Le but du TP est de manipuler des piles et des files avec les modules `Stack` et `Queue` :

```
Stack.create: unit -> 'a Stack.t
Stack.is_empty: 'a Stack.t -> bool
Stack.push: 'a -> 'a Stack.t -> unit
Stack.pop: 'a Stack.t -> 'a
```

```
Queue.create: unit -> 'a Queue.t
Queue.is_empty: 'a Queue.t -> bool
Queue.push: 'a -> 'a Queue.t -> unit
Queue.pop: 'a Queue.t -> 'a
```

Exercice 1.

Récupérer le fichier annexe disponible sur la page du cours :

`https://informatique-lhp.fr/opt-mpsi.html`

Prévoir la réponse d'OCaml lors de l'évaluation de chaque ligne du fichier, puis le vérifier sur machine.

Exercice 2. Piles

1. Écrire une fonction de type « `'a array -> 'a Stack.t` » qui créé une pile contenant tous les éléments du tableau donné en entrée. Vérifier en particulier que l'élément se trouvant au sommet de la pile est le dernier élément du tableau.

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
```

2. 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.
3. 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 3. Nombres de Hamming

Un nombre de Hamming est un entier naturel de la forme $2^p 3^q 5^r$ où $p, q, r \in \mathbb{N}$. Soit $n \in \mathbb{N}$, on souhaite créer une liste L_n contenant les n premiers nombres de Hamming triés par ordre croissant. Par exemple L_{15} est la liste :

`[1; 2; 3; 4; 5; 6; 8; 9; 10; 12; 15; 16; 18; 20; 24]`

À l'aide de files, écrire une fonction de complexité $\mathcal{O}(n)$ qui prend en entrée n et renvoie L_n. Vérifier que le $15^{\text{ème}}$ nombre de Hamming est 24 et que le $1111^{\text{ème}}$ est 102 515 625. On pourra utiliser la fonction `Queue.peek` qui renvoie le premier élément d'une file (sans le supprimer de la file, contrairement à la fonction `Queue.pop`).

Exercice 4. 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 $4 + 6$.

Outre la notation infixe, il existe 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)` »

Évaluation d'une expression infixe

6. 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.

Conversions entre notation préfixe, infixe et postfixe

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 d'une pile, écrire la fonction « `inf_of_post: expr_post -> expr_inf` » pour convertir la notation postfixe en notation infixe.
10. 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 préfixe ou postfixe

11. 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 5. 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 `premier` est l'indice du premier élément arrivé dans `f` et `nb_elements` est le nombre d'éléments dans `f`. La file est pleine lorsqu'elle contient n éléments où n est la taille du tableau `elements`.

```
type 'a file = {
    elements: 'a array;
    mutable premier: int;
    mutable nb_elements: int
  };;
```

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

```
(* File ( 4 3 2 1 ) *)
let f1 = {
    elements = [|10;14;1;2;3;4;100;8|];
    premier = 2;
    nb_elements = 4;};;
```

```
(* File ( 4 3 2 1 ) *)
let f2 = {
    elements = [|3;4;10;14;1;2|];
    premier = 4;
    nb_elements = 4;};;
```

Dans `f1` : `elements.(0)`, `elements.(1)`, `elements.(6)` et `elements.(7)` sont en dehors de la file, et peuvent donc prendre des valeurs arbitraires. De plus, on considère que le tableau `elements` 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 `elements`.
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 pourra afficher « `( 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 6. 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 4 (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)`