I Structure de données

I.1 Définition

Définition A

Une **structure de données** est la description d'un ensemble organisé d'objets, et des opérations qui lui sont associées. On distingue :

- Le **type de données abstrait**, qui correspond à la description mathématique de la structure de données et de ses opérations. Il ne dépend pas du langage de programmation utilisé.
- L'**implémentation**, qui correspond à sa réalisation technique, et dépend du langage de programmation.

Les opérations usuelles des structures de données sont :

- Création
- Consultation
- Modification (ajout/suppression)

Remarque B : Caractéristiques des structures de données

Lors de la création d'une structure de données, on prend en compte différents éléments :

- L'espace mémoire utilisé :
  - Si l'espace alloué est de taille fixe, on dit que la structure de donnée est **statique**.
  - Sinon, on dit qu'elle est **dynamique**.
- La complexité temporelle des différentes opérations,
- les opérations de modifications autorisées : si le contenu est modifiable, la structure est dite **mutable**.

Une structure statique et non mutable est dite **persistante** : on doit créer une nouvelle instance de la structure si on souhaite la modifier. Cela offre notamment la possibilité de garder en mémoire les différentes modifications effectuées. Dans le cas contraire, on parle de structure **impérative**.

Exemple C

En Python :

- La structure de `list` est dynamique et mutable :

```
>>> l = ['a', 1, 2.5]
>>> l.append((1,2))
>>> l[1] = 6
>>> l
['a', 6, 2.5, (1, 2)]
```

- Le type `str` est statique et non mutable :

```
>>> s = "Bonjour"
>>> s[3]
'j'
>>> s[3] = 'd'
TypeError: 'str' object does not support item assignment
>>> s.append('a')
AttributeError: 'str' object has no attribute 'append'
```

Exercice 1

Parmi les types OCaml suivants, déterminer lesquels sont mutables et/ou dynamiques :

- `array`
- `string`
- `char * char`
- `int * int ref`
- `int array ref`

I.2 Création de type en Caml

Pour implémenter des structures de données, l'utilisateur OCaml peut choisir de créer de nouveaux types.

La syntaxe générale est `type` *nomdutype* = *expression*. Le nom du type doit toujours commencer par une lettre minuscule. L'expression peut être de la forme :

- Un type. C'est ce qu'on appelle un alias (on renomme un type déjà existant).

```
type tableaubooleen = bool array;;
```

- Un nom seul, commençant par une capitale. Cela s'apparente à une constante.

```
type constante = Const;;
```

- Un nom, commençant par une capitale, avec en paramètre un type.

```
type proba = Prob of float;;
```

- Un type somme, constitué de plusieurs expressions séparées par le caractère `|`. Le type somme correspond au OU LOGIQUE.

```
type reel =
   | Zero
   | Entier of int
   | Rationnel of int * int
   | Irrationnel of float;;
```

⚠ La définition d'un type et son utilisation ont des syntaxes différentes. Les types sommes sont souvent traités par du filtrage :

```
# let signe = function
     | Zero -> 0
     | Entier n when n > 0 -> 1
     | Rationnel (p, q) when p * q > 0 -> 1
     | Irrationnel r when r > 0. -> 1
     | _ -> -1;;
val signe : reel -> int = <fun>
```

Ci-dessus, le mot-clé `of` n'apparaît pas.

- Un type produit, correspondant à une liste de caractéristiques du type créé. La syntaxe est `{ carac1 : type1 ; carac2 : type2 ; ... ; caracN : typeN }`. Les noms de caractéristiques doivent commencer par des minuscules (comme les noms de type). Le type produit correspond au ET LOGIQUE.

```
# type complexe = {norme : float ; argument : float};;
# let c = {norme = 3. ; argument = 0.785};;
val c : complexe = {norme = 3.; argument = 0.785}
# c.argument;;
- : float = 0.785
```

Une caractéristique peut être rendue mutable dans un type produit. Cela autorise sa modification une fois l'objet créé :

```
# c.norme <- 5.;;
Characters 1-14:
  c.norme <- 5.;;
  ^^^^^^^^^^^^^
Error: The record field norme is not mutable
# type complexe = {mutable norme : float ; argument : float};;
# let c = {norme = 3. ; argument = 0.785};;
# c.norme <- 5.;;
- : unit = ()
```

Exercice 2

Les entiers naturels se définissent dans l'arithmétique de Peano de la façon suivante :

- Zéro est un entier naturel.
- Si n est un entier naturel, alors son successeur $S(n)$ est un entier naturel.

Ainsi, le nombre $S(S(S(S(0))))$ est un entier naturel (correspondant à 4).

1. Écrire un type `nat` correspondant aux entiers naturels de Peano.
2. Écrire une fonction `nul : nat -> bool` qui teste si un entier est nul ou non.
3. Écrire une fonction récursive `entier : nat -> int` qui calcule la valeur entière d'un entier naturel de Peano. Écrire sa fonction réciproque `peano : int -> nat`.
4. Écrire une fonction récursive `addition : nat -> nat -> nat` qui calcule la somme de deux entiers naturels de Peano. Faire de même pour une fonction `multiplication`.
5. Écrire une fonction `plus_grand : nat -> nat -> bool` qui compare deux entiers naturels.

II Listes chaînées

II.1 Type de données abstrait

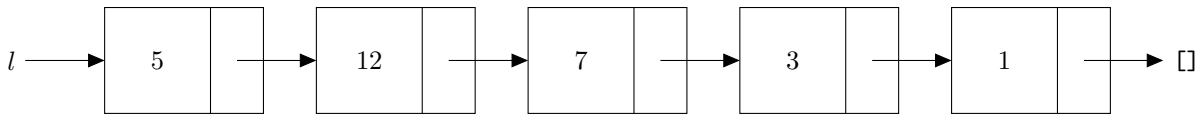
Définition D

Une **liste chaînée** ℓ est :

- Soit la liste `Vide`, notée généralement `[]` (qui a l'avantage de correspondre à la syntaxe Python et OCaml).
- Soit un maillon reliant un élément x et une autre liste chaînée q. On notera alors $\ell = M(x, q)$.

Dans le deuxième point, on appellera x la **tête** de ℓ et q la **queue** de ℓ.

Schématiquement, on représente une liste chaînée de la façon suivante :



Ci-dessus, la représentation de $M(5, M(12, M(7, M(3, M(1, []))))))$.

Une liste chaînée peut être arbitrairement grande, mais on ne peut accéder qu'à son premier élément. La consultation d'un élément quelconque se fait donc avec une complexité temporelle $O(n)$ où n est la taille de la liste. En terme de capacité mémoire, une liste ne stocke pas dans son espace mémoire l'intégralité de ses éléments, mais uniquement l'élément de tête et un **pointeur** vers la queue, autrement dit l'adresse mémoire où trouver la queue.

II.2 Implémentation Caml

En OCaml, `[]` représente la liste vide. La construction d'un maillon se fait à l'aide de l'opérateur `::` (prononcé « Cons », comme constructeur). On peut créer directement une liste avec plusieurs éléments :

```
# let vide = [];;
val vide : 'a list = []
# let l = [1; 2; 3];;
val l : int list = [1; 2; 3]
# let m = 4 :: l;;
val m : int list = [4; 1; 2; 3]
```

On constate ci-dessus qu'après avoir ajouté un élément, la liste a un type dépendant du type de l'élément ajouté. Comme pour les tableaux, les listes ne doivent contenir que des éléments du même type. Seule la liste vide est polymorphe.

Pour « explorer » une liste, on dispose des fonctions `List.hd` et `List.tl` renvoyant respectivement la tête et la queue de la liste :

```
# List.hd m;;
- : int = 4
# List.tl m;;
- : int list = [1; 2; 3]
```

Toutefois, on préfèrera généralement traiter les listes par du filtrage plutôt qu'en utilisant ces fonctions. Voici par exemple un moyen de recoder ces fonctions :

Exemple E

```
let tete = function
    | [] -> failwith "Liste vide"
    | x :: q -> x;;

let queue = function
    | [] -> failwith "Liste vide"
    | x :: q -> q;;
```

Le `failwith` permettant de renvoyer un message d'erreur le cas échéant.

Il est à noter que `::`, `List.hd` et `List.tl` sont des fonctions en temps constant.

Exercice 3

Écrire une fonction `deuxieme` qui renvoie le deuxième élément d'une liste, s'il existe, et renvoie un message d'erreur sinon.

II.3 Utilisation de la récursivité

De par sa définition inductive, la structure de liste est adaptée à la programmation récursive. Par exemple, la fonction suivante calcule la longueur d'une liste :

```
let rec longueur = function
    | [] -> 0
    | x :: q -> 1 + longueur q;;
```

Il existe de nombreuses fonctions récursives déjà implémentées en OCaml. Elle seront reprogrammées en TP.

- `List.length : 'a list -> int` : renvoie le nombre d'éléments dans une liste.
- `List.mem : 'a -> 'a list -> bool` : teste l'appartenance d'un élément à une liste.

– `List.append : 'a list -> 'a list -> 'a list` : concatène deux listes. L'opérateur `@` est un raccourci pour la concaténation.

```
# [1; 2; 3] @ [4; 5];;
- : int list = [1; 2; 3; 4; 5]
```

II.4 Fonctionnelles sur les listes

Certaines fonctions agissent sur tous les éléments d'une liste. C'est ce qu'on appelle des **fonctionnelles**.

– La fonction `List.map : ('a -> 'b) -> 'a list -> 'b list` prend en argument une fonction $f : A \longrightarrow B$, une liste $[a_0, a_1, \dots, a_{n-1}] \in A^n$ et renvoie la liste $[f(a_0), f(a_1), \dots, f(a_{n-1})]$.

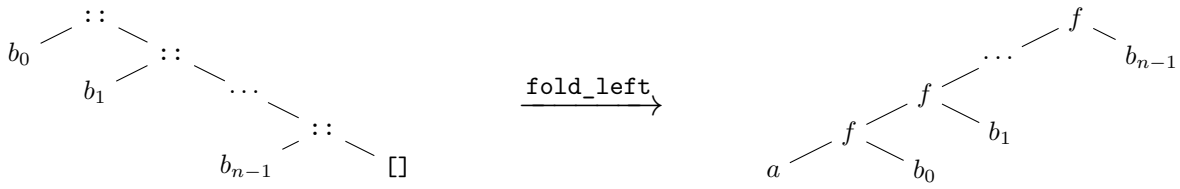
```
# let l = ["ab"; "cde"; "efgh"];;
val l : string list = ["ab"; "cde"; "fghi"]
# List.map String.length l;;
- : int list = [2; 3; 4]
```

– La fonction `List.iter : ('a -> unit) -> 'a list -> unit` prend en argument une fonction f qui renvoie un type `unit`, c'est-à-dire qui modifie l'environnement, et l'applique sur chaque élément de la liste. Cela revient à effectuer : `begin f a1; f a2; ...; f an end`

```
# List.iter print_string l;;
abcdefghi- : unit = ()
```

– La fonction `List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` prend en argument une fonction $f : A \times B \longrightarrow B$, une liste $[a_0, a_1, \dots, a_{n-1}] \in A^n$, un élément $b \in B$ et calcule $f\Big(a_0, f\big(a_1, \dots, f(a_{n-1}, b)\dots\big)\Big)$.
Schématiquement, cela ressemble à :

– La fonction `List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` prend en argument une fonction $f : A \times B \longrightarrow A$, un élément $a \in A$, une liste $[b_0, b_1, \dots, b_{n-1}] \in A^n$ et calcule $f\Big(\dots f\big(f(a, b_0), b_1\big)\dots, b_{n-1}\Big)$. Schématiquement, cela ressemble à :



Exercice 4

En utilisant les fonctions `List.fold_right` ou `List.fold_left`, réécrire les fonctions `List.length`, `List.mem`, `List.map` et `List.iter`.

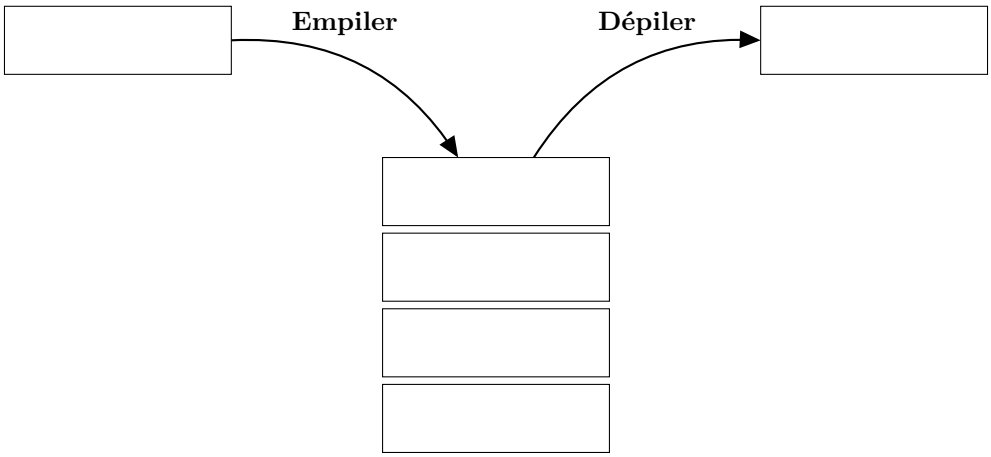
III D'autres structures linéaires

Les piles et les files sont des structures linéaires qui se différencie par leurs fonctions d'ajout et de suppression. Pour chacune de ces structures, on veut pouvoir :

- Créer une structure vide,
- ajouter un élément, ou supprimer un élément et récupérer sa valeur,
- tester si la structure est vide.

III.1 Piles

Une pile (*Stack* en anglais) est une structure dynamique basée sur le principe LIFO (Last In, First Out). C'est le principe d'une pile d'assiette : la dernière assiette lavée et posée sur la pile sera la première à être sortie pour le prochain repas.



Le module `Stack` existe déjà en OCaml, mais nous allons utiliser un type mutable pour redéfinir les piles. Les piles sont particulièrement bien représentées par les listes chaînées, car les éléments sont insérés et enlevés toujours au même endroit.

```
type 'a pile = {mutable liste : 'a list};;
```

Exercice 5

Écrire les fonctions suivantes :

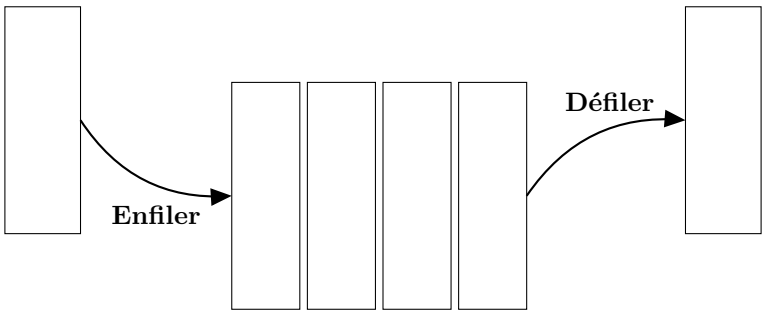
- `creer_pile : unit -> 'a pile` qui crée une pile vide.
- `pile_vide : 'a pile -> bool` qui teste si une pile est vide.
- `empiler : 'a pile -> 'a -> unit` qui empile un élément.
- `sommet : 'a pile -> 'a` qui renvoie la valeur du sommet de la pile (sans l'enlever).
- `depiler : 'a pile -> 'a` qui renvoie la valeur du sommet de la pile et le dépile.

Les deux dernières fonctions renverront un message d'erreur si la pile est vide.

Quelle est la complexité temporelle de ces fonctions ?

III.2 Files

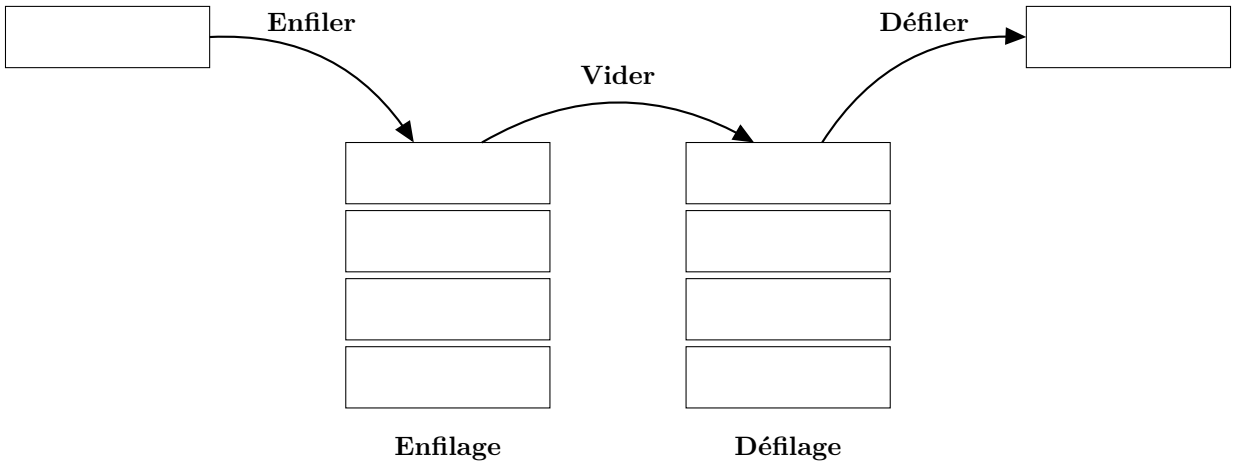
Une file (*queue* en anglais) est une structure basée sur le principe FIFO (First In, First Out), comme dans une file d'attente.



On peut envisager d'utiliser à nouveau une liste pour représenter la file. Cependant, il ne sera pas possible d'obtenir une complexité en $O(1)$ pour l'enfilage et pour le défilage, car atteindre la fin d'une liste a une complexité linéaire en sa taille.

Pour contourner ce problème, nous utiliserons deux piles :

- Une pile d'enfilage, dans laquelle chaque élément enfilé est rajouté,
- une pile de défilage, dans laquelle le premier élément est celui qui a été rajouté à la file avant tous les autres.



L'opération de vidage doit se faire à chaque fois que la pile de défilage est vide et que l'utilisateur souhaite défiler un élément, ou consulter le dernier élément. Dans ce cas, on dépile les éléments un par un de la pile d'enfilage, et on les empile dans la pile de défilage. Cela permet de « retourner » la pile, et donc d'assurer que le premier arrivé est bien le premier à sortir.

Exercice 6

1. Créer le type `'a file` correspondant à cette structure.
2. Écrire les fonctions équivalentes à celles sur les piles, c'est-à-dire :
   - `creer_file : unit -> 'a file`
   - `file_vide : 'a file -> bool`
   - `enfiler : 'a file -> 'a -> unit`
   - `premier : 'a file -> 'a` et `dernier : 'a file -> 'a`
   - `defiler : 'a pile -> 'a`
3. Quelle est la complexité des 4 premières fonctions ?
4. Les deux dernières fonctions sont-elles toujours de complexité constante ? Quelle est leur complexité moyenne ?