CS 3110 Spring 2015
Problem Set 1
Version 2 (last modified February 5, 2016)

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.z
- 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.

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) `3 + 5`

(b) `40 +. 2`

(c) `("zardoz", 3+5, true)`

(d) `["zardoz"; 3+5; true]`

(e) `fun x y -> if x then y else x`

(f) `fun a (b, c) -> a c (b c)`

(g) `(+) :: []`

(h) `(+) :: ((^) :: [])`

(i) `if 1 then true else false`

(j) `fun x -> fun y -> x ^ y`

Exercise 2.

[written] Give expressions having the following types.

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

(b) `(int -> int) -> int`

(c) `int -> (int -> int)`

(d) `'a -> 'a`

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

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

Code Style

Exercise 3.

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

```
let amm rjr shh =
let mcl pdz =
let rec rrs skc snb =
match skc with
[] -> snb
| tcw::tiw -> rrs tiw (tcw::snb) in
rrs pdz [] in
let hck mhk =
let rec aab acc akd =
match acc with [] -> akd
| asg::asw -> aab asw (akd @ asg) in
aab mhk [] in
let atn awl blc =
let rec cy gjm hmd hme =
match hmd with
[] -> hme
| irk::jag -> cy gjm jag ((gjm irk) :: hme) in
mcl (cy awl blc []) in
let jdh je jl = hck (atn je jl) in
jdh rjr shh
```

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 4.

(a) [code] Complete the implementation of the `fib` function, where `fib n` returns the n^{th} Fibonacci number. Indexing for the numbers begin at 0, and we use the seed values 0 and 1. Thus the 0^{th} through the 4^{th} Fibonacci numbers are $0, 1, 1, 2, 3, ...$

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

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

(b) [code] A naive implementation of the Fibonacci function above runs in time $O(2^n)$, which is less than ideal. Now let's write an improved Fibonacci function `acc_fib` that runs in $O(n)$:

```
let improved_fib (n: int) : int =
  let rec helper (n': int) (a: int) (b: int) =
    (* complete this function *)
  in
  helper n 0 1
```

Here `a` and `b` are both accumulators where `a` represents the $n - 2^{th}$ Fibonacci number and `b` represents the $n - 1^{th}$ Fibonaaci number for $n > 1$.

Exercise 5.

Recall the *rational numbers*, which intuitively are fractions with integer components. We can represent rationals as ordered pairs of type `int * int`, where the first component is the numerator and the second component is the denominator. We consider any ordered pair of type `int * int` as a valid rational except for those pairs with a 0 in the second component, as we cannot have 0 in the denominator.

(a) [code] Implement the `gcd` function, which computes the greatest common denominator of two integers.

```
    let rec gcd (a: int) (b: int) : int = ...
```

(b) [code] Implement the `equals` function, which takes two rationals `x` and `y` and returns `true` if and only if they represent the same rational number. You may assume inputs are valid rationals.

```
  let equals (x: int * int) (y: int * int) : bool = ...
```

(c) [code] Implement the `lowest_terms` function, which takes a rational `x` and returns the rational `y` such that `equals x y = true` and `y` is in lowest terms. You may assume inputs are valid rationals.

```
let lowest_terms (x: int * int) : int * int = ...
```

(d) [code] Implement the `plus` and `mult` functions, which denote the addition and multiplication operations over rationals, respectively. The outputs of these functions must be rationals in lowest terms. You may assume inputs are valid rationals.

```
let plus (x: int * int) (y: int * int) : int * int = ...

let mult (x: int * int) (y: int * int) : int * int = ...
```

Exercise 6.

An *unlabeled binary tree* is a binary tree with no values associated with any of its nodes. Thus, we do not differentiate between internal nodes with values and leaves with no values. For this problem, we are only concerned with the structure of binary trees.

Implement the `count_bin_trees` function, which returns the number of structurally unique unlabelled binary trees with n nodes. We define `count_bin_trees 0 = 1`. It is also clear that `count_bin_trees 1 = 1`. See figures (1) and (2) for the structurally unique unlabelled binary trees for 2 and 3 nodes. The behavior of the function is undefined for negative inputs.

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

Hint: Any binary tree necessarily has a root. For the case of $n > 1$, fix the root of the tree, and first consider the possible sizes of the left subtree (or symmetrically the right subtree).

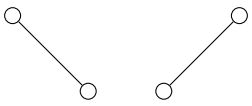

Figure 1: The possible unlabeled binary trees with $n = 2$ nodes.

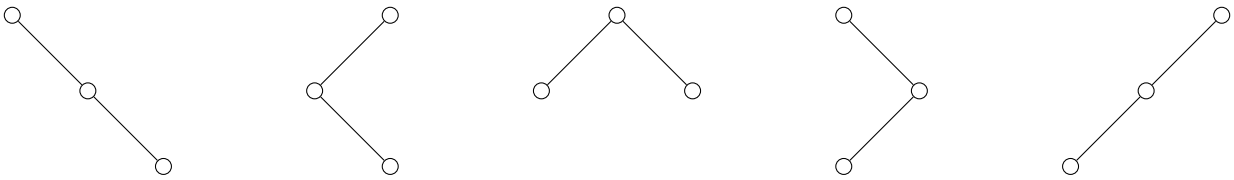

Figure 2: The possible unlabeled binary trees with $n = 3$ nodes.

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 filter (predicate: 'a -> bool) (l: 'a list) : 'a list = ...
```

`filter` takes two inputs: a function `predicate` that takes an element of the list and returns a boolean, and a list `l` of values. The `filter` function should return a list of elements such that satisfies `predicate`. For example:

```
# filter (fun x -> x mod 3 = 2) [1; 2; 3; 4; 5];;
- : int list = [2; 4]

# filter (fun x -> String.length x <= 4) ["cs"; "3110"; "zardoz"];;
- : string list = ["cs"; "3110"]
```

The order of the output list does not matter.

(b) [code] Implement the following function:

```
let rec pair_filter (compare: 'a -> 'a -> 'a) (l: 'a list) : 'a list =
```

`pair_filter` takes a function `compare` that returns one of two values, and a list `l` of values. The `pair_filter` function should break `l` into adjacent pairs, and apply `compare` to each pair. It should return a list containing the resulting values. For example:

```
# pair_filter max [0; 5; 7; 2; 3; -10]
  ↓ [max 0 5; max 7 2; max 3 -10]
  ↓ [5; 7; 3]
```

If the input list has odd length, then the last element should be returned as well. For example,

```
# pair_filter min [1; 2; 3; 4; 5]
  ↓ [min 1 2; min 3 4; 5]
  ↓ [1; 3; 5]
```

(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.

Higher Order Functions

Exercise 8.

(a) [code] Implement the following function:

```
let compose_2 (f: 'a -> 'a) (g: 'a -> a') : ('a -> 'a) = ...
```

`compose_2 f g` is equivalent to the composition $f \circ g$. Recall that $(f \circ g)(x) = f(g(x))$.

(b) [code] Implement the following function:

```
let rec compose_n (l: ('a -> 'a) list) : ('a -> 'a) = ...
```

`compose_n` takes in a list `l` of functions $[f_1; f_2; ...; f_n]$ and returns $f_1 \circ f_2 \circ ... \circ f_n$. If the list `l` is empty, `compose_n` returns the identity function. For example:

```
# let succ n = n + 1 in
  let plus_3 = compose_n [succ; succ; succ] in
  plus_3 0
- : int = 3

# let succ n = n + 1 in
  let mult_two n = n * 2 in
  let two_n_plus_one = compose_n [succ; mult_two] in
  two_n_plus_one 4
- : int = 9

# let identity = compose_n [] in
  identity "zardoz"
- : string = "zardoz"
```

(c) **[karma]**

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

[code] Implement the following function:

```
let rec list_to_func (l: 'a list) : (int -> 'a option) = ...
```

`list_to_func` takes in a list `l` of values `[x_1; x_2; ...; x_n]` and returns a function `f` where `f i` returns `Some x_i`. If `i < 0` or `n < i`, `f i` returns `None`. For example:

```
# let get_n = list_to_func [1; 3; 5; 7] in
  get_n 2
- : int option = Some 5

# let get_n = list_to_func ["cs"; "3110"; "zardoz"] in
  get_n 5
- : string option = None
```

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`