exercise07-solution-p

December 1, 2018

0.0.1 Assignment 7.1 (P) The List Module (part 2)

In order to write short and easy to understand programs, it is crucial to use existing library functions for regularly appearing problem patterns. The most frequently used library functions are clearly those of the `List` Module and the following in particular. Check the documentation for their types and functionality:

- `List.map : ('a -> 'b) -> 'a list -> 'b list` Transforms all elements in the list with the given function.
- `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` Step by step combines all elements in the list from left to right with an initial value.
- `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` Step by step combines all elements in the list from right to left with an initial value.
- `List.find_opt : ('a -> bool) -> 'a list -> 'a option` Searches the list for an element for which the given function returns true. If such an element `x` is found, it is returned as `Some x`, otherwise `None` is returned.
- `List.filter : ('a -> bool) -> 'a list -> 'a list` Constructs a new list, containing only the elements for which the function returns true.

Implement the following functions without using any recursive functions:

1) `squaresum : int list -> int` computes $\sum_{i=1}^{n} x_i^2$ for a list $[x_1, \ldots, x_n]$.
2) `float_list : int list -> float list` converts all ints in the list to floats.
3) `to_string : int list -> string` builds a string representation of the given list.
4) `part_even : int list -> int list` partitions all even values to the front of the list.

```
In [ ]: let squaresum l = List.fold_left (fun a x -> a + x * x) 0 l

        let float_list l = List.map float_of_int l

        let to_string l =
            "[" ^ (List.fold_left (fun a x -> a ^ (string_of_int x) ^ ";") "" l) ^ "]"

        let part_even l =
            let even = List.filter (fun x -> x mod 2 = 0) l in
            let odd = List.filter (fun x -> x mod 2 <> 0) l in
```

```
    even @ odd
(* or better:
    List.partition (fun x -> x mod 2 = 0) l
*)
```

0.0.2 Assignment 7.2 (P) Mappings

There is a need to represent and modify a mapping from one set of values to another set of values in many applications. Remember how we stored the grades in the student record. Storing a list of pairs `'a * 'b` is a simple way to represent a mapping from type `'a` to type `'b`. This kind of list is typically referred to as an associative list:

1) Implement these functions to work with mappings based on associative lists:

- `is_empty : ('k * 'v) list -> bool`
- `get : 'k -> ('k * 'v) list -> 'v option`
- `put : 'k -> 'v -> ('k * 'v) list -> ('k * 'v) list`
- `contains_key : 'k -> ('k * 'v) list -> bool`
- `remove : 'k -> ('k * 'v) list -> ('k * 'v) list`
- `keys : ('k * 'v) list -> 'k list`
- `values : ('k * 'v) list -> 'v list`

In [ ]:
```
let is_empty m = m = empty

let rec get k = function [] -> None
    | (k',v')::ms -> if k' = k then Some v' else get k ms

let put k v m = (k,v)::m
(* or better:
let put k v m = (k,v)::remove k m
*)

let contains_key k m = (get k m) <> None

let rec remove k = function [] -> []
    | (k',v')::ms -> if k' = k then ms else (k',v')::remove k ms

let keys m = List.map fst m

let values m = List.map snd m
```

2) Check the `List` module for functions that already provide (some of) these functionalities.

- `assoc_opt` is `get`
- `mem_assoc` is `contains_key`
- `remove_assoc` is `remove`
- `split` followed by `fst` is `keys`
- `split` followed by `snd` is `values`

An alternative to associative lists is to use functions of type `'k -> 'v option` directly. So for example, the function `fun x -> x * x + 1` respresents a very efficient mapping from any number to the successor of its square.

3) Implement the above functions again for mappings based on functions. Some of these functions cannot be implemented, however:

```
In [ ]: let is_empty m = failwith "impossible"

        let get k m = m k

        let put k v m = fun x -> if x = k then Some v else m x

        let contains_key k m = (get k m) <> None

        let remove k m = fun x -> if x = k then None else m x

        let keys m = failwith "impossible"

        let values m = failwith "impossible"
```

4) Discuss: What are the advantages of either approach? When would you use which?

The function approach is very efficient when the mapping is changed rarely or not at all, because the chain of nested function calls grows with every modification. Querying values from the list is linear in the length of the list, however, if mappings change a lot, it is still much better than the other approach or if functionality like `keys` or `values` is required, which the function implementation could only do by iterating over all values in the domain type (e.g. from `min_int` to `max_int` for `'k = int`).

0.0.3 Assignment 7.3 (P) Operator Functions

In OCaml, infix notation of operators is just syntactic sugar for a call to the corresponding function. The binary addition + merely calls the function `(+) : int -> int -> int`.

1) Discuss why this is a very useful feature.

First, you can define your own operator functions and thus add additional infix operators or change the semantics of an existing one. See the `(=.)` operator defined in the homework tests for example. Second, operators can be used directy in contexts where a function is required. Instead of defining a new function `fun a b -> a + b` the operator `(+)` can be used when folding over a list: `fold_left (+) 0 l`.