CS 3110 Spring 2015	**Soft deadline: 03/05/15 11:59 pm**
Problem Set 2	**Hard deadline: 03/07/15 11:59 pm**
Version 2 (last modified February 26, 2015)	

Revision log

In the main body of the writeup, revisions are marked in orange.

- [02/26/15] Corrected example output in exercise 16. Also corrected public test file.

Objectives

- Practice using the environment model of evaluation.
- Explore the tradeoffs between recursive functions, folds, and library functions.
- Learn about the `List` library module.
- Use trees to represent program expressions and their evaluation.
- Generalize the idea of folding to trees.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 4, 5, and 6; recitations 4, 5, and 6
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1–3
- The OCaml List Module
- The OCaml Pervasives Module (especially exceptions, `unit`, and floating-point arithmetic)
- The Assertions module (especially `assert_true`, `assert_raises`, and `assert_equal`)

What we supply

We provide these files as part of the release code:

1. `ps2.ml`, which is a template for your solution. You do not strictly need to use this file, but it should help get you started.
2. `ps2.mli`, which is an interface file giving the names and types of the values that your solution must define. You are required to adhere to these names and types. Solutions that do not adhere to them will be penalized.
3. `ps2_test_public.ml`, which is a small unit test file. Solutions that do not pass these test cases will receive minimal credit.

To ensure that your solution is compatible with the course staff's test suite, we strongly encourage you never to modify either the provided interface or the provided public test cases.

What to turn in

Submit these files on CMS:

1. A file `ps2written.pdf` containing your answers to the written portions of this problem set, which are identified below as "[written]".
2. Files `ps2.ml` and `ps2_test.ml` containing your solutions and unit tests for the coding exercises of this problem set, which are identified below as "[code]".

Academic integrity

This problem set is to be completed individually. Sharing of code is strictly prohibited. Please review the course policy on academic integrity.

Prohibited OCaml features

Imperative features—such as `ref`'s, the `Array` module, and `mutable` fields—are not permitted in your solutions to this problem set. You have not seen these features in lecture or recitation, so we doubt you'll be tempted. Your solutions may not link in any additional libraries.

Grading issues

Names and types: You are required to adhere to the names and types given in the problem set instructions. If your code does not compile against and pass all the tests in the provided public test file(s), your solution will receive minimal credit.

Code style: Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct will nonetheless be penalized. Take extra time to think and find elegant solutions.

Late submissions: Carefully review the course policy on submission and late assignments. Verify before the deadline on CMS that you have submitted the correct version.

Part 1: Evaluation (15 points)

Exercise 1.

[written] Show the evaluation of each of the following OCaml expressions. Use the environment model (with the rule of lexical scoping), and justify your answer by showing the evaluation of all subexpressions.

(a) `let y = 10 in let x = 3 in x+y`

(b) `let f x a = x::a in f 42 [3110]`

For part (b), you will need to do some desugaring of the OCaml expression before attempting to evaluate it.

Here is an example of the kind of answer we're expecting:

```
{} :: let x = 3+5 in x+x || 16
    {} :: 3+5 || 8
        {} :: 3 || 3
        {} :: 5 || 5
        3+5 = 8
    {x=8} :: x+x || 16
        {x=8} :: x || 8
        {x=8} :: x || 8
        8+8 = 16
```

Exercise 2.

[written] Consider the following program:

```
let x = 1 in
let f = fun y ->
  (let x = y + 1 in
   fun z -> x + y + z) in
let x = 3 in
let g = f 4 in
let y = 5 in
let z = g 6 in
z
```

(a) To what value does this program evaluate under the rule of lexical scope?

(b) To what value does this program evaluate under the rule of dynamic scope?

(c) Using the environment model, explain the key place(s) where evaluation of the above program becomes different under the two scoping rules.

Function Specification and Testing

Complete each of the coding exercises below by following these instructions:

1. Write a function with the appropriate name and type.
2. Write a specification comment above the definition of the function that documents a concise and accurate description of the function's *precondition* and *postcondition*. Also document a brief description of the function (one or two sentences) and/or a brief description of each argument (a couple of words), if your pre- and postconditions do not already address those descriptions.
3. Write unit tests that demonstrate the function's correctness. If the function is named `f`, then these tests should be named `f_test1`, `f_test2`, ...`f_testn`. How many unit tests should you write? As many as necessary to make you confident that your solution is correct. Your tests should be in a separate file named `ps2_test.ml`.

Some Advice on Style

- Write helper functions as needed.
- Use `List.hd` and `List.tl` only if you must, and only in situations where it is guaranteed they cannot raise exceptions.
- Consider efficiency, particularly when choosing between `fold_left` and `fold_right`, and when deciding whether to use the append operator `@`.

Part 2: `rec` vs. `fold` vs. `List` (30 points)

[code] Write each of the following functions in each of three ways:

(i) as a recursive function, without using the `List` module (append `_rec` to the function name for this implementation),

(ii) using `List.fold_left` or `fold_right`, but not other `List` module functions or the `rec` keyword (append `_fold` to the function name for this implementation), and

(iii) using any combination of `List` module functions other than `fold_left` or `fold_right`, but not the `rec` keyword (append `_lib` to the function name for this implementation).

You will therefore implement nine functions in this part of the problem set.

Exercise 3.

Write a function `lst_and: bool list -> bool`, such that `lst_and [a1; ...; an]` returns whether all elements of the list are `true`. That is, it returns `a1 && a2 && ... && an`. The `lst_and` of an empty list is `true`. For example:

- `lst_and [true; true] = true`
- `lst_and [true; false] = false`
- `lst_and [] = true`

Exercise 4.

Write a function `exists: ('a -> bool) -> 'a list -> bool`, such that `exists p [a1; ...; an]` returns whether at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`. The `exists` of an empty list is `false`. For example:

- `exists (fun i -> i > 0) [2; 3] = true`
- `exists ( (<) 0 ) [(-1); 2] = true`
- `exists ( (=) 0) [(-1); 2] = false`
- `exists (fun p -> true) [] = false`

Exercise 5.

Write a function `first: ('a -> bool) -> 'a list -> int`, such that `first p l` returns the index of the first element of list `l` that satisfies predicate `p`. The head of a list is at index 0. Raise `Not_found` if there is no element that satisfies the predicate. For example:

- `first (fun f -> (f 3100) = 3110) [(+) 3; (+) 1; (+) 10] = 2`
- `first (fun f -> (f 3100) = 3410) [(+) 3; (+) 1; (+) 10]` raises `Not_found`

Part 3: Folding (20 points)

[code] Complete these exercises using `List.fold_left` or `List.fold_right`, and without using the `rec` keyword or any other `List` module functions. The most elegant solutions will need only one line of code.

Exercise 6.

Write a function `alt_sum` of type `int list -> int` that computes the alternating sum of the list, such that `alt_sum [a1; ...; an] = a1 - a2 + a3 - a4 +...`

- `alt_sum [3105;(-3);2] = 3110`
- `alt_sum [7] = 7`
- `alt_sum [] = 0`

Exercise 7.

Write a function `poly` that takes a list of floats $[a_0, a_1, \ldots, a_{n-1}]$ and produces a function that takes an argument x and evaluates the polynomial

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}.$$

If the list of floats is empty, the function produced should always return `0.`. For example:

```
# let p1 = poly [1.;1.;1.];;
val p1 : float -> float = <fun>
# p1 1.0;;
- : float = 3.
# p1 2.0;;
- : float = 7.

# let p2 = poly [];;
val p2 : float -> float = <fun>
# p2 4.0;;
- : float = 0.
```

Part 4: Matrices (50 points)

[code] A *matrix* can be thought of as a 2-dimensional array. OCaml has an `Array` module, but we won't use that in this problem. Instead, let's define a representation for matrices of integers as follows:

```
type vector = int list
type matrix = vector list
```

For example, the matrix

$$m = \begin{bmatrix} 1 & 2 & 3 \\ 42 & 41 & 40 \end{bmatrix}$$

would be represented in *row major* form as follows:

```
let m = [[1;2;3];[42;41;40]]
```

Each list in this `matrix` represents a row in m.

A *valid* matrix is one in which all rows have the same length—i.e., the matrix is rectangular—and that length must be at least 1. Examples of invalid matrices include `[]`, `[[];[]]`, and `[[1];[2;3]]`. The behavior of your functions on invalid matrices is *unspecified*, meaning that we leave it up to you how to handle such inputs—and we will not be testing your functions on them. Your functions might raise an exception, or perhaps they might require validity as a precondition. Whatever you choose, be clear in your specification comment about what your function does.

You may choose any implementation strategy—`rec`, `fold`, or `List`—for these exercises. But choose wisely, because some choices are likely to be much easier than others to implement. Many of these exercises can be solved with just one line of code.

Exercise 8.

Implement a function `show: matrix -> unit` that prints the elements of the input matrix. For example,

```
# show m;;
1 2 3
42 41 40
- : unit = ()
```

We will not grade this exercise, but we strongly suggest that you complete it, because it is likely to be helpful in debugging later exercises.

Exercise 9.

Write a function `is_square: matrix -> bool` that returns whether the input matrix is square. A matrix is *square* if the number of rows is equal to the number of columns.

Exercise 10.

Implement a function `insert_col: matrix -> vector -> matrix` which takes a matrix `m` and a vector `c` and inserts `c` as the right-most column of `m`. If the sizes of the matrix and the vector do not match, the behavior is unspecified. For example, if `m` is the matrix defined above, then

- `insert_col m [6;7] = [[1;2;3;6];[42;41;40;7]]`
- `insert_col m [42;42;42]` is unspecified

Exercise 11.

Using `insert_col`, write a function `transpose: matrix -> matrix` that returns the *transpose* of a matrix. For example, if `m` is the matrix defined above, then

- `transpose m = [[1;42];[2;41];[3;40]]`

Exercise 12.

Implement a function `add_matrices: matrix -> matrix -> matrix` for entry-wise matrix addition. If the two input matrices are not the same size, the behavior is unspecified. For example,

- if `m1 = [[1;2;3];[4;5;6]]` and `m2 = [[42;42;42];[43;43;43]]`, then `add_matrices m1 m2 = [[43;44;45];[47;48;49]]`

Exercise 13.

Write a function `trace: matrix -> int` that returns the trace of a square matrix. The *trace* of a square matrix is the sum of the entries on the diagonal. The behavior is unspecified on non-square matrices. For example,

- `trace [[1;2;3];[2;3;4];[3;31;38]] = 42`
- `trace [[5;4];[21;22];[10;11]]` is unspecified

Exercise 14.

Write a function `inner: vector -> vector -> int` that returns the inner product of two vectors. The behavior is unspecified when given two vectors of different length. For example,

- `inner [1;2;3;4] [2;4;1000;25] = 3110`
- `inner [5;4] [21;22;10;11]` is unspecified

Exercise 15.

Implement a function `multiply_matrices: matrix -> matrix -> matrix` that returns the matrix product of the two input matrices. If the two input matrices are not of sizes that can be multiplied together, the behavior is unspecified. For example,

- if `m1 = [[1;2;3];[4;5;6]]` and `m2 = [[7;8];[9;10];[11;12]]`, then `multiply_matrices m1 m2 = [[58;64];[139;154]]`

Exercise 16.

Write a function `outer: vector -> vector -> matrix` that returns the outer product of two vectors. The behavior is unspecified when given two vectors of different length. For example,

- `outer [1;2;3] [4;5;6] = [[4; 5; 6]; [8; 10; 12]; [12; 15; 18]]`
- `outer [5;4] [21;22;10;11]` is unspecified

Part 5: Expression Trees (35 points)

[code] This problem explores the use of a tree data structure to represent computations. Consider the following type:

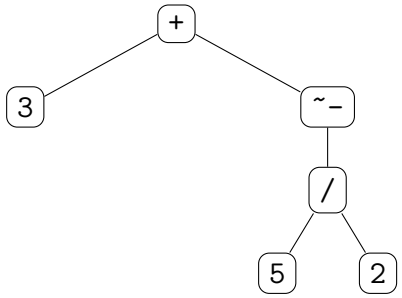
```
type 'a exprTree =
  | Val of 'a
  | Unop of ('a -> 'a) * 'a exprTree
  | Binop of ('a -> 'a -> 'a) * 'a exprTree * 'a exprTree
```

Each node in an `exprTree` can be thought of as either a value or the application of a function to the node's children. You could represent `3 + (-(5/2))` as the following `exprTree`:

```
Binop ((+), Val 3, Unop ((~-), Binop ((/), Val 5, Val 2)))
```

But that expression is a little difficult to read. To improve readability, we could use whitespace:

```
Binop ((+),
       Val 3,
       Unop ((~-),
             Binop ((/),
                    Val 5,
                    Val 2)))
```



Exercise 17.

Implement a function `count_vals: 'a exprTree -> int` that returns the number of values (i.e., the number of `Val` nodes) in the input tree. For example,

```
# count_vals (Binop ((+), Val 2, Val 3));;
- : int = 2

# let t =
  Binop ((+),
```

```
          Val 3,
          Unop ((~-),
                Binop ((/),
                       Val 5,
                       Val 2)))
val t : int exprTree =
  Binop (<fun>, Val 3, Unop (<fun>, Binop (<fun>, Val 5, Val 2)))

# count_vals t;;
- : int = 3
```

Exercise 18.

Write a function `eval: 'a exprTree -> 'a` that evaluates the expression described by the tree. Your function should behave just as the REPL would behave when asked to evaluate the equivalent OCaml expression. For example,

- `eval (Unop ((~-), Val 5)) = -5`
- `eval (make_asum_tree 6) = 21`
- `eval (Binop((/), Val 1, Val 0))` should raise `Division_by_zero`.

Exercise 19.

Consider the following function:

```
(* returns:  the sum of the integers 1..n
 * requires: n >= 1 *)
let rec asum n =
  if n=1 then 1 else n + (asum (n-1))
```

This function generates the sum `n + (... + (4 + (3 + (2 + 1))))`. Write a function `make_asum_tree: int -> int exprTree`, which generates a tree representing that sum. For example, `make_asum_tree 4` would return the following tree:

```
Binop ((+),
       Val 4,
       Binop ((+), Val 3,
              Binop ((+), Val 2, Val 1)))
```

(As usual, the REPL will print `<fun>` instead of `(+)` in that tree.) If `n` is not a positive integer, then `make_asum_tree n` should raise `Failure "make_asum_tree error"`.

Hint 1: For testing purposes, it might help to recall (from CS 2800) that

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

Hint 2: When writing unit tests, you will find that you cannot compare or pattern match on functions. (Determining equality of functions is, in general, undecidable—see CS 4810.) So you will need to write the best unit tests you can, given this limitation.

Exercise 20.

It is possible to define *generalized fold operations* (aka. *catamorphisms*—see CS 6117) over any datatype—not just lists. In this exercise, we'll explore folding over `exprTree`.

The basic idea of a generalized fold operator is that it traverses a data structure, consuming the elements of that structure, and producing an *accumulator*. The fold operator (we'll henceforth omit the word "generalized" for brevity) needs to know, for each variant of the datatype, how to incorporate the data carried by that variant into the accumulator.

For example, consider this simplified list datatype:

```
type intlist = Cons of int * intlist | Nil
```

Let's assume the accumulator is of type `'a`. The fold operator needs to know how to produce an accumulator value out of the data carried by the `Cons` variant—that is, the `int` and the `intlist`. The easy part of that is the `intlist`, because the fold operator can recursively call itself on that `intlist`, producing an accumulator value. The hard part is taking that accumulator, incorporating the `int` into it, and producing a new accumulator. There's no single way to do that; it depends on what kind of computation the client wants to do. For example, the client might want to sum the list elements, multiply them, etc. So instead, the client of the fold operator needs to provide a function `foldCons` that does the incorporation, where `foldCons: int -> 'a -> 'a`. When the fold operator reaches a `Cons(i,l)`, it will recurse on `l`, producing an accumulator `a`, then call `foldCons i a` to incorporate `i` and produce a new accumulator.

The fold operator also needs to know to produce an accumulator value out of the data carried by the `Nil` variant. Actually, there is no data carried by that variant, so the client of the fold operator needs to provide a value `foldNil: 'a` that the fold operator can use to produce an accumulator value anytime it finds a `Nil`.

Given `foldCons` and `foldNil`, we could write a fold operator as follows:

```
let rec intlist_fold (foldCons : int -> 'a -> 'a)
                     (foldNil : 'a) : intlist -> 'a = function
| Cons(i,l) -> foldCons i (intlist_fold foldCons foldNil l)
| Nil -> foldNil
```

It turns out we've reinvented `fold_right`! In fact, `intlist_fold` is the same algorithm as `List.fold_right`, but the implementation has the order of the final two arguments swapped:

```
let rec fold_right (f : 'a -> 'b -> 'b)
                   (l : 'a list) (acc : 'b) : 'b =
  match l with
    [] -> acc
  | x :: xs -> f x (List.fold_right f xs acc)
```

The argument that `fold_right` names `f` serves the same purpose as `foldCons`; likewise, for `acc` and `foldNil`.

The technique we used to derive `intlist_fold` above works for any OCaml datatype:

- Write a recursive `fold` function that takes in one argument for each variant of the datatype.
- That `fold` function matches against the datatype variants, calling itself recursively on any instance of the datatype that it encounters.
- When a variant carries data of a type other than the datatype being folded, use the appropriate argument to `fold` to incorporate that data.
- When a variant carries no data, use the appropriate argument to `fold` to produce an accumulator.

Use this technique to write a generalized fold operator for `exprTree` with the following type:

```
exprTree_fold : ('a -> 'b) ->
                (('a -> 'a) -> 'b -> 'b) ->
                (('a -> 'a -> 'a) -> 'b -> 'b -> 'b) ->
                'a exprTree ->
                'b
```

Exercise 21.

Reimplement `count_vals` and `eval` with `exprTree_fold`. Name your new functions `count_vals_fold` and `eval_fold`.

Part 6: Comments (0 points)

[written,ungraded] In `ps2written.pdf`, please include any comments you have about the problem set or about your solutions. This would be a good place to list any known problems with your submission that you weren't able to fix, or to give us general feedback about how to improve the problem set.