TP6 : Types et structures de données

Lundi 6 novembre

1 Types construits

Exercice 1. Géométrie

On s'intéresse à des types qui nous permettent de faire la représentation d'objets de la géométrie plane.

1. Proposer un type point qui correspond aux coordonnées d'un point dans le plan.
2. Proposer un type triangle composé de trois points.
3. Proposer une fonction egal de signature triangle -> triangle -> bool qui détermine si deux triangles sont égaux.

   Deux triangles sont égaux si leurs points sont égaux à une permutation près.
4. On se donne le type polygone suivant qui représente un polygone par la liste de ses points :

```
type polygone = point list
```

   (a) Proposer une fonction de signature triangle -> polygone qui convertit un triangle en polygone.
   (b) Proposer une fonction de signature quadrilatere -> polygone qui convertit un quadrilatère en polygone.
   (c) Proposer une fonction de signature polygone_egal de signature polygone -> polygone -> bool qui renvoie si deux polygones sont égaux.

   Deux polygones sont égaux si leurs points sont égaux à une rotation circulaire près ou à une inversion de sens près. C'est à dire que [p1; p2; p3; p4 et [p3; p2; p1; p4] sont égaux, mais pas [p1; p2; p3; p4] et [p1; p2; p4; p3].
5. On s'intéresse aux cercles du plan.
   (a) Proposer un type cercle pour représenter les cercles du plan.
   (b) Proposer une fonction polygone_inscrit_dans de signature polygone -> cercle -> bool qui vérifie si un polygone est inscrit dans un cercle.
   (c) Proposer une fonction cercle_inscrit_dans de signature cercle -> cercle -> bool qui vérifie si un cercle est inscrit dans un autre.

Exercice 2. Égalité structurelle et égalité physique

En OCaml, comme dans la plupart des langages, on peut faire la distinction entre l'égalité structurelle et physique. La première correspond à l'égalité entre deux objets au sens mathématiques est testée par le symbole d'égalité simple =, tandis que la seconde est l'égalité au sens de la mémoire physique qui teste avec le symbole d'égalité double == si deux éléments correspondent au même objet en mémoire.

1. Combien valent les deux valeurs suivantes ?

```
[1; 2] == [1; 2]
```

```
[1; 2] = [1; 2]
```

   Que peut-on en déduire ?

2. De même, comment fonctionnent les égalités structurelle et physique pour les tableau, les n-uplets et les chaînes de caractères ?
3. Que se passe-t-il lors de la comparaison de structures imbriquées ?
4. Combien valent les deux valeurs suivantes ?

```
1 == 1
```

```
1 = 1
```

   Que peut-on en déduire ?

5. Combien valent les deux valeurs suivantes ?

```
'a' == 'a'
```

```
'a' = 'a'
```

   Que peut-on en déduire ?

6. Combien valent les deux valeurs suivantes ?

```
1.0 == 1.0
```

```
1.0 = 1.0
```

De la même manière qu'on distingue l'égalité structurelle et l'égalité physique, il existe une différence entre l'inégalité structurelle <> et l'inégalité physique !=.

Exercice 3. Égalité et comparaison sur les types construits

1. Comment se comporte l'égalité physique = et l'opérateur de comparaison < sur le type somme suivant ?

```
type a = A | B of int | C of float
```

2. Comment se comporte l'égalité physique = et l'opérateur de comparaison < sur le type somme suivant ?

```
type a = {a: int ; b: float}
```

Exercice 4. Évaluation paresseuse

De manière similaire à Python, OCaml évite quand possible les expressions qui se situent à droite d'un opérateur booléen.

```
false && (print_string "boum" ; true)
```

```
true || (print_string "boum" ; false)
```

Ainsi, on peut s'en servir pour interrompre plus tôt un programme :

```
let rec trouver x liste = match liste with
| [] -> false
| p::q -> p = x || trouver x q
```

Tous les appels récursifs n'auront pas lieux : dès qu'on atteint un élément qui est égal à x, `p = x` est évalué à vrai, et l'expression `trouver x q` n'est donc pas calculée.

1. Dans quels cas l'évaluation paresseuse a lieu pour les opérateur && et || ?
2. En déduire une implémentation d'une fonction verifier_tous de signature bool list -> bool qui vérifie si tous les éléments d'une liste sont égaux à true qui arrête le parcours dès que le résultat peut être déduit.
3. En déduire une implémentation d'une fonction verifier_un de signature bool list -> bool qui vérifie si au moins un élément d'une liste est égal à true qui arrête le parcours dès que le résultat peut être déduit.

Exercice 5. Référence

1. Proposer un type 'a maref qui corresponde à une référence.
   On pourra utiliser le mot-clef mutable dans un type produit.
2. Proposer une foction acceder de signature 'a maref -> 'a qui accède à la valeur contenue dans un tel type.
3. Proposer une fonction affecter de signature 'a maref -> 'a -> unit qui modifie la valeur d'une valeur contenue dans un tel type.
4. En déduire une fonction echanger de signature 'a maref -> 'a maref -> unit qui échange la valeur contenue dans deux variables d'un tel type.
   On utilisera les fonction précédemment définies.

Exercice 6. Calcul qui peut ne pas aboutir avec le type option

On veut utiliser le type float option pour représenter le résultat d'un calcul sur les flottants qui peut aboutir ou pas.

Essentiellement, on cherche à pouvoir effectuer les calculs normalement (en rajoutant des Some partout) :

```
ajouter (Some 1.0) (Some 2.0) (* Some 3.0 *)
```

Mais aussi renvoyer None quand le calcul n'aboutit pas :

```
diviser (Some 1.0) (Some 0.0) (* None *)
```

Quand on a obtenu un None, on veut que l'impossibilité de faire le calcul se propage.

```
multiplier (None) (Some 0.0) (* None *)
```

1. Proposer une fonction ajouter de signature float option -> float option -> float option qui calcul le résultat de la somme des deux options de flottants si les deux ne sont pas None, et qui renvoie None sinon.

   ```
   ajouter (Some 1.0) (None) (* None *)
   ```

   De la même manière on peut construire les fonctions multiplier et soustraire.
2. Proposer une fonction diviser de signature float option -> float option -> float option qui réalise la divison de deux options de flottants quand possible, et renvoie None sinon.
3. Proposer une implémentation de la fonction racine.
4. Proposer une implémentation de la fonction plus_grand de signature float option -> float option -> bool option qui renvoie si possible si la première entrée et plus grande que la seconde, et None sinon.

Exercice 7. Flottants ou entier

On désire construire un type qui permette d'avoir soit des entiers, soit des flottants.

L'idée est de procéder comme en Python, si on fait des opérations d'addition, de soustraction ou de multiplication avec au moins un flottant, on reste dans les flottants, et sinon on travaille dans les entiers.

1. Proposer un type `nombre` qui puisse contenir un entier ou un flottant.
2. Proposer une fonction `ajouter` de signature `nombre -> nombre -> nombre` qui ajoute deux variables de ce type.
3. De la même manière proposer des fonctions `soustraire` et `multiplier`.
4. Proposer une fonction `diviser`, lors de la division, on passe dans les flottants si la deuxième entrée ne divise pas la première.
5. Proposer une fonction `plus_grand` de signature `nombre -> nombre -> bool` qui renvoie `true` si et seulement si le premier élément est plus grand que le second argument
6. Proposer une fonction `tronquer` de signature `nombre -> nombre` qui convertit un nombre flottant vers un entier.

Exercice 8. Ordinaux de Von Neumann

Il existe plusieurs manière de construire les entiers, et l'une d'entre elle qui s'inscrit dans la théorie des ensembles sont les entiers de Von Neumann.

L'idée, comme pour la plupart des constructions, est de fonctionner par récurrence : on a besoin d'un 0, et d'une manière d'avoir le successeur d'un entier.

On note E_n l'ensemble qui représente un entier n. 0 est représenté par l'ensemble vide $E_0 = \varnothing$, et le successeur d'un entier E_{n+1} est égal à $E_n \cup \{E_n\}$.

Ainsi, on a :

$$E_0 = \varnothing$$
$$E_1 = \{\varnothing\}$$
$$E_2 = \{\varnothing, \{\varnothing\}\}$$
$$\dots$$

1. Quel est le cardinal de E_n pour tout $n \in \mathbb{N}$?
2. Que peut-on dire des éléments de E_n ?
3. On propose d'utiliser le type suivant `type entier = Vide | Union of entier * entier` pour représenté les ordinaux de Von Neumann : $\varnothing$ est représenté par `Vide` et l'union $x \cup f$ est représentée par `Union (e,f)`.

   Montrer que pour un ensemble qui représente un entier, soit cet ensemble est l'ensemble vide, soit il existe une représentation d'ensemble `e` telle qu'il soit égale à `Union(e, e)`.
4. Proposer une fonction `successeur` de signature `entier -> entier` qui donne le successeur d'un entier passé en argument.
5. Proposer une fonction `ajouter` de signature `entier -> entier -> entier` qui construit l'entier résultant des deux entiers en argument.
6. Proposer une fonction `entier_de_int` de signature `int -> entier` qui convertit un entier OCaml vers la représentation d'un entier de Von Neumann.
7. Proposer une fonction `int_de_entier` qui fait la conversion inverse.
8. Proposer une fonction `multiplier` de signature `entier -> entier -> entier` qui renvoie la représentation de deux entiers passés en arguments.

L'autre méthode, plus proche de la logique de définir les entiers, sont les entiers définis par les axiomes de Peano.

Exercice 9. Entiers de Church

Dans la théorie du Lambda-calcul, une manière de représenter les entier sont les entiers de Church.

Les termes du lambda-calcul peuvent être représentées par le type suivant :

```
type terme = Var of string | App of terme * terme | Abs of string * terme
```

Un lambda-termes est donc soit une variable notée x (ou y, ou z, ou tout autre symbole), soit une application de deux lamdba-terme t_1 et t_2 notée $t_1\ t_2$, soit une abstraction à partir d'une variable x et un lambda-terme t noté $\lambda\, x.t$.

Ces termes portent un sens dans le lambda-calcul, mais essentiellement, il s'agit de la base de la programmation fonctionelle, la syntaxe $\lambda\, x.t$ correspondant à la définition d'une fonction qui à un terme t' associe t où toutes les instances de x ont été remplacées par t'.

Pour ce qui est de représenter les entiers, l'idée est de représenter 0 par le terme $\lambda\, f \lambda\, x.x$, qui est représenté par :

```
Abs("f",Abs("x", Var("x")))
```

Et ensuite on itère f un nombre de fois égal à l'entier qui nous intéresse. Ainsi, 1 est représenté par $\lambda\, f \lambda\, x.f\ x$, qui est représenté par :

```
Abs("f",Abs("x", App(Var("f"),Var("x"))))
```

2 est représenté par $\lambda\, f \lambda\, x.f\ (f\ x)$ qui est représenté par :

```
Abs("f",Abs("x", App(Var("f", App(Var("f"),Var("x"))))))
```

Ainsi, en s'accordant la notation d'itération $f^{(n)}\ x$ équivalente à $f\ (...(f\ x))$ où f est itére n fois, on peut écrire que n est représenté par $\lambda\, f \lambda\, x.f^{(n)}\ x$.

1. Proposer une fonction `succ` de signature `terme -> terme` qui construit le successeur d'un entier.

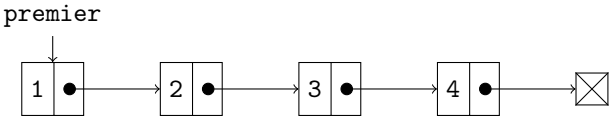
   On fera attention à utiliser le même nom de variable que ceux représentés dans l'entier en entrée : ainsi `succ Abs("g",Abs("y", App(Var("g"),Var("y"))))` *doit renvoyer* `Abs("g",Abs("y", App(Var("g", App(Var("g"),Var("y")))))))`.
2. Proposer une fonction `ajouter` de signature `terme -> terme -> terme` qui construit la somme de deux entiers.
3. Proposer une fonction `church_de_int` de signature `int -> terme` qui convertit un entier OCaml vers sa représentaton en entier de church.
4. Proposer une fonction `int_de_church` qui fait la conversion inverse.
5. Proposer une fonction `multiplier` de signature `terme -> terme-> terme` qui construit le produit de deux entiers.

Il y a beaucoup de choses à dire sur le lambda-calcul, mais nous pouvons nous contenter de cela pour le moment. Le lambda-calcul n'est pas au programme de Prépa, mais il s'agit de la base qui justifie beaucoup de choix faits en OCaml, dont le fait d'avoir l'application à l'aide l'espace.

2 Structures linéaires

Exercice 10. Liste chaînée mutable

On se concentre sur les listes chaînées, on rappelle qu'une liste chaînée est une succession de cellules telles que l'on peut obtenir la suivante à partir de chaque cellule.



On utilise l'implémentation de liste chaînée mutable suivante :

```
type 'a listmut = Vide | Cell of {valeur: 'a ; mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a listmut}
```

Ainsi, le type `'a liste_chainee` est une référence vers une succession de cellules terminée par le constructeur `Vide`.

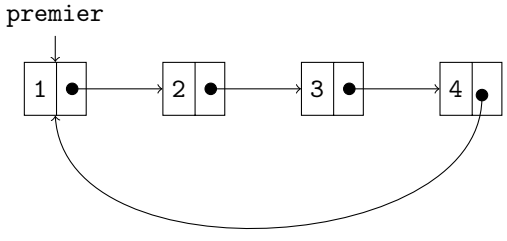
L'intérêt d'avoir un champ mutable qui référence le premier élément est de pouvoir rendre totalement mutable la liste chaînée.

De cette manière, on peut modifier la liste chaînée soit en modifiant la variable mutable dans le champs `premier`, soit en modifiant les données qui se trouvent dans le champs `suivant` des cellules.

1. Comment construire une telle liste qui corresponde à la liste `[1; 2; 3]` ?
2. Proposer une fonction `liste_vide` de type `unit -> 'a liste_chainee` de sorte à ce que `liste_vide ()` renvoie une liste chaînée vide.
3. Proposer une fonction `longueur` de signature `'a liste_chainee -> int` qui renvoie la longueur d'une liste passée en argument.
4. Proposer une fonction `affichier_liste` de signature `int liste_chainee -> unit` qui affiche le contenu d'une liste chainée.
5. Proposer une fonction `ajouter_en_tete` de signature `'a liste_chainee -> 'a -> unit` qui rajoute un élément passé en argument en tête d'une liste chaînée passée en argument en modifiant cette liste chaînée.
6. Proposer une fonction `ajouter_fin` de signature `'a liste_chainee -> 'a -> unit` qui rajoute un élément passé en argument à la fin d'une liste chaînée passée en argument en modifiant cette liste chaînée.
7. Proposer une fonction `retirer_premier` de signature `'a liste_chainee -> unit` qui retire le premier élément d'une liste chaînée.
8. Proposer une fonction `retirer_dernier` de signature `'a liste_chainee -> unit` qui retire le dernier élément d'une liste chaînée.
9. Proposer une fonction `retirer` de signature `'a liste_chainee -> 'a -> unit` qui retire un élément passé en argument d'une liste chaînée passée en argument. On pourra supposer que l'élément est présent au plus une fois dans la liste.
10. Proposer une fonction `range` de signature `int -> 'a liste_chainee` qui à partir d'un entier n renvoie la liste chainée dont les éléments dans l'ordre sont de 0 à $n-1$.
11. Proposer une fonction `vers_liste` de signature `'a liste_chainee -> 'a list` qui convertit une liste chaînée vers une liste OCaml classique.
12. Proposer une fonction `depuis_liste` de signature `'a list -> 'a liste_chainee` qui convertit une liste classique OCaml en liste chaînée.
13. Proposer une fonction `inverser` de signature `'a liste_chainee -> unit` qui modifie une liste chaînée pour obtenir la liste inversée.
14. Proposer une fonction `concatener` de signature `'a liste_chainee -> 'a liste_chainee -> unit` qui modifie la première liste passée en argument de sorte à ce que la seconde lui soit concaténée.
15. Quelle est la complexité de chacune de ces opérations ?
16. Proposer une fonction `modifier` de signature `'a liste_chainee -> int -> 'a -> unit` qui modifie une liste chaînée à l'indice passé en argument pour avoir une valeur égale à une valeur passée en argument.

Exercice 11. Liste chaînée cyclique

Il est parfois intéressant d'avoir une liste chaînée cyclique au lieu d'une liste chaînée simple.



On utilise l'implémentation de liste chaînée cyclique suivante :

```
type 'a listmut = Vide | Cell of {valeur: 'a ; mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a listmut}
```

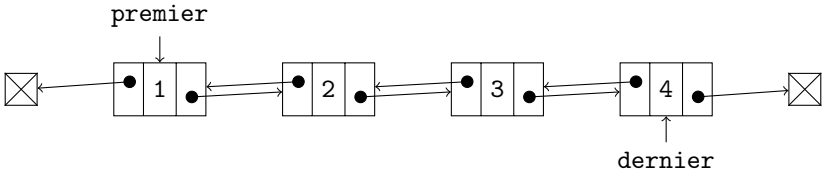
Mêmes questions qu'à l'exercice précédent.

On pourra utiliser l'égaliter physique == au lieu de l'égalité structurelle = pour vérifier si on est revenu à la première cellule.

Exercice 12. Liste doublement chaînée

On peut être intéressé par, au lieu d'avoir une liste simplement chaînée, une liste doublement chaînée dans les deux sens de sorte à pouvoir descendre ou remonter dans une liste en fonction de ce qui est nécessaire.

On dispose d'un accès au premier élément, et au dernier élément directement, qui nous permet de choisir le sens dans lequel nous voulons faire un parcours.



On utilise l'implémentation de liste doublement chaînée suivante :

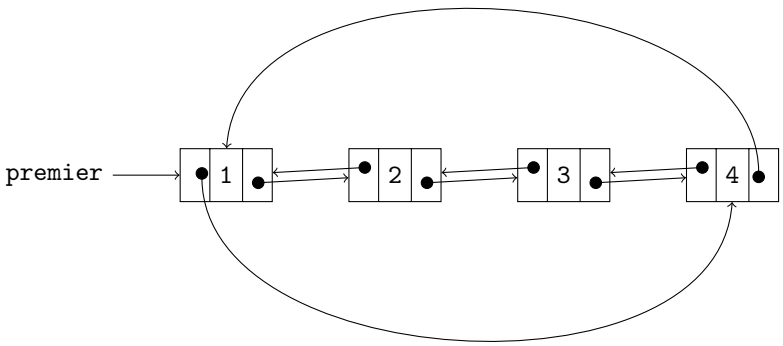
```
type 'a listmut = Vide | Cell of {valeur: 'a ; mutable suiv: 'a listmut ; mutable
    prec: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a listmut, mutable dernier : 'a listmut}
```

Le champs premier permet d'avoir accès au premier élément de la liste (qui est potentiellement le vide), et dernier permet d'avoir accès au dernier élément de la liste (qui est le vide si et seulement si le premier élément est le vide).

Mêmes questions qu'à l'exercice précédent.

Exercice 13. Liste doublement chaînée cyclique

La liste chaînée doublement cyclique permet d'avoir les avantages d'une liste chaînées cyclique et d'une liste doublement chaînée.



On utilise l'implémentation de liste doublement chaînée suivante :

```
type 'a listmut = Vide | Cell of {valeur: 'a ; mutable suiv: 'a listmut ; mutable
    prec : 'a listmut}
type 'a liste_chainee = {mutable premier : 'a listmut}
```

Mêmes questions qu'à l'exercice précédent.

Exercice 14. Détection de cycle avec l'algorithme du lièvre et de la tortue

On reprend le type que l'on a utilisé pour les listes simplement chaînées mutables cyclique et non cyclique :

```
type 'a listmut = Vide | Cell of {valeur: 'a ; mutable suiv: 'a listmut}
type 'a liste_chainee = {mutable premier : 'a listmut}
```

On remarque qu'on a pas besoin d'avoir une structure strictement linéaire en rajoutant par exemple des cycles (très justement ce que l'on a fait pour les listes cyclique), et on aimerait pouvoir détecter ces cycles.

Un cycle est une suite de cellule $a_0....a_n$ telle que $a_0 = a_n$ et telle que pour tout $0 \leqslant i \leqslant n$ a_i ait pour suivant a_{i+1}. Si les a_i sont à deux à deux distincts sauf pour a_0 et a_n, on dit que ce cycle est de longueur n.

Attention : un cycle ne commence pas nécessairement à la première cellule de la liste chaînée.

1. Proposer une implémentation d'un cycle de longeur 1.
2. Proposer une fonction de signature `'a liste_chainee -> bool` qui renvoie si une liste chaînée contient un cycle en utilisant une liste qui contient toutes les cellules vus depuis le début du parcours.

   On pourra utiliser le test d'égalité physique ==.
3. Quelle est la complexité temporelle de la fonction proposée dans le pire des cas en fonction du nombre de cellules accessibles dans la liste chaînée ?
4. Quelle est la complexité spatiale dans le pire des cas en fonction du nombre de cellules accessible dans la liste chaînée ?

   Un algorithme un peu plus efficace est l'algorithme du lièvre et de la tortue qui consiste à utiliser deux parcours de la liste en parrallèle, l'un qui avance d'une cellule à la suivante à chaque étape (la tortue), et l'autre qui avance d'une cellule à la suivante de sa suivante à chaque étape (le lièvre). L'algorithme conclut que la liste comprenait un cycle si jamais le lièvre et la tortue arrivent sur la même case simultanément à une étape après la première. Si l'algorithme arrive à la fin de la liste (c'est-à-dire lorsque l'une des cellules suivantes est vide), il conclut qu'il n'y a pas de cycles.
5. On dit qu'une suite (u_n) est ultimement périodique s'il existe $N \in \mathbb{N}$ et $k \in \mathbb{N}$ tels que pour tout $n \geqslant N$, on ait $u_{n+k} = u_n$.

   Ainsi, une suite ultimement périodique est une suite qui est périodique à partir d'un certain rang.

   On considère les éléments dans l'ordre (c_n) des cellules en prenant c_0 la première cellule de la liste chaînée, et en prenant c_{n+1} égal à la cellule suivante de la cellule c_n.

   Montrer que les (c_n) forment une suite ultimement périodique si et seulement si la liste chaînée comprend un cycle.

   Pour une suite ultimement périodique, on nomme λ le plus petit N à partir duquel la suite est périodique, et on nomme μ le plus petit k tel qu'à partir de ce λ, on ait $c_{n+k} = c_k$.

   λ correspond aux éléments à l'extérieur du cycle, et μ correspond à la plus petite période dans le cycle, c'est-à-dire au nombre d'éléments dans le cycle.
6. On se donne une suite (u_n) telle que pour tout i et j, si $u_i = u_j$ alors $u_{i+1} = u_{j+1}$.

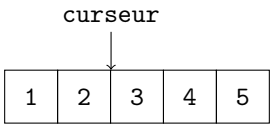
   Montrer que (u_n) est ultimement périodique si et seulement si il existe $n \geqslant 1$ tel que $u_n = u_{2n}$.

7. En déduire que l'algorithme du lièvre et de la tortue est correcte et termine.
8. Montrer que le plus petit $n \geqslant 1$ tel que $u_n = u_{2n}$ noté n_0 est majoré par $\lambda + \mu$.
9. En déduire une complexité asymptotique pour la complexité temporelle de l'algorithme du lièvre et de la tortue. Quelle est sa complexité spatiale ?
10. Proposer une implémentation du lièvre et de la tortue qui vérifie s'il y a un cycle dans une liste chaînée.

Cet algorithme est aussi parfois algorithme de détection de Floyd.

Exercice 15. Zipper

Un zipper est une structure persistante qui permet de représenter les éléments d'une liste où on a rajouté un curseur de sorte à pouvoir avoir un accès privilégié aux informations au voisinage de notre curseur.



Pour représenter le zipper, on utilise une liste pour les éléments à gauche du curseur, ordonné de droite à gauche, et une liste pour les éléments à droite du curseur, avec les éléments de gauche à droite.

On utilise le type suivant pour représenter un zipper :

```
type 'a zipper = {gauche : 'a list ; droite : 'a list}
```

Ainsi, le zippepr représenté en haut est représenté par une valeur de type `int zipper` égal à :

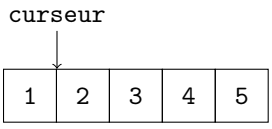
```
{gauche = [2;1] ; droite = [3; 4; 5]}
```

.

On remarque qu'il s'agit bien d'une structure persistante : elle n'est pas mutable est on doit donc reconstruire un nouvel état.

1. Proposer une fonction `creer_zipper` de signature `unit -> 'a zipper` de sorte à ce que `creer_zipper ()` renvoie le zipper vide.
2. Proposer une fonction `vers_gauche` de signature `'a zipper ->'a zipper` qui déplace le curseur vers la gauche.

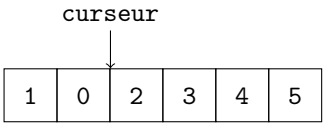
   Par exemple, après avoir appliqué `vers_gauche` au zipper précédent, on obtient le zipper suivant :



   De même on peut construire une fonction `vers_droite` qui déplace le curseur vers la droite.
3. Un des intérêt d'un zipper est de pouvoir modifier facilement les données aux alentours du point.

   Proposer une fonction `ajouter`de signature `'a zipper -> 'a -> 'a zipper` qui ajoute un élément à gauche du curseur.

   Après l'ajout de 0 à gauche du curseur représenté dans la question précédente, on obtient :



4. De la même manière proposer une fonction `retirer` qui retire un élément directement à gauche du curseur.

5. Proposer une fonction qui transforme un zipper en une liste en conservant l'ordre des éléments.
6. On peut se servir d'un zipper pour simuler une bande infinie : au lieu de renvoyer une erreur quand on essayer de se déplacer dans une direction où la liste est vide, on complète avec un élément par défaut qui correspond à un élément vide (par exemple 0 pour les entiers).

   Proposer des versions modifiées de `vers_gauche` et `vers_droite` qui prennent en argument la valeur par défaut et rajoute cette valeur à gauche ou à droite de sorte à toujours pouvoir réaliser le mouvement.
7. Le problème de cette implémentation, est que si on effectue de nombreux déplacement à gauche, puis qu'on revient à droite, on a effectivement rempli beaucoup d'éléments par défaut qui prennent de la place dans la mémoire.

   On aimerait pouvoir effacer les valeurs à gauche ou à droite s'il ne s'agit que d'éléments par défaut.

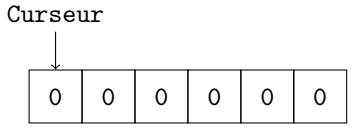
   Proposer un type modifié pour zipper, ainsi qu'une version modifié des fonctions précédentes pour garder l'invariant que, à la fin d'une opération, si tous les éléments sont par défaut dans une liste à gauche ou à droite, alors cette liste doit être vide.

   On pourra rajouter une information sur le nombres d'éléments à gauche et à droite qui ne sont pas la valeur par défaut.

Exercice 16. Interprétation d'un langage exotique

Le langage BF est un langage exotique qui est composé de huits instructions et qui opère sur une bande mémoire.

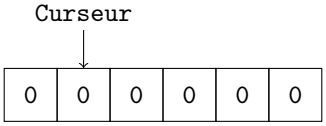
Initialement, la bande mémoire est composée intrégralement d'entiers égaux à 0. On suppose qu'elle est de taille n fixe, et nous utiliserons un OCaml un tableau d'entiers de type `int array`. Initialement, un curseur pointe vers l'élément d'indice 0.



Les entiers sont supposés positifs, et on s'assurera qu'ils le reste dans le code.

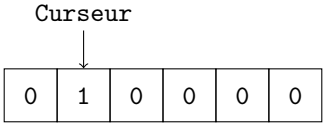
Un programme BF est une chaîne de caractère qui contient les instructions suivantes. Lors de l'exécution on lit les caractères dans l'ordre jusqu'à arriver à la fin du programme.

$>$ déplace le curseur vers la droite d'une case. Ainsi, depuis la situation précédente, on obtient la disposition suivante :



$<$ réalise l'opération inverse, décalant le curseur vers la gauche d'une case.

$+$ incrémente la case indiquée par le curseur de 1. À partir de la situation précédente, on obtient la situation suivante :



– réalise l'opération inverse qui décrémente la case indiquée par le curseur de 1.

. imprime sur la sortie standard le contenu de la case indiquée. Dans notre cas, nous utiliserons un `print_int` pour afficher la valeur dans la case.

[et] permettent de faire des boucles. Le crochet ouvrant saute au crochet fermant associé (en terme de parenthèsage) si la valeur de la case sous le curseur est 0, et le crochet fermant saute au crochet ouvrant associé si la valeur de la case sous le curseur est différente de 0.

, demande une entrée et met le résultant dans la case indiquée par le curseur. En OCaml on utilisera `read_line` de signature `unit -> string` qui lit la ligne écrite en entrée dans la console (et `int_of_string`).

Usuellement, les entrées et sorties en BF sont codés en ASCII. Dans notre cas, on affichera simplement les nombres tels quels, et de même pour récupérer des informations.

Par exemple le programme BF ,[–>+<]>. lit un nombre en entrée, fait une boucle pour le déplacer dans la case suivante (en mettant la première case à 0), puis renvoie cette valeur.
Le programme ,>,<[–>+<]>. lit deux entiers mis dans les deux première case, puis augmente la deuxième case d'une valeur égale à la première puis renvoie la somme des entiers.

1. Proposer un programme BF qui lit un nombre, et affiche 1 si ce nombre est non nul, et 0 sinon.
2. Que fait le programme BF suivant ,>,[<–>–]<. ?
3. On cherche désormais à réaliser un interpréteur pour le langage BF, c'est-à-dire un programme qui exécute un programme BF en gérant la mémoire et les différentes entrées et sorties.
   (a) Proposer une fonction `trouver_fermant` de signature `string -> int -> int` qui à l'aide d'un programme BF et d'un indice qui correspond à un crochet ouvrant dans le programme, trouve l'indice du crochet fermat associé dans le programme :

```
trouver_fermant ",[+>]" 1 (* 4 *)
```

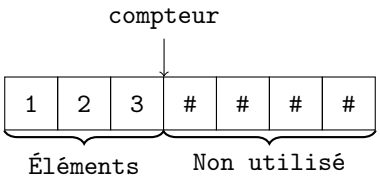
```
trouver_fermant ",[+[+>]]" 1 (* 7 *)
```

   (b) De même, proposer une fonction `trouver_ouvrant` qui trouve l'indice du crochet ouvrant associé à un crochet fermant dans le programme.
   (c) En déduire une fonction `interpreter` de signature `string -> int -> unit` qui à partir d'un programme BF et d'une taille n de tableau mémoire interprète le programme BF sur une mémoire de cette taille.
4. On aimerait ne pas calculer à chaque fois la parenthèse qui correspond à une autre lors de l'exécution du programme. Comment pourrait-on réaliser un pré-calcul à l'aide d'une pile de sorte à ce que l'on puisse ensuite accéder à la parenthèse correspondante en $O(1)$? On aimerait que ce pré-calcul soit en $O(n)$ où n est la longueur du programme BF.
5. Pour l'instant, la bande mémoire est de taille fixe : si le programme BF a besoin d'une valeur à un indice inférieur à 0, ou bien d'une valeur à un inidice supérieur à n, ce que l'on ne peut pas savoir à priori sans exécuter le programme pour un programme arbitraire, notre code va planter.
   En utilisant une structure de zipper telle que présentée à l'exercice précédent, proposer une version d'un interpréteur qui utilise une bande mémoire virtuellement infinie.

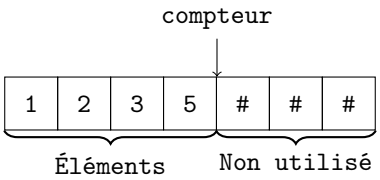
3 Piles, Files

Exercice 17. Pile avec un tableau

On propose d'implémenter une pile avec un tableau. Le tableau est de taille fixe, mais on se souviendra du nombre d'élément dans la pile à l'aide d'un compteur.



Ainsi, après, l'ajout d'un élément 5 dans la pile précédente, on obtient :



Les valeurs qui se trouvent dans le tableau à droite du compteur (ici indiqués par des #) ne sont pas importantes : on ne peut pas y accéder, mais on peut les modifier si on les remplace en ajoutant des éléments dans la pile.

On utilisera le type de pile suivante :

```
type 'a pile = {contenu : 'a array ; mutable nb : int}
```

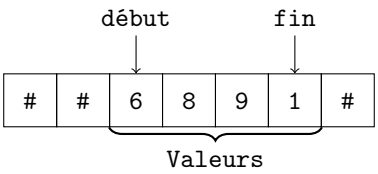
1. Proposer une fonction `creer_pile` de signature `int -> 'a -> 'a pile` qui à partir d'une taille de tableau et d'une valeur par défaut renvoie une pile vide.
2. Proposer une fonction `est_vide` de signature `'a pile -> bool` qui renvoie si une pile passée en entrée est vide.
3. Proposer une fonction `empiler` de signature `'a pile -> 'a -> unit`qui ajoute un élément dans une pile.
   On lèvera une erreur si le tableau est déjà plein.
4. Proposer une fonction `depiler` de signature `'a pile -> 'a` qui retire un élément d'une pile et le renvoie.
   On lèvera une erreur si la pile est vide.
5. Quelle est la complexité de chacune de ces opérations ?

Exercice 18. Deux piles sur un même tableau

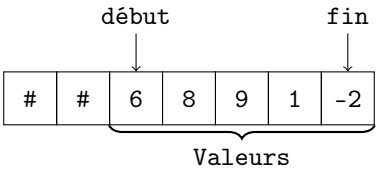
Proposer une implémentation de deux piles à l'aide d'un tableau unique de sorte à ce qu'il n'y ait de dépassement de mémoire que si la somme des tailles des deux piles soit supérieur à la taille du tableau. L'ajout et le retrait dans chacune des piles devra être en $O(1)$ dans le pire des cas.

Exercice 19. File avec un tableau circulaire

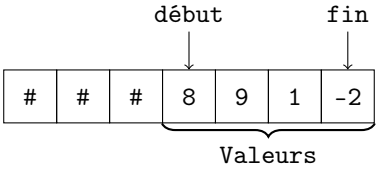
On propose d'implément une file avec un tableau circulaire : on utilise un tableau



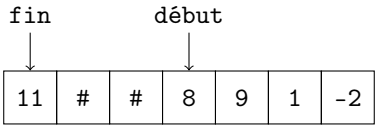
Après enfilement d'un élément −2, on obtient :



En défilant un élément, on obtient :



Enfin, si on rajoute encore un élément 11, on décale le compteur de fin vers le début du tableau :



Par ailleurs, pour pouvoir indiquer quand la file est vide, on suppose que la valeur contenue dans `fin` est égale à -1 si et seulement si la file est vide.

Les valeurs dans les cases qui ne sont pas entre début et fin (ici indiqués par des #) ne sont pas importantes.

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

1. Proposer une fonction `creer` de signature `int -> 'a -> 'a file` qui à partir d'une taille de tableau et d'une valeur par défaut renvoie une file
2. Proposer une fonction `est_vide` de signature `'a file -> bool` qui renvoie si une file est vide.
3. Proposer une fonction `enfiler` de signature `'a file -> 'a -> unit` qui enfile un élément passé en argument dans une file passée en argument.
4. Proposer une fonction `defiler` de signature `'a file -> 'a` qui retire le premier élément dans la file et le renvoie.
5. Quelle est la complexité de ces opérations ?

Exercice 20. File avec une liste doublement chaînée

À l'aide d'une liste doublement chaînée, proposer une implémentation d'une file. Quelles sont les complexités des opérations `creer_file`, `enfiler`, `defiler` et `est_vide` ?

Exercice 21. Pile de file

On suppose qu'on dispose d'une implémentation de file et des fonctions `creer_file`, `est_file_vide`, `defiler` et `enfiler`.

1. Proposer une implémentation d'une pile et des fonctions `creer_pile`, `est_pile_vide`, `depiler` et `empiler`.

   On pourra utiliser deux files, une pour stocker les valeurs, et l'autre pour stocker temporairement les données lors du parcours. On veillera à ne pas utiliser plus qu'une mémoire constante lors de l'implémentation.
2. Quelle est la complexité des opérations proposées en supposant que l'implémentation de file donne une complexité constante pour toutes les opérations sur les files ?

Exercice 22. File de pile

On suppose qu'on dispose d'une implémentation de pile et des fonctions `creer_pile`, `est_pile_vide`, `depiler` et `empiler`.

1. Proposer une implémentation d'une file et des fonctions `creer_file`, `est_file_vide`, `defiler` et `enfiler`.
2. Quelle est la complexité des opérations proposées en supposant que l'implémentation de pile donne une complexité constante pour toutes les opérations sur les piles ?
3. Comment avoir une implémentation d'une file de complexité amortie efficace à partir de deux piles ?

Exercice 23. Pile et file à l'aide d'un tableau dynamique

Dans cet exercice, il n'est pas nécessaire de réaliser l'implémentation exacte en OCaml, mais il est nécessaire de décrire exactement comment fonctionnent les différentes opérations.

1. À l'aide d'un tableau dynamique, proposer une implémentation d'une pile. Quels sont les complexités dans le pire des cas et amorties des opérations d'empilement et de dépilement ?
2. En vous inspirant d'une implémentation d'un tableau dynamique et de l'implémentation d'une file à l'aide d'un tableau circulaire, proposer une implémentation d'une file. Quelles sont les complexités dans le pire des cas et amorties des opérations d'enfilement et de défilement ?

Exercice 24. File d'attente à double extremités

Une file d'attente à double extrémités (parfois nommée deque par anglicisme) est une structure linéaire dans laquelle on peut ajouter et retirer des éléments aux deux extrémités. Il y a donc 4 opérations pour la modifier `ajouter_gauche`, `ajouter_droite`, `retirer_gauche` et `retirer_droite`.

1. Proposer une implémentation d'une file d'attente à double extrémités qui utilise une liste doublement chaînée.
2. Proposer une implémentation d'une file d'attente à double extrémités qui utilise une liste circulaire.
3. À l'aide de l'implémentation d'une file d'attente à double extrémités, implémenter une pile.
4. À l'aide de l'implémentation d'une file d'attente à double extrémités, implémenter une file.

Exercice 25. Vérification de parenthèsesage

On s'intéresse à une séquence de parenthèses ouvrantes et fermantes, de crochets ouvrants et fermants, et d'accolades ouvrantes et fermantes. On veut vérifier que le parenthèsage est correcte au sens où une parenthèse, un crochet ou une accolade ouvrante doit être fermée par le symbole fermant associé dans la suite de la chaîne et toutes les parenthèses, crochets ou accolades qui ont été ouvertes entre temps doivent avoir été fermées.

En utilisant une Pile, proposer une fonction `verifier` de signature `string -> bool` qui vérifie si une chaîne de caractère correspond à un parenthèsage correcte.

```
verifier "{}([]())" (* true *)
```

```
verifier "(}" (* false *)
```

```
verifier "({)}" (* false *)
```

Exercice 26. Dérécursivation

On cherche à transformer une fonction récursive en fonction non-récursive en faisant manuellement la simulation de la pile des appels.

La fonction que l'on cherche dérécursiver est l'implémentation naïve du calcul de Fibonacci :

```
let rec fibo n = match n with
| 0 -> 0
| 1 -> 1
| _ -> fibo (n-1) + fibo (n-2)
```

Pour ce faire, on utilise une pile de type `'a pile` pour laquelle on suppose qu'on a une fonction `creer_pile`, `empiler` et `depiler`. On suppose qu'on dispose aussi de fonctions `est_vide` et `regarder` qui respectivement vérifie si la pile est vide ou renvoie la valeur au sommet de la pile sans la retirer.

Le type des éléments sur la pile est le suivant :

```
type pelem = Appel of int | Resultat of int
```

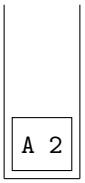
Ainsi, les éléments sur la pile sont soit des appels, soit des résultats.

Au début, on ajoute dans la pile l'appel qui est nécessaire, et ensuite, à chaque étape, on retire l'élément au sommet de la pile :

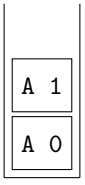
— Si c'est un appel, on le remplace soit par le résultat de l'appel si c'est un cas de base (c'est-à-dire 0 ou 1), soit on rajoute deux appels avec $n-1$ et $n-2$.

— Si c'est un résultat, on a plusieurs cas possibles : soit la pile est vide après ce résultat et on peut renvoyer ce resultat car c'était le résultat du dernier appel ; soit le sommet de la pile aussi un résultat, et dans ce cas, on doit rajouter la somme des deux résultats ; soit c'est un appel et dans ce cas on retire l'appel, ajoute le résultat, puis rajoute l'appel qu'on a retiré pour permettre de le traiter.

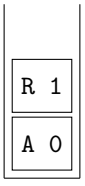
Par exemple, dans le calcul de `fibo 2`, on commence par ajouter `Appel(2)` (on représentera les appels avec l'argument x par « A x », et les résultats de x par « R x ») :



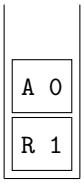
Ensuite, on regarde au sommet de la pile : il y a un appel qui n'est pas un cas de base, on le remplace par ses deux appels récursifs (peu importe l'ordre) :



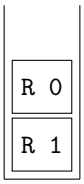
On regarde à nouveau au sommet de la pile : c'est un appel qui est un cas de base, et on peut le remplacer par sa valeur :



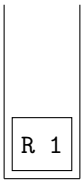
On retire cette valeur et on regarde le sommet de la pile, on a un appel en dessous d'un résultat : on doit inverser les deux en mettant le résultat puis l'appel dans la pile de sorte à obtenir :



On remplace l'appel par sa valeur de base :



On a un résultat au dessus d'un résultat : on peut remplacer le résultat par la somme des deux résultats, on obtient la pile suivante :



On a un résultat seul dans la pile, c'est le résultat qu'on renvoie.

Proposer une implémentation dérécursivée de Fibonacci de cette manière.

Exercice 27. Pile pour l'expression d'un calcul avec opérateur postfix

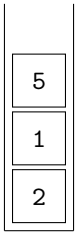
L'écriture post-fixe consiste à mettre les opérateurs après leurs arguments. Par exemple, au lieu d'écrire $4 + 5$, on écrit $4\ 5\ +$. Lorsque plusieurs calculs s'enchaînent, on n'a pas besoin de parenthèses en faisant les caculs de gauche à droite. Par exemple, lorsqu'on lit $4\ 3\ +\ 2\ -$, on commence à calculer $4\ 3\ +$ qui est égal à 7, puis on fait le calcul de $7\ 2\ -$ qui donne 5.

Une méthode pour évaluer une expression arithmétique post-fixe, on utiliser une pile. On lit les éléments dans l'ordre de gauche à droite :

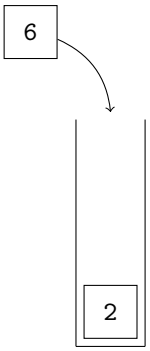
— Quand on voit un nombre, on l'ajoute à la pile ;
— si on voit un opérateur, on prend les deux éléments en haut de la pile, on fait le calcul, puis on remet le résultat dans le pile.

À la fin, il ne reste qu'un seul élément dans la pile : le résultat.

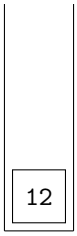
Par exemple, pour le calcul de $2\ 1\ 5\ +\ \times$, on commence par ajouter dans l'ordre, 2, 1, et 5 pour obtenir la pile suivante :



Puis on lit le +, ce qui nous contraint à récupérer les deux éléments du sommet de la pile 5 et 1, qui après ajout nous donne 6, on rajoute donc 6 à la pile :



Enfin, après la lecture de × on récupère les deux éléments au sommet de la pile 2, et 6 qu'on multiplie pour obtenir 12 que l'on remet dans la pile de sorte à obtenir :



Il ne reste bien qu'un élément dans la pile : c'est le résultat de l'expression.

On représente une expression postfixe avec le type suivant :

```
type lexeme = Val of int | Ajouter | Soustraire | Multiplier
type expression = lexeme list
```

Dans ce type, pour représenter 4 3 + 2 −, on a donc la représentation :

```
[Val(4); Val(3); Ajouter ; Val(2); Soustraire]
```

1. Proposer une fonction `evaluer` de signature `expression -> int` qui donne le résultat d'une expression passée en argument sous forme infixe.
2. Proposer une fonction `convertir` de signature `string -> expression` de sorte à ce que `convertir s` renvoie l'expression qui correspond

Exercice 28. Tri par base

On suppose qu'on a une implémentation d'une file qui permette d'enfiler et de défiler un élément en $O(1)$. On suppose qu'on dispose par ailleurs d'une opération de concaténation en $O(1)$.

On essaye d'utiliser cette structure pour trier des entiers qui disposent tous d'une même taille dans une base donnée à l'aide du tri par base.

On représente les entiers par un tableau de leur chiffres :

```
type entier = int array
```

Tous les entiers sont représentés par des tableaux de même tailles. On pourra supposer qu'il existe une constante qui contient la taille des entiers :

```
let taille_entier = 8
```

Les entiers sont représentés dans une base donnée qui est constante. On supposera qu'il existe une constante qui contient la taille de cette base :

```
let base = 10
```

Ainsi, dans ce cas, tous les entiers sont représentés par un tableaux de 8 entiers qui peuvent valoir de 0 à 9.

L'idée du tri est de trier les entiers selon chacun des chiffres en commençant par le chiffre *le moins important*. À chaque fois, on enfile les éléments dans l'ordre dans base files qui contiennent chacun les valeurs qui correspondent à un chiffre. Ensuite on concatène les files par ordre de chiffres croissants pour obtenir une nouvelle file *en conservant l'ordre des nombres dans chaque file*.

On répète ensuite les étapes en utilisant le deuxième chiffre le moins importants.

Si on veut trier les 3 valeurs en base 2 $\overline{11}^2$, $\overline{10}^2$ et $\overline{00}^2$, on commence par les ranger par leurs valeurs selon le chiffre le moins important (le dernier) pour obtenir les deux files suivantes :

$\overline{00}^2$

$\overline{11}^2$	$\overline{01}^2$

On concatène toutes les files par ordre croissant de chiffre, et on obtient :

$\overline{00}^2$	$\overline{11}^2$	$\overline{01}^2$

On redistribue les files cette fois-ci selon leur chiffre le plus important (celui de gauche), et on obtient :

$\overline{00}^2$	$\overline{01}^2$

$\overline{11}^2$

Enfin, on concatène les listes, et on obtient la file triée :

$\overline{00}^2$	$\overline{01}^2$	$\overline{11}^2$

On suppose que tous les éléments sont stockés dans une file.

1. Montrer qu'à la fin de l'étape k, les entiers sont triés en ne prenant en compte que les k chiffres les moins importants.
2. En déduire que le tri par base trie bien les éléments par ordre croissant.
3. Proposer une implémentation du tri par base à l'aide de ces structures. Le tri prendra en argument b la base utilisée et la file des éléments à trier. Le tri renverra la file triée.
4. Quelle est la complexité de ce tri en fonction de n le nombre d'éléments dans le tableau, b la base utilisée, et m la taille des entiers en terme de chiffres ?
5. Quelle implémentation d'une file pouvons nous utiliser pour avoir les opérations proposées avec leur complexité en début d'exercice ?