CSE-321 Assignment 1 - *Fun with Objective CAML* (100 points)

gla@postech

Due at 11:59pm, Feb 27

Welcome to CSE-321 Programming Languages! In this assignment, you will familiarize yourself with functional programming in Objective CAML (OCAML) by implementing functions of various types. Each function can be implemented with no more than a few lines of code, but requires a bit of thinking. So this assignment should not be painstaking programming; rather it will be great fun!

In order to assist the teaching staff in grading your assignment, you should *strictly* follow the submission instruction.

1 Submission instruction

Download the zip file `hw1.zip` from the course webpage or from `/home/class/cs321/` on `programming2.postech.ac.kr`, and unzip it:

```
lngsg@programming2:~ $ unzip hw1.zip
Archive:  hw1.zip
   creating: hw1/
  inflating: hw1/hw1.mli
  inflating: hw1/hw1.ml
  inflating: hw1/.depend
  inflating: hw1/Makefile
```

You will write code in `hw1.ml` and never touch other files. The stub file `hw1.ml` looks like:

```
exception Not_implemented

type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree

let rec sum _ = raise Not_implemented
let rec power _ _ = raise Not_implemented
let rec gcd _ _ = raise Not_implemented
let rec combi _ _ = raise Not_implemented

...
```

1. Fill the function body with your own code *only if you have a correct implementation of the function.* This is absolutely crucial; if you leave code that does not compile, you will receive no credit. If you cannot implement a function, just leave it intact! Make sure that your program compiles by running `make`:

```
lngsg@programming2:~/hw1 $ ls
hw1.ml  hw1.mli  Makefile
lngsg@programming2:~/hw1 $ make
ocamlc  -c hw1.mli -o hw1.cmi
ocamlc  -c hw1.ml -o hw1.cmo
ocamlc  -o hw1 hw1.cmo
```

2. To run your program on the OCAML interpreter, use the command `#use`. Here is a sample session:

```
lngsg@programming2:~/hw1 $ ocaml
        OCaml version 4.05.0

# #use "hw1.ml";;
exception Not_implemented
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
val sum : 'a -> 'b = <fun>
val power : 'a -> 'b -> 'c = <fun>
val gcd : 'a -> 'b -> 'c = <fun>
val combi : 'a -> 'b -> 'c = <fun>
...
# sum 10;;
Exception: Not_implemented.
#
```

3. When you have the file `hw1.ml` ready for submission, copy it to your hand-in directory on `programming2.postech.ac.kr`. For example, if your Hemos ID is `foo`, copy it to:

    `/home/class/cs321/handin/foo/`

2 Functions to be implemented

For this assignment, do not use any library functions provided by OCAML.

2.1 Functions on integers

2.1.1 `sum` for adding integers 1 to n (inclusive) [4 points]

(Type) `sum : int -> int`

(Description) `sum` n returns $\sum_{i=1}^{n} i$.

(Invariant) $n > 0$.

(Example)

```
# sum 10 ;;
- : int = 55
```

2.1.2 `power` for powers [4 points]

(Type) `power: int -> int -> int`

(Description) `power` x n returns x^n. x^0 is computed as 1.

(Invariant) $n \geq 0$.

2.1.3 `gcd` for finding the greatest common divisor [4 points]

(Type) `gcd: int -> int -> int`

(Description) `gcd` m n returns the greatest common divisor of m and n, using Euclid's algorithm.

(Invariant) $m \geq 0$, $n \geq 0$, $m + n > 0$.

(Example) The result below is specific to the sample solution. Your code may be differ in behavior.

```
    gcd 15 20
↦   gcd 5 15
↦   gcd 0 5
↦   5
```

2.1.4 `combi` for calculating a combination [4 points]

(Type) `combi: int -> int -> int`

(Description) `combi` n k returns a combination of n things taken k at a time without repetition.

(Invariant) $n > 0$, $k \geq 0$, $k \leq n$

(Example) `combi 5 2` returns `10`.

2.2 Functions on binary trees

2.2.1 `sum_tree` for computing the sum of integers stored in a binary tree [4 points]

(Type) `sum_tree : int tree -> int`

(Description) `sum_tree` t returns the sum of integers stored in the tree t.

(Example) `sum_tree (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `17`.

2.2.2 `depth` for computing the depth of tree [4 points]

(Type) `depth : 'a tree -> int`

(Description) `depth` t returns the length of the longest path from the root to leaf.

(Example) `depth (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `2`.

2.2.3 `bin_search` for searching an element in a binary search tree [4 points]

(Type) `bin_search : int tree -> int -> bool`

(Description) `bin_search` t x returns `true` if the number x is found in the binary search tree t; otherwise it returns `false`.

(Invariant) t is a binary search tree: all numbers in a left subtree are smaller than the number of the root, and all numbers in a right subtree are greater than the number of the root. We further assume that all numbers are distinct.

(Example) `bin_search (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 2` returns `true`. `bin_search (Node (Node (Leaf 1, 2, Leaf 3), 4, Leaf 7)) 5` returns `false`.

2.2.4 `postorder` for a postorder traversal of binary trees [4 points]

(Type) `postorder: 'a tree -> 'a list`

(Description) `postorder` t returns a list of elements produced by a postorder traversal of the tree t.

(Example) `postorder (Node (Node (Leaf 1, 3, Leaf 2), 7, Leaf 4))` returns `[1, 2, 3, 4, 7]`.

2.3 Functions on lists of integers

2.3.1 `max` for finding the largest integer in a list of integers [4 points]

(Type) `max: int list -> int`

(Description) `max` l returns the largest integer in the list l. If an empty list is given, return 0.

(Example) `max [5; 3; 6; 7; 4]` returns `7`.

2.3.2 `list_add` for adding each pair of integers from two lists [4 points]

(Type) `list_add: int list -> int list -> int list`

(Description) `list_add` $[a;\ b;\ c;\ \ldots]$ $[x;\ y;\ z;\ \ldots]$ returns $[a{+}x;\ b{+}y;\ c{+}z;\ \ldots]$. If one list is longer than the other, the remaining list of elements is appended to the result.

(Example) `list_add [1; 2] [3; 4; 5]` returns `[4; 6; 5]`.

2.3.3 `insert` for inserting an element into a sorted list [4 points]

(Type) `insert: int -> int list -> int list`

(Description) `insert` m l inserts m into a sorted list l. The resultant list is also sorted.

(Invariant) The list l is sorted in ascending order.

(Example) `insert 3 [1; 2; 4; 5]` returns `[1; 2; 3; 4; 5]`.

2.3.4 `insort` for insertion sort [4 points]

(Type) `insort: int list -> int list`

(Description) `insort` l returns a sorted list of elements in l.

(Example) `insort [3; 7; 5; 1; 2]` returns `[1; 2; 3; 5; 7]`.

(Hint) Use `insert` above.

2.4 Higher-order functions

2.4.1 `compose` for functional composition [4 points]

(Type) `compose: ('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`

(Description) `compose` f g returns $g \circ f$. That is, `(compose` f g`)` x is equal to g $(f$ $x)$.

2.4.2 `curry` for currying [4 points]

(Type) `curry: ('a * 'b -> 'c) -> ('a -> 'b -> 'c)`

(Description) We have a choice of how to write functions of two or more arguments. Functions are in *curried form* if they take arguments one at a time. *Uncurried* functions take arguments as a pair. `curry` f transforms an uncurried function f into a curried version.

(Example)

```
let multiply x y = x * y        (* curried function *)
let multiplyUC (x, y) = x * y   (* uncurried function *)
```

If we apply `curry` to `multiplyUC`, we get `multiply`.

2.4.3 `uncurry` for uncurrying [4 points]

(Type) `uncurry: ('a -> 'b -> 'c) -> ('a * 'b -> 'c)`

(Description) See the above exercise. `uncurry` f transforms an curried function f into a uncurried version.

(Example) If we apply `uncurry` to `multiply`, we get `multiplyUC`.

2.4.4 `multifun` for applying a function n-times [4 points]

(Type) `multifun : ('a -> 'a) -> int -> ('a -> 'a)`

(Description) `(multifun` f n`)` x returns the result of $\underbrace{f(f(\cdots f}_{n-times}(x)\cdots))$

(Example) `(multifun (fn x => x + 1) 3) 1` returns `4`.
`(multifun (fn x => x * x) 3) 2` returns `256`.

(Invariant) `n` $\geq$ `1`.

2.5 Functions on `'a list`

2.5.1 `ltake` for taking the list of the first i element of l [4 points]

(Type) `ltake: 'a list -> int -> 'a list`

(Description) `ltake` l n returns the list of the first n elements of l.
If `n` is larger than the length of `l`, then return `l`.

(Example) `ltake [3; 7; 5; 1; 2] 3` returns `[3; 7; 5]`.
`ltake [3; 7; 5; 1; 2] 7` returns `[3; 7; 5; 1; 2]`.
`ltake ["s"; "t"; "r"; "i"; "k"; "e"; "r"; "z" ] 5` returns
`["s"; "t"; "r"; "i"; "k"]`.

2.5.2 `lall` for examining a list [4 points]

(Type) `lall : ('a -> bool) -> 'a list -> bool`

(Description) `lall` f l returns `true` if for every element x of l, f x evaluates to `true`; otherwise it returns `false`. In other words, `lall` f l tests if all elements in l satisfy the predicate f. If l is empty, `lall` f l returns true regardless of f.

(Example) `lall (fun x => x > 0) [1; 2; 3]` evaluates to `true`.
`lall (fun x => x > 0) [-1; -2; 3]` evaluates to `false`.

2.5.3 `lmap` for converting a list into another list [4 points]

(Type) `lmap : ('a -> 'b) -> 'a list -> 'b list`

(Description) `lmap` f l applies f to each element of l from left to right, returning the list of results.

(Example) `lmap (fun x => x + 1) [1; 2; 3]` returns `[2; 3; 4]`.

2.5.4 `lrev` for reversing a list [4 points]

(Type) `lrev: 'a list -> 'a list`

(Description) `lrev` l reverses l.

(Example) `lrev [1; 2; 3; 4]` returns `[4; 3; 2; 1]`.

2.5.5 `lflat` for flattening a list [4 points]

(Type) `lflat : 'a list list -> 'a list`

(Description) `lflat` l flattens l.

(Example) `lrev [[1; 2]; [3; 4; 5]; [6]]` returns `[1; 2; 3; 4; 5; 6]`.

2.5.6 `lzip` for pairing corresponding members of two lists [4 points]

(Type) `lzip: 'a list -> b' list -> ('a * 'b) list`

(Description) `lzip` $[x_1; \cdots; x_n]$ $[y_1; \cdots; y_n] \Rightarrow$ `[(`x_1`, `y_1`); ···; (`x_n`, `y_n`)]` .
If two lists differ in length, ignore surplus elements.

(Example) `lzip ["Rooney"; "Park"; "Scholes"; "C.Ronaldo"] [8; 13; 18; 7; 10; 12]` returns `[("Rooney", 8); ("Park", 13); ("Scholes", 18); ("C.Ronaldo", 7)]`.

2.5.7 `split` for splitting a list into two lists [4 points]

(Type) `split: 'a list -> 'a list * 'a list`

(Description) `split` l returns a pair of two lists. The first list consists of elements in odd positions and the second consists of elements in even posistions in a given list respectively. For an empty list, `split` returns $([\,], [\,])$. For a singleton list $[x]$, `split` returns $([x], [\,])$.

(Example) `split [1; 3; 5; 7; 9; 11]` returns `([1; 5; 9], [3; 7; 11])`.

2.5.8 `cartprod` for the Cartesian product of two sets [4 points]

(Type) `cartprod: 'a list -> 'b list -> ('a * 'b) list`

(Description) `cartprod S T` returns the set of all pairs (x, y) with $x \in S$ and $y \in T$.
The order of elements is important:
`cartprod` $[x_1; \cdots; x_n]$ $[y_1; \cdots; y_n] \Rightarrow [(x_1, y_1); \cdots; (x_1, y_n); (x_2, y_1); \cdots; (x_n, y_n)]$.

(Example) `cartprod [1; 2] [3; 4; 5]` $\Rightarrow$ `[(1, 3); (1, 4); (1, 5); (2, 3); (2, 4); (2, 5)]`.

2.5.9 `powerset` for the power set of a set [4 points]

(Type) `powerset: 'a list -> 'a list list`

(Description) `powerset S` returns the list of all subsets of S.

(Example) `powerset [1; 2]` returns `[[]; [1]; [2]; [1; 2]]`.
You don't have to consider the order of the result. For example, both `[[2; 1]; [1]; [2]; []]` and `[[]; [1]; [2]; [1; 2]]` are the correct answers for `powerset [1; 2]`.

(Invariant) All elements of S are unique.