3 抽象データ構造

ベクトルの抽象データ構造

```
(* 多相型ベクトルのインターフェース *)
module type VectP = sig
  type 'a tp
  val length: 'a tp -> int
  val get: 'a tp -> int -> 'a
  val init: int -> (int -> 'a) -> 'a tp
  val to_list: 'a tp -> 'a list
end

module ArrayVectP : VectP =
  struct
    include Array                                    (* 関数が全て定義されている *)
    type 'a tp = 'a array
  end
module ArrayVect : VectP                             (* 実装を見せない *)

module ListVectP : VectP =
  struct
    type 'a tp = 'a list
    let length = List.length
    let get = List.nth
    let init n f =
      let rec loop i = if i < n then f i :: loop (i+1) else [] in loop 0
    let to_list l = l
  end
module ListVect : VectP

# let arr = ArrayVectP.init 5 (fun x -> x*x);;
val arr : int ArrayVectP.tp = <abstr>
# ArrayVectP.to_list arr;;
- : int list = [0; 1; 4; 9; 16]
# let lst = ListVectP.init 5 (fun x -> x*x);;
val lst : int ListVectP.tp = <abstr>
# ListVectP.to_list lst;;
- : int list = [0; 1; 4; 9; 16]                      (* 働きは実装によらない *)
# arr = lst;;
This expression has type int ListVectP.tp but is here used with type
  int ArrayVectP.tp
```

データ型を抽象型にして，各関数の働き決めると，外部に影響せずに内部の実装が変えられる．二つの実装に互換性がない可能性があるので，各抽象型が異なるものになっている．

データ構造をパラメーターに持ったファンクター

```
module Poly(G : Group)(V : VectP) = struct
  let add p1 p2 =
```

```
    let l1 = V.length p1 and l2 = V.length p2 in
    V.init (max l1 l2)
      (fun i -> if i < l1 then if i < l2 then G.add (V.get p1 i) (V.get p2 i)
                else V.get p1 i else V.get p2 i)
end
module Poly :
  functor (G : Group) ->
    functor (V : VectP) -> sig val add : G.t V.tp -> G.t V.tp -> G.t V.tp end

# module IntAPoly = Poly(IntGroup)(ArrayVectP) ;;
module IntAPoly : sig val add : IntGroup.t ArrayVectP.tp -> ... end
# ArrayVectP.to_list (IntAPoly.add arr arr) ;;
- : int list = [0; 2; 8; 18; 32]
```

練習問題 3　*1.* 実は to_list を VectP のインターフェースに入れなくても，同じ働きの関数が外で定義できる．V.to_list を使わずに，Poly の中に to_list : 'a V.tp -> 'a list を定義せよ．

2. 練習 *2* で定義した mult を参考にして，Ring と VectP をもらい積を定義せよ．
ヒント：ベクトルの先頭に値を追加する関数 cons を先に定義するといい．

引数間の依存関係　VectP というインターフェースには欠点がある．中身の型が決っているデータ構造を抽象化できない．例えば，文字列を char tp にしたいが，VectP は 'a tp しか許さない．それを避けるために，より具体的なインターフェースを定義する．

```
module type Vect = sig
  type t
  type elt
  val length: t -> int
  val get: t -> int -> elt
  val init: int -> (int -> elt) -> t
end

module StringVect : (Vect with type elt = char) =
  struct
    type t = string                                    (* ベクトルは文字列 *)
    type elt = char                                  (* ベクトルの要素は文字 *)
    let length = String.length                (* String から length と get *)
    let get = String.get
    let init n f =
      let s = String.create n in
      for i = 0 to n-1 do s.[i] <- f i done;
      s
  end
module StringVect :
  sig
    type t
    type elt = char
    val length : t -> int
    val get : t -> int -> elt
    val init : int -> (int -> elt) -> t
  end
```

ここでは t を抽象型にしているが，elt を公開している．そうしないとベクトルの中身が見れない．

```
module Poly(G : Group)(V : Vect with type elt = G.t) = struct
  let add p1 p2 =
    let l1 = V.length p1 and l2 = V.length p2 in
    V.init (max l1 l2)
      (fun i -> if i < l1 then if i < l2 then G.add (V.get p1 i) (V.get p2 i)
                else V.get p1 i else V.get p2 i)
end
module Poly :
  functor (G : Group) ->
    functor (V : Vect with type elt = G.t) ->
      sig val add : V.t -> V.t -> V.t end

module IntArrayVect : (Vect with type elt = int) =
  struct
    include ArrayVectP
    type t = int tp
    type elt = int
  end

# module IntAPoly = Poly(IntGroup)(IntArrayVect) ;;
module IntAPoly :
  sig val add : IntArrayVect.t -> IntArrayVect.t -> IntArrayVect.t end
```

Poly の中の Vect with type elt = G.t は依存関係を表している．G.t と V.elt は同じ型でなければならない．

練習問題 4 二つの Vect 型のモジュール A と B をもらって，以下の変換関数を返すファンクター ConvertVect を定義せよ．

```
val a_to_b : A.t -> B.t
val b_to_a : B.t -> A.t
```