Exercice 1. Arithmétique de Peano

Dans l'arithmétique de Peano, les entiers naturels sont définis récursivement :

→ `Zero` est un entier de Peano.

→ Si `n` est un entier de Peano alors « `S n` » est un entier de Peano. L'objet « `S n` » est appelé le ***successeur*** de `n`.

Par exemple, 0 est représenté par « `Zero` », 1 est représenté par « `S Zero` », 2 est représenté par « `S (S Zero)` », 3 est représenté par « `S (S (S Zero))` » ... Notez que le fait de représenter les entiers de cette façon est très inefficace, mais l'efficacité n'est pas le but recherché.

1. Sans utiliser le type `int`, définir un nouveau type `intP` pour représenter les entiers de Peano.

Pour chaque question ci-dessous, vous devez écrire une fonction récursive mais pas de fonction intermédiaire.

2. Écrire une fonction « `est_nul: intP -> bool` » qui renvoie `true` si son argument est nul et `false` sinon.
3. Écrire une fonction « `add: intP -> intP -> intP` » qui renvoie la somme de ses paramètres d'entrée.
4. Écrire une fonction « `mult: intP -> intP -> intP` » qui renvoie le produit de ses paramètres d'entrée.
5. Écrire une fonction « `int_of_peano: intP -> int` » qui renvoie l'entier correspondant à son paramètre d'entrée.
6. Écrire une fonction « `peano_of_int: int -> intP` » qui renvoie l'entier de Peano correspondant à son paramètre d'entrée. Lorsque ce n'est pas possible, déclenchez une exception `Negatif` que vous aurez définie.
7. À l'aide des fonctions `int_of_peano` et `peano_of_int`, écrire deux nouvelles fonctions

   `add_bis: intP -> intP -> intP` `mult_bis: intP -> intP -> intP`

   pour calculer la somme et le produit de deux entiers de Peano.
8. À l'aide de la fonction `Random.int` et d'une boucle `while`, générer 10000 couples d'entiers `(n1, n2)` compris entre 0 et 200, et vérifier que les appels aux fonctions `add` et `add_bis`, ainsi que `mult` et `mult_bis` renvoient les mêmes valeurs. La boucle `while` s'arrêtera dès que deux valeurs sont différentes. La boucle doit prendre quelques secondes pour s'exécuter.

Exercice 2. Listes doublement chaînées circulaires (facultatif - difficile)

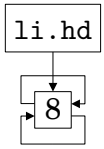

Figure 1

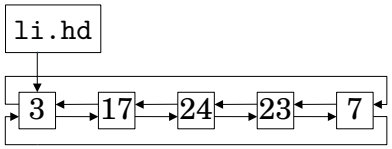

Figure 2

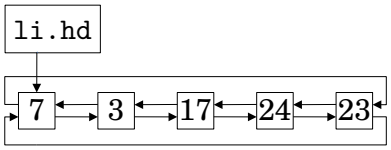

Figure 3

Une ***liste doublement chaînée circulaire*** (LDCC) contient des ***cellules*** composées d'un élément, d'une référence vers la cellule précédente et d'une référence vers la cellule suivante. Dans une LDCC, les cellules forment un cycle : la première cellule possède une référence vers la dernière cellule et inversement. En d'autres termes, on considère que la première cellule est la cellule suivante de la dernière cellule et que la dernière cellule est la cellule précédente de la première. Par exemple, la figure 1 présente une LDCC dont l'unique élément est 8 et la figure 2 présente une LDCC de longueur 5 dont les éléments sont $3, 17, 24, 23, 7$.

En OCaml, on utilisera le type « `'a ldcc` » pour représenter les listes doublement chaînées circulaires dont les éléments sont de type `'a`. Ce type est défini dans le fichier annexe disponible sur la page du cours :

`https://informatique-lhp.fr/opt-mpsi.html`

Le type « `'a option` » est prédéfini dans la bibliothèque standard, vous n'avez donc pas besoin de le redéfinir. Il permet d'indiquer de manière explicite la présence (avec `Some`) ou l'absence de valeur (avec `None`). Le champ `long` du type « `'a ldcc` » contient le nombre d'éléments de la liste. Le champ `hd` contient `None` si la liste est vide, et sinon il pointe vers la première cellule de la LDCC.

Dans le fichier annexe, la variable `li_fig1` représente la liste de la figure 1 dont l'unique cellule est `cell_fig1`. Notez que les champs `cell_fig1.suiv` et `cell_fig1.prec` pointent vers la cellule elle-même d'où la présence du « `rec` » dans la définition. Enfin, les fonctions `hd`, `get_long` et `del_ldcc` illustrent l'utilisation des LDCC, vous pouvez les réutiliser librement dans la suite.

1. (a) Écrire une fonction « `creer_ldcc : unit -> 'a ldcc` » qui renvoie une LDCC vide. Votre fonction s'exécutera en temps constant.
   (b) Définir une variable globale `li_Q1b` contenant une LDCC de longueur 3 dont les éléments sont 123, 456 et 789.

On souhaite maintenant ajouter et supprimer des éléments dans une LDCC. On se contentera d'écrire une fonction qui ajoute un élément au début de la liste et une fonction qui supprime le dernier élément de la liste. Pour ajouter ou supprimer un élément à un autre endroit, on commencera par effectuer une permutation sur les éléments de la liste. Par exemple, la figure 3 représente la liste obtenue après application d'une rotation droite sur les éléments de la LDCC de la figure 2.

2. (a) Écrire une fonction « `rot_droite : 'a ldcc -> unit` » qui effectue une rotation droite sur une LDCC. Notez que comme pour la plupart des fonctions de cet exercice, le type de retour de la fonction `rot_droite` est `unit`. Elle doit donc modifier la liste donnée en entrée et ne rien renvoyer. Votre fonction s'exécutera en temps constant.
   (b) Écrire une fonction « `ajout_ldcc : 'a ldcc -> 'a -> unit` » qui ajoute un élément à la fin d'une LDCC. Votre fonction s'exécutera en temps constant.
   (c) Écrire une fonction « `suppr_hd : 'a ldcc -> unit` » qui supprime le premier élément d'une LDCC. Votre fonction s'exécutera en temps constant.

L'un des intérêts des LDCC par rapport aux listes chaînées habituelles est qu'on peut concaténer deux LDCC en temps constant.

3. Écrire une fonction « `concat_ldcc : 'a ldcc -> 'a ldcc -> 'a ldcc` » qui prend en entrée deux LDCC notées `li1` et `li2`, et renvoie une liste contenant tous les éléments de `li1` suivis de tous les éléments de `li2`. On pourra supposer que `li1` et `li2` ne contiennent pas de cellule en commun. À la fin de la fonction, on détruira les deux listes initiales avec la fonction `del_ldcc`. Votre fonction s'exécutera en temps constant.
   Remarque : pour concaténer deux LDCC en temps constant, il faut nécessairement modifier les cellules de ces listes. C'est pour cela que la fonction `concat_ldcc` détruit `li1` et `li2` : leurs cellules ayant été modifiées, ces objets ne sont plus utilisables.

Lorsqu'on manipule une liste de type « `'a list` », il est très courant d'appliquer successivement une certaine fonction « `f: 'a -> unit` » aux éléments de la liste. La fonction « `List.iter: ('a -> unit) -> 'a list -> unit` » permet de faire cela. Par exemple, `somme_list` calcule la somme des éléments d'une liste d'entiers.

```
let somme_list (li: int list) =
  let res = ref 0 in
  let f x = res := !res + x in
  List.iter f li;
  !res;;
```

4. (a) Écrire une fonction « `iter_ldcc: ('a -> unit) -> 'a ldcc -> unit` » qui prend entrée une fonction `f` ainsi qu'une LDCC notée `li` et applique `f` à chacun des éléments de `li`. En d'autres termes, si on note `x1, x2, ... , xn` les éléments de `li` alors un appel à « `iter_ldcc f li` » est équivalent à exécuter « `f x1; f x2; ... ; f xn` ». Votre fonction s'exécutera en temps linéaire en la longueur de la LDCC.
   (b) Écrire une fonction « `print_ldcc: int ldcc -> unit` » qui affiche les éléments d'une LDCC d'entiers. Par exemple, sur la liste de la figure 2, votre fonction affichera `3 17 24 23 7`.

Exercice 3. Polynômes

Polynômes creux Un polynôme est dit ***creux*** si la plupart de ses coefficients sont nuls. Dans ce cas, il est intéressant de représenter le polynôme comme la liste de ses coefficients non nuls. Un monôme sera représenté par un couple d'entiers formé du degré et du coefficient du monôme. Un polynôme sera représenté par une liste chaînée de monômes dont les puissances sont triées par ordre décroissant.

1. Définir un type `monome` et un type `poly_creux`. Le type `monome` sera un type produit et `poly_creux` sera un alias.
2. Définir une variable `poly1` représentant le polynôme $X^{4321} - 2X^{1234} + 3X - 4$.
3. Écrire une fonction « `addition: poly_creux -> poly_creux -> poly_creux` » qui ajoute les deux polynômes donnés en argument.
4. Écrire une fonction « `produit: poly_creux -> poly_creux -> poly_creux` » qui multiplie les deux polynômes donnés en argument.
5. Écrire une fonction « `derivee: poly_creux -> poly_creux` » qui renvoie le polynôme dérivé du polynôme donné en argument.

Remarque. Avez vous fait en sorte que les listes obtenues en sortie des fonctions ne contiennent aucun coefficient nul ? Si ce n'est pas le cas, corrigez vos fonctions.

Polynômes pleins Un polynôme est dit ***plein*** si la plupart de ses coefficients sont non nuls. Dans ce cas, il est intéressant de représenter le polynôme par un tableau `poly` tel que `poly.(i)` est le coefficient devant $X^{\mathtt{i}}$. Afin de garantir l'unicité, on impose que le dernier élément de `poly` soit non nul.

6. Définir un type `poly_plein`.
7. Écrire une fonction « `addition: poly_plein -> poly_plein -> poly_plein` ».
8. Écrire une fonction « `produit: poly_plein -> poly_plein -> poly_plein` ».
9. Écrire une fonction « `derivee: poly_plein -> poly_plein` ».

Remarque. Avez vous fait en sorte que le dernier élément des tableaux obtenus en sortie des fonctions soit non nul ? Si ce n'est pas le cas, corrigez vos fonctions.

Conversion

10. Écrire les fonctions :

```
creux_of_plein: poly_plein -> poly_creux
plein_of_creux: poly_creux -> poly_plein
```

pour convertir entre les deux types.

Exercice 4. Nombres complexes

1. Définir un nouveau type pour représenter les nombres complexes.
2. Définir une variable globale `un` représentant le nombre complexe 1, une variable `zero` représentant le nombre complexe 0 et une variable `i` représentant le nombre complexe i.
3. Écrire une fonction qui prend en entrée un couple de réels et renvoie le nombre complexe correspondant.
4. Écrire une fonction qui prend en entrée un nombre complexe et renvoie le couple de réels correspondant.
5. Sans utiliser les fonctions des questions précédentes, écrire des fonctions qui permettent de calculer (pensez à traiter les cas où ce n'est pas possible à l'aide d'un `failwith` ou d'une exception) :
   - → Le conjugué d'un nombre complexe.
   - → La somme de deux nombres complexes.
   - → Le produit de deux nombres complexes.
   - → Le module d'un nombre complexe. Notez qu'en OCaml, `mod` et `module` ne sont pas des noms de fonctions valides.
   - → Le quotient de deux nombres complexes (lorsque c'est possible).
   - → L'argument d'un nombre complexe. On pourra utiliser la fonction `atan` et les formules :

$$\pi = 4\arctan(1) \qquad\qquad \forall \theta \in \left]-\pi\,;\pi\right[: \theta = 2\arctan\left(\frac{\sin\theta}{1+\cos\theta}\right)$$

   - → Le nombre complexe élevé à la puissance $n \in \mathbb{Z}$ à l'aide d'une exponentiation rapide.

Exercices à rendre au plus tard le 30/03/2025 à 20h

Exercice 5. L'anneau $\mathbb{Z}[\sqrt{2}]$

L'anneau $\mathbb{Z}[\sqrt{2}]$ est l'ensemble défini par : $\mathbb{Z}[\sqrt{2}] = \left\{a + b\sqrt{2} : a, b \in \mathbb{Z}\right\}$.

1. Définir un type noté `z_sqrt2` pour représenter les éléments de $\mathbb{Z}[\sqrt{2}]$.
2. Écrire une fonction « `opp_zs2: z_sqrt2 -> z_sqrt2` » qui renvoie l'opposé de son argument.
3. Écrire une fonction « `add_zs2: z_sqrt2 -> z_sqrt2 -> z_sqrt2` » qui renvoie la somme de ses deux arguments.
4. À l'aide des fonctions `opp_zs2` et `add_zs2`, écrire une fonction « `sub_zs2: z_sqrt2 -> z_sqrt2 -> z_sqrt2` » qui renvoie la différence de ses deux arguments.
5. Le conjugué du nombre $a+b\sqrt{2}$ est le nombre $a-b\sqrt{2}$. Écrire une fonction « `conj_zs2: z_sqrt2 -> z_sqrt2` » qui renvoie le conjugué de son argument.
6. Écrire une fonction « `mult_zs2: z_sqrt2 -> z_sqrt2 -> z_sqrt2` » qui renvoie le produit de ses deux arguments.

Exercice 6. Droite réelle achevée

1. Définir un type `r_barre` pour représenter l'ensemble $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$.

Pour faire des opérations sur $\overline{\mathbb{R}}$, on utilisera les mêmes règles que lors d'un calcul de limite. Par exemple :

$$4 + (+\infty) = +\infty \qquad\qquad (+\infty) + (-\infty) \text{ est une forme indéterminée.}$$

2. Écrire des fonctions pour tester l'égalité entre deux éléments de $\overline{\mathbb{R}}$, puis pour additionner, soustraire, multiplier et diviser deux éléments de $\overline{\mathbb{R}}$. Lorsque l'opération n'est pas possible (forme indéterminée ou division par 0), votre fonction déclenchera une erreur :

```
egal_rb: r_barre -> r_barre -> bool
add_rb : r_barre -> r_barre -> r_barre
diff_rb: r_barre -> r_barre -> r_barre
mult_rb: r_barre -> r_barre -> r_barre
div_rb : r_barre -> r_barre -> r_barre
```

Exercice 7. Utilisation d'exceptions (facultatif)

Étant donnée une liste chaînée d'entiers `li`, on souhaite écrire une fonction `f` qui renvoie `0` si `li` contient au moins un élément nul et la somme des éléments de `li` sinon. Par exemple, on obtient `0` pour `[1;5;-4;0;8;0]` et `16` pour `[2;-8;6;9;4;3]`. Pour cela, on définit l'exception suivante :

```
exception Zero;;
```

1. (a) Écrire une fonction `f_aux` qui prend en entrée la liste `li` et qui :
   - Déclenche l'exception `Zero` si `li` contient au moins un élément nul.
   - Renvoie la somme des éléments de `li` sinon.

   (b) En déduire la fonction `f`.
2. Modifier les fonctions précédentes pour écrire une fonction `g` qui prend en entrée `li` et renvoie :
   - Le nombre de zéros si `li` contient au moins un élément nul.
   - Le dernier élément strictement négatif si `li` ne contient pas d'élément nul mais au moins un élément strictement négatif.
   - La somme des éléments de `li` sinon.

   Il est attendu que vous suiviez la même approche que dans la question précédente. Vous devez donc définir deux nouvelles exceptions pour gérer les deux premiers points et écrire deux fonctions `g_aux` et `g`. En revanche, il est interdit d'écrire une autre fonction intermédiaire (en particulier, pas de fonction qui renvoie le nombre d'éléments nuls dans une liste quelconque).