Data Structures and Functional Programming	**Problem Set 2**
CS 3110, Spring 2014	**Thursday, February 13**
Version: 1	**Last Modified: February 7, 2014**

Overview

This assignment contains a number of short exercises centered around **folding**, a powerful technique used in many functional programs.

Objectives

- Use folds over lists to implement a variety of tasks
- Gain familiarity with the `List` module
- Generalize the `fold_right` function on lists to other types
- Use lists, options, tuples, and higher-order functions
- Reason about the impact of code style on readability and maintainability

Recommended reading

The following supplementary materials may be helpful in completing this assignment:

- Lectures 4 and 5
- Recitations 4 and 5
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1-3
- The OCaml List module documentation

What to turn in

You should complete the files `warmup.ml`, `lists.ml`, `binary.ml`, `trees.ml`, and `arithmetic.ml` with your solutions for exercises 1, 2–3, 4, 5–6 and 7–8 respectively. Any comments you wish to make can go in `comments.txt` or `comments.pdf`.

If you choose to submit any Karma work, you may submit the file `karma.ml` (be sure to describe what you've done in your `comments.txt`).

Folding on lists

Exercise 1:

To start, you will warm up with a few simple questions using `List.fold_left`. In these exercises you are **not** permitted to use the `rec` keyword.

(a) Write a function `sum (lst : int list) : int` that sums up the elements of `lst`.

(b) Write a function `rev (lst : 'a list) : 'a list` that reverses `lst`.

(c) Write a function `max2 (lst : 'a list ) : int` that returns the second greatest unique element in `lst`, or fails if the list contains fewer than two distinct elements.

Examples

```
# max2 [0; 1; 4; -13];;
- : int = 1

# max2 [1.; 3.5; 3.5; 5.];;
- : float = 3.5

# max2 ["one"; "two"; "three"];;
- : string = "three"

# max2 ["whoops"; "whoops"];;
Exception: Failure "max2: Fewer than two distinct elements"

# max2 [5; 5; 1; 2];;
- : int = 2
```

Exercise 2:

Folds are very general list operations. However, the OCaml List module has many other useful functions that are often better suited to a particular task than a fold. Other times, a library function may not even be the best choice.

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),

(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).

(a) Write a function `lengths (lsts : 'a list list) : int list` that computes an `int list` containing the length of each element of `lsts`. You may use `List.length` in all three strategies for this problem.

```
# lengths_lib [["zar"; "doz"]; []; ["ocaml"; "rocks"]];;
- : int list = [2; 0; 2]
```

(b) Write a function `find_first_value (lst : ('a * 'b) list) (x : 'a) : 'b option` that evaluates to `Some z` for the first pair `(y,z)` in the list such that `x` equals `y`. Return `None` if no such pair exists.

```
# find_first_value_rec [('x', 2); ('y', 4); ('z', 8); ('x', 12)] 'x';;
- : int option = Some 2

# find_first_value_fold [('x', 2); ('y', 4); ('z', 8); ('x', 12)] 'w';;
- : int option = None
```

Exercise 3:

Now write the following functions, using any of the three strategies. It is worth taking some time to choose which strategy or combination of strategies will be most effective before you start coding.

(a) Write a function `confirm_outputs (fs : ('a -> 'b) list) (i : 'a) (o : 'b): bool` that evaluates to `true` if and only if each function in `fs` evaluates to `o` when applied to `i`.

```
# confirm_outputs [(+) 1; (+) 1; fun  x -> 2] 1 2;;
- : bool = true
# confirm_outputs [(+) 1; (+) 2; (+) 1] 1 2;;
- : bool = false
```

(b) Write a function `total_length (lsts : 'a list list) : int` that evaluates to the total number of elements in all the lists in `lsts`.

```
# total_length [[]; [17;2];[5]] ;;
- : int = 3
# total_length [[7;8;9]; [17;2];[5]] ;;
- : int = 6
```

(c) Write a function `find_last_value (lst : ('a * 'b) list) (x : 'a) : 'b option` that evaluates to `Some z` for the last pair `(y,z)` in the list such that `x` equals `y`. Return `None` if `x` does not appear in any pair.

```
# find_last_value [('x', 2); ('y', 4); ('z', 8); ('x', 12)] 'x';;
- : int option = Some 12
# find_last_value [('x', 2); ('y', 4); ('z', 8); ('x', 12)] 'w';;
- : int option = None
```

(d) Write a function `median (lst : 'a list) : 'a option` that evaluates to the median of the list. For even sized lists, choose the lesser of the two middle objects. For empty lists, return None.

```
# median [3;6;9;12;0;2];;
- : int option = Some 3
# median [3;6;12;-4;0;2];;
- : int option = Some 2
# median [];;
- : int option = None
```

Exercise 4:

In this exercise we will represent non-negative integers in binary using the following types:

```
type bit = Zero | One
type bits = bit list
```

As an example, the integer 3110 is represented as the list:

```
[One; One; Zero; Zero; Zero; Zero; One; Zero; Zero; One; One; Zero]
```

We assume that most significant `bit` is at the head of the list. By convention, values of type `bits` should not have leading `Zeros`. For example, we represent 0 as the empty list `[]` and not `[Zero]`.

(a) Write functions

```
bits_to_int : bits -> int
int_to_bits : int -> bits
```

that convert between non-negative integers and their binary representations using `bits`.

```
# int_to_bits 0;;
- : bits = []
# int_to_bits 1;;
- : bits = [One]
# int_to_bits 2;;
- : bits = [One; Zero]
# int_to_bits 42;;
- : bits = [One; Zero; One; Zero; One; Zero]
# bits_to_int [Zero; One; Zero; One; Zero];;
- : int = 10
# bits_to_int [One; One; One; One; One; One; One];;
- : int = 127
```

Your implementation of `int_to_bits` should not produce leading zeros in its output, although `bits_to_int` should correctly handle leading zeroes in the input.

(b) Write a function `binary_addition` that implements addition of numbers represented in binary using `bits`.

```
binary_addition: bits -> bits -> bits
```

Your implementation should remove any leading zeros from the output (although it should correctly handle leading zeroes in the input).

```
# binary_addition [Zero; One; Zero] [Zero; Zero; One];;
- : bits = [One; One]
# binary_addition [Zero] [Zero; Zero; Zero];;
- : bits = []
```

Your implementation must also work on the `bits` directly (in particular, operations such as `(+)` or `(-)` are forbidden).

We have provided you with some helper functions in the `Binary` module (`binary.ml` and `binary.mli`) that you may wish to use. You may also find the library functions `List.fold_left2` and `List.fold_right2`, as well as the helper function `normalize` useful in developing your solution.

Folding on trees

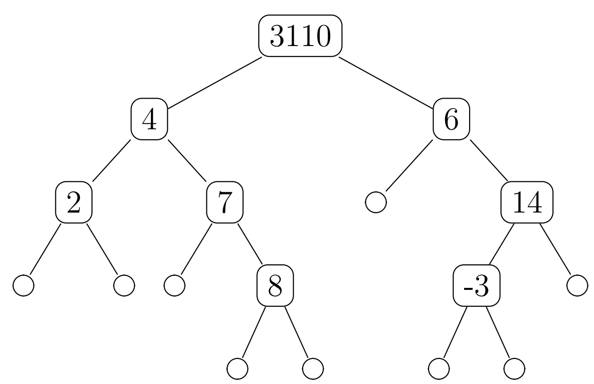
In this section, we will work with the following binary tree type:

```
type 'a bintree = Leaf | Node of 'a bintree * 'a * 'a bintree
```

We will refer to the following tree in our examples:

```
# let example_tree =
  Node (
    Node (
      Node (Leaf, 2, Leaf),
      4,
      Node (Leaf,
            7,
            Node (Leaf, 8, Leaf))),
    3110,
    Node (
      Leaf,
      6,
      Node (Node (Leaf, -3, Leaf),
            14,
            Leaf)));;

example_tree : int bintree = ...
```



Exercise 5:

(a) Write a function `tree_sum : int bintree -> int` such that `tree_sum t` evaluates to the sum of the elements in `t`. Example:

```
# tree_sum example_tree;;
- : int = 3148
```

(b) Write a function `tree_mem : 'a -> 'a bintree -> bool` that evalutates to `true` if the element exists in the tree and `false` otherwise. Example:

```
# tree_mem 8 example_tree;;
- : bool = true

# tree_mem 8 Leaf;;
- : bool = false

# tree_mem 100 example_tree;;
- : bool = false
```

(c) Write a function `tree_preorder : 'a bintree -> 'a list` such that `tree_preorder t` evaluates to a list containing the data in `t` ordered by preorder traversal.

```
# tree_preorder example_tree;;
- : int list = [3110; 4; 2; 7; 8; 6; 14; -3]
```

(d) Write a function `tree_inorder : 'a bintree -> 'a list` such that `tree_inorder t` evaluates to a list containing the data in `t` ordered by inorder traversal.

```
# tree_inorder example_tree;;
- : int list = [2; 4; 7; 8; 3110; 6; -3; 14]
```

(e) Write a function `tree_postorder : 'a bintree -> 'a list` such that `tree_postorder t` evaluates to a list containing the data in `t` ordered by postorder traversal.

```
# tree_postorder example_tree;;
- : int list = [2; 8; 7; 4; -3; 14; 6; 3110]
```

Exercise 6:

(a) Identify the similarities and differences between the functions you implemented in Exercise 5. Write a single higher-order function

```
tree_fold : 'b -> ('a -> 'b -> 'b -> 'b) -> 'a bintree -> 'b
```

that abstracts their common structure. For example, you should be able to write the following:

```
# let tree_sum_fold = tree_fold 0 (fun l x r -> l + x + r)
tree_sum_fold : int bintree -> int

# tree_sum_fold example_tree
- : int = 3148

# let tree_count_fold = tree_fold 0 (fun l x r -> l + 1 + r)
tree_count_fold : 'a bintree -> int

# tree_count_fold example_tree
- : int = 8
```

(b) Use `tree_fold` to reimplement the functions from the previous exercise in the file `trees.ml`. Name them `tree_sum_fold`, `tree_mem_fold`, `tree_preorder_fold` `tree_inorder_fold`, and `tree_postorder_fold`. You may not use the `rec` keyword for these implementations.

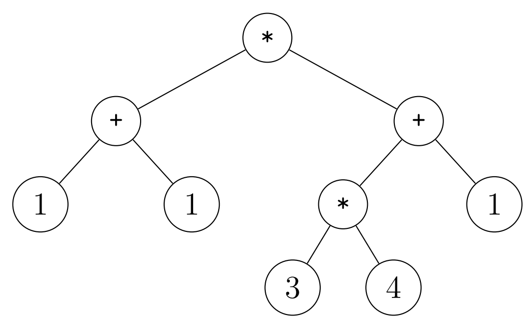
Folding on expressions

In this section we will work with arithmetic expressions constructed from integers and the operators `+` and `*`. We represent these expressions using values of the following type:

```
type exp = Val of int | Plus of exp * exp | Times of exp * exp
```

For example, the expression `(1 + 1) * ((3 * 4) + 1)` would be represented as follows:

```
# let example_exp =
  Times (
    Plus ( Val 1 , Val 1 ) ,
    Plus (
      Times ( Val 3 , Val 4 ) ,
      Val 1
      )
    );;
example_exp : exp = ...
```



Exercise 7:

Write a function

```
exp_fold (val_op   : int -> 'a)
         (plus_op  : 'a -> 'a -> 'a)
         (times_op : 'a -> 'a -> 'a)
         (exp : exp) : 'a
```

that recursively processes each subexpression of `exp` and then combines their results using the following scheme:

- It applies the operation `val_op` to each `Val` subexpression.
- It applies the operation `plus_op` to each `Plus` subexpression.
- It applies the operation `times_op` to each `Times` subexpression.

Exercise 8:

Use your implementation of `exp_fold` to complete the following functions. You may not use the `rec` keyword in this exercise.

(a) Write a function `eval : exp -> int` that takes an input expression and evaluates it according to the rules of ordinary integer arithmetic.

```
# eval example_exp;;
- : int = 26
```

(b) Write a function `to_string : exp -> string` that returns a fully parenthesized string representation of the input expression, with no spaces between operators and their operands.

```
# to_string example_exp;;
- : string = "((1+1)*((3*4)+1))"
```

Comments

[written] At the end of the file, please include any comments you have about the problem set, or about your implementation. This would be a good place to document any extra Karma problems that you did (see below), to list any problems with your submission that you weren't able to fix, or to give us general feedback about the problem set.

Release files

We have provided you with the files `warmup.ml`, `lists.ml`, `mystery.ml`, `trees.ml`, and `arithmetic.ml`, with signatures under which your solutions should go. Each also has a corresponding `.mli` file.

Karma Suggestions

> **Note:** We encourage you to think about different directions that you can take the problem sets or different parts of OCaml or functional programming that you are curious about.
>
> You may submit any extra work you do in the Karma section of each problem set. We will comment on any extra work that you turn in, but

Karma **is completely optional and will not affect your grade in any way**.

Here are some suggestions that may pique your interest:

- Implement `median` as a recursive function. This can be done in $O(n)$ time.
- Implement `fold_left` in terms of `fold_right` and vice-versa.
- Write a version of `tree_fold` to work for nodes with arbitrarily many children.
- Write a version of `tree_fold` to work for nodes with **infinitely** many children.
- Extend the `expr` type to include `if-then-else` expressions. Extend `expr` to include other OCaml features.