

```

1  (***** Algorithmes combinatoires et monades *****)
2
3  module type FONCTEUR =
4    sig
5      type 'a t
6      val map : ('a -> 'b) -> ('a t -> 'b t)
7    end
8
9  module type MONADE =
10   sig
11     include FONCTEUR
12     val return : 'a -> 'a t
13     val (>=) : 'a t -> ('a -> 'b t) -> 'b t
14   end
15
16  module type MONADE_PLUS =
17   sig
18     include MONADE
19     val zero : 'a t
20     val (++) : 'a t -> 'a t -> 'a t
21   end
22
23  (* interface incluant l'affichage des éléments calculés *)
24  (* pour les listes d'entiers uniquement *)
25  module type MONADE_PLUS_PRINT =
26   sig
27     include MONADE_PLUS
28     val print : Format.formatter -> int list t -> unit
29   end
30
31  (* fonction auxiliaire pour compter le nombre maximum d'octets alloués en
   mémoire *)
32  let max_bytes () =
33    let stat = Gc.stat () in
34    8. *. (stat.minor_words +. stat.major_words -. stat.promoted_words)
35
36  (* fonction auxiliaire pour afficher une liste d'entiers *)
37  let print_int_list fmt l =
38    begin
39      Format.fprintf fmt "[";
40      List.iter (Format.fprintf fmt "%d; ") l;
41      Format.fprintf fmt "]"
42    end
43
44  (* implantation de la monade ND avec des listes *)
45  (* ne fonctionne qu'en l'absence de doublons *)
46  module NDList : MONADE_PLUS_PRINT =
47   struct
48     type 'a t = 'a list
49     let map = List.map
50     let return v = [v]
51     let (>=) s f = List.flatten (List.map f s)
52     let zero = []
53     let (++) = (@)
54
55     (* fonction d'affichage pour les tests *)
56     let print fmt =
57       List.iter (Format.fprintf fmt "%a@." print_int_list)

```

```

59   end
60
61  (** Combinaisons d'une liste **)
62
63  module Exo1 (ND : MONADE_PLUS) =
64   struct
65     (* CONTRAT
66      *
67      * Fonction qui renvoie toutes les combinaisons possible de k éléments
   d'une liste l
68      * Paramètre k : le nombre d'éléments dans la liste retournée
69      * Précondition : k >= taille de l
70      * Paramètre l : la liste dans laquelle on prend les éléments
71      * Résultat : les combinaisons de k éléments choisis dans l
72      *)
73     let rec combinaisons k l =
74       match k, l with
75       | 0, _ -> ND.return []
76       | _, [] -> ND.zero
77       | _, t::q -> ND.(combinaisons k q ++ (combinaisons (k-1) q >= fun
   combi -> return (t::combi)))
78
79   end
80
81  (* TESTS *)
82  let test1 (module ND : MONADE_PLUS_PRINT) =
83    let module M = Exo1 (ND) in
84    let old_bytes = max_bytes () in
85    let result = M.combinaisons 4 [1;2;3;4;5;6;7;8;9;10] in
86    begin
87      Format.printf "@.TEST combinaisons@.memory used: %f@.result:@. %a@."
   (max_bytes () -. old_bytes) ND.print result
88    end
89
90  let _ = test1 (module NDList)
91
92  (** Permutations d'une liste **)
93
94  module Exo2 (ND : MONADE_PLUS) =
95   struct
96     (* CONTRAT
97      *
98      * Fonction prend en paramètre un élément e et une liste l et qui insère
   e à toutes les positions possibles dans l
99      * Paramètre e : ('a) l'élément à insérer
100     * Paramètre l : ('a list) la liste initiale dans laquelle insérer e
101     * Résultat : toutes les insertions possible de e dans l
102     *)
103
104     let rec insertion e l =
105       match l with
106       | [] -> ND.return [e]
107       | t::q -> ND.(return (e::l) ++ (insertion e q >= fun ins -> return
   (t::ins)))
108
109     (* CONTRAT
110      *
111      * Fonction qui renvoie la liste des permutations d'une liste
112      * Paramètre l : une liste
113      * Résultat : les permutations de l (toutes différentes si les éléments
   de l sont différents deux à deux)

```

```

113 *)
114 let rec permutations l =
115   match l with
116   | [] -> ND.return []
117   | t::q -> ND.(permutations q >=> fun perm -> insertion t perm)
118 end
119
120 (* TESTS *)
121 let test2 (module ND : MONADE_PLUS_PRINT) =
122   let module M = Exo2 (ND) in
123   let old_bytes = max_bytes () in
124   let result = M.permutations [1;2;3;4;5] in
125   begin
126     Format.printf "@.TEST permutations@.memory used: %f@.result:@. %a@."
127     (max_bytes () -. old_bytes) ND.print result
128   end
129 let _ = test2 (module NDList)
130
131 (** Partition d'un entier **)
132
133 module Exo3 (ND : MONADE_PLUS) =
134   struct
135     (* CONTRAT
136      partition int -> int list
137      Fonction qui calcule toutes les partitions possibles d'un entier n
138      Paramètre n : un entier dont on veut calculer les partitions
139      Préconditions : n > 0
140      Résultat : les partitions de n
141      *)
142     let partitions n =
143       let rec partitions_aux n t =
144         if t > n then ND.zero
145         else if t = n then ND.return [t]
146         else (* t < n *) ND.((partitions_aux (n-t) t >=> fun part -> return
147 (t::part)) ++ partitions_aux n (t+1))
148       in partitions_aux n 1
149     end
150
151     (* TESTS *)
152     let test3 (module ND : MONADE_PLUS_PRINT) =
153       let module M = Exo3 (ND) in
154       let old_bytes = max_bytes () in
155       let result = M.partitions 5 in
156       begin
157         Format.printf "@.TEST partitions@.memory used: %f@.result: %a@."
158         (max_bytes () -. old_bytes) ND.print result
159       end
160     let _ = test3 (module NDList)
161
162     (* fonction auxiliaire pour réaliser tous les tests des fonctions
163     combinatoires *)
164     let test_combinatoire (module ND : MONADE_PLUS_PRINT) =
165       begin
166         test1 (module ND);
167         test2 (module ND);
168         test3 (module ND)

```

```

169 end
170
171 (** Autre implantation de ND par itérateurs **)
172
173 module NDIter : MONADE_PLUS_PRINT =
174   struct
175     type 'a t = Tick of ('a * 'a t) option Lazy.t
176
177     let next (Tick it) = Lazy.force it
178
179     let rec map f (it : 'a t) =
180       Tick (lazy (
181         match next it with
182         | None -> None
183         | Some (a, it') -> Some (f a, map f it')
184       ))
185
186     let return a =
187       Tick (lazy (Some (a, Tick (lazy None))))
188
189     let zero =
190       Tick (lazy None)
191
192     let rec (++) it1 it2 =
193       Tick (lazy (
194         match next it1 with
195         | None -> next it2
196         | Some (a1, it1') -> Some (a1, it1' ++ it2)
197       ))
198
199     let rec (>=>) it f =
200       Tick (lazy (
201         match next it with
202         | None -> None
203         | Some (a, it') -> next (f a ++ (it' >=> f))
204       ))
205
206     (* fonction d'affichage pour les tests *)
207     let rec print fmt it =
208       match next it with
209       | None -> Format.fprintf fmt "@."
210       | Some (a, it') -> Format.fprintf fmt "%a@.%a" print_int_list a print
211       it'
212     end
213
214     (* TESTS *)
215     let test_iter () =
216       begin
217         Format.printf "@.TEST itérateur@";
218         test_combinatoire (module NDIter)
219       end
220
221     (** Autre implantation de ND par tirage aléatoire **)
222
223     module NDRandom : MONADE_PLUS_PRINT =
224       struct
225         type 'a t = unit -> 'a option
226
227

```

```

228 let map f it =
229     fun () -> match it () with
230     | None    -> None
231     | Some a  -> Some (f a)
232
233 let return a = fun () -> Some a
234
235 let (>=>) it f =
236     fun () -> match it () with
237     | None    -> None
238     | Some a  -> f a ()
239
240 let zero = fun () -> None
241
242 let (++) it1 it2 =
243     fun () -> if Random.bool ()
244     then
245         match it1 () with
246         | None -> it2 ()
247         | r   -> r
248     else
249         match it2 () with
250         | None -> it1 ()
251         | r   -> r
252
253 (* fonction d'affichage pour les tests *)
254 let print fmt it =
255     match it () with
256     | None    -> Format.fprintf fmt "@."
257     | Some v  -> Format.fprintf fmt "%a@." print_int_list v
258 end
259
260 (* TESTS *)
261 let test_random (module ND : MONADE_PLUS_PRINT) =
262     begin
263         Format.printf "@.TEST aléatoire@"; test_combinatoire (module ND)
264     end
265
266 let _ = test_random (module NDRandom)
267
268 (** Autre implantation de ND par tirage aléatoire équitale **)
269
270 module NDFairRandom : MONADE_PLUS_PRINT =
271     struct
272         type 'a t = int * (unit -> 'a)
273
274         let map f (n, it) = (n, fun () -> f (it ()))
275
276         let return a = (1, fun () -> a)
277
278         let zero = (0, fun () -> raise Not_found)
279
280         let (++) (n1, it1) (n2, it2) =
281             match n1, n2 with
282             | 0, 0 -> zero
283             | 0, _ -> (n2, it2)
284             | _, 0 -> (n1, it1)
285             | _   -> (n1+n2, fun () -> if Random.int (n1+n2) < n1 then it1 () else
it2 ())
286

```

```

287 let (>=>) (n, it) f =
288     match n with
289     | 0 -> zero
290     | _ -> match f (it ()) with
291     | (p, _) -> (n*p, fun () -> snd (f (it ())) ())
292
293 (* fonction d'affichage pour les tests *)
294 let print fmt (n, it) =
295     match n with
296     | 0 -> Format.fprintf fmt "@."
297     | _ -> Format.fprintf fmt "%a@." print_int_list (it ())
298 end
299
300 (* TESTS *)
301 let _ = test_random (module NDFairRandom)
302

```