CS 3110 Spring 2017
Problem Set 1
Version 1 (last modified February 7, 2017)

Overview

This assignment will take you through a sequence of small problems that will introduce you to writing (good) functional programs in OCaml. This assignment must be done individually.

Objectives

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

Recommended reading

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

- Lectures 1 2 3
- 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 `ps1.ml` and will be graded automatically. Exercises marked [written] should be placed in `written.txt` or `written.pdf` and will be graded by hand.

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.

Warm-up

Exercise 1.

[written] Identify the types and values of the following expressions. If the expressions are not well typed, briefly explain why not.

> **Note:** Although the toplevel will give you the answers to these questions, we recommend that you try them on your own before checking them against the toplevel. Figuring out types is a good skill to have for reading OCaml code and for passing 3110 exams.

(a) `14 * 3`

(b) `14 ** 3`

(c) `3 > 2`

(d) `3.1 > 3`

(e) `3.1 > 3.`

(f) `("zardoz", 42, true)`

(g) `["zardoz"; 42; true]`

(h) `fun a b c -> (a ^ b) :: c`

(i) `fun x -> (+) :: []`

(j) `fun x -> fun y -> if x then y + 3 = 4 else x`

Exercise 2.

[written] Give expressions having the following types. Your type must match these types exactly.

(a) `int -> int -> int`

(b) `(bool -> bool) -> bool`

(c) `bool -> (bool -> bool)`

(d) `'a -> 'a`

(e) `'a * 'b list -> ('a * 'b) list`

(f) `('a * 'b -> 'c) -> ('a -> 'b -> 'c)`

(g) `ta`

(h) `ta -> ta list -> int`

(i) `bool option`

(j) `unit -> int`

For (g) and (h) use the following types:

```
type deg = UGRAD | MENG | PHD;;
type ta = { name : string ; course : int; degree : deg };;
```

Evaluation Rules

Exercise 3.

> **Note:** Again, we recommend you try this exercise on your own before running toplevel.

[written] The following two expressions behave differently. Explain what happens in the first expression, explain what happens in the second expression, and explain how these differ. How does this relate to eager evaluation, lazy evaluation, and canonical values in OCaml?

```
let rec loop a = loop a in
(fun x -> 3) (loop 4)
```

```
let rec loop a = loop a in
(fun x -> 3) (fun y -> loop 4)
```

Code Style

Exercise 4.

[written] The following function executes correctly, but was written with poor style. Rewrite it with better style. Please consult the CS 3110 style guide.

```
let process m =
let foo t =
  match t with (x, y, z) ->
  match x with
  | 0 -> (
    match y with
    | 0 -> z
    | _ -> z = false
  )
  | 1 -> false
  | -1 -> false
  | 2 -> not (y > 0)
  | _ -> true in
let bar x =
let square y = y * y in
square x in
if not (foo (fst m, (let v = snd m in v * v), false)) then bar (fst m) else -1
```

Examples

The expression

```
fun x ->
  match 3110 with
  | 3110 -> if x=true then "cs" else "3110"
  | _ ->  "2110"
```

This will become

```
fun x -> if x then "cs" else "3110"
```

OCaml programming with Ints

Exercise 5.

[code] Complete the implementation of the `factorial` function, where `factorial n` returns the $n!$.

If $n < 0$, the behavior is undefined.

```
let rec factorial (n: int) : int = ...
```

Exercise 6.

(a) [code] Implement the `is_prime` function, which outputs whether or not a number is prime.

```
let rec is_prime (n : int) : bool = ...
```

(b) [code] Implement the `nth_prime` function, which outputs the n^{th} prime number. We will 0 index, so the 0^{th} prime number is 2. Behavior is undefined for inputs < 0 (you can return whatever you want in these cases)

```
let rec nth_prime (n : int) : int = ...
```

(c) [code] Implement the `prime_factorization` function, which takes an integer `n` and returns the list of prime factors in any order. Multiplicity matters. Behavior is undefined on inputs < 2

```
let rec prime_factorization (n: int) : int list = ...
```

For example:

```
# prime_factorization 12;
- : int list = [2; 3; 2]
```

OCaml programming with Lists

You may NOT use any List module functions (except for cons (::) and concat (@))

Exercise 7.

(a) [code] Implement the following function:

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

`map` takes two inputs: a function `f` that maps an element of type a to an element of type b, and a list `l` of type b. The `map` function should return a list that results from applying the function `f` on each element in the list `l`. Output order should correlate to input order. For example:

```
# map (fun x -> x * 2) [1; 2; 3; 4; 5];;
- : int list = [2; 4; 6; 8; 10]

# map (fun x -> x ^ "fez") ["cs"; "3110"; "zardoz"];;
- : string list = ["csfez"; "3110fez"; "zardozfez"]
```

(b) [code] Implement the following function:

```
let composition (f: 'a -> 'b ) (g: 'b -> 'c) (l: 'a list) : 'c list = ...
```

`composition` takes a function `f` that maps an element of type a to an element of type b a function `g` that maps an element of type b to an element of type c, and a list `l` of values of type a. The `composition` function should apply both functions to the list and return the output list of type 'c. For example:

```
# composition (fun x -> x + 1) (fun x -> string_of_int x) [1; 2; 3; 4; 5]
- : string list = ["2"; "3"; "4"; "5"; "6"]
```

(c) [code] Implement the following function:

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

`powerset` takes a set S represented as a list `l` and returns the powerset of S. Recall that sets have no duplicates and are unordered, and you may assume that input lists are valid representations of a set. Also recall that the **powerset** of a set S is the set of all subsets of S. For example,

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

# powerset [];;
- : 'a list list = [[]]
```

Your function must output a valid representation of a set.

You may use sort here (exercise 8 onward, not for any prior exercises). You may NOT use any other List module functions (except for cons (::) and concat (@))

Exercise 8.

(a) [code] Implement the `list_union` function:

```
let rec list_union (l1 : 'a list) (l2: 'a list) : ('a list) = ...
```

`union` takes the union of two lists. Note that lists can have duplicate elements. The number of copies of an element in our output list is to be equal to the maximum number of times that element occurs in an input list. Output order does not matter.

```
# list_union [4; 1; 1; 2] [1; 2; 5];;
- : int list = [1; 1; 2; 4; 5]
```

(b) [code] Implement the `list_intersection` function:

```
let rec list_intersection (l1 : 'a list) (l2: 'a list) : ('a list) = ...
```

`list_intersection` takes the intersection of two lists. Note that lists can have duplicate elements. The number of copies of an element in our output list to be equal to the minimum number of times that element occurs in an input list. Output order does not matter.

```
# list_intersection [4; 1; 1; 1; 2] [1; 1; 2; 5];;
- : int list = [1; 1; 2]
```

> **Note:** Consider looking up the relationship between the lowest common multiple, greatest common denominator, and prime factors. We will not test numbers <= 1. Recursion is not allowed here.

(c) [code] Implement the `lowest_common_multiple` function:

```
let lowest_common_multiple (n1 : int) (n2 : int) : (int) = ...
```

`lowest_common_multiple` outputs the lowest common multiple of n1 and n2 (defined here as the smallest positive number that both n1 and n2 divide). Recursion is not allowed.

```
# lowest_common_multiple 30 50;;
- : int = 150
```

(d) [code] Implement the `greatest_common_factor` function:

```
let greatest_common_factor (n1 : int) (n2 : int) : (int) = ...
```

`greatest_common_factor` outputs the greatest common factor of the n1 and n2 (defined here as the largest number that divides both n1 and n2). Recursion is not allowed.

```
# greatest_common_factor 20 30;;
- : int = 10
```

> **Note:** The next few problems are karma problems (8e, 8f, and 9). These problems have no bearing on your grade and are entirely optional.

For the following karma problems, you may also used the fold functions and the mem function

(e) [**karma**] **Note that karma is completely optional and will not affect your grade in any way.**

[code] Implement the `list_lcm` function:

```
let list_lcm (l1 : int list) : (int) = ...
```

`list_lcm` outputs the lowest common multiple of the list (defined here as the smallest positive number that all items in the list divide). The list will not be empty, and will contain only numbers >= 2. Recursion is not allowed here. Output order does not matter.

```
# list_lcm [20; 30; 50];;
- : int = 300
```

(f) [**karma**] **Note that karma is completely optional and will not affect your grade in any way.**

[code] Implement the `list_gcf` function:

```
let list_gcf (l1 : int list) : (int) = ...
```

`list_gcf` outputs the greatest common denominator of the list (defined here as the the largest number that divide all items in the list). The list will not be empty, and will contain only numbers >= 2. Recursion is not allowed here. Output order does not matter.

```
# list_gcf [20; 30; 40];;
- : int = 10
```

Exercise 9.

[karma]

Note that karma is completely optional and will not affect your grade in any way.

[code] Jek and Net are fencing in a direct single elimination tournament together. In a direct elimination tournament with n people, seed 1 fences seed n, seed 2 fences seed $(n-1)$, etc. The loser of each match is eliminated each round and the process is repeated in a new round with $n/2$ people. Jek is going to win the tournament, but Jek received a random seed. Net knows he will beat everyone he fences except for Jek, and Net knows he will lose to Jek. Net knows Jek's seed. Net wants to find the best seed he can take (lowest number) to receive second place overall (this means fencing Jek in the final round).

[code] Implement the following function:

```
let rec find_seed (jek_seed : int) (number_of_fencers : int) : int = ...
```

For example:

```
# find_seed 1 128;;
- : int = 2

# find_seed 3 8;;
- : int = 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

The accompanying release file `ps1.zip` contains the following files:

- `writeup.pdf` is this file.
- `release/ps1.ml` and `written.txt` are templates for you to fill in and submit.
- `ps1.mli` contains the interface and documentation for the functions that you will implement in `ps1.ml`