CS 3110 Spring 2017
Problem Set 2
Version 1 (last modified February 23, 2017)

Overview

This assignment reinforces the use of recursion when approaching certain types of problems. We will emphasize the use of `map`, `fold\_left`, and `fold\_right` as well as work on many methods involving list manipulation. Furthermore, this problem set contains two sections that introduce the use of data structures in OCaml by defining new types. **This assignment must be done individually**.

Objectives

- Gain familiarity and practice with folding and mapping.
- Practice writing programs in the functional style using immutable data, recursion, and higher-order functions.
- Introduce data structures in OCaml and get familiar with using defined types for manipulating data.

Recommended reading

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

- Lectures 2 3 4 5
- The CS 3110 style guide
- The OCaml tutorial
- Introduction to Objective Caml
- Real World OCaml, Chapters 1-3

What to turn in

Exercises marked [code] should be placed in corresponding `.ml` files and will be graded automatically. Please do not change any `.mli` files. Exercises marked [written], if any, should be placed in `written.txt` or `written.pdf` and will be graded by hand. **Karma questions are not required and will not affect your grade in any way. They are there as challenges that we think may be interesting.**

Compiling and testing your code

For this problem set we have provided a Makefile and testcases.

- To compile your code, run `make main`
- To test your code, run `make test`
- To remove files from a previous build, run `make clean`

Please note that any submissions that do not compile will get an immediate 10% deduction from their original grade.

Some final notes

For loops and while loops are not allowed; just try to think recursively!

We are not looking for highly optimized or efficient code, just code that shows good use of functional programming concepts. You are free to write internal helper functions (but not external helper functions). If a function contains rec, your internal functions may also be recursive (and if a function does not contain rec, recursive helper functions will result in loss of points). You are free to use any previously defined function in subsequent problems.

Between The Folds

[code] All the subparts of this exercise must be implemented **entirely** by

- Writing one or more internal 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.

The signatures for all these functions are in the file fold_recs.ml. Note that not all these steps are necessary for every exercise. You may also use `List.rev`, as well as `List.hd` and `List.tl` (but beware of using the latter two 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 List module functions other than `fold_left`, `fold_right`, `rev`, `hd`, and `tl`.** These prohibitions are designed to ensure that you get practice using folding.

(a) [code] Implement the `tail_rec_map` function, a tail recursive implementation of List.map.

```
let tail_recursive_map (f: 'a -> 'b) (l: 'a list) : 'b list = ...
```

For example:

```
# tail_recursive_map (fun x -> x+1) [1; 2; 3];;
- : int list = [2; 3; 4]
```

(b) [code] Implement the `mapi` function. Function f is applied to the index of the element as the first argument and the element itself as the second argument.

```
let mapi (f: int -> 'a -> 'b) (l: 'a list) : 'b list = ...
```

For example:

```
# mapi (fun i elt -> if (i mod 2 = 0) then elt+1 else elt+2)
  [1; 2; 3; 4; 5];;
- : int list = [2; 4; 4; 6; 6]
```

(c) [code] Implement the `count_vote` function, which takes a boolean list and outputs a tuple (t, f) containing t, the number of true elements in the list, and f, the number of false elements in the list.

```
let count_vote (l: bool list) : int * int = ...
```

For example:

```
# count_vote [true; true; true; false; true; false];;
- : int * int = (4, 2)
```

(d) [code] Implement the `partition` function, which takes a predicate function and a list, and outputs a tuple (l1, l2) where l1 contains the elements of the list that satisfy the predicate, and l2 contains the elements of the function that do not.

```
let partition (p: 'a -> bool) (l: 'a list) : 'a list * 'a list = ...
```

For example:

```
# partition (fun x -> x > 3) [1; 3; 4; 2; 5; 2; 0; 8];;
- : int list * int list = ([4; 5; 8], [1; 3; 2; 2; 0])
```

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:

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

(e) [code] Implement the two functions, `scan_left` and `scan_right`.

```
let scan_left (f: 'a -> 'b -> 'a) (acc: 'a)
  (l: 'b list) : 'a list = ...
```

```
let scan_right (f: 'a -> 'b -> 'b) (l: 'a list)
  (acc: 'b)  : 'b list = ...
```

Recursion Returns

[code] All the subparts of this exercise must be implemented **entirely** by using recursion. You can **not** use list functions, with the exception of List.rev and List.length. The signature of these subparts are in the fold_recs.ml file.

(a) [code] Implement the `sep_list` function, which separates a list of elements into consecutive element sublists.

```
let rec sep_list (l: 'a list) : 'a list list = ...
```

For example:

```
# sep_list [1; 1; 1; 2; 3; 3; 1; 1; 5; 5; 5; 5];;
- : int list list = [[1; 1; 1]; [2]; [3; 3]; [1; 1]; [5; 5; 5; 5]]
```

(b) [code] Implement the `slice_list` function, which extracts a slice from a list.

```
let rec slice_list (l: 'a list) (i: int) (j: int) : 'a list = ...
```

Given indices i and j, `slice_list` returns the list of elements from index i to j, inclusive. Assume $0 \leq$ i and j $<$ length of l. For indices that are out of bounds, the behavior is unspecified. For example:

```
# slice_list ["a"; "b"; "c"; "d"; "e"; "f"; "g"] 1 4;;
- : string list = ["b"; "c"; "d"; "e"]
```

(c) [code] Implement the `rotate_list` function, which rotates a list's elements by n places to the left. If n $>$ length of the list, the function should continue to rotate.

```
let rec rotate_list (l: 'a list) (n: int) : 'a list = ...
```

For example:

```
# rotate_list ["a"; "b"; "c"; "d"; "e"; "f"; "g"] (2);;
- : string list = ["c"; "d"; "e"; "f"; "g"; "a"; "b"]

# rotate_list ["a"; "b"; "c"; "d"; "e"; "f"; "g"] (-2);;
- : string list = ["f"; "g"; "a"; "b"; "c"; "d"; "e"]

# rotate_list ["a"; "b"; "c"; "d"; "e"; "f"; "g"] (10);;
- : string list = ["d"; "e"; "f"; "g"; "a"; "b"; "c"]
```

For the next questions, we define our own list type:

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

(d) [code] Implement the `fold_left` function for 'a mylists.

```
let rec fold_left (f: 'a -> 'b -> 'a) (acc: 'a)
  (l: 'b mylist) : 'a = ...
```

(e) [code] Implement the `fold_right` function for 'a mylists.

```
let rec fold_right (f: 'a -> 'b -> 'b) (l: 'a mylist)
  (acc: 'b) : 'b = ...
```

The Matrix

For this problem, you will write your solutions in the last section of the fold_recs.ml file. 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, we will represent matrices as int list lists. For example, the matrix m, represented as follows,

```
let m = [[1; 2; 3]; [4; 5; 6]]
```

has element 2 at location [0][1].

A valid matrix is an int list list that has at least one row, at least one column, and in which every column has the same number of rows. There are many values of type int list list that are invalid. For example:

```
[]
[[1; 2]; [3]]
```

(a) [code] Implement the `is_valid_matrix` function, which returns whether the input matrix is valid.

```
let rec is_valid_matrix (m: 'a list list) : bool = ...
```

For example:

```
# is_valid_matrix [[2; 3]; [4; 5]; [0; 1]] ;;
- : bool = true
# is_valid_matrix [[2; 3]; [5]; [0; 1]] ;;
- : bool = false
```

(b) [code] Implement the `add_matrices` function, which performs matrix addition. If the two input matrices are not the same size, the behavior is unspecified.

```
let rec add_matrices (m1: int list list)
  (m2: int list list) : int list list = ...
```

For example:

```
# add_matrices [[2; 3]; [4; 5]; [0; 1]] [[0; 2]; [3; 2]; [1; 1]];;
- : int list list = [[2; 5]; [7; 7]; [1; 2]]
```

(c) [code] KARMA: Implement the `mult_matrix` function, which performs matrix multiplication. If the two input matrices are not of sizes that can be multiplied together, the behavior is unspecified.

```
let rec mult_matrices (m1: int list list)
  (m2: int list list) : int list list = ...
```

For example:

```
# mult_matrices [[2; 1]; [3; 2]] [[-8; -4; 3]; [-2; 1; 4]] ;;
- : int list list = [[-18; -7; 10]; [-28; -10; 17]]
```

Tries - A New Hope

For this problem, we will ask you to implement some functions that allow users to create, manipulate, and work with a very interesting type of data structure called Trie. Tries, simply put, are a type of search tree that are usually used to store strings. The general purpose of a trie is to determine whether a word has been inserted into the data structure in linear time. What makes tries so special is that they allow users to quickly determine if a given string is a prefix of any of the words we have stored in the trie. This is why tries are very useful for certain applications involving dictionaries.
For this particular problem, we will consider a more general variation of tries that will allow us to insert duplicate words into the data structure so that we can later count how many words have been inserted in total. The definition of a trie, for the sake of this problem set, is the following one:

- A trie is a search tree T where the every node in T has a number associated to it and every edge in T has a label, which in this case will be a character, associated to it. We will refer to the number associated to node v as the count of v.
- When we initialize an empty trie, we will construct a root node v_o whose count is initialized as zero and edge set is empty. The count in this root node will represent the number of times we have inserted the empty string "" into our trie T.
- For every node w in T, if there is a path from v_o to w with edges $e_1, e_2, ..., e_m$ and corresponding labels (i.e. letters) $c_1, c_2, ..., c_m$, then the number of times we have inserted the word $c_1c_2 \cdots c_m$ in T is going to be equal to the number stored in node w (i.e. the count of w).

For our implementation, we will assume that all words are in lowercase and they have no blank-spaces in them. For example, a trie containing the words "ocaml", "ocam", "help", "hello", "hell", "help" will look like the tree in figure 1. Note that in that figure the count of each node is the integer inside the node itself and the red characters are the labels of each edge.

Now in order to implement in this in OCaml, we will define the following type:

```
type trie = Trie of int * ((char*trie) list)
```

Here we recursively encode a trie T as a tuple (c_T, L_T) where c_T represents the count at the root of T, and L_T is a list of elements of the form (c_i, T_i) that represent the set of edges for the root ((c_i, T_i) represents an edge with label c_i that goes to sub-tree T_i). For example, under this definition we can define an empty trie as

```
let empty_trie = Trie (0, [])
```

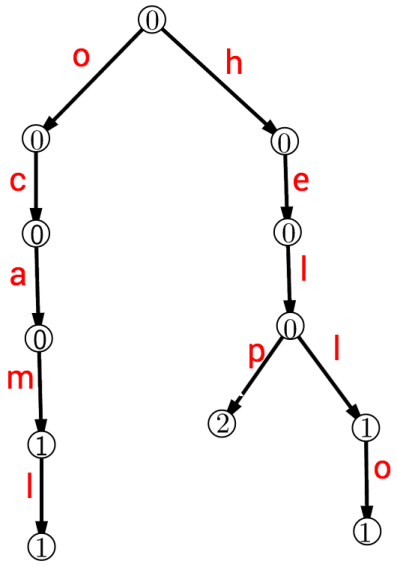

Figure 1: example of a Trie.

With this type definition in hand, please refer to tries.ml in order to complete the following methods:

1. [code] Implement the function `add_string` that takes an instance `t` of type `trie` and a string s and returns a **new** instance of type `trie` that is exactly like `t` but with the string s now inserted into the trie. You may assume that the input string is in lowercase and does not contain any whitespace.

2. [code] Implement the function `remove_string` that takes a `(t:trie)` and a `(s:string)` and returns a trie that represents the result of removing **one copy** of `s` from `t`. If s is not a string in t, then this method simply returns the original trie. Furthermore, if the trie has two or more copies of `s` inserted in it, then we simply remove one of the copies.
Hint: this is easier than what it sounds. As far as we are concerned, we do not care about memory consumption right now so you don't have to remove extra nodes from the trie if this makes removal easier.

3. [code] Implement the function `total_num` that takes a trie and returns **the total number** of strings that have been inserted into the trie. For the example in figure 1, this function would return 6.

4. [code] Implement the function `count_string` that takes a trie and a string and returns the number of times we have inserted the given string into the trie. If the string has not been inserted, then this function should return zero.

5. [code] Implement the function `get_unique_strings` that takes a trie and returns a string list that contains **all** the unique strings that have been inserted into the trie. This is, if a string has been inserted twice, we just include it once in the resulting list. The

ordering of the strings in the resulting list can be arbitrary. For our example in figure 1, this function would return the list

```
["ocaml"; "ocam"; "help"; "hello"; "hell"]
```

6. [code] Implement the function `prefix_strings` that takes a trie and a string representing a prefix and returns a list containing all unique strings that have been previously inserted into the trie and start with the given prefix.
   For example, if we try this function on the trie of figure 1 with prefix "he", the resulting list will be

   ```
   ["help"; "hell"; "hello"]
   ```

   If there are no strings with the given prefix, then simply return an empty list. **Hint:** you are allowed to reuse any of the functions above to help you implement this function.

KARMA A Game of Words

After retiring from the presidency, Barack Obama finds himself pretty bored and constantly looking for ways to spend his free time. One afternoon, while talking about life and how short it is with his wife Michelle, they decided to learn Spanish; thinking that learning this language will let them enjoy a nice vacation together in the Galapagos Islands. In order to facilitate this learning process, they decide to play the following game: Michelle will select a random set of non-empty words $\mathcal{W}$ from a Spanish dictionary she owns. With this set of words in hand, they will construct a word in Spanish together by starting with the empty string and, **in alternating turns**, add one letter to the string with the only condition that, **at all times**, the resulting string must be a valid prefix of **at least one string** in $\mathcal{W}$. However, in order to make their time together even more fun they agreed that they will play this game T times and the ultimate-winner of the day will be the person that wins the T^{th} game. Because Michelle was the one who selected the words in $\mathcal{W}$, they agreed that **Barack will be the first player of the first game** and whomever looses the i^{th} game will be the first player of the $(i+1)^{th}$ game.

Given all of this, implement the function `word_play_winner` that takes a list of strings `words` and a positive integer n and returns `"Barack"` if Barack has a winning strategy to obtain the title of "ultimate-winner", assuming they both play optimally, and `"Michelle"` otherwise. You may use any of the functions/data structures implemented in this problem set to solve this problem.
Hint: tries and prefixes usually like to hang-out a lot together.

KARMA A Clockwork Church

The objective of this problem is to get you more comfortable with the use of high-level functions by introducing a concept that is very interesting and mathematically elegant. In this section, we will work describe a way to represent natural numbers using only functions. This form of encoding was first described by the mathematician Alonzo Church in order to show that his lambda calculus, the most primitive and simple functional language, could encode any data structure with the simple use of function applications. This is a very powerful idea as it shows how simple things can build up to encode the complexity we usually attribute to most computations.

The idea behind this encoding is the following: we will encode a natural number n with a high-order function that takes as an argument an arbitrary function f and a value x. The function returns the result of the n-fold application of function f on value x. In other words, the integer n is represented by a function $g_n(f, x)$ that returns the following value:

$$g_n(f, x) = f^n(x) = \underbrace{f(f(\cdots f(x))\cdots)}_{\text{n times}}$$

With this definition, we can nicely and elegantly encode operations of naturals such as addition, subtraction, multiplication, and exponentiation. For example, the function that encodes the number 0 will be the identity function $g_0(f, x) = x$.

The function f that is fed into our encoding $g_n(f, x)$ for a natural number n must have type $\alpha \to \alpha$. The type of x must also be α. This would imply that the type of a church encoding for n will be $(\alpha \to \alpha) \to \alpha \to \alpha$. In order to work with these encodings in OCaml, we will define the following OCaml type that allow us to represent a high-order function whose type is $(\alpha \to \alpha) \to \alpha \to \alpha$:

```
type church = Ch of ((church -> church) -> church -> church)
```

For example, under this scheme we can define an encoding for zero as

```
let zero = Ch (fun f -> fun x -> x)
```

Furthermore, we can make functions that operate on Church encodings like the following function `inc` that takes a Church encoding of n and returns the Church encoding of $n + 1$

```
let inc (Ch n:church) : church = (fun f -> fun x -> f (n f x))
```

Note that all we do is return a high-order function that takes a function f, and element x and returns $f(g_n(f, x))$. This is equivalent to applying f $n + 1$ times on argument x. With this as a tool, we will ask you to implement the following functions:

1. [code] Implement the constant `one` that represents the church encoding that represents the natural number 1.

2. [code] Implement the function `add` that takes two church encodings n and m and returns the church encoding of the addition of n and m.

3. [code] Implement the function `mult` that takes two church encodings n and m and returns the church encoding of the product of n and m.

4. [code] Implement the function `exp` that takes two church encodings n and m and returns the church encoding of n^m. In this case the first argument is n and the second argument is m.

5. [code] Now that we can encode natural numbers, we can easily encode booleans as well through the following definitions:

   ```
   let true_c = one
   let false_c = zero
   ```

   Using this definitions, implement the function `is_zero` that takes a church numeral n and returns `true_c` if n is equal to zero, otherwise it returns `false_c`.

Comments

[written] 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

The accompanying release file `ps2.zip` contains the following files:

- `writeup.pdf` is this file.
- `release/fold_recs.ml`, `release/trie.ml`, `release/word_game.ml`, and `release/church.ml` are the files that you need to fill in.
- `release/fold_recs.mli`, `release/trie.mli`, `release/word_game.mli`, and `release/church.mli` contain the interface and documentation for the functions that you need to implement.
- `Makefile` a make file to build the files related to this assignment. Please do not change this file.
- `main.ml` is a dummy file used to compile all the files for this assignment.

- `test.ml` has tests for the first two parts of the assignments.

  Important note: these tests are not comprehensive and we strongly suggest that you should further test your implementation.

- `updates.txt` information on revisions of this pset