Chapitre 6

Structures de données

1 Type de données abstrait **1**
- 1.1 Définitions . . . 1
- 1.2 Piles . . . 2
- 1.3 Files . . . 3
- 1.4 Files de priorité . . . 4
- 1.5 Dictionnaires . . . 4

2 Types personnalisées **5**
- 2.1 Abréviations . . . 5
- 2.2 Type construit . . . 5
- 2.3 Type somme . . . 6
- 2.4 Type enregistrement . . . 8

3 Application à la structure de pile **9**
- 3.1 Structure persistante . . . 9
- 3.2 Structure impérative . . . 11

1 Type de données abstrait

1.1 Définitions

Définition.

Une **structure de donnée** est l'association :

- d'un ou de plusieurs types servant à contenir des données,
- de fonctions, appelée **primitives d'accès**, servant à manipuler ces données.

Exemple. Pour pouvoir manipuler des polynômes, on peut imaginer une structure de données qui disposerait des primitives d'accès suivantes :

- `creer_polynome` : construire le polynôme nul ;
- `etre_nul` : tester si un polynôme est nul ;
- `ajouter_coefficient` : ajouter le coefficient constant ; cette opération permet d'obtenir le polynôme $XP(X) + c$ à partir d'un polynôme P déjà construit et d'une constant c ;
- `renvoyer_coefficient` : renvoyer le coefficient constant ; cette opération permet d'obtenir le reste dans la division par X d'un polynôme P ;
- `retirer_coefficient` : retirer le coefficient constant ; cette opération permet d'obtenir le quotient dans la division par X d'un polynôme P ;

Toutefois, cette structure correspond à un modèle abstrait plus général détaillé plus loin : la structure de pile.

Nous distinguerons la structure de données abstraite de la structure concrète :

Définition.

- La **structure abstraite** décrit la structure du point de vue du programmeur qui l'utilise (c'est le mode d'emploi de la structure), indépendant du langage.
- La **structure concrète** est la façon dont est implémentée la structure, dépendant du langage.

Les types abstraits deviennent des types OCaml parfaitement définis et les primitives des fonctions OCaml codés selon des algorithmes précis.

Définition.

Dans son implémentation, une structure de données a différentes caractéristiques :

- Si l'espace mémoire alloué à la structure est de taille fixe, on dit que la structure est **statique**. Sinon, on dit qu'elle est **dynamique**.
- Si le contenu déjà présent dans la structure est modifiable, on dit que la structure est **mutable**.

Une structure **statique** et **non mutable** est dite **persistante**. Sinon, elle est dite **impérative**.

Dans la suite de cette partie, nous allons présenter quelques structures de données présentes dans la plupart des langages informatiques.

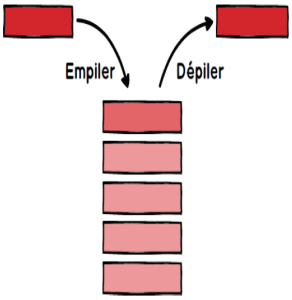
1.2 Piles

Une **pile** est une structure de données avec priorité LIFO (pour *Last In, First Out*) : le dernier élément inséré dans la structure est le premier à en être retiré. Plus précisément :

Définition.

Le type **pile** est une structure de donnée munie des opérations suivantes :

- `creer_pile_vide` : construire une pile vide ;
- `etre_pile_vide` : tester si une pile est ou non vide ;
- `empiler` : empiler un élément dans une pile ;
- `depiler` : dépiler et renvoyer le dernier élément empilé dans une pile non vide.



Exemples. Voici quelques exemples de piles :

- gestion de l'historique des pages visitées dans un navigateur internet ou des actions dans un traitement de texte pour l'annulation `Ctrl+Z`... : on empile à chaque instant la dernière page visitée ou la dernière action effectuée ; la touche *back* ou *undo* correspond à une action pour dépiler ;
- exécution d'une fonction récursive : chaque appel récursif est ajouté dans une pile et lorsque l'on arrive à un cas de base (c'est-à-dire une exécution de la fonction qui n'utilise pas la récursivité), on dépile pour passer aux appels antérieurs ;
- calcul d'une expression arithmétique écrite en notation polonaise inverse : on lit l'expression, on empile les opérandes et pour chaque opération, on dépile le bon nombre d'opérandes puis on empile le résultat de l'opération.

Remarque. Il existe en OCaml un module `Stack` permettant de modéliser des piles. Les fonctions sont les suivantes, avec un type de la forme `'a Stack.t` :

- `Stack.create : unit -> 'a Stack.t` permet de créer une pile vide ;
- `Stack.is_empty : 'a Stack.t -> bool` permet de tester si une pile est vide ;
- `Stack.push : 'a -> 'a Stack.t -> unit` ajoute un élément en haut de la pile ;
- `Stack.pop : 'a Stack.t -> 'a` enlève un élément en haut d'une pile et le renvoie.

Exercice 1

1. Écrire en OCaml une fonction qui intervertit les deux éléments situés au sommet d'une pile contenant plus de deux éléments.
2. Écrire en OCaml une fonction qui prend une pile `p` en argument et renvoie une copie `q` de cette pile.

   La pile `p` ne doit pas être modifiée à l'issue de cette opération.
3. Écrire une fonction qui réalise la rotation d'une pile, c'est-à-dire que l'élément au somment se retrouve en bas de la pile.

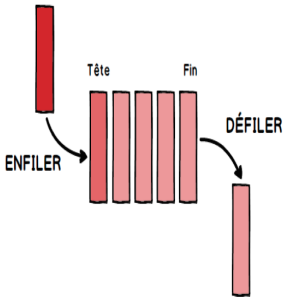
1.3 Files

Une **file** est une structure de données avec priorité FIFO (pour *First In, First Out*) : l'élément retiré est le plus ancien dans la file. Plus précisément :

Définition.

Le type **file** est une structure de donnée munie des opérations suivantes :

- `creer_file_vide` : construire une file vide ;
- `etre_file_vide` : tester si une file est ou non vide ;
- `enfiler` : ajouter un élément à une file ;
- `defiler` : retirer et renvoyer l'élément en tête d'une file non vide, c'est-à-dire l'élément le plus ancien dans la file.



Exemples. Voici quelques exemples de files :

- gestion des clients dans une file d'attente devant un guichet avec une file unique ;
- serveur d'impression sur une imprimante avec les requêtes d'un ou plusieurs ordinateurs.

Remarque. Il existe en OCaml un module `Queue` permettant de modéliser des files par une structure impérative. Les fonctions sont les suivantes, avec un type de la forme `a' Queue.t` :

- `Queue.create : unit -> 'a Queue.t` permet de créer une file vide ;
- `Queue.is_empty : 'a Queue.t -> bool` permet de tester si une file est vide ;
- `Queue.push : 'a -> 'a Queue.t -> unit` ajoute un élément à l'entrée de la file ;
- `Queue.pop : 'a Queue.t -> 'a` enlève un élément à la sortie de la file et le renvoie.

Exercice 2

Les nombres de Hamming sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5 :

$$1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, ...$$

Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (songez que le 1999-ième entier de Hamming est égal à 8 100 000 000 et le 2000-ième à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).
On adopte donc la démarche suivante : on utilise trois files f_2, f_3, f_5 contenant initialement le nombre 1, et on suit la démarche suivante :

- on détermine le plus petit des trois têtes de file, que l'on note k et que l'on imprime à l'écran ;
- on retire cet élément des files où il se trouve ;
- on insère en queue des files f_2, f_3 et f_5 les entiers $2k$, $3k$ et $5k$.

Vous l'avez compris : cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.

1. On admet qu'il existe une instruction `Queue.peek f` qui renvoie la tête de file, mais sans modifier cette dernière. Rédiger une fonction en OCaml permettant l'affichage des n premiers nombres de Hamming.
2. L'inconvénient de la démarche précédente est que le même nombre peut se retrouver dans plusieurs des trois files. Modifier votre fonction pour que cela ne soit plus le cas.

1.4 Files de priorité

Une **file de priorité** est une structure de données où l'ordre pour le retrait d'un élément est donné par une clé (un attribut) associée à la donnée, appelée priorité. Plus précisément :

Définition.

Le type **file de priorité** est une structure de données munie des opérations suivantes :

- `creer_fdp_vide` : construit une file de priorité vide ;
- `etre_fdp_vide` : tester si une file de priorité est vide ;
- `inserer` : ajouter un élément (avec sa priorité) à la file de priorité ;
- `modifier_priorite` : modifier la priorité d'un élément déjà présent ;
- `retirer_prioritaire` : retirer et renvoyer l'élément avec la plus grande priorité de la file.

Exemples. Voici quelques exemples de files de priorité :

- liste de devoirs notée dans l'ordre de distribution, la priorité étant la date pour rendre le devoir ;
- gestion des patients aux urgences en fonction de la gravité de leurs problèmes médicaux.

1.5 Dictionnaires

Un **dictionnaire** est une structure de données où l'on stocke des éléments dotés d'une clé, de sorte à pouvoir extraire l'élément correspondant à une clé donnée. Plus précisément :

Définition.

Le type **dictionnaire** est une structure de données abstraite munie des opérations suivantes :

- `creer_dictionnaire_vide` : construire un dictionnaire vide ;
- `etre_dictionnaire_vide` : tester si un dictionnaire est vide ;
- `inserer` : insérer un couple (clé, élément) dans le dictionnaire ;
- `chercher` : retrouver, s'il existe, l'élément du dictionnaire correspondant à une clé donnée ;
- `supprimer` : supprimer, s'il existe, l'élément du dictionnaire correspondant à une clé donnée ;
- `modifier` : modifier, s'il existe, l'élément du dictionnaire correspondant à une clé donnée.

Exemples.

- L'exemple typique est un dictionnaire au sens usuel : on y rentre des définitions avec les mots à définir comme clés. Ainsi, on veut pouvoir consulter la définition d'un mot donné.
- Outre le dictionnaire, on peut citer un annuaire téléphonique et plus généralement toute base de données.

2 Types personnalisées

Pour implémenter de nouvelles structures de données concrètes en OCaml correspondant à certaines structures de données abstraites, il est intéressant de créer ses propres types avec l'instruction `type` :

- soit en donnant un autre nom à un type déjà existants ;
- soit en construisant un type particulier qu'on souhaite distinguer des autres types déjà existants ;
- soit en donnant la réunion disjointe de toutes les valeurs possibles (on parle de type somme) ;
- soit en utilisant une sorte de produit cartésien de types existants (on parle de type produit ou enregistrement).

2.1 Abréviations

Lorsqu'un type déjà existant revient souvent dans du code, on peut en faciliter la lecture en lui donnant un **alias**. Par exemple :

```
type coordonnees = float * float ;;
```

2.2 Type construit

Il est possible en OCaml de construire de nouveaux types autrement que par simple assemblage de types déjà existants. On obtient ce qu'on pourrait appeler des types formels et que OCaml préfère appeler des **types construits**.

Pour cela, considérons un identificateur `Constr` appelé **constructeur de type**. Alors la déclaration

```
type monType = Constr ;;
```

construit un nouveau type (c'est-à-dire un nouvel ensemble de valeurs) de nom `monType` n'ayant qu'une seule valeur à savoir `Constr` (un peu analogue à la définition d'une constante mathématique de nom `Constr` qui ne serait jamais évaluée). On dit alors que `Constr` est un **constructeur constant** (nous avons en fait déjà rencontré quatre constructeurs constants, à savoir `true`, `false`, `[]` et `()`). Par exemple :

```
type couleur = Bleu ;;

Bleu ;;
- : couleur = Bleu
```

Ainsi, `Bleu` est l'unique objet qui est de type `couleur`.

Attention.

Les constructeurs doivent nécessairement commencer par une lettre capitale alors que les noms de types commencent par une minuscule.

De la même façon, si on désigne par `leType` un type déjà connu, alors la déclaration

```
type monType = Constr of leType
```

construit un nouveau type (c'est-à-dire un nouvel ensemble de valeurs) de nom `monType` dont les valeurs sont exactement les assemblages `Constr x` où `x` est un objet de type `leType` ; ceci est tout à fait analogue à une fonction formelle mathématique qui ne serait jamais évaluée (de la même façon que, dans une expression mathématique, on manipule le symbole sans jamais l'évaluer). On dit alors que `Constr` est un **constructeur non constant**. Par exemple :

```
type logarithme = Ln of float ;;

Ln 2.3 ;;
- : logarithme = Ln 2.3
```

On peut alors définir des primitives sur ce nouveau type. Par exemple :

```
let evaluation x = let Ln a = x in log a ;;
val evaluation : logarithme -> float = <fun>

evaluation (Ln 2.3) ;;
- : float = 0.83290912293510388
```

```
let somme_algebrique x y = let Ln a = x and Ln b = y in Ln (a*.b) ;;
val somme_algebrique : logarithme -> logarithme -> logarithme = <fun>

somme_algebrique (Ln 1.2) (Ln 3.2) ;;
- : logarithme = Ln  3.84
```

2.3 Type somme

On peut mêler à volonté au sein d'un même type des types construits que ce soit avec des constructeurs constants ou avec des constructeurs non constants. On obtient ainsi des **types sommes**, tout à fait analogues à une réunion disjointe d'ensembles mathématiques. Pour cela il suffit de séparer les différentes constructions de type par une barre verticale.

Par exemple, pour définir un nouveau type qui mêle à la fois deux constantes `Gamma` et `Epsilon`, des "sinus" d'entiers `Sin x` et des couples formés d'une chaîne de caractère et d'un réel, on pourra poser :

```
type idiot = Gamma | Epsilon | Sin of int | Couple of string * float ;;
```

A partir de cette déclaration, nous disposons de nouvelles valeurs :

- les constantes `Gamma` et `Epsilon`,
- les valeurs du type `Sin n` où `n` est un entier (ne pas confondre avec la fonction `sin` définie sur les réels),
- les valeurs du type `Couple (s,x)` où `s` est une chaîne de caractères et `x` un réel en virgule flottante.

Maintenant nous pouvons poser :

```
let x1 = Gamma ;;
val x1 : idiot = Gamma

let x2 = Sin 2 ;;
val x2 : idiot = Sin 2

let x3 = Couple ("abcd",3.1) ;;
val x3 : idiot = Couple ("abcd", 3.1)
```

De cette manière nous avons construit une réunion disjointe de deux singletons, de l'ensemble des sinus d'entiers et du produit cartésien de l'ensemble des chaînes de caractères par l'ensemble des réels en virgule flottante.

Rien ne nous interdit alors de définir une fonction `f` qui associera à `Gamma` la valeur $0,577$, à `Epsilon` la valeur 10^{-10} , à un entier considéré comme un nombre réel son sinus (le vrai cette fois) et à un couple formé d'une chaîne de caractères et d'un réel, ce réel. Par reconnaissance de motif, c'est très facile :

```
let evaluation_idiote x = match x with
    | Gamma -> 0.577
    | Epsilon -> 1e-10
    | Sin theta -> sin (float_of_int theta)
    | Couple (a,b) -> b
;;
val evaluation_idiote : idiot -> float = <fun>

evaluation_idiote x1 ;;
- : float = 0.577

evaluation_idiote x2 ;;
- : float = 0.90929742682568171

evaluation_idiote x3 ;;
- : float = 3.1
```

On remarque que la reconnaissance de motif suit exactement la définition du type. Ce n'est pas indispensable, mais si nous oublions d'examiner un cas, OCaml nous en avertira en précisant que la liste des motifs n'est pas exhaustive pour le type en question (autrement dit que le domaine de définition de notre fonction n'est pas l'ensemble considéré tout entier).

On peut également définir des types sommes génériques, c'est-à-dire dont le type dépend de certains paramètres qui sont eux-mêmes des types (analogues aux vecteurs, listes ou références qui dépendent du type auxquels on les applique). Pour cela, on fait précéder le nom du nouveau type des paramètres de type qu'il recevra, séparés par des virgules et encadrés par des parenthèses s'il y en a plusieurs ; bien entendu, comme on l'a déjà vu, les paramètres de type sont précédés d'une apostrophe.

Par exemple, on peut définir un type générique `bidon` qui associera à deux types quelconques `type1` et `type2` un nouveau type `(type1,type2) bidon` comprenant une constante universelle `Rho` et des triplets `(x,y,z)` d'un élément de type `type1`, d'un élément de type `type2` et d'un entier de la façon suivante :

```
type ('a,'b) bidon = Rho | Triplet of 'a * 'b * int ;;

Rho ;;
- : ('a, 'b) bidon = Rho

Triplet (1,1.2,3) ;;
- : (int, float) bidon = Triplet (1, 1.2, 3)
```

On peut constater que OCaml a reconnu que `Triplet (1,1.2,3)` est de type `(int, float) bidon`.

Exercice 3

1. Définir un type somme `complexe` de deux constructeurs `Cartesien (x, y)` et `Polaire (r, t)` où `(x, y)` et `(r, t)` sont de type `float * float`.
2. Écrire en OCaml une fonction permettant de calculer le module de `z` de type `complexe`.

Exercice 4

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

- *Zero* est un entier naturel ;
- si n est un entier naturel, alors son successeur $S(n)$ est un entier naturel.

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

1. Définir un type somme `nat` pour les entiers naturels.
2. Écrire en OCaml une fonction `nul : nat -> bool` qui teste si un entier est nul ou non.
3. (a) Écrire en OCaml une fonction récursive `entier : nat -> int` qui calcule la valeur entière d'un entier naturel de Peano.
   (b) Écrire en OCaml sa fonction réciproque `peano : int -> nat`.
4. (a) Écrire en OCaml une fonction récursive `addition : nat -> nat -> nat` qui calcule la somme de deux entiers naturels de Peano.
   (b) Faire de même pour la fonction multiplication.

Exercice 5

Une liste chainée ℓ d'éléments de type a est une structure de données non mutable qui est :

- soit la liste `Vide` ;
- soit un couple (x, q) (ou maillon de chaine) composé d'un élément x de type a et d'une autre liste q d'éléments de type a.

On dit que x est la tête de la liste ℓ et que q est sa queue.

1. Implémenter les listes chainées en utilisant un type somme `maListe`.

   On désignera par `Ajout` le constructeur du maillon.
2. Définir la liste $[1; 2; 3]$.
3. Définir les primitives d'accès `tete`, `queue` et `longueur` et les tester sur l'exemple précédent.

2.4 Type enregistrement

Le dernier type d'objets utilisé par OCaml est le **type enregistrement**. Il part de la constatation qu'un objet courant est souvent construit à partir d'un certain nombre de caractéristiques (un nom, un prénom, une adresse, un numéro de compte, un montant, etc.), les unes étant fixes (nom, prénom, numéro de compte), les autres étant variables (montant, éventuellement adresse).

OCaml (comme Pascal ou C) permet de définir des objets de type enregistrement, chacun ayant un certain nombre de **champs** qui sont eux mêmes de objets OCaml. Chaque champ est décrit par un identificateur (un nom) appelé une **étiquette**.

Prenons l'exemple d'un compte en banque. Dans un tableur, on présenterait par exemple l'ensemble des comptes en banque sous la forme :

nom	*prenom*	*adresse*	*numero*	*montant*
etc...				

De la même façon, nous représenterons chaque ligne par un objet OCaml composé de cinq champs : un champ `nom` de type `string`, un champ `prenom` de type `string`, un champ `adresse` de type `string`, un champ `numero` de type `int` et un champ `montant` de type `float`. Pour cela nous ferons une déclaration de type sous la forme

```
type compte_en_banque = {nom : string; prenom : string; mutable adresse : string;
numero : int; mutable solde : float} ;;
```

Le mot clé `mutable` précède les étiquettes qui seront modifiables, un peu à la manière d'une référence.

Attention.

Les étiquettes décrivant les champs dans un type enregistrement doivent nécessairement commencer par une lettre minuscule.

On peut alors poser :

```
let compte_ex = {nom = "moi"; prenom = "encore moi"; adresse = "ici";
numero = 123456789; solde = 2.3} ;;
```

A partir de cet instant, l'identificateur `compte_ex` est lié à un objet composé de 5 champs, dont deux seront modifiables. On accèdera à un champ donné d'un objet sous la forme :

```
nom_objet.etiquette
```

Par exemple :

```
compte_ex.nom ;;
- : string = "moi"

compte_ex.solde ;;
- : float = 2.3
```

Les champs modifiables sont modifiés comme les éléments d'un tableau sous la forme :

```
nom_objet.etiquette <- valeur
```

Par exemple :

```
compte_ex.adresse <- "là bas" ;;
- : unit = ()

compte_ex.solde <- compte_ex.solde +. 1. ;;
- : unit = ()

compte_ex ;;
- : compte_en_banque = {nom = "moi"; prenom = "encore moi"; adresse = "la bas";
numero = 123456789; solde = 3.3}
```

Exercice 6
Les entiers de Gauss sont les éléments de la forme $a + ib$ où a et b sont des entiers.

1. Définir un type enregistrement pour les entiers de Gauss.
2. Affecter une variable `i` avec la base des imaginaires purs.
3. Écrire les fonctions `addition`, `multiplication` et `conjugaison` qui prennent des arguments entiers de Gauss.

Exercice 7
On dispose d'une pile d'assiettes bleues ou rouges numérotées disposées dans le désordre. Comment procéder pour former une pile dans laquelle les assiettes bleues sont situées sous les assiettes rouges, mais en faisant en sorte que pour chacune des deux couleurs l'ordre relatif ne soit pas modifié ? Autrement dit, si l'assiette bleue i est située sous l'assiette bleue j dans la pile initiale, ce sera toujours le cas dans la pile finale.
On définit les types :

```
type couleur = Bleue | Rouge and assiette = {c : couleur ; n : int} ;;
```

1. Définir une fonction `tri p b r` qui étant données trois piles d'assiettes `p`, `b` et `r`, vide la pile `p` en ajoutant ses assiettes bleues à la pile `b` et ses assiettes rouges à la pile `r`.
2. Définir une fonction `empile a p` qui empile la pile `a` sur la pile `p`.
3. En déduire une fonction `rangement p` qui range les assiettes bleues de `p` sous ses assiettes rouges.

---

3 Application à la structure de pile

Nous allons dans cette dernière partie implémenter la structure de pile à l'aide des types personnalisées vus précédemment.

3.1 Structure persistante

Implémentation à l'aide d'un constructeur de type

Pour cette représentation, les fonctions ne modifient pas les piles, mais renvoient de nouveaux objets.

```
type 'a pile = PileVide | Ajout of 'a * 'a pile ;;
```

On peut alors définir les primitives d'accès de la façon suivante :

Implémentation à l'aide d'un alias

Pour cette représentation, le type n'est qu'un alias du type de liste chaînée. Les fonctions ne modifient pas les piles, mais renvoient de nouveaux objets.

```
type 'a pile = 'a list ;;
```

On peut alors définir les primitives d'accès de la façon suivante :

3.2 Structure impérative

Implémentation à l'aide de références de listes

Il s'agit de s'assurer que la structure est modifiable. Pour cela on peut utiliser des références de listes.

```
type 'a pile = 'a list ref ;;
```

On peut alors définir les primitives d'accès de la façon suivante :

Implémentation à l'aide d'un type enregistrement

Pour que la structure soit modifiable, on peut aussi utiliser un type enregistrement.

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

On peut alors définir les primitives d'accès de la façon suivante :

Exercice 8

On souhaite effectuer le tri par insertion d'une pile d'entiers (implémentée à l'aide d'un type enregistrement).

1. Écrire en OCaml une fonction récursive `insere x p` qui place un entier `x` à la bonne place dans une pile `p` triée.
2. Écrire en OCaml une fonction récursive `tri_insertion_pile p` qui trie une pile `p` en suivant le principe du tri par insertion.

Exercice 9

En notation postfixe (ou NPI, notation polonaise inversée), l'évaluation d'une expression se fait par lecture de gauche à droite et les parenthèses sont inutiles. Par exemple, l'expression mathématique écrite en mode infixe

$$\frac{y+\sqrt{x-1}}{3+2\,\cos(z)}$$

se traduit en mode postfixe par :

$$x \quad 1 \quad - \quad \sqrt{} \quad y \quad + \quad z \quad \cos \quad 2 \quad * \quad 3 \quad + \quad /$$

Il n'est pas nécessaire de mémoriser d'autre résultat intermédiaire que le(s) argument(s) du prochain opérateur, dont l'action est alors immédiate. Pour cette raison, l'évaluation d'une expression postfixée se prête très bien à l'utilisation d'une pile de type LIFO.
L'évaluation immédiate nécessite que tous les opérateurs aient une arité (un nombre d'arguments) constante. Il faut en particulier différencier la soustraction (binaire) du changement de signe (unaire).

1. Définir sur OCaml une expression mathématique postfixée comme une liste de symboles, un symbole étant soit un nombre entier, soit une opération unaire (c'est-à-dire une fonction `int -> int`), soit une opération binaire (c'est-à-dire une fonction `int -> int -> int`).
2. Définir sur OCaml l'expression mathématique postfixée de $3 * (10 + 5)$.
3. L'idée pour évaluer une expression mathématique postfixée est de définir une autre pile (modélisée par une liste) qui empile les têtes de l'expression jusqu'à tomber sur une opération :
   - si c'est une opération unaire : on dépile alors le dernier nombre de cette pile, on effectue l'opération sur ce nombre et on rempile le résultat ;
   - si c'est une opération binaire : on dépile alors les deux derniers nombres de cette pile, on effectue l'opération sur ces nombres et on rempile le résultat.

   On continue de manière récursive jusqu'à ce que la première pile soit vide.

   Écrire en OCaml une fonction permettant d'évaluer une expression mathématique postfixée en suivant le modèle décrit précédemment.
