COS 326 Functional Programming — Midterm Exam, Fall 2016

Name (print clearly):

Login:

You are allowed to bring a hand-written sheet (8.5x11in page, both sides) into the exam containing any information you want. Other than that, the work on this midterm must be your own without any outside help. ***A few students are taking this exam on Thursday. Do not talk about this exam outside the exam room or with your friends until Friday.***

Honor Pledge and signature:

General Instructions:

- **Write your name and login on this page. Write and sign the Honor Code pledge**.
- You have **1 hour and 20 minutes** to complete this exam.
- There are **7 parts** to the exam (each part may have several questions). Write answers in the spaces provided. We will give partial credit on questions. So part of a solution, if it is correct, or at least sensible, is better than no solution at all.
- Minor errors in OCaml syntax will not be penalized significantly, if at all. However, we have no choice but to penalize errors that render a solution incomprehensible.
- If you cannot complete the details of a proof but can show that you know how to structure the proof, you will receive some credit. Show that you know how to break down a proof into appropriate cases. Show that you know what the induction hypothesis is by writing it down clearly. Show that you know what must be proven
- Do not give two answers to a question hoping that one of them is correct. If you give two different answers to a question, you will receive no credit if one of the answers is correct and one is incorrect. If ambiguous, circle the answer you intend.

Grades on the 7 questions:

1 __________ / 2	4 ___________ / 3	**7________________ / 4**
2__________ / 3	5 ___________ / 8	**Total __________ / 32**
3__________ / 6	6 ___________ / 6	

There are 11 pages in all

<u>Problem 1</u> [2 points]: Answer in the space provided.

Give an example of a feature of OCaml that is not present in Java (and not easily replicated in Java) and that helps a programmer write more reliable programs. Explain briefly (a sentence or two or using an example) how the feature helps improve program reliability.

Many answers. Here are two:

-- OCaml has both the type int * int and the type (int * int) option. That means that unlike in Java, programmers can use the type system to control whether arguments to functions may be null or are guaranteed not to be null. In OCaml, you can't "forget" that an object is an option type and may be null, and hence you don't get accidental null pointer exceptions.

-- OCaml has data types against which the compiler will check to determine whether a pattern match statement is exhaustive. If a match is non-exhaustive, the compiler reports this fact and gives an example of an unmatched value. This helps ensure programmers have covered all cases.

Problem 2 [3 points]

Using the substitution model of evaluation for OCaml, show how the following expression evaluates in a step-by-step manner. (Do not just give the final answer; show each step independently.) Show that you know the order of evaluation of OCaml programs.

```
let x = (fun x -> x) (let x = 4 + 1 in x) in x + (let x = 2 in x)

--> let x = (fun x -> x) (let x = 5 in x) in x + (let x = 2 in x)

--> let x = (fun x -> x) 5 in x + (let x = 2 in x)

--> let x = 5 in x + (let x = 2 in x)

--> 5 + (let x = 2 in x)

--> 5 + 2

--> 7
```

Problem 3 [6 points]:

Give the most general (most polymorphic) types for each of these functions,

or write “does not typecheck”:

```
type ('k,'v) tree =
    Leaf
  | Node of 'k * 'v * ('k,'v) tree * ('k,'v) tree

let rec tree_fold f a l =
  match l with
      Leaf -> a
    | Node (k,v,left,right) ->
        f k v (tree_fold f a left) (tree_fold f a right)

let rec balanced n g =
  if n <= 0 then
    Leaf
  else
    Node (n, g n, balanced (n-1) g, balanced (n-1) g)

let folded f =
  tree_fold f Leaf (balanced 5 (fun x -> x))
```

```
tree_fold :
('a -> 'b -> 'c -> 'c -> 'c) -> 'c -> ('a,'b) tree -> 'c

balanced :
int -> (int -> 'a) -> (int, 'a) tree

folded :

(int -> int -> ('a, 'b) tree -> ('a, 'b) tree) -> ('a, 'b) tree)
-> ('a, 'b) tree)
```

most students, naturally, used the polymorphic type names from the type definition, resulting in ('k,'v) trees throughout rather than the ('a,'b) trees offered by the default type inference algorithm's preference to sequentially as they appear label types alphabetically.

Problem 4 [3 points] Consider the following code, theorem and proof.

```
let rec iterate (f:int list -> int list) (x:int list) : int list =
  match x with
    [] -> []
  | hd::tl -> iterate f (f tl)

let rec iterate2 (x:int list) (f:int list -> int list) : int list =
  match x with
    [] -> []
  | hd::tl -> iterate2 (f tl) f
```

Theorem:

```
for all f:int list -> int list.
for all x:int list.
 iterate f x == iterate2 x f
```

Proof:

```
Pick any f:int list -> int list

Now, we must show that:

for all x:int list.
 iterate f x == iterate2 x f

The proof is by induction on the structure of the list x.

case x = []:
   iterate f []                                          (LHS)
== match x with [] -> [] | hd::tl -> iterate f (f tl)    (eval)
== []                                                    (eval)
== match x with [] -> [] | hd::tl -> iterate2 (f tl) f   (reverse eval)
== iterate2 [] f                                         (reverse eval)

case x = hd::tl:
   iterate f (hd::tl)                                        (LHS)
== match hd::tl with [] -> [] | hd::tl -> iterate f (f tl)   (eval)
== iterate f (f tl)                                          (eval)
== iterate2 (f tl) f                                         (by IH)
== match hd::tl with [] -> [] | hd::tl -> iterate2 (f tl) f (rev. eval)
== iterate2 (hd::tl) f                                       (rev. eval)
```

QED

The ***proof*** of this theorem is **not correct**. In other words, it contains a significant ***reasoning error***. The reasoning error is not something that can be easily fixed – it is not a minor typo or minor omission of some kind. What is the significant reasoning error? Be as precise as you can – vague answers will receive fewer points than precise answers. Show that you really know what you are talking about by giving an example of a function f that causes a problem of some kind. Your explanation should be brief but precise. **Note**: your job is not to determine whether or not the theorem is true, it is to find an error in the proof. A proof can be wrong, even if the theorem is true. Use the next page (but you definitely do not need the whole page – just 2-3 sentences and an appropriate example of an f)

This is not a valid reasoning step in the proof:

```
   iterate f (f tl)
== iterate2 (f tl) f                                       (by IH)
```

The proof is by induction on the structure of the list x, which means that in our proof, we can use an equation of the form:

```
iterate f L == iterate2 L f
```

in the case for x = hd::tl only when we know that L is structurally smaller than hd::tl. However, (f tl) may be not be smaller. For instance, f may be the function:

let f xs = 0::xs

which would create a list of the same size as hd::tl. Hence, the induction hypothesis does not apply here.

Problem 5 [8 points] Consider the following code:

```
type exp =
    Int of int
  | Add of exp * exp

let rec eval1 (e:exp) : int =
  match e with
      Int i -> i
    | Add (e1, e2) -> (eval1 e1) + (eval1 e2)

let rec eval2 (e:exp) (n:int) : int =
  match e with
      Int i -> i + n
    | Add (e1, e2) -> eval2 e1 (eval2 e2 n)
```

Prove the following theorem:

```
For all e:exp.
eval1 e = eval2 e 0
```

Your proof may use any standard facts about addition you find useful.

Note: If you can't figure out how to prove this theorem, at least explain clearly where you get stuck. A partial proof and a clear explanation of where you get stuck, and why, is worth more partial credit than a proof with erroneous justifications/incorrect proof steps.

(Your proof may continue to the next page.)

A direct attempt to prove to prove the theorem by induction on the structure of the expression e fails. Consider the case for e = Add (e1,e2):

```
eval1 (Add (e1,e2)                (LHS)
== (eval1 e1) + (eval1 e2)        (eval)
== eval2 e1 0 + eval2 e2 0        (IH)

???

== eval2 e1 (eval2 e2 0)
== eval2 (Add(e1,e2)) 0           (eval reverse -- RHS)
```

```
There is no way to equate the LHS and RHS.  The proof fails.  We
must prove a more general theorem first.  See the next page.
```

One approach is to prove the following more general lemma:

```
Lemma 1:  For all e:exp. For all n:int. eval1 e + n == eval2 e n
Proof:  By induction on the structure of e.

To carry out the proof, we must now prove

For all n:int. eval1 e + n == eval2 e n

To do so, let’s pick an arbitrary n.

Now, there are 2 cases:

Case e = Int i:

   eval1 (Int i) + n   (LHS)
== i + n               (eval eval1)
== eval2 (Int i) n     (eval eval2 reverse == RHS)

Case e = Add(e1,e2)
  IH1:  For all n’:int. eval1 e1 + n’ == eval2 e1 n’
  IH2:  For all n’:int. eval1 e2 + n’ == eval2 e2 n’

   eval1 (Add(e1,e2)) + n         (LHS)
== (eval1 e1 + eval1 e2) + n      (eval eval1)
== eval1 e1 + (eval1 e2 + n)      (associativity of +)
== eval1 e1 + (eval2 e2 n)        (IH2 with n substituted for n’)
== eval2 e1 (eval2 e2 n)          (IH1 with (eval2 e2 n) for n’)
== eval2 (Add(e1,e2)) n           (eval eval2 reverse == RHS)

QED

Using Lemma 1, we can prove our overall theorem:

Pick an arbitrary e:exp:

   eval1 e       (LHS)
== eval1 e + 0   (by math)
== eval2 e 0     (by Lemma 1 -- RHS)
```

Problem 6 [6 points]
An integer matrix can be represented as a list of lists of integers, with each inner list being a row the matrix.

type matrix = int list list

For example, [[1;2;3];[4;5;6]] represents a matrix with 2 rows. The first row is [1;2;3]. The second row is [4;5;6].

Write the function valid, which tests to see whether every row in a matrix has the same length WITHOUT using the "rec" keyword. eg:

```
valid [[1;2;3];[4;5;6]] == true
valid [[1;2;3];[4;5];[7;8;9]] == false
valid [] == true
valid [[]] == true
valid [[]; []] == true
```

You may use `List.map, List.filter, List.fold_left, and List.length` but no other functions from the list library. Recall:

```
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_left f a [b1; ...;bn] == f (... (f (f a b1) b2) ...) bn.
```

Hint: For clarity and ease of acquiring partial credit, break your function up into parts (ie: more than one function or let declaration). Describe the goal of each part to the grader (very briefly) in a comment.

Give your answer on the following page.

There is more than one solution. Here are three of the most common:

```
let valid1 m =
  let test_lengths (len:int) (ls:int list) : bool =
    let f (b:bool) (l:int) = b && (l = len) in
    List.fold_left f true ls
  in
  let lengths = List.map List.length m in
  match lengths with
  | [] -> true
  | hd::tl -> test_lengths hd tl

let valid2 m =
  let lengths = List.map List.length m in
  match lengths with
  | [] -> true
  | hd::tl -> List.filter ((=) hd) tl = tl

let valid3 m =
  let lengths = List.map List.length m in
  match lengths with
  | [] -> true
  | hd::tl -> List.filter ((<>) hd) tl = []
```

<u>Problem 7 [4 points]:</u> Consider the following code

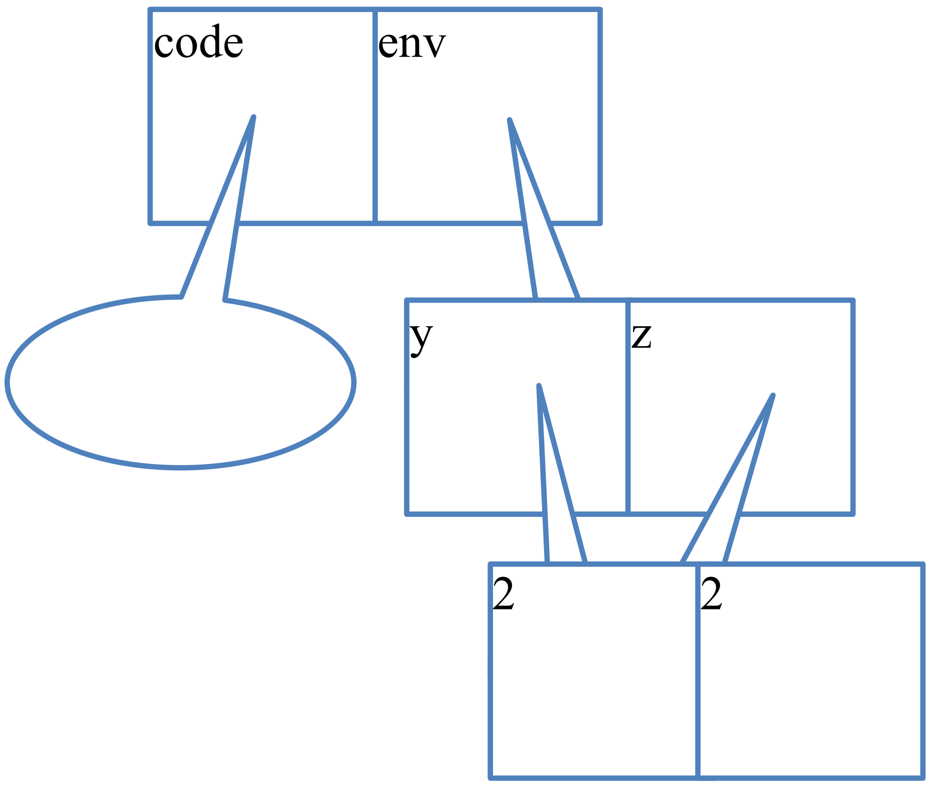
```
type foo = Baz of int | Bar of int * foo * foo

let x = Bar (4, Baz 1, Baz 2) in
let z = (2,2) in
let y = z in
let w = 15 in
(fun x -> match (y,x) with (y,w) -> (z,()))
```

Assume the code above is evaluated to completion, producing a value. Draw a picture of the memory layout of the **value** produced using the conventions discussed in class. Assume that the OCaml compiler performs an optimization similar to the one described in part 1.3 of assignment 4 --- in other words, it minimizes closure size based on the free variables of the function body. **Draw only the data structures reachable from that value; draw no other data structures.** If you find this question ambiguous, or under-specified, simply state any reasonable assumptions you are making about the memory layout. Feel free to write a few brief comments to clarify any decisions you make.

The value produced is the closure for the function:

```
fun x -> match (y,x) with (y,w) -> (z,())
```

The closure's environment should include all free variables of the function (and only those variables, due to the closure pruning requirement) — this includes y and z but not w or x. Note that y and z should point to the same pair.


code
env
y
z
2
2