Data Structures and Functional Programming	**Problem Set 1**
CS 3110, Fall 2013	**Due at 11:59 PM, Thursday, September 5**
Version: 1	**Last Modified: Wednesday, August 28**

This assignment has four parts. You should write your solution to each part in a separate file: `part1.txt`, `part2.ml`, `part3.ml` and `part4.ml`. To help get you started, we have provided a stub file for each part on CMS. You should download and edit these files. Once you have finished, submit your solution using CMS, linked from the course website. The solutions were compiled with OCaml version 4.00.1.

Instructions

Compile Errors

All code you submit must compile. **Programs that do not compile will be heavily penalized.** If your submission does not compile, we will notify you immediately. You will have 24 hours after the submission date to supply us with a patch. If you do not submit a patch, or if your patched code does not compile, you will receive an automatic zero.

Naming

We will be using an automatic grading script, so it is **crucial** that you name your functions and order their arguments according to the problem set instructions, and that you place the functions in the correct files. Incorrectly named functions are **treated as compile errors** and you will have to submit a patch.

Code Style

Finally, please pay attention to style. Refer to the CS 3110 style guide and lecture notes. Ugly code that is functionally correct may still lose points. Take the extra time to think out the problems and find the most elegant solutions before coding them up. Even though only the second part of the assignment explicitly addresses the style issue, good programming style is also required for the other parts of this assignment, and for all the subsequent assignments in the rest of the semester.

Late Assignments

Please carefully review the course website's policy on late assignments, as **all** assignments handed in after the deadline will be considered late. Verify on CMS that you have submitted the correct version, **before** the deadline. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

You MAY NOT use side-effects in this assignment.

Part One: Expressions & Types

Exercise 1

Give the types of each of the following OCaml expressions. For example, the type of `fun x -> 42` is `'a -> int`.

(a) `[(true,[]); (false,[["yolo"]])]`

(b) `[Some ((fun x -> 3110) 2110)]`

(c) `if "zardoz"="zardoz" then false else 1 <> 1`

(d) `fun x (y,z) -> (x (Some (y, None))) (z ((())))`

(e) `fun x y -> (x y) (y (fun x -> x))`

Exercise 2

Give expressions that have the following OCaml types. You may **not** use explicit type annotations in the expression, as in `let f (x : int) : string =` $\cdots$. Your expressions should not cause the OCaml compiler to emit a warning.

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

(b) `bool * (int -> int) * float`

(c) `unit option`

(d) `'a -> 'b -> 'a list list`

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

Exercise 3

In each of the following, replace the `???` with an expression that

1. Makes the code type check correctly.
2. Does not cause the code to raise an exception.
3. Does not raise any compiler warnings.

Give the type of the expression you chose.

(a)

```
let zardoz (zar,doz) =
  match zar doz with
  | (zardoz, Some(zar))::doz -> zardoz "cs3110"
  | ???                      -> 3110
  | []                       -> failwith "unclear"
```

(b)

```
let cs3110 = ??? in
  let hope zar doz yolo =
    match zar yolo with
    | [[wepa]] -> fun x -> doz (wepa x)
    | _        -> fun y -> doz y in
  let you = function
    | Some(Some([[zardoz]])) -> [[fun x -> x]]
    | Some(None)             -> [[(+) 3110]]
    | None                   -> [[(-) 3110]]
    | _                      -> failwith "I'm a camel" in
  let enjoy swerve =
    let chirp = you (Some(Some([[3110]]))) in
    if (List.hd (List.hd chirp)) swerve > 3110 then begin
      match chirp with
      | we::pa -> "zardoz"
      | _      -> "zodraz"
    end
    else failwith "too easy" in
  hope you enjoy cs3110
```

Part Two: Code Style

Exercise 4

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 bkc blg =
(*rc and mg recommend starting early*)
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
  (*bz and wab are pro*)
  jdh bkc blg
```

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"
```

Part 3: Functional Programming

Exercise 5

Write a recursive function

```
  apply_n_times : int -> ('a -> 'a) -> 'a -> 'a
```

Which, given a non-negative integer n and a function f outputs the n-fold composition of f with itself. Namely,

$$\text{apply\_n\_times } f\ n\ x = f(f(f\cdots(f\ x)\cdots)))$$

Observe that `apply_n_times 0 f` is equivalent to the identity function and that `apply_n_times 1 f` is equivalent to `f` itself.

Examples

```
# let pred = fun x -> x-1;;
val pred : int -> int

# apply_n_times 3000 pred 6110;;
- : int = 3110

# apply_n_times 3 (( * ) 2) 1;;
- : int = 8

# apply_n_times 4 (( * ) 2) 5;;
- : int = 80
```

Exercise 6

Write a function

```
succ : (('a -> 'b) -> 'c -> 'a) -> ('a ->'b) -> 'c -> 'b)
```

that, when given a function equivalent to `apply_n_times n` as in the previous exercise, will produce a function equivalent to `apply_n_times (n+1)`. The functionality of `succ` is demonstrated in the following:

Examples

```
# let square = fun x -> x*x;;
val square : int -> int

# apply_n_times 0 square 2;;
- : int = 2

# succ (apply_n_times 0) square 2;;
- : int = 4

# succ (succ (apply_n_times 0)) square 2;;
- : int = 16

# succ (succ (succ (apply_n_times 0))) square 2;;
- : int = 256
```

Exercise 7

Consider a polynomial $p(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_{n-1} x^{n-1} + \alpha_n x^n$. We can encapsulate the relevant data in $p(x)$ by representing it as a list of its coefficients $[\alpha_0; \alpha_1; \cdots; \alpha_{n-1}; \alpha_n]$. This suggests the following type definition:

```
type polynomial = float list
```

Write a function

```
d : polynomial -> polynomial
```

which, given a polynomial $p(x)$ returns its derivative. You may assume that the input to `d` is that a valid polynomial (i.e. a non-empty list). You may raise an exception if the input to `d` is invalid.

One of the deficiencies of our OCaml abstraction of polynomials is that the representation of a given polynomial is not unique. For example the polynomial $p(x) = 1 + x$ could be represented as the OCaml values `[1.;1.]` as well as `[1.;1.;0.]`, `[1.;1.;0.;0.]` and so on. In order to overcome this we have provided you with a function

```
drop_trailing_zeroes : polynomial -> polynomial
```

which removes the trailing zeroes of a given polynomial.

Examples

```
# d [3110.];;
- : polynomial = [0.]

# d [3.;1.;1.;0.];;
- : polynomial = [1.;2.]

# d [0.;1.;2.;3.;4.;5.];;
- : polynomial = [1.;4.;9.;16.;25.]
```

Part 4: Relating Induction and Recursion

As you recall, mathematical induction is a method for proving statements about numbers by reducing them to statements about smaller numbers. There are two steps to prove a that a property P holds for every number $n \geq 1$. First, one proves the "base case" (namely $P(1)$) holds. Second, one must provide an "inductive step" that shows $P(n)$ holds whenever $P(n-1)$ holds.

For example, it is well known that for all n,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

To prove this by induction, we first prove the base case: that $1 + \cdots + 1 = 1 \cdot 2/2$. This follows immediately from the definition.

Next, we must prove the inductive step. Assuming that $P(n-1)$ holds, we must show that $P(n)$ holds. In this case, we assume that

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \qquad \text{(inductive hypothesis)}$$

and we must prove that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

This is a simple matter of algebraic manipulation:

$$\begin{aligned}
\sum_{i=1}^{n} i &= n + \sum_{i=1}^{n-1} i \\
&= n + \left(\frac{(n-1)n}{2}\right) \qquad \text{(by the inductive hypothesis)} \\
&= \frac{2n + (n-1)n}{2} \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

Because we have shown that the claim holds for 1, and that whenever it holds for $n-1$ it also holds for n, we can build up an argument that it holds for an arbitrary number by repeatedly applying the inductive step.

Exercise 8

In this exercise, you will replace induction with recursion. Instead of producing a single inductive proof that works for any n, you will write a recursive function that outputs a non-inductive proof for any given n.

Write a function

```
prove : int -> string
```

which, given an integer $n \geq 1$ will produce a string containing the argument given above. Your function should display an error message if the input is less than 1.

We have provided you with a helper function

```
inductive_step : int -> string
```

That given an integer n will produce a portion of the argument. Use this function in your implementation of `prove`. In order to view the output we have provided you with a wrapper function

```
write_proof : int -> unit
```

which prints out the result of your `prove` implementation.

Examples

```
# write_proof 3;;
"how do we know that 1 + ... + 3 = 4 * 3 / 2?
  Well, another way to write '1 + ... + 3' is '(1 + ... + 2) + 3'.
  And we know that 1 + ... + 2 = 2 * 3 / 2.
  So 1 + ... + 3 = 2 * 3 / 2 + 2 * 3/2
     = (2 * 3 + 2 * 3) / 2
     = 4 * 3 / 2.
and how do we know that 1 + ... + 2 = 3 * 2 / 2?
  Well, another way to write '1 + ... + 2' is '(1 + ... + 1) + 2'.
  And we know that 1 + ... + 1 = 1 * 2 / 2.
  So 1 + ... + 2 = 1 * 2 / 2 + 2 * 2/2
     = (1 * 2 + 2 * 2) / 2
     = 3 * 2 / 2.
and finally, 1 + ... + 1 is 1 by definition."
- : unit = ()

# write_proof 4;;
"how do we know that 1 + ... + 4 = 5 * 4 / 2?
  Well, another way to write '1 + ... + 4' is '(1 + ... + 3) + 4'.
  And we know that 1 + ... + 3 = 3 * 4 / 2.
  So 1 + ... + 4 = 3 * 4 / 2 + 2 * 4/2
                 = (3 * 4 + 2 * 4) / 2
                 = 5 * 4 / 2.
and how do we know that 1 + ... + 3 = 4 * 3 / 2?
  Well, another way to write '1 + ... + 3' is '(1 + ... + 2) + 3'.
  And we know that 1 + ... + 2 = 2 * 3 / 2.
  So 1 + ... + 3 = 2 * 3 / 2 + 2 * 3/2
                 = (2 * 3 + 2 * 3) / 2
                 = 4 * 3 / 2.
and how do we know that 1 + ... + 2 = 3 * 2 / 2?
  Well, another way to write '1 + ... + 2' is '(1 + ... + 1) + 2'.
  And we know that 1 + ... + 1 = 1 * 2 / 2.
  So 1 + ... + 2 = 1 * 2 / 2 + 2 * 2/2
                 = (1 * 2 + 2 * 2) / 2
                 = 3 * 2 / 2.
and finally, 1 + ... + 1 is 1 by definition."
- : unit = ()

# write_proof (-3110);;
Exception: Failure "We only prove this for n > 0.".
```