt) ->
      if x = n then t
      else if n < x then Node (insert lt n, x, rt)
      else Node(lt, x, insert rt n)
  end
```

**2.1:** **(5 points)** Check "Good" below each tree that satisfies the binary search tree invariant and "Bad" below the ones that don't. (Note that we have omitted the Empty nodes from these pictures.)

```
4            3            2            3           7
 \          / \          / \            \           \
  5        2   6        1   6            6           6
   \          /              \          / \           \
    6        5                4        2   7           5
     \                                                  \
      7                                                  4
```

(a) ☒ Good ☐ Bad    (b) ☒ Good ☐ Bad    (c) ☐ Good ☒ Bad    (d) ☐ Good ☒ Bad    (e) ☐ Good ☒ Bad

**2.2:** **(4 points)** For each definition below, check the letter of the tree above that it constructs or "none of the above."

```
let t1 : int tree =
  Node(Node(Empty, 1, Empty), 2, Node(Empty, 6, (Node (Empty, 4, Empty))))
```

☐ (a)  ☐ (b)  ☒ (c)  ☐ (d)  ☐ (e)  ☐ None of the above

```
let t2 : int tree =
  insert (insert (insert (insert Empty 4) 5) 6) 7
```

☒ (a)  ☐ (b)  ☐ (c)  ☐ (d)  ☐ (e)  ☐ None of the above

```
let t3 : int tree =
  insert (insert (insert (insert Empty 2) 5) 3) 6
```

☐ (a)  ☐ (b)  ☐ (c)  ☐ (d)  ☐ (e)  ☒ None of the above

```
let t4 : int tree =
  Node(Empty, 3, Node(Node (Empty, 2, Empty), 6, (Node (Empty, 7, Empty))))
```

☐ (a)  ☐ (b)  ☐ (c)  ☒ (d)  ☐ (e)  ☐ None of the above

**2.3: (8 points)** Consider the following function `foo`:

```
let rec foo (x:int) (t : int tree) : int tree =
  begin match t with
  | Empty -> Empty
  | Node (lft, y, rgt) ->
     if x = y then lft
     else if x < y then
       foo x lft
     else
       Node (lft, y, foo x rgt)
```

Suppose `t` is this tree:

```
      5
     / \
    3   6
   / \
  0   4
```

**a.** Draw the result of `foo 1 t` (where `1` is the number one).

```
  0
```

**b.** Draw the result of `foo 5 t`.

```
    3
   / \
  0   4
```

**c.** Draw the result of `foo 4 t`.

```
    3
   /
  0
```

## 3: Java concepts (16 points)

**3.1:** If `B` is a subtype of `A`, then `List<B>` is a subtype of `List<A>`.

☐ True [✓ False]

**3.2:** An object's *static* type is always a subtype of its *dynamic* type.

☐ True [✓ False]

**3.3:** A static method dispatch `C.m()` implicitly pushes a binding for `this` onto the stack.

☐ True [✓ False]

**3.4:** If `s` is an instance of a class that does not override the default implementation of `equals` (from class `Object`), and if `t` is an instance of some different class, then `t.equals(s)` should always return `false`, in order to comply with Java's rules about the proper behavior of `equals`. (The documentation for the `equals` method can be found on page 4 of the appendix.)

[✓ True] ☐ False

**3.5:** Suppose we have the declaration `final String s = "CIS 120";`. Then the expression `s.equals("CIS 120")` always returns `true`.

[✓ True] ☐ False

**3.6:** If `s` and `t` are Java objects of the same type with `s.hashCode() == t.hashCode()`, then `s == t` must return `true`.

☐ True [✓ False]

**3.7:** All Java threads share the same workspace, stack, and heap.

☐ True [✓ False]

**3.8:** In Swing, the `paintComponent` method only executes the very first time a component is displayed (when the application initially starts).

☐ True [✓ False]

## 4: Java Typing and Dynamic Dispatch

Consider the following class definitions:

```
 1 public class A {
 2    void m() {
 3       System.out.println("A.m");
 4       this.n();
 5    }
 6    void n() {
 7       System.out.println("A.n");
 8    }
 9 }
10 public class B extends A {
11    void m() {
12       System.out.println("B.m");
13       super.m();
14    }
15    @Override
16    void n() {
17       System.out.println("B.n");
18       // this .m();
19    }
20    void q() {
21       System.out.println("B.q");
22       this.m();
23    }
24 }
```

And the following variable declaration:

```
A a0 = new B();
```

**4.1: (1 points)** The static (i.e. compile-time) type of `a0` is A.

[✓ True]    False

**4.2: (1 points)** The dynamic (i.e. run-time) type of `a0` is A.

True    [✓ False]

**4.3: (2 points)** What is printed if we evaluate `(new A()).m()`?

Answer: Prints
```
A.m
A.n
```

**4.4: (2 points)** What is printed if we evaluate `(new B()).m()`?

Answer: Prints
```
B.m
A.m
B.n
```

**4.5: (2 points)** If `this.m()` is uncommented at line 18 above, what is now printed if we evaluate the expression `(new B()).m()`?

Answer: Prints
```
B.m
A.m
B.n
```
repeatedly until there is a `StackOverflowError`.

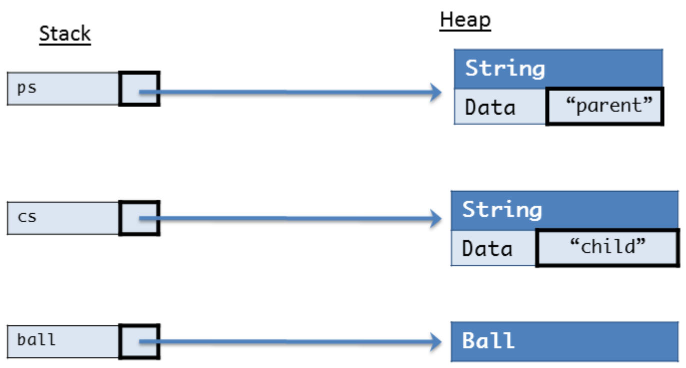
**4.6:** **(2 points)** What happens if we write `a0.q()`?

Answer: Compilation error

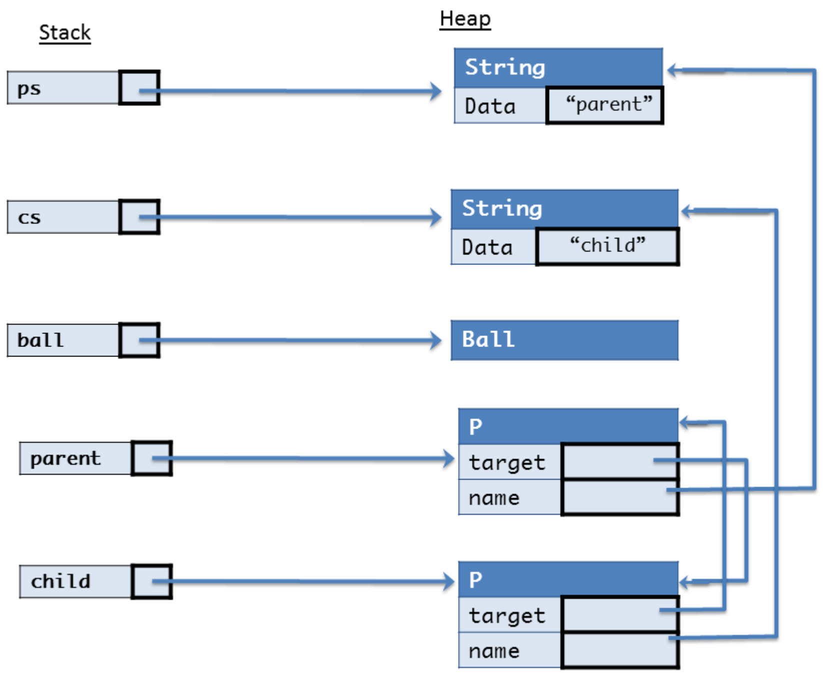
## 5: Java Exceptions

Suppose we define the classes `P` and `Ball` as shown on page 2 in the appendix, and suppose the code from the bottom of the same page is placed in the workspace of the Java ASM.

**5.1: (10 points)** The stack and heap diagram that corresponds to the point in the execution marked `START` is shown below. Extend this diagram so that it displays the stack and heap at the point when execution terminates. *Note: There is no need to show the workspace or the class table!*



Answer:



**5.2: (4 points)** Now suppose the following code is placed on the workspace.

```
parent.aim(ball);
```

What happens next? Check the correct behavior from the choices below.

☐ The console prints

```
parent is throwing the ball.
child caught the ball.
```

and execution terminates.

☐ The console prints

```
parent is throwing the ball.
parent caught the ball.
```

and execution terminates.

☐ Nothing is printed to the console, and the program immediately terminates.

☐ The console prints

```
child is throwing the ball.
parent caught the ball.
parent is throwing the ball.
child caught the ball.
...
```

repeatedly and then eventually produces a `StackOverflowError`.

☒ The console prints

```
parent is throwing the ball.
parent caught the ball.
child is throwing the ball.
child caught the ball.
...
```

repeatedly and then eventually produces a `StackOverflowError`.

☐ The console prints

```
child is throwing the ball.
child caught the ball.
parent is throwing the ball.
parent caught the ball.
...
```

repeatedly and then eventually produces a `StackOverflowError`.

☐ Nothing is printed to the console, and the program produces a `StackOverflowError`.

**5.3:** **(4 points)** Suppose that the definition of the class `P` is changed by adding a `finally` clause in the `aim` method:

```
void aim(Ball ball) {
  try {
    System.out.println(name + " is throwing the ball.");
    throw ball;
  } catch (Ball b) {
    System.out.println(name + " caught the ball.");
    target.aim (b);
  } finally {
    System.out.println(name + " laughs happily");
  }
}
```

and again all of the following code is placed on the workspace of the Java ASM:

```
System.out.println("Executing");
String ps = "parent";
String cs = "child";
Ball ball = new Ball();
// START
P parent = new P(ps,null);
P child = new P(cs,parent);
parent.target = child;
parent.aim(ball);
```

Now what happens? Check the correct behavior from the choices below:

☐ The same behavior as in the previous question.

☐ Now every instance of

    parent caught the ball

is immediately followed by

    parent laughs happily

☐ Now every instance of

    child caught the ball

is immediately followed by

    child laughs happily

☐ Both of the two answers above are true.

☒ None of the above.
*This question is actually harder than we intended. The right answer is*
```
Parent is throwing the ball
Child catches the ball
```
*repeatedly until a* `StackOverflow` *exception occurs, after which all the* `finally` *clauses unwind, printing (depending on when exactly stack overflow occurred)*
```
Child laughs happily
Parent laughs happily
```
*repeatedly. So "None of the above" is the correct answer, but we gave "Same behavior as previous question" half credit.*

**5.4: (4 points)** Finally, suppose that the definition of the class `P` is changed by moving the call to the `aim` method into the `finally` clause:

```
void aim(Ball ball) {
  try {
     System.out.println(name + " is throwing the ball.");
     throw ball;
  } catch (Ball b) {
     System.out.println(name + " caught the ball.");
  } finally {
     System.out.println(name + " laughs happily");
     target.aim (ball);
  }
}
```

and once again all of the following code is placed on the workspace of the Java ASM:

```
System.out.println("Executing");
String ps = "parent";
String cs = "child";
Ball ball = new Ball();
// START
P parent = new P(ps,null);
P child = new P(cs,parent);
parent.target = child;
parent.aim(ball);
```

Now what happens? Check the correct behavior from the choices below:

☐ The same behavior as in question 5.2.

☐ Now every instance of

```
parent caught the ball
```

is followed by

```
parent laughs happily
```

☐ Now every instance of

```
child caught the ball
```

is followed by

```
child laughs happily
```

☒ Both of the two answers above are true.

☐ None of the above.

**6: Java: Aliasing and Inner Classes**

Consider the Java class `WeirdCounter` from page 3 of the Appendix.

The following test succeeds:

```
public void test1() {
   WeirdCounter c = new WeirdCounter();
   assertEquals("x1y1", c.tick());
}
```

Fill in the missing strings so that the following tests also succeed. (Note that each test will start from a completely fresh machine state, with static fields initialized to 0.)

**6.1: (4 points)**

```
public void test2() {
   WeirdCounter c1 = new WeirdCounter();
   WeirdCounter c2 = new WeirdCounter();
   assertEquals("x1y1", c1.tick());
   assertEquals("x1y2", c2.tick());
}
```

**6.2: (4 points)**

```
public void test3() {
   WeirdCounter c1 = new WeirdCounter();
   WeirdCounter.Inner d1 = c1.new Inner();
   WeirdCounter.Inner d2 = c1.new Inner();
   assertEquals("x1y1z1", d1.tick());
   assertEquals("x2y2z1", d2.tick());
}
```

**6.3: (8 points)**

```
public void test4() {
   WeirdCounter c1 = new WeirdCounter();
   WeirdCounter.Inner d1 = c1.new Inner();
   WeirdCounter c2 = new WeirdCounter();
   WeirdCounter.Inner d2 = c2.new Inner();
   assertEquals("x1y1z1", d1.tick());
   assertEquals("x1y2z1", d2.tick());
   assertEquals("x2y3", c1.tick());
   assertEquals("x2y4", c2.tick());
}
```

## 7: Java: Iterators

The documentation for the `Iterator` interface can be found on page 5 of the appendix.

Suppose `base` is an `Iterator` such that successive calls to `base.next()` will produce the values `1`, `2`, and `3` and we want an iterator that will produce the values `1`, `1`, `2`, `2`, `3`, `3`. How can we build the latter from the former?

Let's follow the standard design pattern (the first two steps are given to you).

**7.1: Understand the problem**. A general approach to this situation is to build an "iterator wrapper" — a class `Doubler` whose constructor takes an iterator `base` as argument and whose `next` and `hasNext` methods call `base.next()` and `base.hasNext()` "every other time."

**7.2: Formalize the interface**. Since what we want at the end is a new iterator, we should define `Doubler` to extend the `Iterator` interface. Moreover, the `Doubler` constructor should take an `Iterator` as an argument. This leads us to the following skeleton:

```
public class Doubler<E> implements Iterator<E> {

   public Doubler(Iterator<E> b) {
      // ...
   }

   public boolean hasNext() {
      // ...
   }

   public E next() {
      // ...
   }
}
```

**7.3: (8 points) Implement test cases**. Now it's your turn. Here is a JUnit test that exercises the behavior of `Doubler`'s `next` method:

```
@Test
public void test1() {
   LinkedList<String> l = new LinkedList<String>();
   l.add("1");
   l.add("2");
   Iterator<String> i = new Doubler(l.iterator());
   assertEquals(i.next(), "1");
   assertEquals(i.next(), "1");
   assertEquals(i.next(), "2");
   assertEquals(i.next(), "2");
}
```

Complete the following test so that it checks the behavior of `hasNext` similarly:

```
@Test
public void test2() {
   LinkedList<String> l = new LinkedList<String>();
   l.add("1");
   Iterator<String> i = new Doubler(l.iterator());
   assertEquals(i.hasNext(), true);
   i.next();
   assertEquals(i.hasNext(), true);
```

```
      i.next();
      assertEquals(i.hasNext(), false);
   }
```

**7.4: (16 points) Implement the functionality**. Complete the following definition of `Doubler`.

```
public class Doubler<E> implements Iterator<E> {

   private Iterator<E> base;
   // ADDITIONAL FIELDS IF NEEDED (FILL IN):
   private E saved = null;

   public Doubler(Iterator<E> b) {
      base = b;
      // ADDITIONAL INITIALIZATION IF NEEDED (FILL IN):
   }

   public boolean hasNext() {
      // FILL IN:
      if (saved != null)
         return true;
      else
         return base.hasNext();
   }

   public E next() {
      // FILL IN:
      if (saved != null) {
         E r = saved;
         saved = null;
         return r;
      } else {
         saved = base.next();
         return saved;
      }
   }

}
```

# Appendix

Do not write answers in this portion of the exam. (But feel free to use it as scratch paper.)

Do not open until the exam begins.

## Transform and fold

```
let rec transform (f: 'a -> 'b) (l: 'a list): 'b list =
  begin match l with
  | [] -> []
  | hd :: tl -> (f hd) :: (transform f tl)
  end

let rec fold (combine: 'a -> 'b -> 'b) (base: 'b) (l: 'a list) : 'b =
  begin match l with
  | [] -> base
  | hd :: tl -> combine hd (fold combine base tl)
  end
```

## Classes Ball and P

```
public class Ball extends Throwable {
}

public class P {
  public P target;
  public String name;

  P (String name, P target) { this.name = name; this.target = target; }

  void aim(Ball ball) {
     try {
       System.out.println(name + " is throwing the ball.");
       throw ball;
     } catch (Ball b) {
       System.out.println(name + " caught the ball.");
       target.aim (b);
    }
  }
}
```

## Initial workspace for Ball problem

```
    System.out.println("Executing");
    String ps = "parent";
    String cs = "child";
    Ball ball = new Ball();
    // START
    P parent = new P(ps,null);
    P child = new P(cs,parent);
    parent.target = child;
```

## WeirdCounter class

```
public class WeirdCounter {
   private int x = 0;
   private static int y = 0;

   class Inner {
      private int z = 0;

      public String tick() {
         x++;
         y++;
         z++;
         return("x" + x + "y" + y + "z" + z);
      }
   }

   public String tick() {
      x++;
      y++;
      return("x" + x + "y" + y);
   }
}
```

## Object.equals() documentation

```
public boolean equals(Object obj)
```

Indicates whether some other object is ”equal to” this one. The equals method implements an equivalence relation on non-null object references:

- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

## Iterator documentation

`public interface Iterator<E>`

Type Parameters:

- E - the type of elements returned by this iterator

Methods:

- `boolean hasNext()`

  Returns true if the iteration has more elements. (In other words, returns true if next() would return an element rather than throwing an exception.) Returns: true if the iteration has more elements.

- `E next()`

  Returns the next element in the iteration. Returns: the next element in the iteration Throws: NoSuchElementException - if the iteration has no more elements.