CS 3110 Fall 2014 **Due at 11:59 PM, Thursday, 09/18/14**
Problem Set 2
Version 3.1 (last modified September 16, 2014)

Revision log

- [09/16/14] Corrected argument order in example of `scan_right` that uses `(+)`.
- [09/15/14] Corrected argument order and type of `scan_right`. Clarified that printing is an imperative feature but nonetheless allowed in problem 3, exercise 1. Clarified that behavior on "empty" matrices in problem 3 is undefined. Re-clarified that any `List` function may be used in problem 3. Added `@` as a permitted operator in problem 2. *Addendum: Fixed typo in description of* `Structor` *matching.*
- [09/12/14] Corrected example in problem 1, exercise 2. Clarified behavior of `eval` in problem 1, exercise 3. Added `List.hd` and `List.tl` as permitted functions in problem 2. Clarified that any `List` function may be used in problem 3.

Objectives

- Gain familiarity with higher-order functions, particularly *folding*.
- Use *trees* to represent program expressions and their evaluation.
- Deeper understanding of *pattern matching* by implementing a fragment of it.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 4, 5, and 6; recitations 3 and 4
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1–3
- The OCaml List Module

What we supply

We are providing a template file `ps2.ml` that you can download from CMS. Although you do not strictly need to use this file, it should help get you started.

What to turn in

Submit these files on CMS:

- A file `ps2written.pdf` containing your comments about the problem set, as described at the end of this writeup. There are no actual written problems that you need to solve.
- 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]".
- A commit log `ps2log.txt` documenting your activity on the repository.

Partners and source control

You are required to work with a partner for this problem set. Each partner is responsible for understanding all parts of the assignment. You need not use the same partner as in PS1.

You are required to use `git`, a version control system, to work with your partner. Your repository must be private. We expect the `git` log you submit to show evidence of work over a period of time, not just a single commit at the end.

Grading issues

Compilation errors: All code you submit must compile. If your submission does not compile, we will notify you immediately. You will have 48 hours after the due date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming: We use automated grading, so it is crucial that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions will be treated as compilation errors.

Code style: Refer to the CS 3110 style guide and lecture notes. Ugly code that yet 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.

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`.

No Imperative 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.

Problem 1: Expression Trees (40 points)

[code] In other courses, trees may have been useful for representing hierarchical data or speeding up searches on a collection of objects. This is still true in functional programming. But now, with the help of functional paradigms—in particular, first-class functions—you can use trees not only to store data, but to represent computations.

Consider the following type:

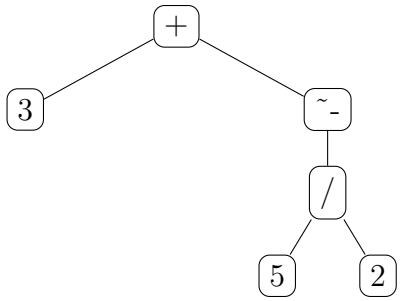
```
type 'a exprTree =
  | Val of 'a
  | Unop of ('a -> 'a) * 'a exprTree
  | Binop of ('a -> 'a -> 'a) * 'a exprTree * 'a exprTree
```

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 tree:

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

But that expression is a little difficult to read. To improve readability in the rest of this writeup, we will use whitespace so that nodes at each level of the tree have the same horizontal position:

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



In the following exercises, you may (and in fact should!) use the `rec` keyword.

Exercise 1.

Implement a function `count_ops: 'a exprTree -> int` that returns the number of function applications in the input tree. For example,

```
# count_ops (Binop ((+), Val 2, Val 3));;
- : int = 1

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

This can be roughly thought of as the "cost" of the tree, or how computationally expensive it would be to evaluate the tree.

Exercise 2.

Recall the factorial function:

```
let rec fact n = if n=0 then 1 else n * (fact (n-1))
```

Write a function `make_fact_tree: int -> int exprTree`, which recursively generates a tree representing the execution of `fact` on the input integer. For example, `make_fact_tree 3` would return the following tree:

```
Binop (( * ),
```

```
       Val 3,
       Binop (( * ), Val 2,
              Binop (( * ), Val 1, Val 1)))
```

Although the REPL will print that tree as follows:

```
Binop (<fun>,
       Val 3,
       Binop (<fun>, Val 2,
              Binop (<fun>, Val 1, Val 1)))
```

If `make_fact_tree` is called on a negative integer, it should raise `Failure`.

Exercise 3.

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_fact_tree 5) = 120`
- `eval (Binop((/), Val 1, Val 0))` should raise an exception

Problem 2: Folding (70 points)

[code] All the exercises in this problem should be completed entirely by

- writing one or more helper functions;
- calling `List.fold_left` or `List.fold_right` with one of those helper functions, some initial accumulator, and the input list; and
- doing a single pattern-match, if necessary, to extract the answer from the return value of the fold.

Note that not all these steps are necessary for every exercise.

You may also use `List.rev` and the cons `::` constructor. And you may use `List.hd`, `List.tl`, and the append `@` operator (which is really `List.append`), but beware of using these with good style. You should consider the efficiency of your solution, particularly when choosing between `fold_left` and `fold_right`. **You may not use the `rec` keyword in your solutions, nor may you use any other `List` module functions not mentioned above.** These prohibitions are designed to ensure that you get practice using folding.

Exercise 1.

Write a function `product: float list -> float` that takes a float list and returns the product of the elements of that list. The product of an empty list is `1.0`. For example,

- `product [777.5; 4.] = 3110.`
- `product [] = 1.`

Exercise 2.

Write two functions, `concat_left` and `concat_right`, both of type `string list -> string`. Both functions should have identical behavior: given a string list, return the in-order concatenation of all strings in the input list. For example,

- `concat_left ["cs"; "3110"] = "cs3110"`
- `concat_right ["cs"; "3110"] = "cs3110"`

In your implementation, `concat_left` must use `List.fold_left`, and `concat_right` must use `List.fold_right`.

Exercise 3.

(a) Implement `mapi_lst`, which has type

```
(int -> 'a -> 'b) -> 'a list -> 'b list
```

It is the same as `map`, except that the function is applied to the index (counting from 0) of the element as the first argument and the element itself as the second argument. For example,

- `mapi_lst (+) [3; 0; -1; -3] = [3; 1; 1; 0]`

(b) Use `mapi_lst` to implement a function `outline : string list -> string list` that produces a numbered outline from strings. Your function should prepend a number, a period, and a space to each string. For example,

- `outline ["point 1"; "point 2"; "point 3"]`
  `= ["1. point 1"; "2. point 2"; "3. point 3"]`

Exercise 4.

Folding functions return the accumulator after the entire input list has been processed. *Scanning functions* instead return a list of each value taken by the accumulator during processing. For example,

- `List.fold_left (+) 0 [1; 2; 3]` would return `6`, but
- `scan_left (+) 0 [1; 2; 3]` would return `[0; 1; 3; 6]`.

Here are some additional examples of scanning:

- `scan_right (+) [1; 2; 3] 0 = [0; 3; 5; 6]`
- `scan_left (^) "swag" ["zar"; "doz"] = ["swag"; "swagzar"; "swagzardoz"]`
- `scan_right (^) ["zar"; "doz"] "swag"= ["swag"; "dozswag"; "zardozswag"]`

(a) Implement two functions, `scan_left` and `scan_right`, with the following types:

```
scan_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a list

scan_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b list
```

(b) Use `scan_left` or `scan_right` to implement a function `fact_list : int -> int list` that takes an int `n` and returns `[1!; 2!; ... ; (n − 1)!; n!]`. Your implementation may assume that n ≥ 1. You may also use the following helper function, which uses the `rec` keyword:

```
(* requires: n >= 1
   returns: the list [1;2;...;n] *)
let countup (n:int) : int list =
  (* tail-recursive helper function for countup:
       accumulate the answer in l,
       starting from n and working down *)
  let rec countup' i l =
    if i<=0 then l
    else countup' (i-1) (i::l)
  in countup' n []
```

Problem 3: Matrices (70 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. That length must be at least 1.

Complete the following exercises using any functions you like from the `List` module. **You may not use the `rec` keyword in your solutions.** Again, this prohibition is designed to ensure that you practice folding and other higher-order functions.

Your functions should raise `MatrixFailure` when a matrix argument is invalid or does not conform to any specified preconditions about sizes:

```
exception MatrixFailure of string
```

The behavior of your functions is allowed to be undefined on empty matrices, including matrices such as `[]` and `[[];[]]`. You could raise an exception, or you could do whatever else seems reasonable.

The efficiency of your solutions will be considered when we evaluate style.

Exercise 1.

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 = ()
```

You might find this function helpful in the later exercises! Note: we won't be picky about your formatting when we grade this exercise. Also note: although printing is truly an imperative feature, we nonetheless allow (in fact, require) it in this one exercise.

Exercise 2.

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 function should fail. 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]` should raise `MatrixFailure`

Exercise 3.

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 4.

Implement a function `add_matrices: matrix -> matrix -> matrix` for entry-wise matrix addition. If the two input matrices are not of the same size, `add_matrices` should fail. 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 5.

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, this function should fail. 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]]`

Problem 4: Pattern Matching (60 points)

[code] We have touched upon the benefits of pattern matching, a functional paradigm that you will use heavily in this course. In this problem, we'll implement pattern matching!

Consider the following types:

```
type pat =
  | WCPat (*"wildcard", i.e., underscore*)
  | VarPat of string
  | UnitPat
  | ConstPat of int
  | TuplePat of pat list
  | StructorPat of string * pat option  (*"constructor"*)

type value =
```

```
  | ConstVal of int
  | UnitVal
  | TupleVal of value list
  | StructorVal of string * value option

type bindings = (string * value) list option
```

Throughout the course, you'll use pattern matching to solve a wide variety of problems. In fact, you've already used pattern matching extensively in this problem set—both `List.fold_left` and `List.fold_right` are internally implemented with pattern matching, as you see below:

```
let rec fold_left f acc l =
  match l with
  | [] -> acc
  | hd::tl -> fold_left f (f acc hd) tl
```

In general, the goal of pattern matching is to take some value and determine whether it conforms to a pattern. If so, the match produces a new set of bindings (which can be thought as a list of string-value pairs) that will be used in an expression following the pattern. For example, in `fold_left`, the `hd::tl` case will match any non-empty list, and then bind the head of that list to variable `hd`, and the tail of the list to `tl`. Then, the expression `fold f (f acc hd) tl` is executed with the updated definitions for `hd` and `tl`.

More formally, we can use the following rules to determine whether a pattern matches a value, and the bindings produced by that match:

- `WCPat` can match any value, and it doesn't produce any bindings.
- `VarPat s` also can match any value. It produces the one-element binding `[(s,v)]`.
- `UnitPat` matches only `UnitVal` and doesn't produce any bindings.
- `ConstPat i` matches `ConstVal i` and doesn't produce any bindings.
- `TuplePat pats` matches `TupleVal vals` when `pats` and `vals` have the same size, and for all i, element i of `vals` matches the element i of `pats`. It produces the bindings, appended together, from all the sub-matches.
- `StructorPat (s,p_opt)` matches `StructorVal (s', v_opt)` when `s = s'` (where `=` represents conventional string equality), and either
  - `p_opt = Some p` and `v_opt = Some v` and `p` matches `v`, or
  - `p_opt = None` and `v_opt = None`.

  In the latter case, no new bindings are produced. In the former case, the bindings from matching `p` with `v` are produced.

Exercise 1.

(a) To familiarize you with the `pat` and `value` datatypes, we've provided a function `z`. Write a specification comment for `z`. Try to avoid describing **how** the computation is achieved. Do try to give the reader some intuitive idea of **what** `f1` and `f2` represent. You don't need to submit anything for this sub-exercise.

(b) Use `z` to implement `count_wcs`, a function that counts the number of wildcards found in a pattern.

(c) Use `z` to implement `count_wcs_and_var_lengths`, which counts the wildcards in a pattern and adds that to the sum of lengths of the variable names in the pattern.

(d) Use `z` to implement `count_var`, which counts how often a variable name occurs in a pattern.

Exercise 2.

Implement a function `all_vars_unique: pat -> bool` that determines whether all variables in the pattern have unique names. In addition to writing `all_vars_unique`, implement these *required* helper functions:

```
extract_names: pat -> string list
has_dups: 'a list -> bool
```

The first function should produce a list of the variable names occurring in the pattern. If a variable name occurs n times in a pattern, then there should be n copies in the resulting list. Order does not matter. The second function should check whether duplicates exist in a list of values.

Exercise 3.

Implement a function `all_answers: ('a -> 'b list option) -> 'a list -> 'b list option` that applies the function argument to every element of the list argument. If the function is applied successfully to each element (i.e., each function call returns `Some b_lst`), take each resulting list, and append them together in any order you wish. Then return `Some l`, where `l` is the appended lists. But if any of the function applications returns `None`, then `all_answers` should return `None`. Calling `all_answers f []` should return `Some []`.

Exercise 4.

Implement a function `match_pat: value * pat -> bindings` that checks whether a value matches a pattern. If it does, return `Some l`, where `l` is the (possibly empty) list of bindings produced by the match. If it doesn't match, return `None`.

Exercise 5.

Implement `first_answer: ('a -> 'b option) -> 'a list -> 'b` that applies the function argument to elements of the list argument until that function returns `Some v`—in which case, `first_answer` will return `v`. If `first_answer` never encounters an element that produces `Some v`, it should raise the exception `NoAnswer`.

Exercise 6.

Implement `match_pats: value * pat list -> bindings` that checks whether a value matches any of the patterns in the list argument. If so, return `Some b`, where `b` is the list of bindings produced by the first pattern that matches. Otherwise, return `None`.

Problem 5 (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.

Also, include a statement of what work in this problem set was done by which partner. The ideal case is that each of you contributed to every problem. But (especially since this exercise is ungraded) please be honest about how you divided the work.