Exercice 1 : Ordre lexicographique

On rappelle qu'en OCaml la fonction `(<=) : 'a -> 'a -> bool` permet de comparer deux éléments de même type. L'*ordre lexicographique* sur des listes d'éléments de type `'a` est l'ordre du dictionnaire, où les listes jouent le rôle de mots dont les lettres sont les éléments contenus dans la liste. Par exemple `[1; 2; 2]` est lexicographiquement plus petit que `[1; 3]`, `[1; 2]` est lexicographiquement plus petit que `[1; 2; 2]`. `[1; 2; 2]` est lexicographiquement plus petit que `[1; 3; 0]`.

Q. 1 Définir une fonction `lexico_inf : 'a list -> 'a list -> bool` prenant en paramètres deux listes et testant si la première est, ou non, strictement inférieure à la seconde, pour l'ordre lexicographique.

Solution

```
let rec lexico_inf (l1: 'a list) (l2: 'a list): bool =
  match l1, l2 with
  | [], _ :: _           -> true
  | x1 :: l1', x2 :: l2' -> (x1 < x2) || ((x1 = x2) && (lexico_inf l1' l2'))
  | _                    -> false
```

Q. 2 Comment rendre cette fonction paramétrique en l'opérateur de comparaison des éléments dans les listes ?

Solution

On passe en paramètre à la fonction, une fonction de comparaison. Une fonction de comparaison est une fonction de type `'a -> 'a -> int` prenant en paramètres deux éléments à comparer et retournant un nombre < 0 si le premier est inférieur strictement au deuxième, retournant 0 si le premier est égal au deuxième, retournant un nombre > 0 si le premier est strictement supérieur au deuxième.

```
let rec lexico_inf_p (c: 'a -> 'a -> int) (l1: 'a list) (l2: 'a list): bool =
  match l1, l2 with
  | [], _ :: _           -> true
  | x1 :: l1', x2 :: l2' ->
    let comp = c x1 x2 in
    (comp < 0) || ((comp = 0) && (lexico_inf l1' l2'))
  | _                    -> false
```

Exercice 2 : Sous listes contiguës

Q. 1 Écrire une fonction `sous_listes_contigues: 'a list -> 'a list list` prenant en paramètre une liste `l` et calculant l'ensemble de ses sous-listes non vides d'éléments contigus dans la liste `l`.

```
sous_listes_contigues [1; 2; 3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [2]; [2; 3]; [3]]
```

Solution

```
(* Retourne un couple :
   - la liste des sous-listes contiguës de la liste passée en paramètre ;
   - la liste des sous-listes contiguës (de la liste passée en paramètre)
   dont le premier élémént est l'élément le plus à gauche de la liste. *)
let rec sous_listes_contigues_et_contigues_colles_a_gauche (l: 'a list)
```

```
  : 'a list list * 'a list list =
  match l with
  | []     -> [], []
  | x :: l ->
    let sls, sls_cg = sous_listes_contigues_et_contigues_colles_a_gauche l in
    (* les sous-listes collées à gauche, avec un x en tête. *)
    let sls_cg_avec_x = List.map (fun sl -> x :: sl) sls_cg in
    ([x] :: sls_cg_avec_x @ sls, [x] :: sls_cg_avec_x)

let sous_listes_contigues (l: 'a list) : 'a list list =
  fst (sous_listes_contigues_et_contigues_colles_a_gauche l)
```

Exercice 4 : Ordre lexicographique

On rappelle qu'en OCaml la fonction `(<=) : 'a -> 'a -> bool` permet de comparer deux éléments de même type. L'*ordre lexicographique* sur des tableaux d'éléments de type `'a` est l'ordre du dictionnaire, où les tableaux jouent le rôle de mots dont les lettres sont les éléments contenus dans la liste. Par exemple `[|1; 2; 2|]` est lexicographiquement plus petit que `[|1; 3|]`, `[|1; 2|]` est lexicographiquement plus petit que `[|1; 2; 2|]`. `[1; 2; 2|]` est lexicographiquement plus petit que `[1; 3; 0|]`.

Q. 1 Définir une fonction `lexico_inf : 'a array -> 'a array -> bool` prenant en paramètres deux tableaux et testant si la première est, ou non, strictement inférieure à la seconde, pour l'ordre lexicographique.

Solution

```
let lexico_inf (t1: 'a array) (t2: 'a array): bool =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  (* (aux i) retourne si oui ou non le sous-tableau t1[i:len(t1)] est
     strictement inférieur, pour l'ordre lexicographique au tableau
     t2[i:len(t2)]. *)
  let rec aux (i: int): bool =
    (i = n1 && n1 <> n2)
    || (i < n1 && i < n2 && (
        t1.(i) < t2.(i)
        || (t1.(i) = t2.(i) && aux (i + 1))))
  in aux 0
```

Q. 2 Comment rendre cette fonction paramétrique en l'opérateur de comparaison des éléments dans les tableaux ?

Solution

On passe en paramètre à la fonction, une fonction de comparaison. Une fonction de comparaison est une fonction de type `'a -> 'a -> int` prenant en paramètres deux éléments à comparer et retournant un nombre < 0 si le premier est inférieur strictement au deuxième, retournant 0 si le premier est égal au deuxième, retournant un nombre > 0 si le premier est strictement supérieur au deuxième.

```
let lexico_inf (c: 'a -> 'a -> int) (t1: 'a array) (t2: 'a array): bool =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  (* (aux i) retourne si oui ou non le sous-tableau t1[i:len(t1)] est
     strictement inférieur, pour l'ordre lexicographique au tableau
     t2[i:len(t2)]. *)
  let rec aux (i: int): bool =
    (i = n1 && n1 <> n2)
    || (i < n1 && i < n2 && (
        let comp = c t1.(i) t2.(i) in
        comp < 0
        || (comp = 0 && aux (i + 1))))
  in aux 0
```

Exercice 5 : Nombre d'occurrences

Q. 1 Définir une fonction `histogramme (donnees: int list) (pas: int) (max: int): int array` prenant en paramètres une liste d'entiers (`donnees`) dont les éléments sont tous dans l'intervalle entier $[\![0, \texttt{max}-1]\!]$, un pas entier (`pas`) et une valeur entière `max` et retournant un tableau

`t` de taille $\lceil \frac{\texttt{max}}{\texttt{pas}} \rceil$ contenant dans sa case d'indice `i` un entier indiquant le nombre d'entiers de la liste `donnees` se trouvant dans l'intervalle entier $[\![i \times \texttt{pas}, (i+1) \times \texttt{pas} - 1]\!]$.

Solution

```
let histogramme (donnees: int list) (pas: int) (max: int): int array =
  let taille = (max / pas) + (if max mod pas = 0 then 0 else 1) in
  let rep = Array.make taille 0 in
  List.iter (fun donnee ->
      rep.(donnee / pas) <- rep.(donnee / pas) + 1
    ) donnees;
  rep
```

Exercice 7 : Remplissage d'un tableau

Q. 1 Définir une fonction `remplissage : int array -> int list -> unit` prenant en paramètres un tableau d'entiers `t` et une liste *non vide* `l` et remplaçant, dans le tableau `t` les occurrences du nombre -1 par des valeurs piochées dans la liste `l`. Les valeurs de la liste `l` devront être utilisées dans l'ordre, si on atteint la fin de la liste, on recommence. Par exemple si `t` est le tableau `[|1; 2; -1; 0; -2; -1; -1|]` l'appel `(remplissage t [4; 5; 6; 8])` l'appel devra modifier le tableau `t` en la valeur `[|1; 2; 4; 0; -2; 5; 6|]`. `(remplissage t [4; 5])` devra modifier le tableau `t` en la valeur `[|1; 2; 4; 0; -2; 5; 4|]`.

Solution

```
let remplissage (t: int array) (l: int list): unit =
  (* Remplis le tableau t[i:len(t)] avec les éléments de la liste ll. *)
  let rec aux (i: int) (ll: int list): unit =
    match t.(i), ll with
    | -1, x :: ll' -> t.(i) <- x ; aux (i + 1) ll'
    | -1, []       -> aux i l
    | _            -> aux (i + 1) ll
  in
  aux 0 l
```

Exercice 8 : Fusion de listes en tableau

Q. 1 Définir une fonction `fusion : 'a list -> 'a list -> 'a array` prenant en paramètres deux listes *triées par ordre croissant* et retournant un tableau `t`, *trié par ordre croissant* et dont les éléments sont ceux se trouvant dans l'union des listes en paramètres.

Solution

```
(* Si l1 ou l2 est non vide, retourne un élément se trouvant dans l1 ou
   dans l2 *)
let trouve_un (l1: 'a list) (l2: 'a list): 'a option =
  match l1, l2 with
  | x :: l', _ | _, x :: l' -> Some x
  | _                       -> None

let rec fusion (l1: 'a list) (l2: 'a list): 'a array =
  match trouve_un l1 l2 with
  | None    -> [||]
  | Some(x) ->
    let n1 = List.length l1 in
    let n2 = List.length l2 in
    let rep = Array.make (n1 + n2) x in
    (* Remplis rep[i:len(rep)] avec les éléments se trouvant dans ll1 et
       ll2, en préservant les croissances. *)
    let rec aux (i: int) (ll1: 'a list) (ll2: 'a list): unit =
      match ll1, ll2 with
      | x1 :: ll1', x2 :: ll2' when x1 < x2 ->
        rep.(i) <- x1;
        aux (i + 1) ll1' ll2
      | x1 :: ll1', x2 :: ll2' (* when x1 >= x2 *) ->
        rep.(i) <- x2;
        aux (i + 1) ll1 ll2'
      | x :: ll', _ | _, x :: ll' ->
        rep.(i) <- x;
```

```
        aux (i + 1) ll' []
      | _ -> ()
    in aux 0 l1 l2;
    rep
```

Exercice 9 : Sous-séquences monotones

Q. 1 Définir une fonction `sseq_monotones : 'a array -> int list` prenant en paramètres un tableau d'éléments et retournant la liste des longueurs des sous-séquences monotones du tableau en paramètre. La longueur d'une séquence monotone est le nombre d'éléments se trouvant dans la séquence décrémenté de 1. Par exemple le tableau `[|1; 5; 8; 4; 6; 9; 8; 7; 6; 1; 9|]` contient les séquences monotones : `[|1; 5; 8|]`, `[|8; 4|]`, `[|4; 6; 9|]`, `[|9; 8; 7; 6; 1|]` et finalement `[|1; 9|]` de longueurs respectives : 2, 1, 2, 4, 1. Ainsi le résultat attendu pour cette entrée est la liste `[2; 1; 2; 4; 1]`. *Indication :* On prendra soin de définir des fonctions auxiliaires et d'expliciter leurs comportements.