CSE-321 Assignment 2 - *More Fun with Objective CAML* (100 points)

gla@postech

Welcome to the second assignment of CSE-321 Programming Languages! In this assignment, you will further familiarize yourself with functional programming in Objective CAML (OCAML) by implementing various tail-recursive functions, sorting algorithms, and structures.

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

1 Submission instruction

Download the zip file `hw2.zip` from the course webpage or from `/home/class/cs321/` on `programming2.postech.ac.kr`, and unzip it:

```
gla@ubuntu:~/temp$ unzip hw2.zip
Archive:  hw2.zip
   creating: hw2/
  inflating: hw2/hw2.mli
  inflating: hw2/.depend
  inflating: hw2/hw2.ml
  inflating: hw2/Makefile
```

You will write code in `hw2.ml` and never touch other files. The stub file `hw2.ml` looks like:

```
exception NotImplemented

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

(** Recursive letctions **)

let rec lconcat _ = raise NotImplemented

let rec lfoldl _ _ _ = raise NotImplemented

(** Tail recursive letctions  **)

let fact _ = raise NotImplemented

...
```

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`:

```
gla@ubuntu:~/temp/hw2$ ls
hw2.ml  hw2.mli  Makefile
gla@ubuntu:~/temp/hw2$ make
ocamlc  -c hw2.mli -o hw2.cmi
ocamlc  -c hw2.ml -o hw2.cmo
ocamlc  -o hw2 hw2.cmo
```

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

```
gla@ubuntu:~/temp/hw2$ ocaml
        OCaml version 4.01.0

# #use "hw2.ml";;
exception NotImplemented
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
val lconcat : 'a list list -> 'a list = <fun>
val lfoldl : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
val fact : int -> int = <fun>
val power : int -> int -> int = <fun>
val fib : int -> int = <fun>
val lfilter : ('a -> bool) -> 'a list -> 'a list = <fun>
val ltabulate : int -> (int -> 'a) -> 'a list = <fun>
val union : 'a list -> 'a list -> 'a list = <fun>
val inorder : 'a tree -> 'a list = <fun>
...
# lconcat [[1]; [2; 3]; [4; 5; 6]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

3. When you have the file `hw2.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 Recursive functions

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

2.1 `lconcat` for concatenating a list of lists [5 points]

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

(Description) `lconcat` l concatenates all elements of l.

(Example) `lconcat [[1; 2; 3]; [6; 5; 4]; [9]]` returns `[1; 2; 3; 6; 5; 4; 9]`.

2.2 `lfoldl` for left folding a list [5 points]

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

(Description) `lfoldl` f e l takes e and the first item of l and applies f to them, then feeds the function with this result and the second argument and so on.

> `lfoldl` f e $[x_1; x_2; ...; x_n]$ returns $f(x_n, ..., f(x_2, f(x_1, e))...)$ or e if the list is empty.

(Note) You may not use `List.fold_left`.

3 Tail recursive functions

For each description below, give a tail recursive implementation. Usually you want to introduce a tail recursive helper function; the main function is not recursive but just invokes the helper function with appropriate arguments. For example, a tail recursive implementation of `fact` may look like

```
let fact n =
  let rec fact_aux n acc =
    ...
  in fact_aux n 1
```

where `fact_aux` is tail-recursive.

You should fill in the code **only if it is a correct tail recursive implementation.** If you are unsure that it is a correct tail recursive implementation, do not fill in the code.

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

3.1 `fact` for factorials [3 points]

(Type) `fact: int -> int`

(Description) `fact` n returns $\prod_{i=1}^{n} i$.

(Invariant) $n \geq 0$.

3.2 `power` for powers [3 points]

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

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

(Invariant) $n \geq 0$.

3.3 `fib` for Fibonacci numbers [3 points]

(Type) `fib: int -> int`

(Description)
`fib` n returns `fib` $(n-1)$ + `fib` $(n-2)$ when $n \geq 2$.
`fib` n returns 1 if $n = 0$ or $n = 1$.

(Invariant) $n \geq 0$.

(Hint) Perhaps you want to use the idea of dynamic programming?

3.4 `lfilter` for filtering a list [3 points]

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

(Description) `lfilter` p l returns all elements of l that satisfies the predicate p.

(Example) `lfilter (fun x -> x > 2) [0; 1; 2; 3; 4; 5]` returns `[3; 4; 5]`.

3.5 `ltabulate` [3 points]

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

(Description) `ltabulate` n f applies f to each element of a list `[0; 1; ...; n-1]`.

(Example) `ltabulate 4 (fun x -> x * x)` returns `[0; 1; 4; 9]`.

(Invariant) $n \geq 0$

3.6 `union` for union of two sets [5 points]

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

(Description) `union` S T returns a set that includes all elements of S and T without duplication of any element. Note that all list elements have an equality type as indicated by equality type variable `'a`. The order of elements in the return value does not matter.

(Invariant) Each input set consists of distinct elements.

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

(Hint) You can implement `union` without introducing an auxiliary tail recursive function. That is, `union` itself can be implemented as a tail recursive function.

3.7 `inorder` for an inorder traversal of binary trees [8 points]

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

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

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

(Hint) `inorder` can be implemented as follows:

```
let inorder t =
  let rec inorder' (t' : 'a tree) (post : 'a list) : 'a list = ...
  in
    inorder' t [ ]
```

`post` will be a list of elements to be appended to the result of an inorder traversal of `t'`. For example, when `inorder'` visits the node marked `2` in the tree below, `post` will be bound to `[1; 6; 3; 7]`.

```
      1

   2     3

 4   5  6   7
```

3.8 `postorder` for a postorder traversal of binary trees [8 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]`.

3.9 `preorder` for a preorder traversal of binary trees [8 points]

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

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

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

4 Sorting in the ascending order

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

4.1 `quicksort` for quick sorting [8 points]

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

(Description) `quicksort` l implements quick sorting by selecting the first element of l as a pivot. To compare elements of type `'a`, use the primitive operators `<`, `>`, and `=`.

(Example) `quicksort [3; 7; 5; 1; 2]` selects 3 as a pivot to obtains two sublists `[1; 2]` and `[5; 7]` to be sorted independently.

4.2 `mergesort` for merge sorting [8 points]

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

(Description) `mergesort` l divides l into two sublists, sorts each sublist, and then merges the two sorted sublists. If the length of l is even, then the two sublists are of equal length. If not, one sublist has one more element than the other. To compare elements of type `'a`, use the primitive operators `<`, `>`, and `=`.

5 Structures

The goal of this part is to learn *modular programming* in OCAML — structures and signatures. We will first implement a structure for heaps. You should keep in mind that this data structure is not an ordinary heap data structure. You had better think of it as a mechanism for dynamic memory allocation. See the explanation below carefully.

Remark. In the signatures `HEAP` and `DICT`, `empty` is given types `unit -> 'a heap` and `unit -> 'a dict`, respectively. A better design would be to use types `'a heap` and `'a dict`, but in order to facilitate grading, we decided to change it into a function.

For this part, you may use the `List` library of OCAML.

5.1 `Heap` for heaps [10 points]

The structure `Heap` conforms to the signature `HEAP`. A heap is a mechanism for dynamic memory allocation.

```
module type HEAP =
sig
  exception InvalidLocation
  type loc
  type 'a heap
  val empty : unit -> 'a heap
  val allocate : 'a heap -> 'a -> 'a heap * loc
  val dereference : 'a heap -> loc -> 'a
  val update : 'a heap -> loc -> 'a -> 'a heap
end
```

- `loc` is the internal representation of location, which is similar to the *pointer* of C language. `type loc` is not visible to the outside of the structure.
- `'a heap` is a heap for the type `'a`.
- `empty ()` returns an empty heap.
- `allocate` h v allocates the given value v to a fresh heap cell and returns the pair (h', l) of the updated heap h' and the location l of this cell.
- `dereference` h l fetches the value v stored in the heap cell at location l. `InvalidLocation` is raised if the l is an invalid `loc`.
- `update` h l v updates the heap cell at location l with the given value v and returns the updated heap h'. `InvalidLocation` is raised if the l is an invalid `loc`.

5.2 Signature DICT

DICT is a signature for dictionaries.

```
module type DICT =
sig
  type key
  type 'a dict
  val empty : unit -> 'a dict
  val lookup : 'a dict -> key -> 'a option
  val delete : 'a dict -> key -> 'a dict
  val insert : 'a dict -> key * 'a -> 'a dict
end
```

- `empty ()` returns an empty dictionary.
- `lookup` d k searches the key k in the dictionary d. If the key is found, it returns the associated item. Otherwise, it returns `None`.
- `delete` d k deletes the key k and its associated item in the dictionary d and returns the resultant dictionary d'. If the key does not exist in the dictionary d, it returns the given dictionary d without any modification.
- `insert` d `(`k`,` v`)` inserts the new key k and its associated item v in the dictionary d. If the key k already exists in the dictionary d, it just updates its associated item with the given item v.

5.2.1 Structure DictList [10 points]

Implement the structure `DictList` of signature `DICT` with the definition `'a dict = (key * 'a) list`.

The structure `DictList` uses a list of pairs as the representation of a dictionary. The implementation should be straightforward because a list of pairs itself may be thought of as a dictionary.

5.2.2 Structure DictFun [10 points]

Implement the structure `DictFun` of signature `DICT` with the definition `'a dict = key -> 'a option`.

The structure `DictFun` uses a "functional representation" of dictionaries. The idea is that we represent a dictionary as a function that, given a key, returns an associated item. The implementation of `DictFun` may be either very difficult or just a piece of cake depending on how familiar you are with "functional thinking." Our advice is: forget about everything that you have learned so far about imperative programming; just "think functionally!" You will be amazed at the conciseness of your code once you figure it out.