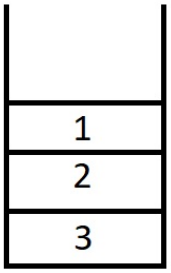
Chapitre 11 - TAD

1. Les piles

Nous allons implémenter le TAD Pile à l'aide du type produit suivant :

```
type pile = PileVide | Empile of int * pile  ; ;
```

1. Définir l'exception `PileVideErreur` qui sera utilisée lorsque les préconditions des opérateurs ne seront pas respectées.
2. Définir la pile $p1$ suivante :



3. Écrire une fonction `estVide : pile -> bool` qui renvoie *true* si la pile est vide.

```
estVide p1  ; ;
(*-  : bool = false*)
```

4. Écrire une fonction `sommet : pile -> int` qui retourne le sommet de la pile ou lève l'exception.

```
sommet p1  ; ;
(*-  : int = 1*)
```

5. Écrire une fonction `empile : int -> pile -> pile` qui empile un élément au sommet de la pile

```
let p2 = empile 0 p1  ; ;
(*p2  : pile = Empile (0, Empile (1, Empile (2, Empile (3, PileVide))))*)
```

6. Écrire une fonction `dépile : pile -> pile` qui élimine le sommet de la pile ou lève l'exception.

```
let p3 = dépile p2  ; ;
(*p3  : pile = Empile (1, Empile (2, Empile (3, PileVide)))*)
```

7. Écrire une fonction récursive `égalité : pile -> pile -> bool` qui teste l'égalité de deux piles.

```
égalité p1 p3 ;;
égalité p1 p2 ;;
(*#- : bool = true
#- : bool = false*)
```

Remarquez que l'on aurait pu simplement écrire `p1 = p3` ...

Expression bien parenthésée

Les expressions algébriques $(a+b)$ et $(ax(b+c))$ sont dites bien parenthésées, à l'inverse, $(a+b$ et $(a+b))$ ne le sont pas. L'objectif est d'écrire une fonction permettant de dire si une expression algébrique (de type `string`) est bien parenthésée ou non. Remarquez qu'il ne suffit pas de compter le nombre de parenthèses ouvrantes et fermantes : $)a+b($ n'est pas non plus bien parenthésée alors qu'elle comporte le même nombre de parenthèses ouvrantes et fermantes.
Nous avons vu l'an dernier, comment procéder à l'aide d'une pile : pour chaque caractère de l'expression, s'il s'agit d'une parenthèse ouvrant, on l'empile, s'il s'agit d'une parenthèse fermante on dépile. Si aucune erreur n'est levée et qu'à la fin de l'expression, la pile est vide alors celle-ci est bien parenthésée.

1. Ecrire une fonction `auxParentheses : string -> pile -> bool` qui, à partir d'une expression et d'une pile, détermine si celle-ci est bien parenthésée.
2. Ecrire une fonction d'appel `parenthese : string -> bool` qui lance la fonction précédente avec une pile vide. On pourra utiliser un *try...with* pour renvoyer $false$ lorsque la fonction précédente lève l'exception `PileVideErreur`.

```
parenthese "(a+b" ;;
parenthese "(a+b)" ;;
parenthese "(a x (b + c))" ;;
parenthese "(a+b))" ;;
parenthese ")a+b(" ;;
(*- : bool = false
#- : bool = true
#- : bool = true
#- : bool = false
- : bool = false*)
```

2. Les files

Le TAD File est caractérisé par le fait que les éléments sont enfilés d'un côté de la file et défilés de l'autre, les deux opérations devant être de complexité constante.
Comme nous ne disposons, en Caml, que de listes ne permettant qu'un accès rapide à la tête, nous allons utiliser deux listes, correspondant au début de la file (à l'endroit) et à la fin de la file (à l'envers).
Nous utiliserons alors le type suivant :

```
type file = {debut : int list ; fin : int list} ;;
```

Ainsi, la file [1;3;4;7;2] pourra être représentée par :

```
{Debut =[1] ; Fin : [2 ;7 ;4 ;3]} ou {Debut =[1 ;3 ;4] ; Fin : [2 ;7]}
ou {Debut =[1 ;3] ; Fin : [2 ;7 ;4]} ou encore {Debut =[] ; Fin : [2 ;7 ;4 ;3 ;1]
```

Nous définissons ainsi :

```
let f1 = {debut = [1 ;3] ; fin = [2 ;7 ;4]} ; ;
(*f1 : file = {debut = [1 ; 3] ; fin = [2 ; 7 ; 4]}*)

let f2 = {debut = [] ; fin = [3 ;2 ;1]} ; ;
(*f2 : file = {debut = [] ; fin = [3 ; 2 ; 1]}*)
```

pour nous servir d'exemples.

1. Définir l'exception `FileVideErreur` qui sera levée lorsque la précondition d'un opérateur ne sera pas vérifiée.
2. Écrire une fonction `estVide : file -> bool` qui détermine si une file est vide.
3. Écrire une fonction `transvase : file -> file` qui, alors que `debut` est vide (ce que l'on supposera sans chercher à le vérifier), transvase le contenu de la liste `fin` dans la liste `debut`.

   ```
   let f3 = transvase f2 ; ;
   (*f3 : file = {debut = [1 ; 2 ; 3] ; fin = []}*)
   ```

4. Écrire une fonction `premier : file -> int` qui retourne le premier élément de la file ou lève l'exception.

   ```
   premier f1 ; ;
   (*- : int = 1*)

   premier f2 ; ;
   (*- : int = 1*)
   ```

5. Écrire une fonction `enfiler : int -> file -> file` qui enfile un élément en fin de file.

   ```
   let f5 = enfiler 0 f2 ; ;
   (*f5 : file = {debut = [] ; fin = [0 ; 3 ; 2 ; 1]}*)
   ```

6. Écrire une fonction `queue : file -> file` qui élimine le premier élément de la file ou lève l'exception.

   ```
   queue : file -> file = <fun>
   (*- : file = {debut = [2 ; 3] ; fin = []}*)
   ```

7. Écrire une fonction `egale : fil -> file -> bool` qui teste l'égalité entre deux files.

```
egale f2 f3 ;;
egale f1 f3 ;;
(*- : bool = true
- : bool = false*)
```

3. Tas (extrait CC 2021)

Rappelons qu'un tas, est un arbre binaire satisfaisant (entre autres) la propriété suivante : la valeur de chaque nœud est supérieure à celles de ses fils.
Nous décidons de représenter les tas par le type :

```
type tas = Feuille of int | Noeud of int * tas * tas ;;
```

1. Écrire une fonction `valMax : tas -> int` qui a un tas, associe sa valeur maximale, c'est à dire, sa racine.

```
#valMax tas1 ;;
- : int = 18

#valMax tas2 ;;
- : int = 26
```

2. Écrire une fonction `verif : tas -> bool` qui vérifie si un arbre binaire satisfait bien la propriété *la valeur de chaque nœud est supérieure à celles de ses fils*.

```
#verif tas1 ;;
- : bool = true

#verif pastas ;;
- : bool = false
```

4. Multiensemble (extrait CC 2021)

Le multiensemble est un TAD qui généralise les ensembles : les répétitions sont autorisées, mais l'ordre des éléments n'a pas d'importance. Par exemple : $m_1 = \{'a';'a';'b';'c';'c';'c'\} = \{'c';'a';'c';'b';'c';'a';'b'\}$ est un multiensemble.
Dans cet exercice, on se limitera aux multiensembles de caractères. Nous les représenterons par une fonction qui à chaque caractère associe sa quantité :

```
let m1 = fun
| 'a' -> 2
| 'b' -> 1
| 'c' -> 3
| _ -> 0 ;;
```

1. Écrire une fonction `combien : ('a -> 'b) -> 'a -> 'b` qui nous dit combien de fois un caractère est présent dans le multiensemble.

```
combien m1 'a' ; ;
(*-  : int = 2*)
```

2. Écrire une fonction `ajoute : 'a -> ('a -> int) -> 'a -> int` qui ajoute une lettre dans un multiensemble.

```
let m2 = ajoute 'b' m1 ; ;
(*m2  : char -> int*)

#combien m2 'a' ; ;
-  : int = 2
#combien m2 'b' ; ;
-  : int = 2
#combien m2 'c' ; ;
-  : int = 3
```

3. Écrire une fonction `union : ('a -> int) -> ('a -> int) -> 'a -> int` de deux multiensembles. L'union fonctionne comme pour les ensembles : $\{`a`;`b`;`b`\} \cup \{`a`;`a`;`b`;`c`;`c`;`c`\} = \{`a`;`a`;`b`;`b`;`c`;`c`;`c`\}$ On donnera des exemples du bon fonctionnement de la fonction `union`.

5. Les tableaux dynamiques d'entiers

Les tableaux imposent un accès direct en temps constant à n'importe quel élément du tableau, quelque soit son indice. La liste n'est donc pas appropriée, aussi, nous allons définir les tableaux dynamiques à l'aide d'une implémentation très fonctionnelles : avec une fonction !
Ainsi, nous pouvons définir le tableau `[|7 ;4 ;3 ;2|]` par la fonction :

```
exception OutOfRange ; ;

let v1 = fun
0->7
|1->4
|2->3
|3->2
|_-> raise OutOfRange ; ;
```

Nous avons alors un accès direct à la case d'indice i par l'appel de fonction `v1(i)` ou `v1 i` (s'il n'y a pas ambiguïté).

1. Écrire une fonction `affecte : ('a -> 'b) -> 'a * 'b -> 'a -> 'b` qui dans un tableau v, remplace la valeur de la case d'indice i par l'entier j.

```
let v2 = affecte v1 (1,5) ; ;
(v1 1, v2 1) ; ;
(v1 3, v2 3) ; ;
(*#v2  : int -> int = <fun>
#-  : int * int = 4, 5
```

```
#-   : int * int = 2, 2*)
```

2. Écrire une fonction `supprime : (int -> 'a) -> int -> int -> 'a` qui supprime le contenu de la case i et décale donc toute la fin du tableau d'un rang sur la gauche.

```
let v3 = supprime v1 1  ; ;
(v1 0, v3 0)  ; ;
(v1 1, v3 1)  ; ;
(v1 2, v3 2)  ; ;
(v1 3, v3 3)  ; ;

(*#v3   : int -> int = <fun>
#-   : int * int = 7, 7
#-   : int * int = 4, 3
#-   : int * int = 3, 2
#Exception non rattrapée : OutOfRange*)
```

3. Écrire une fonction `insere : (int -> 'a) -> int * 'a -> int -> 'a` qui insère la valeur j dans la case d'indice i en décalant la fin du tableau d'un cran sur la droite.

```
let v4 = insere v3 (1,4)  ; ;
(v1 0, v4 0) ; ;
(v1 1, v4 1) ; ;
(v1 2, v4 2) ; ;
(v1 3, v4 3) ; ;
(*#v4   : int -> int = <fun>
#-   : int * int = 7, 7
#-   : int * int = 4, 4
#-   : int * int = 3, 3
#-   : int * int = 2, 2*)
```

Les tableaux $v1$ et $v4$ sont normalement égaux, mais comment le vérifier ? Avec notre type actuel, nous sommes un peu embêtés (pourquoi ?).

Amélioration

Pour améliorer le type ci-dessus et permettre (notamment) de tester l'égalité de deux tableaux, nous définissons le type `tabDyn`.

```
type tabDyn = {taille   : int  ; valeurs   : int -> int } ; ;
```

Le tableau $v1$ devient donc :

```
let d1 = {taille = 4  ; valeurs = v1}  ; ;
(*d1   : tabDyn = {taille = 4 ; valeurs = <fun>}*)
```

1. Écrire une fonction `affecteDyn : tabDyn -> int * int -> tabDyn` qui remplace le contenu d'une case d'indice i par la valeur j dans un tableau d.

```
let d2 = affecteDyn d1 (1,5) ;;
d2.valeurs 1 ;;
(*d2 : tabDyn = {taille = 4 ; valeurs = <fun>}
#- : int = 5*)
```

2. Écrire une fonction `supprimeDyn : tabDyn -> int -> tabDyn` qui supprime le contenu de la case d'indice i et décale le contenu du reste du tableau.

```
let d3 = supprimeDyn d1 1 ;;
(*d3 : tabDyn = {taille = 3 ; valeurs = <fun>}*)
```

3. Écrire une fonction `insereDyn : tabDyn -> int * int -> tabDyn` qui insère la valeur j dans la case d'indice i et décale le contenu du reste du tableau.

```
let d4 = insereDyn d3 (1,4) ;;
(*d4 : tabDyn = {taille = 4 ; valeurs = <fun>}*)
```

4. Écrire une fonction `egalite : tabDyn -> tabDyn -> bool` qui teste l'égalité de deux tableaux dynamiques. *On pourra définir localement une fonction auxiliaire récursive.*

```
egalite d1 d4 ;;
(*- : bool = true*)
```

5. Écrire une fonction `indice : 'a list -> int -> 'a` qui renvoie l'élément d'indice i dans une liste ℓ.
6. Écrire une fonction `longueur : 'a list -> int` qui renvoie la longueur d'une liste.
7. Déduire des deux fonctions précédentes, une fonction `listeVersTableau : int list -> tabDyn` qui construit un tableau dynamique à partir d'une liste passée en paramètre.
8. Inversement, écrire une fonction `tableauVersListe : tabDyn -> int list` qui convertit un tableau en une liste.

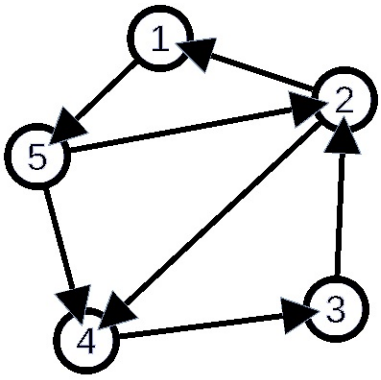
```
let listeTest = [1 ;4 ;7 ;8 ;10] ;;
tableauVersListe (listeVersTableau listeTest) ;;

(*listeTest : int list = [1 ; 4 ; 7 ; 8 ; 10]
- : int list = [1 ; 4 ; 7 ; 8 ; 10]*)
```

6. Les graphes

Nous allons pour terminer, implémenter les graphes (orientés, non valués) à l'aide des listes d'adjacences. Pour cela, nous définissons le type suivant :

```
type graphe = {sommets : int ; adjacences : int -> int list} ;;
```

1. Définir une exception `PasUnSommet`.
2. Définir le graphe $g1$ suivant :



3. Ecrire une fonction `longueur : 'a list -> int` qui détermine la longueur d'une liste.
4. En déduire une fonction `nombreVoisins : graphe -> int -> int` qui détermine le nombre de successeurs d'un sommet s dans un graphe G.

```
nombreVoisins g1 5 ;;
(*- : int = 2*)
```

5. Ecrire une fonction `nombreArcs : graphe -> int` qui renvoie le nombre d'arc d'un graphe orienté G.

```
(*nombreArcs g1 ;;
- : int = 7*)
```

6. Écrire une fonction `initGraphe : int -> graphe` qui crée le graphe à n sommets et sans arc.

```
let g2 = initGraphe 3 ;;
(*g2 : graphe = {sommets = 3 ; adjacences = <fun>}*)
g2.sommets ;;
(*- : int = 3*)
g2.adjacences 1 ;;
(*- : int list = []*)
```

7. Écrire une fonction `ajoutSommet : graphe -> graphe` qui ajoute un $(n+1)$-ème sommet sans arc à un graphe.

```
let g1b = ajoutSommet g1 ;;
(*g1b : graphe = {sommets = 6 ; adjacences = <fun>} *)
g1b.adjacences 1 ;;
(*- : int list = [5]*)
g1b.adjacences 6 ;;
(*- : int list = []*)
```

8. Écrire une fonction `ajoutListe : 'a -> 'a list -> 'a list` qui ajoute un élément x à une liste ℓ **seulement si** celui-ci n'est pas déjà présent dans la liste.
9. En déduire une fonction `ajoutArc : graphe -> int * int -> graphe` qui ajoute un arc (d,f) dans un graphe G seulement si celui-ci n'est pas déjà présent.

```
let g1c = ajoutArc g1b (6,1) ; ;
g1c.adjacences 6  ; ;
g1c.adjacences 1  ; ;
(*g1c  : graphe = {sommets = 6 ; adjacences = <fun>}
#-  : int list = [1]
#-  : int list = [5]*)
```

10. En déduire une fonction `arcToG : int -> (int * int) list -> graphe` qui crée un graphe à partir du nombre de sommets n et de sa liste d'arcs.

```
let listeArc = [(1,5) ;(2,1) ;(2,4) ;(3,2) ;(4,3) ;(5,2) ;(5,4)] ; ;

let g4 = arcToG 5 listeArc  ; ;
(*g4  : graphe = {sommets = 5 ; adjacences = <fun>}*)

g4.adjacences 1  ; ;
(*-  : int list = [5]*)
g4.adjacences 2  ; ;
(*-  : int list = [4 ; 1]*)
```

Parcours en profondeur simple

1. Ecrire une fonction `appartient : 'a -> 'a list -> bool` qui teste l'appartenance d'un élément x dans une liste ℓ.
2. Écrire une fonction `auxProfond : graphe -> int -> int list -> int list` qui prend en paramètre un graphe G, le sommet i en cours de traitement dans le parcours et $listeVisite$ la liste des sommets visités et qui fait progresser le parcours. *On écrira une fonction auxiliaire définie localement, travaillant sur la liste d'adjacence du sommet i.*
3. Ecrire la fonction d'appel `profond : graphe -> int -> int list` qui démarre le parcours en profondeur à partir d'un graphe G et d'un sommet i.

```
profond g1 1  ; ;
(*-  : int list = [1 ; 5 ; 2 ; 4 ; 3]*)
profond g1c 1  ; ;
(*-  : int list = [1 ; 5 ; 2 ; 4 ; 3]*)
profond g1c 6  ; ;
(*-  : int list = [6 ; 1 ; 5 ; 2 ; 4 ; 3]*)
```

Parcours en largeur - Version 1

1. Écrire une fonction `largeur : graphe -> int -> int list` qui effectue un parcours en largeur d'un graphe G à partir d'un sommet i comme vu en classe. On définira localement deux fonction auxiliaire :
   - *ajout $l1$ f lv* qui ajoute les sommets de $l1$ non visités (la liste des sommets visités est lv) dans la file f.
   - *auxLargeur lv f* qui effectue le parcours en largeur à partir de la file f en faisant progresser la liste des sommets visités lv.

```
largeur g1c 1  ; ;
largeur g1c 6  ; ;
(*-  : int list = [1 ; 5 ; 2 ; 4 ; 3]
-  : int list = [6 ; 1 ; 5 ; 2 ; 4 ; 3]*)
```

Parcours en largeur - Version 2

Dans cette seconde version, nous ne nous soucions plus de l'ordre de visite (nous voulons seulement connaître les sommets accessibles). Cette méthode, qui a l'avantage d'être plus simple que la précédente, sera fréquemment employée au prochain semestre !

1. Écrire une fonction `union : 'a list -> 'a list -> 'a list` qui renvoie l'union de deux listes.
2. Écrire une fonction `uneEtape : graphe -> int -> int list` qui a un graphe G et un sommet i, renvoie la liste sans répétition formée du sommet i et de ses voisins (autrement dit, cette fonction renvoie tous les sommets à une distance inférieure ou égale à 1 du sommet i).

   ```
   uneEtape g1 1  ; ;
   (*-  : int list = [5 ; 1]*)
   ```

3. Étendre la fonction précédente en une fonction `listeEtape : graphe -> int list -> int list`, qui pour un graphe G et une liste de sommets ℓ renvoie la liste sans répétition formée des sommets de ℓ et de leurs successeurs.

   ```
   listeEtape g1 [2 ;5]  ; ;
   (*-  : int list = [5 ; 1 ; 4 ; 2]*)
   ```

4. En déduire une fonction `largeurAux : graphe -> int list -> int list` qui effectue un parcours en largeur dans un graphe G à partir d'une liste de sommets ℓ. On prendra le temps de réfléchir à la condition d'arrêt de cette fonction.
5. En déduire une fonction d'appel `largeur2 : graphe -> int -> int list` qui effectue un parcours en largeur à partir du sommet i dans un graphe G.