Les structures de données

MPSI - Prytanée National Militaire

Pascal Delahaye

9 mai 2019

1 Création de nouveaux TYPES sous OCaml

Caml propose une série de types prédéfinis : `bool`, `int`, `float`, `string`, `char`, `'a array`, `'a list`... Mais ces différents types sont parfois inadaptés aux objets que l'on peut être amené à manipuler en informatique.

Or, la programmation de type "objet" repose sur le principe que les objets manipulés par les algorithmes puissent être représentés informatiquement. La plupart des langages informatiques proposent donc à leurs utilisateurs la possibilité de modéliser ces types d'objets en définissant une structure de données adaptée.
Ainsi :

1. Un point du plan pourra être représenté par ses coordonnées du type : `point = float * float`
2. Un cercle pourra être représenter par un couple (point, rayon) du type : `cercle = point * float`
3. Les entiers et les flottants pourront être rassemblés sous le type `nombre`
4. Un arbre généalogique pourra être avantageuseument réprésenté par un objet de type `arbre`

Les différents types possibles se classent en deux catégories : les types *somme* et les types *produit ou enregistrements*.

⚠ Ocaml impose que { les noms des types / les noms des variables } commencent tous par une MINUSCULE.

1.1 Les types somme

On dit qu'un type de données est un *type somme* lorsqu'il réunit des éléments et/ou des types différents.

1.1.1 Cas d'un ensemble fini de valeurs

Lorsqu'on souhaite typer un ensemble fini d'éléments, on utilise la syntaxe suivante :

OCaml
```
type nouveau_type =
         | Elem_1
         | ...
         | Elem_n ;;   (* Majuscule en première lettre *)
```

Remarque 1. Les différents éléments réunis dans ce nouveau type ne sont pas d'un type prédéfinis par OCaml. Il s'agit en fait de simples noms.

`Exemple 1.` Création des types "cartes" et "fleur" :

OCaml
```
type carte =
    | As
    | Roi
    |...
    | Sept ;;

type fleur =
    | Rose
    | Tulipe
    | Lys;;

let couleur = function
                | Rose -> "rose"
                | Tulipe -> "rouge"
                | Lys -> "blanc";;
```

Le programme précédent montre que le type `fleur` *est bien reconnu par OCaml comme un type à part entière.*

1.1.2 Cas d'une réunion de types

La définition sous CAML d'un tel type utilise la syntaxe suivante :

OCaml
```
type nouveau_type = | Nom_1 of type_1      (* Noter l'utilisation obligatoire de la majuscule *)
                    | Nom_2 of type_2
                    | ...
                    | Nom_n of type_n  ;;
```

Remarque 2. ⚠ `Nom_1`, ..., `Nom_n` sont alors des fonctions qui convertissent les types `type_1`, ..., `type_n` en `nouveau_type`.

Pour créer des données ayant le type ainsi défini, on procède donc ainsi :

OCaml
```
let a = Nom_1 valeur ;;
let b = Nom_2 valeur ;;   (* a et b sont alors deux variables de type "nouveau_type" *)
```

`Exemple 2.` Ainsi, on peut créer un type `nombre` qui regroupe les entiers et les flottants :

OCaml
```
type nombre = | Entier of int
              | Reel   of float ;;

let n  = Entier 1 ;;
let x  = Reel 2.5 ;;

let add = function
        | (Entier n),(Entier m) -> Entier (n + m)
        | (Entier n),(Reel x) -> Reel (float_of_int n +. x)
        | (Reel x), (Entier n) -> Reel (float_of_int n +. x)
        | (Reel x), (Reel y) -> Reel (x +. y);;
```

Remarque 3. Nous utiliserons cette structure de *type somme* pour définir plus tard le type récursif *arbre binaire*.

OCaml
```
type organigramme = | F of string
                    | N of organigramme*string*organigramme ;;
```

1.2 Les types produit (ou "enregistrement")

Un *type produit* est un type de données défini par le produit cartésien de types existants.

1.2.1 Cas du renommage d'un n_uplet :

Pour la création de ce nouveau type, on utilise la syntaxe suivante :

OCaml
```
type nouveau_type = type_1 * ... * type_n;;    (* Définition d'un type produit sans étiquette *)
```

`Exemple 3.` Les points du plan sont donnés sous la forme de couples de $\mathbb{R}^2$.
Le type `float*float` n'étant pas particulièrement explicit, il pourra être utile de renommer ce nouveau type en l'appelant `point`.

OCaml
```
type point = float * float;;

let milieu (a : point) (b : point) = let (xa, ya) = a and (xb, yb) = b
                                     in (((xa +. xb) /. 2., (ya +. yb) /. 2.) : point);;
```

Notez bien la syntaxe pour imposer que la fonction `milieu` soit bien de type : `milieu : point -> point -> point`.

`Exemple 4.` Définir des nouveaux types adaptés aux données suivantes :

1. Point du plan
2. Point pondéré du plan
3. Fiche d'identité d'une personne
4. Cercle
5. Rotation dans d'espace
6. Courbe paramétrée

Remarque 4. ⚠ Le problème des objets définis sous la forme de n-uplets est que la récupération des informations contenues dans les différentes composantes se fait globalement sous la forme :

```
let (a,b,c) = objet in ...
```

1.2.2 Cas général :

Pour distinguer les différentes composantes définissant une donnée de type produit, et les pouvoir ainsi les récupérer une à une, on peut ajouter à celles-ci des *étiquettes*. La définition du type se fait alors de la façon suivante :

OCaml
```
type nouveau_type = {nom_1 : type_1 ; nom_2 : type 2 ; ... ; nom_n : type_n};;
```

Remarquez que les étiquettes commencent obligatoirement par une MINUSCULE.
On utilise alors ce nouveau type de la façon suivante :

OCaml
```
let a = {nom_1 = "valeur" ; nom_2 = "valeur" ; ... ; nom_n = "valeur"};;

a.nom_1;;        (* renvoie la valeur contenue dans la composante "nom_1" de la variable a *)
a.nom_2;;        (* renvoie la valeur contenue dans la composante "nom_2" de la variable a *)
```

`Exemple 5.` Définissons le type "client" qui comporte les informations suivantes : nom, année de naissance, ville.

OCaml
```
type client = {nom : string ; année : int ; adresse : string};;

let client1 = {nom = "Martin" ; année = 1982 ; adresse = "Paris 16"};;

client1.nom;;
client1.année;;
client1.adresse;;
```

⚠ Le problème de la construction précédente, c'est qu'elle n'autorise pas la modification des données. Pour autoriser cette modification, il suffit de rajouter la mention "mutable" devant chacune des composantes que l'on souhaite pouvoir modifier :

OCaml
```
type nouveau_type = {mutable nom_1 : type_1 ; ... ; mutable nom_n : type_n};;

let a = {nom_1 = "valeur" ; nom_2 = "valeur" ; ... ; nom_n = "valeur"};;

a.nom_1 <- "nouvelle valeur";;    (* modifie la valeur de la composante "nom_1" de a *)
a.nom_2 <- "nouvelle valeur";;    (* modifie la valeur de la composante "nom_2" de a *)
```

`Exemple 6.` Rendons le type "client" mutable :

OCaml
```
type client = {mutable nom : string ; mutable annee : int ; mutable adresse : string};;

let client1 = {nom = "Martin" ; annee = 1982 ; adresse = "Paris 16"};;
client1.adresse <- "La fleche";;       (* si celui-ci a démenagé ! *)

let new_ad  client c = client.adresse <- c;;
```

Exercice : 1

(∗) On souhaite manipuler sous OCaml les nombres complexes.
Définir en OCaml un type "complexe" et programmer les opérations usuelles portant sur les données de ce type.

1.2.3 Les types polymorphes

Pour définir un nouveau type produit ayant des composantes polymorphes, on ajoute `'a` et éventuellement `('a,'b)` devant le nom du nouveau type.

OCaml
```
type 'a nouveau_type1 = {nom_1 : 'a ; ...};;
type 'a nouveau_type2 = {nom_1 : 'a liste ; ...};;
type ('a,'b) nx_type3 = {l : 'a liste ; v: 'b};;
```

`Exemple 7.` On peut construire le type `'a boite` de la façon suivante :

```
type 'a boite = {nom : string ; mutable contenu : 'a list};;
```

A retenir : création de nouveaux types ! !

1. Type somme fini : `type new_type = | A... | B... | C... | D... | ... ;;`
2. Type somme : `type new_type = | Nom1 of type1 | Nom2 of type2 | ... ;;`
3. Type produit simple `type new_type == type1 * type2 * type3 * ... ;;`
4. Type produit étiqueté `type new_type = {nom1 : type1 ; nom2 : type2 ; ...};;`
5. Type produit étiqueté et mutable `type new_type = {mutable nom1 : type1 ; mutable nom2 : type2 ; ...}`

Bien respecter les règles : { Minuscule en début de variable, de fonctions, de type et d'étiquette ; Majuscule pour les éléments des types somme } .

Les programmes informatiques, traitent des données, créent des données et renvoient des données.
Pour stocker ces données, les langages informatiques proposent différentes structures que nous allons présenter dans ce chapitre.

2 Les structures abstraites de données

Les structures de données les plus simples rencontrées en informatique sont les suivantes :

1. Les nombres entiers (`int` en CAML - 32 bits)
2. Les nombres flottant (`float` en CAML - 64 bits)
3. Les caractères (`char` en CAML) codés à l'aide d'un entier de $[\![1, 256]\!]$: le code ASCII (8 bits)
4. Les booléens (`bool` en CAML) codés sur un seul bit

Remarque 5. Le codage des entiers et des flottants a été vu en début d'année dans le cours d'Informatique pour Tous.

Lorsque l'on souhaite manipuler une série de données, on fait appel à des structures plus complexes adaptées à la nature et à l'utilisation que l'on souhaite faire de ces données.
Dans ce cours, nous ne chercherons pas à comprendre comment ces données sont stockées sur l'ordinateur car leur mode de stockage est souvent spécifique au langage. Nous allons en revanche présenter les propriétés de ces structures en précisant les fonctions (on dit aussi *opérations*) qui permettent de les créer, d'extraire l'information qu'elles contiennent et d'effectuer des tests.

Une structure complexe de données sera donc définie par une série d'opérations que l'on classe en 3 catégories :

1. Les *fonctions de construction* (ou constructeurs) : qui permettent de créer la donnée
2. Les *fonctions de sélection* : qui permettent d'avoir accès aux informations contenues dans la donnée
3. Les *prédicats* : fonctions booléennes qui permettent de tester la nature de la donnée

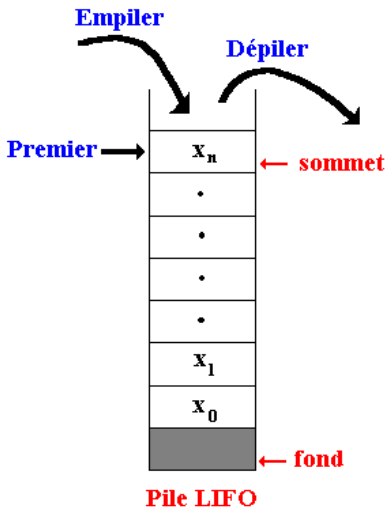
Parmi les structures complexes de données usuelles, on trouve :

1. les piles
2. les files
3. les dictionnaires
4. les arbres

Remarque 6. Les structures présentées ci-dessous se retrouvent rarement telles quelles dans les différents langages informatiques disponibles. Ceux-ci proposent en effet des structures assez proches mais qui leur sont en général spécifiques. Ainsi en OCaml, les structures complexes de données les plus couramment utilisées sont les structures de *liste* et de *array* qui sont des adaptations des structures abstraites *pile* et *dictionnaire*.

2.1 Les Piles

Les *Piles* sont des structures qui vérifient le principe LIFO : "Last In, First Out".
Elles seront surtout utilisées dans le but de stocker momentanément des données générées lors de l'exécution d'un programme.



Les opérations définissant les *Piles* sont les suivantes :

2 Constructeurs	1 Fonctions de sélection	1 Prédicat
f° créant la pile vide [ ] f° ajoutant un élément en tête de pile	f° renvoyant la tête de la pile f° qui élimine la tête de la pile ou f° qui fait les deux à la fois	f° qui teste si une pile est vide

Remarque 7. Selon les langages informatiques qui proposent le type "pile", certains imposent que tous les éléments stockés soient du même type tandis que d'autres, plus souples, autorisent que les éléments soient de types différents.

OCaml
```
(*** En OCaml, le type le plus proche de la structure abstraite PILE est le type LISTE ***)
(*** Mais cette structure n'est pas adaptée car elle est immuable                       ***)
```

`Exemple 8.` La structure de "pile" est en particulier utilisée lors du stockage temporaire d'information :
- Stockage des pages web visitées avec un navigateur internet
- Mémorisation des modifications faites dans un document
- Evaluation d'une expression arithmétique postfixée
- Sauvegarde des arguments lors de l'appel d'une fonction, en particulier lors de l'exécution d'un programme récursif.

Exercice : 2

On peut concevoir sous OCaml le type pile de la façon suivante :

OCaml
```
type 'a pile = {mutable n : int ; content = 'a array};;
```

1. Les constructeurs : déterminer une fonction de construction d'une pile vide et une fonction d'ajout d'un élément.
2. La fonction de sélection : déterminer une fonction qui récupère le sommet d'une pile et l'élimine de celle-ci.
3. Le prédicat : déterminer une fonction booléenne qui nous dit si une pile est vide.
4. Application : tester vos fonctions sur des exemples.

2.1.1 Application 1 : Evaluation d'expressions arithmétiques linéaires

DÉFINITION 1 : **Expression arithmétique sous forme postfixées**
Il s'agit d'une succession de nombres et d'opérateurs que l'on évalue de la façon suivante :
1. la succession de deux nombres ou plus ne génère aucun calcul
2. la succession de deux nombres a et b et d'un opérateur $\star$ entraîne l'opération $a \star b$.

`Exemple 9.` Calculer le résultat des expressions arithmétiques suivantes : $\begin{cases} 1)\ 7\ 2\ +\ 3\ \times \\ 2)\ 5\ 3\ 9\ +\ \times \end{cases}$.

Remarque 8.
1. Cette écriture post-fixée est aussi appelée *notation polonaise inverse*
2. Il existe plusieurs écritures postfixées pour une même expression algébrique.
3. L'intérêt de l'écriture postfixée tient au fait que les données intervenant dans le calcul apparaissent de façon ordonnée de la gauche vers la droite, ce qui rend le traitement algorithmique beaucoup plus simple et plus rapide.

Exercice : 3

1. L'objectif de cet exercice est la construction d'une fonction `eval` permettant d'évaluer une expression arithmétique postfixée ne faisant intervenir que des entiers et les opérateurs + et ×.

   Pour cela :

(a) l'expression arithmétique sera codée sous la forme d'une pile de chaînes de caractères.

(b) la fonction lit les composantes de la pile les unes après les autres.

(c) une pile sert à stocker les résultats intermédiaires.

Algorithme : La fonction empile les entiers jusqu'à ce qu'elle rencontre un opérateur. Lorsque c'est le cas, elle dépile les deux derniers éléments, effectue l'opération puis empile le résultat. En fin de parcours du vecteur, si l'expression arithmétique est correcte, la pile doit contenir le résultat du calcul.

2. Proposer une implémentation en OCaml du programme précédent utilisant la structure de LISTE.

3. Si $7\ 2 + 3 \times$ est la notation postfixée d'une expression arithmétique, alors on dit que $(7+2) \times 3$ est la notation infixée. Comment modifier la fonction précédente pour réaliser une fonction convertissant la notation postfixée en notation infixée ?

2.1.2 Application 2 : Gestion des environnements d'exécution des fonctions

Les piles sont systématiquement utilisées par l'ordinateur pour gérer les "environnements d'exécution" (en d'autres termes : "les arguments et les adresses") des fonctions.
Cette gestion est en général transparente pour l'utilisateur. Elle apparaît cependant lorsque le message d'erreur : `stack overflow` apparaît.

Lors de l'exécution d'un programme, l'appel d'une fonction entraîne la réservation d'un espace mémoire dont l'adresse est stockée (avec l'adresse de la fonction) au sommet d'une pile dite *pile d'exécution*. Lorsqu'on atteind le bout de la chaîne de fonctions, les adresses stockées dans la pile sont récupérées les unes après les autres pour effectuer les calculs.

`Exemple 10.` Considérons la fonction f0 définie de la façon suivante :

OCaml
```
let f2 p q = let x = p - q in .... ;;

let f1 x y z = let a = x+y and
                   b = y+z in ...
                              f2 a b;;

let f0 n = let x = n + 2 and
               y = 3 * n and
               z = n - 4 in ...
                            f1 x y z;;
```

Exécution de la commande `f0 5` :

Remarque 9. Le traitement des fonctions récursives est analogue. Un nouvel environnement d'exécution est créé à chaque appel récursif de la fonction comme s'il s'agissait d'une nouvelle fonction. Ainsi, la pile d'exécution ayant une hauteur bornée, il existe une limitation au nombre d'appels récursifs possibles. D'où parfois le message d'erreur `stack overflow` (n'apparaît plus en OCaml).

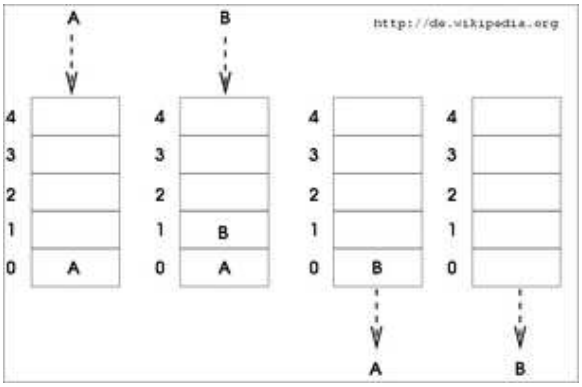
Lors de l'évaluation d'une fonction `f`, si vous souhaitez visualiser les appels aux différentes fonctions intermédiaires, puis l'évaluation de la valeur finale en passant par le calcul des différentes valeurs intermédiaires, pour pouvez utiliser la commande :

```
#trace f;;
```

`Exemple 11.` Appliquer la commande `#trace` à une fonction récursive de votre choix.

2.2 Les files

Comme les piles, il s'agit d'une structure linéaire et dynamique de stockage temporaire. La différence tient dans sa gestion des priorités. Contrairement aux piles qui fonctionnent sur le principe LIFO (*Last In First Out*), les files fonctionnent sur le principe FIFO (*First In First Out*). En pratique, l'insertion d'un nouvel élément se fait à l'extrémité opposée à celle d'où l'on retire les plus anciens (comme dans la file d'attente chez le boulanger !).



Les opérations définissant les *files* sont les suivantes :

2 Constructeurs	1 Fonction de sélection	1 Prédicat
f° créant la file vide [ ] f° ajoutant un élément en tête de file	f° renvoyant l'élément en queue de file en l'éliminant de la file	f° qui teste si une file est vide

`Exemple 12.` Ce sont par exemple des files qui gèrent la lecture de caractères saisis au clavier, l'envoi de textes vers une imprimante, l'acheminement des paquets sur un réseau. On utilise aussi le principe de file pour gérer les stocks d'un produit dans une entreprise afin d'éviter le dépassement des dates de péremption.

OCaml

```
(****   En OCaml, la structure abstraite de FILE n'est pas implémentée   ****)
(****   On peut, comme le montre l'exercice suivant, en proposer des     ****)
(****   implémentations à partir d'une LISTE ou d'un TABLEAU             ****)
```

Exercice : 4

Implémentation du type `file` sous OCaml :
Il s'agit ici de programmer une structure de données `'a file` qui doit permettre les 4 opérations précédentes.

1. Méthode 1 :

   On peut simplement décider de poser le type : `type 'a file = 'a list`.
   Cependant, cette structure étant immuable, elle n'est pas très adaptée...

2. Méthode 2 :

   On peut aussi définir un type `'a file` par un triplet contenant un tableau (de stockage), un indice de début et un indice de fin :

`type 'a file = {tableau : 'a array; mutable debut : int ; mutable fin : int} .`

Pour éviter que le décallage systématique des éléments vers la droite rende notre structure obsolète au bout d'un nombre fini d'opérations, on peut envisager de considérer les indices de fin et de début modulo la longueur du vecteur choisi.
Ecrire les quatres fonctions précédentes et en donner leur complexité.

Remarque 10. Les "files de priorité" sont une variante du type `file`. Chaque élément de la file est alors associé à un indice de priorité (entier de 1 à 3 par exemple). Le principe est alors de maintenir en permanence la file triée selon l'entier associé. Ainsi, au lieu de placer systématiquement le dernier objet inséré en fin de file, celui-ci est placé à sa place dans la file triée.

Remarque 11. En deuxième année, vous verrez le "tri par tas" (ou "heapsort") qui utilise l'implémentation d'une file de priorité à l'aide de la structure d'arbre. Cette implémentation présente l'avantage de donner des opérations d'insertion et d'ajout de complexité $O(\ln n)$. L'algorithme de tri qui consiste à placer chaque élément d'une liste donnée dans une file de priorité du type précédent a, quant à lui, au mieux une complexité en $O(n \ln n)$.

2.3 Les dictionnaires

La structure de données "dictionnaire" stocke des éléments de la forme "(clé, information)". L'exemple le plus simple est justement, un dictionnaire au sens classique. Les clés sont les mots, et les informations sont les définitions.
Les clés doivent être telles qu'il existe une relation d'ordre entre elles. Elles peuvent être définies sur un domaine beaucoup plus vaste que celui qui est effectivement utilisé. Un numéro de registre national belge contient 11 chiffres, il y a donc potentiellement 10^{11} numéros qui peuvent être attribués, même si la population n'est que de 10^6.

Un dictionnaire est en général maintenu trié (selon la clé) afin d'optimiser les temps de recherche.

Les opérations définissant les *dictionnaires* sont les suivantes :

4 Constructeurs	1 Fonctions de sélection	1 Prédicat
f° créant un dictionnaire vide f° ajoutant une donnée au dictionnaire f° modifiant l'information associée à une donnée f° supprimant une donnée d'un dictionnaire	f° renvoyant la donnée associée à une clé donnée	f° qui teste si un dictionnaire est vide

Remarque 12. On peut aussi envisager de programmer des fonctions complémentaires du type :

1. `min : dictionnaire -> donnée` qui renvoie la donnée de clé minimale
2. `max : dictionnaire -> donnée` qui renvoie la donnée de clé maximale
3. `liste : dictionnaire -> cle -> cle -> donnée` qui liste les données de clés comprises entre 2 autres

Exercice : 5
Implémentation impérative du type dictionnaire à l'aide d'un tableau :

Déterminer une implémentation impérative du type "dictionnaire" à l'aide d'un tableau contenant les couples [clé ; contenu] et maintenu trié en permanence.

1. Création du type
2. Programmation des opérations en un coût linéaire

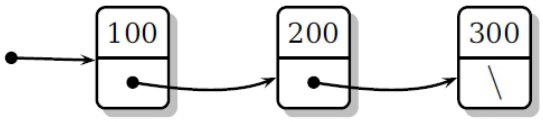
3 Les Listes et les Tableaux sous OCaml

Les *listes* et les *tableaux* sont deux structures de données complexes implémentées et couramment utilisées en OCaml.

3.1 Liste (liste chaînée ou liste dynamique : list)

Il s'agit d'une structure de données de taille variable essentiellement utilisée en programmation récursive et définie de façon récursive par :

- une liste vide appelée "NIL"
- 'a list = 'a : : 'a list



Liste `[100 ; 200 ; 300]`

Chaque composante d'une "liste" contient une valeur (qui intéresse l'utilisateur) et l'adresse de la cellule qui la précède (d'où le nom parfois utilisée de "liste chaînée"), la dernière cellule contenant uniquement l'élément NIL. La liste proprement dite renvoie sur la première cellule qui est aussi la dernière créée. Seule les valeurs contenues dans la liste intéressent les utilisateurs. Les adresses ne servant qu'à lier les éléments entre eux. Ces adresses ne sont pas accessibles en OCaml.

Remarque 13. Il est possible d'implémenter d'autres structures de liste :

1. Les *listes doublement chaînées* dans lesquelles il est possible d'accèder à chacun des éléments placés en bout de liste (le premier et le dernier) et de remonter la liste soit dans un sens, soit dans l'autre.
2. Les *listes circulaires* dans lesquelles la tête de liste est l'élément qui suit le dernier élément de la liste.

Les opérations définissant les *Listes* sont les suivantes :

En O-caml :

2 Constructeurs	1 Fonction de sélection	1 Prédicat
`[]` : création de la liste vide [ ] `a::l` : ajout de "a" en tête d'une liste "l"	`tl::ql = l` : tl tête de la liste "l" : ql queue de la liste "l" ou `List.hd l` : tête de liste `List.tl l` : queue de liste	`l = []` : teste si une liste "l" est vide

Remarque 14.

1. La structure de liste s'apparente à la structure "pile".
   Elle diffère dans la mesure où la récupération de la donnée en "tête de liste" ne l'élimine pas automatiquement de la liste.
2. ATTENTION : Contrairement aux listes en langage Python, l'ajout d'un élément à une liste se fait en tête.

Exercice : 6

Programmer en OCaml les fonctions complémentaires du type "liste" suivantes :

1. longueur
2. max, min
3. valeur du ième élément
4. concaténation
5. image miroire
6. insertion, suppression

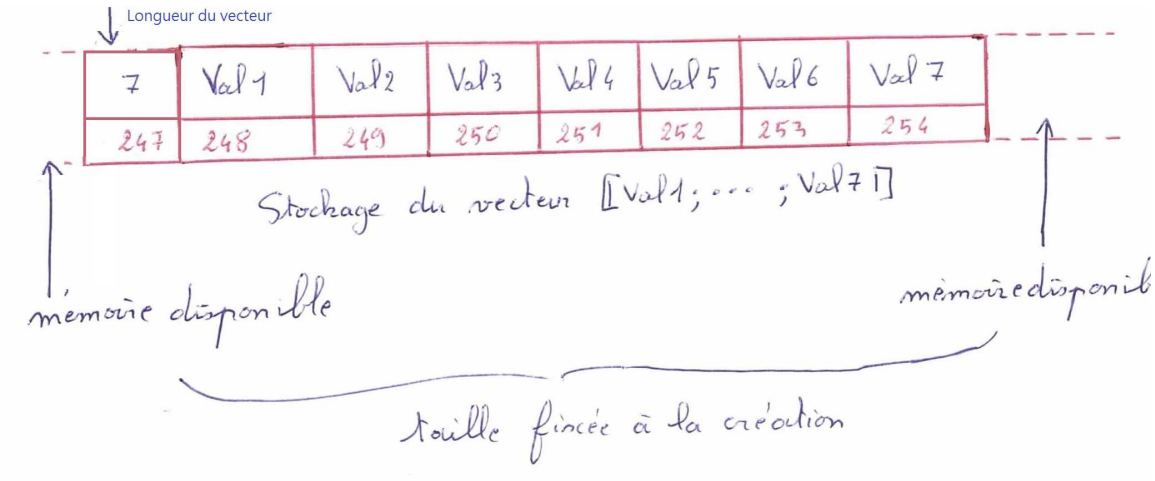
3.2 Tableaux (liste indexée : array)

Contrairement à la structure de liste, la taille d'un *tableau* est fixée à la création : on dit qu'il s'agit d'une structure de données *persistante*.
Cette structure est essentiellement utilisée en programmation itérative (par opposition à la programmation récursive).

La principale caractéristique de ce type est de présenter une fonction d'indexation permettant d'accèder directement à un élément quelconque de la liste. Contrairement aux listes dynamiques, l'accès à un élément se fait à temps constant,

quelque soit la position de cet élément dans la liste.

Stockage informatique : Lors de la création d'un tableau de taille n, un bloc de $n+1$ cases mémoires consécutives est associé à cette structure. La première case contient la taille de la liste, les suivantes son contenu. Les cases mémoires étant consécutives et de même taille, un simple calcul algébrique permet de connaître l'adresse de la case d'indice i ce qui explique pourquoi l'accès à un élément se fait à temps constant. En revanche, une fois le tableau créé, les cases mémoires qui précèdent et qui suivent les éléments du tableau sont considérées disponibles pour d'autres usages : c'est pourquoi il n'est pas possible de modifier la taille d'un tableau.



Enfin, retenons qu'en général, (en particulier sous OCaml) l'indexation d'une liste indexée $\begin{cases} \text{commence à } 0 \\ \text{se termine à } n-1 \end{cases}$ si n est la longueur du tableau.

Les opérations définissant les *Tableaux* sont les suivantes :

3 Constructeurs	`Array.make n 0` : création du tableau $[\|0;...;0\|]$ de taille "n" `[|x1;...;xn|]` : création du tableau $[\|x_1;...;x_n\|]$ `v.(p) <- a` : insertion de "a" dans la composante de "v" d'index "p"
2 Fonctions de sélection	`v.(p)` : valeur contenue dans la composante de "v" d'index "p" `Array.length v` : longueur du tableau "v"
1 Prédicat	`v.(p) = a` : teste si la composante "v.(p)" vaut "a"

Remarque 15. La fonction `Array.create` est équivalente à la fonction `Array.make`.

Exercice : 7

Dans le langage Python, les "listes" sont à la fois des listes indexées ET des listes dynamiques.
A l'aide des structures de données disponibles sous OCaml, programmer cette nouvelle structure de données.

Pour plus de détails sur les structures de LISTE et de TABLEAU, voir le lien suivant : http ://mpechaud.fr/scripts/donnees/listes