Devoir surveillé #3 (avec solutions)

Vendredi 2026-01-23 ; durée : trois heures

Dans tout le sujet, vous pouvez (sauf mention du contraire) réutiliser au sein **d'un même exercice** une fonction demandée à une question précédente, même si la question n'a pas été traitée. Vous pouvez de même utiliser le résultat d'une question précédente même si celle-ci n'a pas été traitée.
Sauf mention du contraire, vous **ne pouvez pas** utiliser de fonctions de la bibliothèque standard OCaml.

Exercice 1. *Liste impérative par référence sur un type immuable*

Le but de cet exercice est d'implémenter (quelques unes des fonctions d') une structure de données de liste impérative en OCaml, en utilisant pour cela un `type 'a ilist = 'a list ref` de référence sur des `'a list`
Dans cette implémentation, on a par exemple qu'une liste vide est représentée par une référence vers une `'a list` vide, et une liste non vide est représentée par une référence vers une `'a list` de motif `x::xs`, c'est à dire dont la tête est `x` et la queue `xs`.

1. Écrivez une fonction OCaml :
   `make_empty : unit -> 'a ilist`
   telle que `make_empty ()` s'évalue en une nouvelle `'a ilist` vide (ne contenant aucun élément).

   On propose :
   ```
   let make_empty () : 'a ilist = ref []
   ```

2. Écrivez une fonction OCaml :
   `insert_head : 'a -> 'a ilist -> unit`
   telle que `insert_head e x` modifie la `'a list` référencée par `x` (appelons-la `x'`) pour la remplacer par la `'a list` dont `e` est la tête et `x'` la queue.

   On propose :
   ```
   let insert_head e (x : 'a ilist) = x := e::!x
   ```

Dans toute la suite de l'exercice, on suppose définie une `exception Empty`.

3. Écrivez une fonction OCaml :
   `remove_head : 'a ilist -> unit`
   qui lève une exception `Empty` si son argument référence une `'a list` vide, et sinon modifie la liste référencée par son argument en la queue de celle-ci.

   On propose :
   ```
   let remove_head (x : 'a ilist)
       = match !x with
       | [] -> raise Empty
       | _::xs -> x := xs
   ```

4. Écrivez une variante :
   `remove_head_s : 'a ilist -> unit`
   de spécifications identiques à `remove_head`, à la seule différence qu'elle ne fait rien (plutôt que de lever une exception) lorsque son argument référence une liste vide. Cette fonction **devra** être implémentée en utilisant `remove_head` en « boîte noire » et en traitant une éventuelle exception `Empty`.

   On propose :
   ```
   let remove_head_s (x : 'a ilist)
       = try remove_head x with Empty -> ()
   ```

5. Écrivez une fonction OCaml :
   `get_head : 'a ilist -> 'a`
   qui lève une exception `Empty` si son argument référence une `'a list` vide, et sinon s'évalue en la tête de la liste référencée.

   On propose :
   ```
   let get_head (x' : 'a ilist)
       = match !x' with
       | [] -> raise Empty
       | x::_ -> x
   ```

Dans toute la suite de l'exercice, on s'autorise l'utilisation de la fonction `rev_append` du module `List`, qui est telle que `rev_append x y` s'évalue en une nouvelle liste obtenue en concaténant le miroir de `x` avec `y`. (autrement dit, `List.rev_append x y` s'évalue identiquement à `(List.rev x) @ y`). Cette fonction est de coût linéaire en la longueur de son premier argument `x`.
On s'**interdit** en revanche toute utilisation des fonctions `List.rev` et `(@)`.

6. Écrivez une fonction OCaml :
   `_remove : 'a -> 'a ilist -> 'a list -> 'a list -> unit`
   telle que `_remove e x p s` lève une exception `Not_found` si la liste `s` ne contient pas d'élément égal à `e`, et sinon modifie la liste référencée par `x` par la liste obtenue en concaténant le miroir de la liste `p` avec la liste `s` dont on a supprimé la première occurrence (en partant de la tête) d'un élément égal à `e`. Cette fonction devra avoir un coût au plus linéaire en la somme des longueurs de ses arguments `p` et `s`, mais on ne demande pas de démontrer ce coût.

   On propose :
   ```
   let rec _remove e (x' : 'a ilist) (p : 'a list) (s : 'a list)
       = match s with
       | [] -> raise Not_found
       | x::xs -> if x = e then
                  x' := List.rev_append p xs else
                  _remove e x' (x::p) xs
   ```

7. Écrivez une fonction OCaml :
   `remove : 'a -> 'a ilist -> unit`
   telle que `remove e x` lève une exception `Not_found` si la liste référencée par `x` ne contient pas d'élément égal à `e`, et sinon remplace cette dernière par une liste égale à la liste initiale dont on a supprimé la première occurrence de `e`. Cette fonction devra être la plus simple possible.

   On propose :
   ```
   let remove e x = _remove e x [] !x
   ```

8. Écrivez une fonction OCaml :
   `remove_s : 'a -> 'a ilist -> unit`
   de spécifications identiques à `remove`, à la seule différence qu'elle ne fait rien (plutôt que de lever une exception) lorsque la liste référencée par `x` ne contient pas d'élément égal à `e`. Cette fonction devra être la plus simple possible.

   On propose :
   ```
   let remove_s e x
       = try remove e x with Not_found -> ()
   ```

Exercice 2. *File à deux bouts par tableau circulaire*
Le but de cet exercice est d'étudier une structure de données abstraite de « file à deux bouts » (*double-ended queue*, ou *deque*) impérative, puis de l'implémenter en C en utilisant un *tableau circulaire*.

Structure de données abstraite de *deque*

On définit « abstraitement » les fonctions suivantes opérant sur une *deque* :

— *make_empty* : constructeur renvoyant une nouvelle *deque* vide.
— *push_left* : transformateur tel que pour une *deque d* et un élément *e*, *push_left e d* modifie *d* pour y ajouter *e* à son extrémité gauche.
— *push_right* : transformateur tel que pour une *deque d* et un élément *e*, *push_right e d* modifie *d* pour y ajouter *e* à son extrémité droite.
— *peek_left* : accesseur tel que pour une *deque d* non vide, *peek_left d* renvoie l'élément de *d* se trouvant à son extrémité gauche.
— *peek_right* : accesseur tel que pour une *deque d* non vide, *peek_right d* renvoie l'élément de *d* se trouvant à son extrémité droite.
— *pop_left* : accesseur & transformateur tel que pour une *deque d* non vide, *pop_left d* renvoie l'élément de *d* se trouvant à son extrémité gauche et modifie *d* pour l'en supprimer.
— *pop_right* : accesseur & transformateur tel que pour une *deque d* non vide, *pop_right d* renvoie l'élément de *d* se trouvant à son extrémité droite et modifie *d* pour l'en supprimer.

On illustre ces différentes fonctions (et leurs interactions) par les résultats renvoyés par la suite d'appels suivants (écrits en « pseudocode » abstrait) :

```
d = make_empty ();
push_left 1 d;
push_left 2 d;
peek_left d;  // renvoie 2
peek_right d; // renvoie 1
push_right 0 d;
peek_right d; // renvoie 0
pop_left d;   // renvoie 2
pop_right d;  // renvoie 0
pop_right d;  // renvoie 1
```

FIGURE 1 – Suite d'appels sur une *deque*

1. Expliquez comment il est possible d'implémenter une structure de donnée abstraite de pile impérative à partir de la structure de données abstraite de *deque* décrite ci-dessus. Plus précisément, expliquez comment implémenter chacune des fonctions :

— *s_make_empty* ;
— *s_push* ;
— *s_pop* ;

d'une telle structure de données abstraite de pile en fonction des fonctions ci-dessus. *Aucune justification n'est attendue.*

Il suffit de prendre :

— *s_make_empty* = *make_empty* ;
— *s_push* = *push_left* (ou *push_right*) ;
— *s_pop* = *pop_left* (ou *pop_right*).

2. Même question pour une structure de donnée abstraite de file impérative : expliquez comment implémenter chacune des fonctions :

— *q_make_empty* ;
— *q_push* ;
— *q_pop* ;

d'une telle structure de données abstraite de file en fonction des fonctions ci-dessus. *Aucune justification n'est attendue.*

Il suffit de prendre :

- *q_make_empty* = *make_empty* ;
- *q_push* = *push_left* (ou *push_right*) ;
- *q_pop* = *pop_right* (ou *pop_left*).

Tableau circulaire

On définit (abstraitement) le type de *tableau circulaire* d'entiers comme le produit :

- d'un entier *capacité* strictement positif ;
- d'un tableau *data* indexé à partir de zéro pouvant contenir *capacité* entiers ;
- d'un entier *longueur* positif ou nul inférieur ou égal à *capacité* ;
- d'un entier *left* positif ou nul inférieur strictement à *capacité* ;
- d'un entier *right* positif ou nul inférieur strictement à *capacité* ;

avec les contraintes que :

- si *len* est égal à zéro, alors on doit avoir *left* égal à *right* ;
- sinon, on doit avoir *right* moins *left* plus un congru à *len* modulo *capacité*.

Dans toute la suite de l'exercice, un tel tableau circulaire sera représenté concrètement par le type C suivant :

```
struct circ
{
    int *data;
    size_t l;   // left
    size_t r;   // right
    size_t len; // longueur
    size_t cap; // capacité
};
```

où `data`, `l`, `r`, `len` et `cap` correspondent respectivement aux *data*, *left*, *right*, *longueur* et *capacité* ci-dessus. On illustre également informellement plusieurs états possibles d'un tel tableau circulaire ci-dessous, où « - » (respectivement « . ») désigne une case du tableau occupée (non occupée) par un élément :

```
// cap = 14
//
// len = 0 :
//
// l, r = 6 :
// [..............]
//        ^
//      l,r
//
// l, r = 0 :
// [..............]
//  ^
// l,r
//
// len = 1 ;
// l, r = 0 :
// [-.............]
//  ^
// l,r
//
// len = 7 ;
// l = 3, r = 9 :
// [...-------....]
//     ^     ^
//     l     r
```

```
//
// len = 6 ;
// l = 11, r = 2 :
// [---.........---]
//    ^         ^
//    r         l
//
// len = 14 ;
// l = 11, r = 10 :
// [--------------]
//             ^^
//             rl
```

Consignes générales. Dans toute la suite de l'exercice, on ne pourra **pas** supposer que les allocations dynamiques de mémoire se déroulent toujours avec succès, **sauf** dans les questions où cela est **explicitement** indiqué. De plus, toute présence d'une fuite mémoire est considérée comme une erreur.

3. Écrivez une fonction C de signature :
`struct circ *circ_make8(void)`
qui crée un objet de type `struct circ` représentant un tableau circulaire vide de capacité 8 (comme décrit ci-dessus) de durée de stockage « allouée » (« alloué sur le tas ») et en renvoie une adresse.
On pourra supposer dans cette question que toutes les allocations dynamiques de mémoire se déroulent toujours avec succès.

On propose :
```c
struct circ *circ_make8(void)
{
    struct circ *c = malloc(sizeof(struct circ));
    c->cap = 8;
    c->data = malloc(c->cap * sizeof(int));
    c->l = 0;
    c->r = 0;
    c->len = 0;

    return c;
}
```

4. Écrivez une fonction C de signature :
`void circ_free(struct circ *c)`
qui libère (ou détruit ; c'est à dire, met fin à la durée de vie de) l'objet de type `struct circ` dont l'adresse est donnée en argument.

Attention aux fuites mémoires et aux *use-after-free* ! On propose :
```c
void circ_free(struct circ *c)
{
    free(c->data);
    free(c);
}
```

5. Écrivez une fonction C de signature :
`size_t mod_incr(size_t i, size_t mod)`
qui pour $i \in [\![0, mod-1]\!]$ renvoie l'unique entier $j \in [\![0, mod-1]\!]$ congru à $i+1$ modulo mod. Pour des raisons de coût, cette fonction ne devra pas utiliser d'opérateur de division (`/` ou `%`).

On propose :
```c
size_t mod_incr(size_t i, size_t mod)
{
```

```
    if (i == mod - 1)
    {
        return 0;
    }
    else
    {
        return i + 1;
    }
}
```

On supposera dans la suite l'existence d'une fonction de signature :
`size_t mod_decr(size_t i, size_t mod)`
qui pour $i \in [\![0, mod-1]\!]$ renvoie l'unique $j \in [\![0, mod-1]\!]$ congru à $i-1$ modulo mod.

6. Écrivez une fonction C de signature :
`void push_left(struct circ *c, int x)`
qui pour un premier argument pointant vers un tableau circulaire (valide) représenté par un objet de type `struct circ` non plein (c'est à dire qui n'est pas tel que `len` est égal à `cap`) le *modifie* pour y **ajouter** un élément à l'indice immédiatement inférieur à `l` ou, si `l` est égal à zéro, à l'indice `cap - 1` ou, si `len` est égal à zéro, à l'indice `l`. Cette fonction doit également faire les éventuelles modifications nécessaires pour conserver une représentation de tableau circulaire valide.

On propose :
```
void push_left(struct circ *c, int x)
{
    assert(c->len < c->cap);

    if (c->len == 0)
    {
        c->data[c->l] = x;
    }
    else
    {
        size_t lnew = mod_decr(c->l, c->cap);
        c->l = lnew;
        c->data[lnew] = x;
    }
    c->len = c->len + 1;

    return;
}
```

On suppose écrite de même une fonction C de signature :
`void push_right(struct circ *c, int x)`
qui pour un premier argument pointant vers un tableau circulaire (valide) représenté par un objet de type `struct circ` non plein (c'est à dire qui n'est pas tel que `len` est égal à `cap`) le modifie pour y ajouter un élément à l'indice immédiatement *supérieur* à `r` ou, si `r` est égal à `cap - 1`, à l'indice `0` ou, si `len` est égal à zéro, à l'indice `r`.

7. Écrivez une fonction C de signature :
`int pop_right(struct circ *c)`
qui pour un premier argument pointant vers un tableau circulaire (valide) représenté par un objet de type `struct circ` non vide (c'est à dire qui n'est pas tel que `len` est égal à `0`) le *modifie* pour y **supprimer** l'élément à l'indice `r` et le renvoie. Cette fonction doit également faire les éventuelles modifications nécessaires pour conserver une représentation de tableau circulaire valide.
Remarque : par *suppression* d'un élément, on entend uniquement le fait qu'il ne sera plus possible d'y accéder légalement.

On suppose écrite de même une fonction C de signature :

`int pop_left(struct circ *c)`
qui pour un premier argument pointant vers un tableau circulaire (valide) représenté par un objet de type `struct circ` non vide le modifie pour y supprimer l'élément à l'indice `l` et le renvoie.

On propose :

```
int pop_right(struct circ *c)
{
    assert(c->len > 0);

    int x = c->data[c->r];
    if (c->len > 1)
    {
        c->r = mod_decr(c->r, c->cap);
    }
    c->len = c->len - 1;

    return x;
}
```

Par la remarque ci-dessus, il n'est pas nécessaire de modifier (par quoi, d'ailleurs ?) la valeur de `data` où se trouvait `x`.

On souhaite maintenant profiter du fait que notre représentation de *deque* permet d'effectuer des « accès aléatoires » efficaces à ses éléments.

8. Écrivez une fonction C de signature :
`int get(struct circ *c, size_t i)`
qui pour un premier argument pointant vers un tableau circulaire (valide) représenté par un objet de type `struct circ` de longueur strictement supérieure à son second argument *i* renvoie la valeur du *i*ème élément du tableau, en indexant à partir de zéro et en partant de la gauche. Par exemple, pour *i* égal à `0`, cette fonction doit renvoyer l'élément de `data` à l'indice `l`, et pour *i* égal à `len - 1` celui à l'indice `r`.
On pourra utiliser les opérateurs de division dans cette question, et supposer l'absence de tout dépassement de capacité sur les types entier.

On propose :

```
int get(struct circ *c, size_t i)
{
    assert(i < c->len);

    return c->data[(c->l + i) % c->cap];
}
```

9. Donnez la suite des états du tableau circulaire de capacité 8 représenté par un objet de type `struct circ` créé puis manipulé par la suite d'instructions de la Figure 1 (concrétisée avec les fonctions ci-dessus), en ignorant toutes les instructions *peek*. On indiquera (éventuellement visuellement) pour chaque état les valeurs des champs `l`, `r` et `len`, ainsi que les `len` valeurs du tableau occupées par des éléments de la *deque* (par ex. séparées par des `|`).
Par exemple, un tableau circulaire contenant trois éléments (de gauche à droite) `1`, `2`, `3` avec `l` égal à `2` et `r` égal à `4` peut se représenter visuellement comme :

```
len = 3
[..1|2|3...]
   ^   ^
   l   r
```

On a (en gardant la syntaxe des appels « abstraits », mais en les supposant réalisés par les fonctions C ci-dessus) :

```
d = make_empty ();
// len = 0
// [........]
//  ^
// l,r
push_left 1 d;
```

```
// len = 1
// [1.......]
//  ^
// l,r
push_left 2 d;
// len = 2
// [1......2]
//  ^      ^
//  r      l
push_right 0 d;
// len = 3
// [1|0.....2]
//    ^     ^
//    r     l
pop_left d;
// len = 2
// [1|0......]
//  ^ ^
//  l r
pop_right d;
// len = 1
// [1.......]
//  ^
// l,r
pop_right d;
// len = 0
// [........]
//  ^
// l,r
```

Exercice 3. *Permutations triables par pile*

Machine à pile

Une *machine à pile* est un modèle de calcul primitif dont la *mémoire* est une pile et les *programmes* des suites d'opérations `Push` et `Pop` manipulant cette pile. L'exécution d'un programme p de machine à pile sur un *flux d'entrée is* se définit ainsi :

- on initialise la mémoire à la pile vide ;
- puis on lit successivement toutes les opérations de p, et :
  - si l'opération est `Push`, le programme échoue si le flux d'entrée est vide, et sinon lit une valeur depuis celui-ci et l'empile dans la mémoire ;
  - si l'opération est `Pop`, le programme échoue si la mémoire est vide, et sinon dépile la valeur en sommet de celle-ci et l'écrit sur le *flux de sortie os* ;

le résultat de l'exécution du programme est alors le flux de sortie obtenu après exécution de toutes les opérations de p (c'est à dire notamment que l'on ignore les éventuelles valeurs encore présentes dans le flux d'entrée et dans la mémoire de la machine à pile).

En toute généralité, une machine à pile peut effectuer d'autres opérations que celles que l'on vient de décrire ; notre modèle de machine à pile est *particulièrement* primitif.

Dans tout cet exercice, on représentera les programmes de machine à pile *via* les types OCaml suivants :

```
type stack_op = Push | Pop
type stack_program = stack_op list
```

Les flux d'entrée et de sortie seront quant à eux représentés par des valeurs de type `'a list`, ordonnées de la tête vers la queue ; c'est à dire que la première valeur à *lire* dans un flux d'entrée (non vide) représenté par une liste `x::xs` est sa tête `x`, et la première valeur à avoir été écrite dans un flux de sortie (non vide) représenté par une liste `x::xs` est également sa tête `x`.

Ainsi l'on a par exemple que le programme :

`[Push; Push; Pop; Push; Pop]`

exécuté sur le flux d'entrée :

`[2; 3; 5; 7; 11]`

produit le flux de sortie :

`[3; 5]`

1. Donnez le flux de sortie obtenu en exécutant le programme :
   `[Push; Push; Pop; Pop; Push]`
   sur le flux d'entrée :
   `[1; 2; 3; 4; 5]`

   On a :
   `[2; 1]`

2. Donnez un programme permettant d'obtenir le flux de sortie :
   `[2; 3; 5; 7]`
   à partir du flux d'entrée :
   `[1; 2; 3; 4; 5; 6; 7; 8; 9]`

   On propose :
   `[Push; Push; Pop; Push; Pop; Push; Push; Pop; Push; Push; Pop]`

Dans toute la suite de l'exercice, on s'autorise l'utilisation de la fonction `rev` du module `List`, qui est telle que `rev x` s'évalue en une nouvelle liste égale au miroir de `x`, et qui est de coût linéaire en la longueur de son argument.

3. Écrivez une fonction OCaml :
   `run_stack_program : 'a list -> stack_program -> 'a list`
   telle que `run_stack_program is sp` lève une exception en cas d'échec de l'exécution du programme `sp` sur le flux d'entrée `is`, et s'évalue sinon en le flux de sortie produit par celle-ci. On garantira pour cette fonction un coût linéaire en la longueur de son second argument, mais on ne demande pas de prouver ce coût.

   On propose :
```
let run_stack_program is sp =
    let rec _run is st out
        = function
        | [] -> out
        | Push::sp' -> (match is with
                        | [] -> failwith "empty input stream"
                        | iss::is' -> _run is' (iss::st) out sp')
        | Pop::sp'  -> (match st with
                        | [] -> failwith "empty memory"
                        | xst::st' -> _run is st' (xst::out) sp')
    in
    _run is [] [] sp |> List.rev (* équivalent à
    List.rev (_run is [] [] sp) >     *)
```

Permutations triables par pile

Soit $n \geq 1$ entier naturel, une *permutation* de $\{1,\ldots,n\}$ est une bijection de $\{1,\ldots,n\}$ dans lui-même. On représentera une telle permutation π en OCaml par la `int list` d'éléments $(\pi(1),\ldots,\pi(n))$. Par exemple, `[1; 2; 3]` et `[2; 3; 1]` représentent des permutations pour $n = 3$, mais `[1; 2; 3; 1]` ou `[1; 2; 4]` n'en représentent pour aucun n.

On s'intéresse dans cet exercice aux *permutations triables par pile* : soit un flux d'entrée *is* représentant une permutation, on cherche à déterminer s'il existe un programme de machine à pile qui quand exécuté sur ce flux produit le flux de sortie représentant $(1, \ldots, n)$.
Par exemple, la permutation représentée par `[1;3;2;4]` est triable par le programme de machine à pile : `[Push; Pop; Push; Push; Pop; Pop; Push; Pop]`

4. Donnez des programmes de machine à pile permettant de trier les permutations représentées par les flux d'entrée :

i. `[1; 2; 3; 4]`
ii. `[4; 3; 2; 1]`
iii. `[1; 2; 3; 5; 4; 6]`
iv. `[1; 6; 2; 4; 3; 5]`

Les programmes sont uniques, et donnés par :

i. `[Push; Pop; Push; Pop; Push; Pop; Push; Pop]`
ii. `[Push; Push; Push; Push; Pop; Pop; Pop; Pop]`
iii. `[Push; Pop; Push; Pop; Push; Pop; Push; Push; Pop; Pop; Push; Pop]`
iv. `[Push; Pop; Push; Push; Pop; Push; Push; Pop; Pop; Push; Pop; Pop]`

5. Montrez que la permutation représentée par `[2; 3; 1]` n'est pas triable par pile (c'est à dire qu'il n'existe aucun programme à pile qui produise le flux de sortie `[1; 2; 3]` quand exécuté sur le flux d'entrée `[2; 3; 1]`).

Pour produire un tel flux de sortie à partir d'un tel flux d'entrée, un programme à pile doit nécessairement commencer par `[Push; Push; Push; Pop]` ; celui-ci ne peut plus alors se compléter en un programme valide (n'échouant pas à l'exécution) et produisant un flux de sortie de longueur trois que d'une unique façon, comme `[Push; Push; Push; Pop; Pop; Pop]` ce qui donne le flux de sortie `[1; 3; 2]` non égal à `[1; 2; 3]`.

6. Écrivez une fonction :
`create_sp : int list -> stack_program`
qui lève une exception si son argument ne représente pas une permutation triable par pile, et sinon s'évalue en un programme de machine à pile triant celle-ci.
On garantira pour cette fonction un coût *linéaire* en la taille des entrées, mais l'on ne demande pas de montrer ce coût.

On propose :

```
let create_sp p =
    let rec _csp p sm st ne
        = match (p, st) with
        | [], [] -> sm
        | [], sts::st' -> if sts = ne
                          then _csp [] (Pop::sm) st' (ne + 1)
                          else failwith "not stack sortable"
        | ps::p', [] -> _csp p' (Push::sm) (ps::st) ne
        | ps::p', sts::st' -> if sts = ne
                              then _csp p (Pop::sm) st' (ne + 1)
                              else _csp p' (Push::sm) (ps::st) ne
    in
    _csp p [] [] 1 |> List.rev (* équivalent à
     List.rev (_csp p [] [] 1) *)
```

7. Montrez que toute permutation triable par pile peut être triée en temps linéaire en sa longueur.

Pour n'importe quelle permutation, il suffit d'utiliser `create_sp` pour déterminer en temps linéaire en sa longueur si elle est triable par pile. Si c'est le cas, cette fonction fournit un programme à pile de longueur (nécessairement) linéaire en le nombre d'éléments n de la permutation, et l'on peut alors exécuter ce programme sur la permutation avec `run_stack_program`, ce qui se fait en temps linéaire en n.

8. Déduisez de ce qui précède une fonction :

`stack_sort : int list -> int list`
qui trie son argument quand cela est possible, et lève une exception quand ce n'est pas le cas.

On propose :
```
let stack_sort p = create_sp p |> run_stack_program p (* équivalent à
let stack_sort p = run_stack_program p (create_sp p) *)
```

Dénombrement des permutations triables par pile

On dit d'un programme (de machine) à pile qu'il est *valide* si son exécution (sur une mémoire initialement vide) n'échoue pas sur un flux d'entrée infini (elle n'émet jamais de `Pop` sur une mémoire vide), et qu'il est *flush* si sa mémoire est vide lorsque son exécution est terminée.
On pourra remarquer que le programme vide `[]` est valide et flush.

9. Montrez que :

i. Si `p` est un programme à pile valide et flush alors `[Push] @ p @ [Pop]` est valide et flush.
ii. Si `p1` et `p2` sont deux programmes à pile valides et flush, alors `p1 @ p2` est valide et flush.

i. Soit `p' = [Push] @ p @ [Pop]` avec `p` valide et flush contenant n_p instructions, l'exécution de la première instruction de `p'` (qui n'échoue pas) dépile une valeur du flux d'entrée et l'empile dans la mémoire, qui contient alors une unique valeur `x`. Ensuite l'exécution des n_p instructions suivantes exécutent le programme `p'` ; par l'hypothèse que celui-ci est valide cette exécution ne produirait aucun `Pop` sur une mémoire vide si les instructions étaient exécutées à partir d'une mémoire vide, ce qui implique que la valeur empilée dans la mémoire par la première instruction n'est jamais dépilée ; par l'hypothèse qu'il est flush, cette exécution sur une mémoire initialement vide se terminerait sur une mémoire vide. Ces deux observations combinées impliquent que l'état de la mémoire à l'issue de l'exécution de `p` est exactement `[x]`, et donc exactement la mémoire vide après exécution de la dernière instruction `Pop` de `p'` (qui n'échoue pas). Il s'ensuit que `p'` est bien valide et flush.
ii. Soit `p = p1 @ p2` et n_{p1} (resp. n_{p2}) le nombre d'instructions de `p1` (resp. `p2`), l'exécution des n_{p1} premières instructions de `p` coïncide exactement avec l'exécution de `p1`. Par l'hypothèse que celui-ci est valide et flush, cette exécution n'échoue pas et termine sur une mémoire vide. L'exécution des n_{p2} dernières instructions de `p` coïncide alors exactement avec l'exécution de `p2`. Par l'hypothèse que celui-ci est valide et flush, et le fait qu'il s'exécute sur une mémoire vide, cette exécution n'échoue pas et se termine sur une mémoire vide. Il s'ensuit donc que `p1 @ p2` est valide et flush.

Dans la suite, on pourra admettre que tout programme à pile valide et flush peut se décomposer comme ci-dessus, c'est à dire s'écrire `[Push] @ p1 @ [Pop]` ou `p1 @ p2` pour des programmes à pile `p1` et `p2` (éventuellement vides) eux-mêmes valides et flush.

10. Montrez que pour tout programme à pile `p` valide et flush produisant un flux de sortie de longueur n, il existe une permutation `x` de n éléments telle que l'exécution de `p` trie `x`.
On pourra se contenter d'une argumentation relativement informelle.

Puisque `p` est valide et flush son exécution ne bloquera pas et la sortie produite contiendra bien n éléments qui étaient présents dans le flux d'entrée, c'est à dire une permutation de ceux-ci.
Cette permutation peut même être retrouvée efficacement, en simulant l'exécution de `p` sur un flux d'entrée indéterminé (constitué de « trous »), et écrivant la « bonne valeur » dans « le bon trou » lorsque l'indéterminée qui lui correspond est finalement « écrite » dans le flux de sortie par le programme.
On donne un exemple complet d'une telle fonction (qui utilise quelques fonctions de la bibliothèque standard OCaml qui ne sont pas explicitement au programme, et dont il faudrait prouver la correction), dans un style fonctionnel-impératif (mélangeant une structure récursive et l'usage de référence pour une gestion efficace des indéterminées (des « trous »)).
```
let vfsp2perm sp =
    let op = List.init ((List.length sp) / 2) (fun _ -> ref None) in
    let rec _v sp p st ne
        = match sp, p, st with
        | [], _, _ -> List.map (fun x -> Option.get !x) op
        | Push::sp', ps::p', _  -> _v sp' p' (ps::st) ne
        | Pop::sp', _, sts::st' -> sts := Some ne ; _v sp' p st' (ne + 1)
```

```
        | _ -> failwith "not valid and flush"
    in
    _v sp op [] 1
```

On peut remarquer que cette fonction est efficace : son coût est linéaire en la longueur de son entrée.

11. Montrez que deux programmes `p1` et `p2` distincts valides et flush produisant un flux de sortie de même longueur n trient des permutations distinctes.

Il suffit de considérer une exécution parallèle de `p1` et `p2` sur un même flux d'entrée représentant une permutation, et de constater qu'ils produisent des flux de sortie distincts (ce qui implique qu'au plus l'un de ces programmes trie la permutation en entrée, et donc qu'ils trient des permutations distinctes).
Par hypothèse `p1` et `p2` sont distincts ; on note `p` leur préfixe commun (nécessairement non vide, mais cela n'a pas d'importance), c'est à dire un programme à pile tel que (sans perte de généralité) l'on a `p1` égal à `p @ [Push] @ p1'` et `p2` égal à `p @ [Pop] @ p2'` pour certains suffixes `p1'` et `p2'`.
Soit n_p la longueur de `p`, l'exécution des n_p premières instructions de `p1` et `p2` produit un flux de sortie `os` et une mémoire `st` identiques pour les deux programmes ; de plus par l'hypothèse que `p1` et `p2` sont valides, et puisque l'instruction suivante de `p2` est `Pop`, l'on a que `st` est non vide et s'écrit donc `sts::st'` pour certains `sts`, `st'`.
Alors, soit `iss` la valeur en tête du flux d'entrée à ce moment, l'exécution de la n_p+1ème instruction :

— de `p1` produit le flux de sortie `os` et la mémoire `iss:sts::st'` ;
— de `p2` produit le flux de sortie `os @ [sts]` et la mémoire `st'` ;

et par la structure d'un programme à pile le flux de sortie produit par `p1` ne pourra pas avoir `os @ [sts]` comme préfixe puisque `sts` apparaît dans la mémoire de `p1` sans s'y trouver en tête, et n'apparaît pas ailleurs dans le flux d'entrée (ou la mémoire) puisque celui-ci représente une permutation.
On a donc bien que les flux de sortie produits par `p1` et `p2` sont distincts.

12. Déduisez de tout ce qui précède une expression récurrente pour compter le nombre C_n de permutations de $\{1,\dots,n\}$ triables par pile.
On pourra se contenter d'une argumentation relativement informelle.

Par les deux questions précédentes, il suffit de compter le nombre de programmes à pile valides et flush produisant des sorties de longueur n. On considère alors les deux écritures données par l'hypothèse suivant la question 9.
Il y a C_{n-1} programmes à pile valides et flush produisant des sorties de longueur n qui ne peuvent pas s'écrire comme la concaténation `p1 @ p2` de deux programmes valides et flush : ce sont exactement les programmes de la forme `[Push] @ p @ [Pop]` avec `p` valide et flush quelconque produisant des sorties de longueur $n-1$.
Les autres programmes s'écrivent alors comme la concaténation `p1 @ p2` de deux programmes non vides `p1` et `p2` valides et flush. Afin d'éviter de compter plusieurs fois un même programme, il est suffisant de considérer pour chacun de ces programmes une écriture où `p1` ne peut pas lui-même s'écrire comme une concaténation `p1p @ p1s` avec `p1p` et `p1s` non vides valides et flush (c'est à dire que l'on ne doit pas pouvoir compter un même programme une fois comme `p1p @ (p1s @ p2)` et une fois comme `(p1p @ p1s) @ p2` ; il n'y a cependant pas de telle restriction pour `p2`). Ainsi pour un programme s'écrivant comme `p1 @ p2` il y a C_{i-1} possibilités pour un programme `p1` valide et flush produisant une sortie de longueur i et ne pouvant pas s'écrire comme une concaténation non triviale (*cf.* ci-dessus), et C_{n-i} possibilités pour `p2` valide et flush produisant une sortie de longueur $n-i$ (sans autre contrainte).
Finalement, $C_0 = C_1 = 1$, et C_n est donné par la « convolution » :

$$C_n = C_{n-1} + \sum_{i=1}^{n-1} C_{i-1}C_{n-i} = \sum_{i=1}^{n} C_{i-1}C_{n-i}$$

Remarque : on peut prouver que $C_n = \frac{1}{n+1}\binom{2n}{n}$: c'est le $n$ième nombre « de Catalan ».