Exercice 1. Piles d'assiettes

On considère une pile d'assiettes dans laquelle chaque assiette est numérotée et possède une couleur (bleu ou rouge) :

```
type assiette =
  | Bleue of int
  | Rouge of int;;
```

```
(* Exemple 1 *)
Rouge 5
Bleue 3
Rouge 2
Bleue 10
Bleue 1
```

```
(* Exemple 2 *)
Bleue 3
Bleue 10
Bleue 1
Rouge 5
Rouge 2
```

1. Écrire une fonction récursive de type « `assiette Stack.t -> unit` » qui affiche une pile d'assiettes. On pourra par exemple obtenir l'affichage de l'exemple 1 où l'assiette du dessus est rouge et porte le numéro 5 et l'assiette du dessous est bleue et porte le numéro 1. Lorsque vous testez votre fonction, **essayez d'afficher la même pile deux fois de suite**.
2. Écrire une fonction de type « `assiette Stack.t -> unit` » qui modifie la pile en entrée en plaçant toutes les assiettes rouges sous les assiettes bleues sans modifier l'ordre des assiettes bleues ni l'ordre des assiettes rouges. Pour tester votre fonction : créez une pile, affichez la, appelez votre fonction, puis afficher de nouveau la pile. Par exemple, à partir de la pile de l'exemple 1, on obtient la pile de l'exemple 2.

Exercice 2. Notation polonaise

Habituellement, lorsqu'on manipule des expressions arithmétiques, on utilise la notation dite ***infixe***. Cela signifie que pour appliquer un opérateur à deux opérandes, on écrit le premier opérande, puis l'opérateur et enfin le second opérande. Par exemple, pour appliquer l'opérateur + aux opérandes 4 et 6, on écrit $4 + 6$.

Deux autres notations existent : la notation ***préfixe*** (dite aussi notation ***Polonaise***) et la notation ***postfixe*** (dite aussi notation ***polonaise inverse***). Pour la notation préfixe, on écrit l'opérateur puis les deux opérandes (par exemple : + 4 6) et pour la notation postfixe, on écrit les opérandes puis l'opérateur (par exemple : 4 6 +). Les notations préfixe et postfixe permettent de se passer de parenthèses. Voici deux exemples d'expressions écrites en notation préfixe, infixe et postfixe :

$\times\ +\ 1\ 23\ 456$,	$((1+23)\times 456)$,	$1\ 23\ +\ 456\ \times$,
$\times\ -\ 2\ 4\ +\ 5\ +\ 8\ 6$,	$((2-4)\times(5+(8+6)))$,	$2\ 4\ -\ 5\ 8\ 6\ +\ +\ \times$.

1. L'expression $((1+2)-(7\times 5))$ est écrite en notation infixe. Donner ses écritures en notation préfixe et postfixe.
2. L'expression $\times\ +\ 5\ 4\ \times\ 3\ +\ 2\ 1$ est écrite en notation préfixe. Donner ses écritures en notation infixe et postfixe.
3. L'expression $9\ 2\ \times\ 5\ 4\ +\ 7\ \times\ +$ est écrite en notation postfixe. Donner ses écritures en notation préfixe et infixe.

En OCaml, on représentera les expressions préfixées et postfixées par une liste de symboles ; un symbole étant soit un entier soit un opérateur sur les entiers.

```
type op =
  | Plus
  | Moins
  | Fois;;
```

```
type symb =
  | Int of int
  | Op  of op;;
```

```
type expr = symb list;;
type expr_pre = expr;;
type expr_post = expr;;
```

Les expressions infixées seront représentées par le type suivant :

```
type expr_inf =
  | Int_inf of int
  | Op_inf  of expr_inf * op * expr_inf;;
```

Affichage d'une expression

4. Écrire une fonction « `print_expr: expr -> unit` » qui affiche l'expression donnée en argument. Par exemple :

   « `print_expr [Op Fois; Op Plus; Int 1; Int 23; Int 456]` » affiche « `* + 1 23 456` »

5. Écrire une fonction « `print_inf: expr_inf -> unit` » qui affiche l'expression donnée en argument. Par exemple :

   « `print_inf (Op_inf (Int_inf 4, Plus, Int_inf 6))` » affiche « `(4+6)` »

Conversions entre notation préfixe, infixe et postfixe

6. À l'aide d'une pile, écrire la fonction « `inf_of_post: expr_post -> expr_inf` » pour convertir la notation postfixe en notation infixe.
7. Écrire les fonctions :

   `pre_of_inf: expr_inf -> expr_pre` `post_of_inf: expr_inf -> expr_post`

   pour convertir la notation infixe en notation préfixe ou postfixe.
8. Écrire la fonction « `inf_of_pre: expr_pre -> expr_inf` » pour convertir la notation préfixe en notation infixe.
9. À l'aide des fonctions précédentes, écrire les fonctions :

   `pre_of_post: expr_post -> expr_pre` `post_of_pre: expr_pre -> expr_post`

   pour faire les conversions entre notation préfixe et postfixe.

Évaluation d'une expression

10. Écrire une fonction « `int_of_inf: expr_inf -> int` » qui prend en entrée une expression sous forme infixe et renvoie l'entier en lequel s'évalue l'expression.
11. À l'aide des fonctions précédentes, écrire les fonctions :

    `int_of_pre: expr_pre -> int` `int_of_post: expr_post -> int`

    qui renvoient l'entier en lequel s'évalue l'expression donnée en entrée.

Exercice 3. Implémentation de dictionnaires (suite)

Dans cet exercice, on va implémenter la structure de dictionnaire à l'aide d'un tableau. Les dictionnaires contiendront des objets de type « `'a elem_dict` » composés d'une clé de type `int` et d'une valeur de type `'a` :

```
type 'a elem_dict = {cle: int; valeur: 'a};;
```

Implémentation 1. Dans cette implémentation, on va stocker les éléments du dictionnaire dans un tableau de taille m. Une case du tableau contiendra « `None` » ou bien « `Some e` » où « `e: 'a elem_dict` » :

```
type 'a dict2 = {
    tab: 'a elem_dict option array;
    mutable nb_elem: int;
  };;
```

Le champ `nb_elem` contient le nombre d'éléments dans le dictionnaire, il faut donc penser à le maintenir à jour. Les éléments du dictionnaire seront stockés dans les cases de `tab` indicées de `0` à `nb_elem - 1` ; les autres cases contiendront `None`.

1. Écrire une fonction « `create_d2: int -> 'a dict2` » qui prend en entrée m et renvoie un dictionnaire vide.

Pour ajouter un élément `e` au dictionnaire `d`, on commence par vérifier si la clé c de `e` apparaît dans `d`. Si c'est le cas, on remplace l'ancien élément de clé c par `e`. Dans le cas contraire, on stocke `e` dans la première case contenant `None`.

2. Écrire la fonction « `add_d2: 'a dict2 -> 'a elem_dict -> unit` ».
3. Écrire la fonction « `find_d2: 'a dict2 -> int -> 'a elem_dict` ».
   Consigne : il est interdit de faire du copier/coller à partir du code de la question précédente : créez des fonctions intermédiaires et utilisez les dans `add_d2` et `find_d2`.

Afin de supprimer un élément de clé c du dictionnaire, on repère l'indice i de cet élément dans le tableau, puis on décale tous les éléments d'indice $j > i$.

4. Écrire la fonction « `remove_d2: 'a dict2 -> int -> unit` ».
   Consigne : le copier/coller est interdit.
5. Évaluer les complexités de ces fonctions.

Implémentation 2. On souhaite implémenter la structure de dictionnaires à l'aide d'une ***table de hachage*** et gérer les collisions par ***adressage ouvert***. Les éléments du dictionnaire seront stockés dans un tableau de taille m en utilisant la fonction de hachage :

$$h : c \mapsto \left\lfloor m \times c\frac{\sqrt{5}-1}{2} \right\rfloor \mod m$$

Un élément de clé c est donc stocké dans la case d'indice $h(c)$ qui contient un seul élément (il n'y a pas de liste chaînée dans cette implémentation). La gestion de collisions sera faite par ***adressage ouvert***. C'est à dire que si la case d'indice $h(c)$ est déjà occupée, l'élément est stocké dans la première case inoccupée parmi les cases $(h(c)+1 \mod m)$, $(h(c)+2 \mod m)$ ... Dans le cas où le tableau est plein, un appel à la fonction `add` déclenche une erreur. De plus, si on ajoute un élément dont la clé appartient déjà au dictionnaire, l'élément déjà présent dans le tableau est remplacé. Enfin, on suppose qu'on ne peut pas supprimer d'élément du dictionnaire (en particulier, on n'implémentera pas de fonction `remove`).

6. Définir un type « `'a dict3` » et écrire les fonctions `create_d3`, `add_d3` et `find_d3`. La fonction « `create_d3 : int -> 'a dict3` » prendra en argument la taille du tableau à créer.
7. Évaluer les complexités de ces fonctions.

Exercice 4. Implémentation de files à l'aide de tableaux

Dans cet exercice, une file est représentée par un objet `f` de type « `'a file` ». L'entier `i0` est l'indice du premier élément arrivé dans `f` et `len` est le nombre d'éléments dans `f`. La file est pleine lorsqu'elle contient n éléments où n est la taille de `elem`.

```
type 'a file = {
    elem: 'a array;
    mutable i0: int;
    mutable len: int
  };;
```

Par exemple, lorsqu'on enfile `1`, `2`, `3`, `4` dans une file initialement vide, le résultat peut être représentée par `f1` ou par `f2` :

```
(* File ( 4 3 2 1 ) *)
let f1 = {
    elem = [|10;14;1;2;3;4;100;8|];
    i0 = 2;
    len = 4;};;
```

```
(* File ( 4 3 2 1 ) *)
let f2 = {
    elem = [|3;4;10;14;1;2|];
    i0 = 4;
    len = 4;};;
```

Dans `f1` : `elem.(0)`, `elem.(1)`, `elem.(6)` et `elem.(7)` sont en dehors de la file, et peuvent donc prendre des valeurs arbitraires. De plus, on considère que le tableau `elem` est circulaire, donc `f1` et `f2` représentent la même file.

1. Écrire une fonction « `is_empty: 'a file -> bool` » ainsi qu'une fonction « `is_full: 'a file -> bool` » qui indiquent si une file est vide ou pleine.
2. Écrire une fonction « `create: int -> 'a -> 'a file` » qui prend en entrée n ainsi qu'une variable « `x: 'a` », et renvoie une file vide de taille maximale n. La variable `x` sera utilisée pour initialiser les cases du tableau `elem`.
3. Écrire une fonction « `push: 'a -> 'a file -> unit` » qui ajoute un élément à la file.
4. Écrire une fonction « `pop: 'a file -> 'a` » qui supprime le premier élément de la file et renvoie cet élément.
5. Écrire une fonction « `print_int_file: int file -> unit` » qui affiche une file contenant des entiers. Par exemple, votre programme affichera « `( 4 10 0 6 )` » pour une file d'entiers dont le premier élément est `6` (c'est le premier élément à être entré) et dont le dernier élément est `4` (c'est le dernier élément à être entré).

Exercice 5. Parseur (exercice facultatif)

Le but de cet exercice est d'écrire un programme d'analyse syntaxique (souvent appelé ***parseur***), c'est à dire qu'on veut traduire une chaîne de caractères contenant une expression arithmétique, en données exploitables par OCaml. Dans la suite, on utilise les types définis dans l'exercice 2 (les fonctions de cet exercice peuvent notamment être utilisées pour faciliter les tests).

1. Écrire une fonction « `expr_of_string: string -> expr` » qui prend en entrée une chaîne de caractères, et renvoie l'expression infixe ou postfixe correspondante. Par exemple :

   « `expr_of_string "* + 1 23 456"` » vaut :
   `[Op Fois; Op Plus; Int 1; Int 23; Int 456]`

2. Écrire une fonction « `inf_of_string: string -> expr_inf` » qui prend en entrée une chaîne de caractères, et renvoie l'expression infixe correspondante. Par exemple :

   « `inf_of_string "((1 + 23) * 456)"` » vaut :
   `Op_inf (Op_inf (Int_inf 1, Plus, Int_inf 23), Fois, Int_inf 456)`