```
type zero = private Dummy1
type _ succ = private Dummy2
type nil = private Dummy3
type 'a list = Nil | Cons of 'a * 'a list


(* Exercice 1 *)
module Exo1 =
  struct
    type ('a, 'n) nlist = Nil : ('a, zero) nlist | Cons : 'a * ('a, 'n) nlist
-> ('a, 'n succ) nlist
    let rec map : type n. ('a -> 'b) -> ('a, n) nlist -> ('b, n) nlist = fun
f l ->
      match l with
      | Nil         -> Nil
      | Cons (t, q) -> Cons (f t, map f q)
    let rec snoc : type n. ('a, n) nlist -> 'a -> ('a, n succ) nlist = fun l
e ->
      match l with
      | Nil         -> Cons (e, Nil)
      | Cons (t, q) -> Cons (t, snoc q e)
    let rec rev : type n. ('a, n) nlist -> ('a, n) nlist = fun l ->
      match l with
      | Nil -> Nil
      | Cons (t, q) -> snoc (rev q) t
  end


(* Exercice 2 *)
module Exo2 =
  struct
    open Exo1
    let rec insert : type n. 'a -> ('a, n) nlist -> ('a, n succ) nlist =
      fun x -> function
      | Nil                   -> Cons(x, Nil)
      | Cons (t, q) as l -> if t < x then Cons (t, insert x q) else Cons (x,
l);;
    let rec insertion_sort : type n. ('a, n) nlist -> ('a, n) nlist =
      function
      | Nil             -> Nil
      | Cons (t, q) -> insert t (insertion_sort q);;
  end


(* Exercice 3 *)
module Exo3 =
  struct
    type _ hlist = Nil : nil hlist | Cons : 't * 'q hlist -> ('t * 'q) hlist
    let tail : type t q. (t * q) hlist -> q hlist = function (Cons (_, q)) ->
q
    let add : type q. (int * (int * q)) hlist -> (int * q) hlist = function
      | Cons (i1, Cons (i2, q)) -> Cons (i1+i2, q)
  end


(* Exercice 4 *)
module Exo4 =
  struct
    type 't expr =
      | Entier : int -> int expr
      | Booleen : bool -> bool expr
      | Plus : int expr * int expr -> int expr
      | Egal : 't expr * 't expr -> bool expr

    let rec eval : type t. t expr -> t =
      function
      | Entier i         -> i
      | Booleen b    -> b
      | Plus (e1, e2) -> eval e1 + eval e2
      | Egal (e1, e2) -> eval e1 = eval e2
  end


(* Exercice 5 *)
module Exo5 =
  struct
    open Exo4
    type valeur = Int of int | Bool of bool
    type code = PushI of int | PushB of bool | Add | Equ | Seq of code * code

    let rec compile : type t. t expr -> code =
      function
      | Entier i         -> PushI i
      | Booleen b     -> PushB b
      | Plus (e1, e2) -> Seq (compile e1, Seq (compile e2, Add))
      | Egal (e1, e2) -> Seq (compile e1, Seq (compile e2, Equ))

    let rec exec code pile =
      match code, pile with
      | PushI i         , _                                          -> (Int
i)::pile
      | PushB b       , _                                          -> (Bool
b)::pile
      | Add             , (Int  i1)::(Int  i2)::reste      -> (Int
(i1+i2))::reste
      | Equ             , (Int  i1)::(Int  i2)::reste      -> (Bool
(i1=i2))::reste
      | Equ             , (Bool b1)::(Bool b2)::reste -> (Bool
(b1=b2))::reste
      | Seq (c1, c2), _                                          -> exec c2
(exec c1 pile)
      | _                                                                ->
failwith "erreur"
  end


(* Exercice 6 *)
module Exo6 =
  struct
    (* Attention au parenthésage des produits de types !!                *)
    (* 'a * 'b * 'c /= 'a * ('b * 'c) /= ('a * 'b) * 'c
          *)
    (* Ici, il faut utiliser 'a * ('b * 'c) pour exprimer la structure de
pile *)
    type ('pile1, 'pile2) code =
      | PushI : int -> ('pile, int * 'pile) code
      | PushB : bool -> ('pile, bool * 'pile) code
      | Add  : (int * (int * 'reste), int * 'reste) code
      | Equ  : ('a * ('a * 'reste), bool * 'reste) code
```

```
      | Seq  : ('pile1, 'pile2) code * ('pile2, 'pile3) code -> ('pile1, 'pile3) code
  end


(* Exercice 7 *)
module Exo7 =
  struct
    open Exo3
    open Exo4
    open Exo6

    let rec compile : type t stackin. t expr -> (stackin, t * stackin) code =
      function
      | Entier i         -> PushI i
      | Booleen b     -> PushB b
      | Plus (e1, e2) -> Seq (Seq (compile e1, compile e2), Add)
      | Egal (e1, e2) -> Seq (Seq (compile e1, compile e2), Equ)

    let rec exec : type pile1 pile2. (pile1, pile2) code -> pile1 hlist -> pile2 hlist = function
      | PushI i  -> (fun pile                          -> Cons (i, pile))
      | PushB b -> (fun pile                          -> Cons (b, pile))
      | Add           -> (function (Cons (i1, Cons (i2, reste))) -> Cons (i1 + i2, reste))
      | Equ           -> (function (Cons (v1, Cons (v2, reste))) -> Cons (v1 = v2, reste))
      | Seq (c1, c2)  -> (function pile                          -> exec c2 (exec c1 pile))
  end

(* Exercice 7 avec variation sur les problèmes de polymorphisme *)
module Exo7bis =
  struct
    open Exo3
    open Exo4
    open Exo6

    type 't gcode = { contents : 'stackin. ('stackin, 't * 'stackin) code }

    let pushI i = { contents = PushI i }
    let pushB b = { contents = PushB b }
    let add c1 c2 = { contents = Seq (c1.contents, Seq (c2.contents, Add)) }
    let equ c1 c2 = { contents = Seq (c1.contents, Seq (c2.contents, Equ)) }

    let rec gcompile : type t. t expr -> t gcode =
      function
      | Entier i      -> pushI i
      | Booleen b     -> pushB b
      | Plus (e1, e2) -> add (gcompile e1) (gcompile e2)
      | Egal (e1, e2) -> equ (gcompile e1) (gcompile e2)

    let rec exec : type pile1 pile2. (pile1, pile2) code -> pile1 hlist -> pile2 hlist = function
      | PushI i  -> (fun pile                          -> Cons (i, pile))
      | PushB b -> (fun pile                          -> Cons (b, pile))
      | Add           -> (function (Cons (i1, Cons (i2, reste))) -> Cons (i1 + i2, reste))
      | Equ           -> (function (Cons (v1, Cons (v2, reste))) -> Cons (v1 = v2, reste))
      | Seq (c1, c2)  -> (function pile                          -> exec c2 (exec c1 pile))

  end

```