CS 3110 Spring 2015	**Soft deadline: 02/12/15 11:59 pm**
Problem Set 1	**Hard deadline: 02/14/15 1:10 pm**
Version 1 (last modified February 3, 2015)	

Objectives

- Gain familiarity with basic OCaml features such as lists, tuples, functions, pattern matching, datatypes, and basic features of the OCaml type system.
- Practice writing programs in the functional style using immutable data and recursions.
- Appreciate the impact of code style on readability, correctness, and maintainability.

Recommended reading

The following materials should be helpful in completing this assignment:

- Course readings: lectures 1, 2, 3, and 4; recitations 1, 2, and 3
- The CS 3110 style guide
- The OCaml tutorial
- Real World OCaml, Chapters 1–3

What we provide

We provide two files as part of the release code:

1. `ps1.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.
2. `ps1_test_public.ml`, which is a minimal test file giving a single unit test for each function you must write. 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 of these files.

What to turn in

Submit these files on CMS:

1. A file `ps1written.pdf` containing your answers to the written portions of this problem set, which are identified below as "[written]".
2. Files `ps1.ml` and `ps1_test.ml` containing your solutions and unit tests for the coding exercises of this problem set, which are identified below as "[code]".

Hard deadline

Please note that, because of February break, the hard deadline will be **Saturday, February 14, at 1:10 pm**. No submissions will be accepted after this time.

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` cells, the `Array` module, and `mutable` fields—are not permitted in your solutions to this problem set. Loops (including `while` and `for`) are also prohibited. 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.

Problem 1 (15 points)

Exercise 1.

[written] Give the OCaml type (if any) of each of the following OCaml expressions, and also give the value to which each well-typed expression reduces. For each expression that is not well-typed, briefly explain why.

(a) `3000 + 110`

(b) `(-) 22 20`

(c) `[3110]`

(d) `"zar"::[3]`

(e) `"zar" ^ "doz"`

(f) `Some ()`

(g) `(fun x -> x)`

(h) `match [1;2;3] with _ -> "fun"`

Exercise 2.

[written] Show the evaluation of each of the following OCaml expressions. Use the substitution model, and show each step of evaluation. For example:

```
let x = 3 + 5 in x + x
--> let x = 8 in x + x  (because 3+5 --> 8)
--> 8 + 8               (because x+x{8/x} is 8+8)
--> 16
```

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

(b)
```
let x = 0 in
let x = 1 in
match x with
| 0 -> "zar"
| 1 -> "doz"
| x -> "zed"
```

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

Problem 2 (10 points)

[written] Give OCaml expressions that have the following types. You may not use type annotations.

(a) `string`

(b) `int list`

(c) `string -> string list`

(d) `string list -> string`

(e) `int -> int -> int`

(f) `int * char -> int`

(g) `int * char list -> (int * char) list`

(h) `student`

(i) `student -> string * string`

(j) `string -> string -> float -> student`

For parts h, i, and j, assume the following type definition:

```
type student = {first_name:string; last_name:string; gpa:float}
```

Problem 3 (10 points)

[written] The following function executes correctly, but it was written with poor style. Your task is to rewrite it with better style. Please consult the CS 3110 style guide on the course website. You should invent a new name for the function that appropriately conveys what it does.

```
let zar (a : 'a list) =
  let rec doz (b : 'a list) (c : 'a list) =
    if b = [] then let d = c in d
    else begin
      let x = List.hd b in
      let y = List.tl b in
      let z = [x] in
      let n = z@c in
      doz y n end
  in
  doz a []
```

Problem 4 (90 points)

[code] Complete each of the 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 brief description of the function (one or two sentences).
   - A brief description of each argument (a couple of words).
   - A concise and accurate description of the function's *precondition* and *postcondition*.

   You do not need to be redundant. For example, if the documentation of the postcondition already describes each argument, there is no need to separately document the arguments.
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 `ps1_test.ml`.

Exercise 1.

Write a function `every_nth : 'a list -> int -> 'a list` that takes any list ℓ and a positive integer n and returns a list containing every nth element of ℓ in the same order that the elements appear in ℓ. If the length of the list is less than n, there is no nth element. For example:

- `every_nth [1;2;3;4;5] 2 = [2;4]`
- `every_nth [1;2;3;4;5;6;7] 3 = [3;6]`
- `every_nth [1;2;3] 1 = [1;2;3]`
- `every_nth [3;6;9;12] 21 = []`

Exercise 2.

Write a function `is_unimodal: int list -> bool` that takes an integer list and returns whether that list is unimodal. A *unimodal list* is a list that monotonically increases to some maximum value then monotonically decreases after that value. Either or both segments (increasing or decreasing) may be empty. For example:

- `is_unimodal [1;2;3;6;9;5;4] = true`
- `is_unimodal [1;3;5;7;5;6] = false`

- `is_unimodal [1;1;2;3;4;4;3;2;2;-1] = true`
- `is_unimodal [] = true`
- `is_unimodal [1;1;1] = true`
- `is_unimodal [1;2] = true`
- `is_unimodal [2;1] = true`

Exercise 3.

Write a function `complete_list: 'a list -> ('a list * 'a list) list` that takes any list ℓ and returns a list of pairs of non-empty lists ℓ_1 and ℓ_2 such that $\ell_1 @ \ell_2 = \ell$. Moreover, the returned list should the biggest such list possible. For example:

- `complete_list [] = []`
- `complete_list [1] = []`
- `complete_list [1;2] = [([1],[2])]`
- `complete_list [1;2;3;4;5] =`
  `[([1;2;3;4],[5]); ([1;2;3],[4;5]); ([1;2],[3;4;5]); ([1],[2;3;4;5])]`

The order of the pairs in the returned list is unspecified. For example, the last invocation above could instead return the output in a different order:

- `complete_list [1;2;3;4;5] =`
  `[([1;2;3],[4;5]); ([1;2],[3;4;5]); ([1],[2;3;4;5]); ([1;2;3;4],[5])]`

Exercise 4.

Write a function `rev_int: int -> int` that takes an integer i and returns an integer whose digits are the reverse of i. The sign should remain unchanged. If the reversed integer is larger than `max_int`, the behavior is unspecified: your function can do whatever you want. For example:

- `rev_int 1234 = 4321`
- `rev_int 4 = 4`
- `rev_int (-1234) = (-4321)`
- `rev_int (-10) = (-1)`
- `rev_int 11111111 = 11111111`
- `rev_int 1073741822 = (*undefined: 2281473701 > max_int *)`

Exercise 5.

Flattening is the process of converting a list of lists into a single list (see `List.flatten`). Write the reverse operation `unflatten: int -> 'a list -> 'a list list option` that takes an integer k and list `lst` and breaks it up into a list of lists, each of size k. In the case that `List.length lst` is not a multiple of k, the last list is allowed to be of size less than k. If `k` ≤ 0, then `unflatten k l` should return `None`. For example:

- `unflatten (-1) [1;2;3;4;5;6] = None`
- `unflatten 0 [1;2;3;4;5;6] = None`
- `unflatten 2 [1;2;3;4;5;6] = Some [[1;2]; [3;4]; [5;6]]`
- `unflatten 3 [1;2;3;4;5;6;7;8] = Some [[1;2;3]; [4;5;6]; [7;8]]`
- `unflatten 6 [1;2;3;4;5;6] = Some [[1;2;3;4;5;6]]`
- `unflatten 7 [1;2;3;4;5;6] = Some [[1;2;3;4;5;6]]`
- `unflatten 2 [] = Some [[]]`

Exercise 6.

A *Roman numeral* can be represented as a list of letters from the set {`I`, `V`, `X`, `L`, `C`, `D`, `M`}. The value of each letter is as follows:

Column 1		Column 2	
`I`	1	`V`	5
`X`	10	`L`	50
`C`	100	`D`	500
`M`	1000		

The letters are usually ordered from greater-valued to smaller-valued. If that ordering is broken, it means that the immediately preceding (lower) value is deemed to be "negative" and should be subtracted from the higher (out of place) value. For example, `IV` represents 4, and `XC` represents 90.

For purposes of this problem (i.e., don't worry about whether this corresponds to other definitions you've learned in the past), define a *valid* Roman numeral as follows:

1. A letter from Column 1 may never appear more than three times in a row, and there may never be more than one additional occurrence of that letter.
2. A letter from Column 2 may never appear more than once.
3. Once a letter Z has been used in a "negative" position, all subsequent letters except the immediately following character may not be greater than Z.

For example, `MMMCX` is valid Roman numeral, but `XCMMM` is not.

We can define a type for Roman numerals in OCaml as follows:

```
type numeral = I | V | X | L | C | D | M
type roman = numeral list
```

Complete `int_of_roman`, assuming only valid Roman numerals as input. Note that your implementation may assume validity—it does not need to check for validity.

```
let rec int_of_roman (r : roman) : int =
  let int_of_numeral = function
    | I -> 1
    | V -> 5
    | X -> 10
    | L -> 50
    | C -> 100
    | D -> 500
    | M -> 1000 in
  ???
```

For example:

- `int_of_roman [I; I; I] = 3`
- `int_of_roman [X; L; I; I] = 42`
- `int_of_roman [M; C; M; X; C; I; X] = 1999`

Problem 5 (0 points)

[written,ungraded] At the end of your file of written problems, 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.