Principles of Programming, Fall 2009
Practice 9
OCaml modules

Woosuk Lee, Suwon Jang, Sungkeun Cho
Programming Research Lab.@SNU

November 9, 2009

parameterized type & module We studied integer stack module in last class. Using polymorphic type we can easily make generalized stack.

```
module type INTSTACK =
sig
  type t
  exception Empty
  val empty : t
  val push : int -> t -> t
  val pop : t -> int * t
end

module type STACK =
sig
  type 'a t
  exception Empty
  val empty : 'a t
  val push : 'a -> 'a t -> 'a t
  val pop : 'a t -> 'a * 'a t
end

module IntStack : INTSTACK =
struct
  type t = int list
  exception Empty
  let empty = []
  let push x t = x :: t
  let pop t =
    match t with
      [] -> raise Empty
```

```
    | h::t -> (h,t)
end

module Stack : STACK =
struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let push x t = x :: t
  let pop t =
    match t with
      [] -> raise Empty
    | h::t -> (h,t)
end
```

exercise 1. Define module `BT` with following signature.

```
module type SIGBT =
sig
  type 'a t = LEAF | NODE of 'a t * 'a * 'a t
  val empty: 'a t
  val mem: 'a -> 'a t -> bool
  val insert: 'a -> 'a t -> 'a t
  val remove: 'a -> 'a t -> 'a t
  val inorder_visit: 'a t -> 'a list
end
```

e.g.

```
module BT : SIGBT =
struct
...
end

open BT

let intTree = insert 5 (insert 1 (insert 2 empty)) ;;
let charTree = insert 'c' (insert 'a' (insert 'b' empty)) ;;
mem 5 intTree ;;
remove 'a' charTree ;;
```

2. OCaml standard library provides module `Map` implemented with functor. But it is possible to make simpler version of module `Map`, because default comparing functions such like `compare`, `(<)`, `(>)`, ... can be applied any value. Define module `MyPMap` with following signature.

```
module type MYPMAP =
```

```
sig
  type ('a, 'b) t
  val empty : ('a, 'b) t
  val is_empty : ('a, 'b) t -> bool
  val add : 'a -> 'b -> ('a, 'b) t -> ('a,'b) t
  val remove : 'a -> ('a, 'b) t -> ('a, 'b) t
  val mem : 'a -> ('a, 'b) t -> bool
  val find : 'a -> ('a, 'b) t -> 'b
  val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit
  val map : ('a -> 'b) -> ('c, 'a) t -> ('c, 'b) t
  val fold : ('a -> 'b -> 'b) -> ('c, 'a) t -> 'b -> 'b
end
```

e.g.

```
module MyPMap : MYMAP =
struct
...
end ;;

open MyPMap

let intCharMap = add 1 'a' (add 2 'b' (add 3 'c' empty)) ;;
let charIntMap = add 'a' 1 (add 'b' 2 (add 'c' 3 empty)) ;;
remove 1 intCharMap ;;
find 'b' charIntMap ;;
map (fun x -> x * 10) charIntMap ;;
```