L1 MISPIC
Programmation fonctionnelle
TP 5

Avant de commencer le TP, créez un répertoire `TP5` dans votre arborescence consacrée à OCaml. Allez dans ce répertoire et ouvrez un fichier `tp5.ml` avec *emacs*. Vous taperez vos fonctions dans ce fichier, et les chargerez dans l'interpréteur OCaml avec `#use "tp5.ml";;` Si une fonction est erronée, corrigez-la dans *emacs*, puis rechargez le fichier dans l'interpréteur. Ceci vous permettra de pouvoir sauvegarder votre travail dans le fichier `tp5.ml`. Gardez à l'esprit que même si la définition d'une fonction ne fait pas d'erreur dans l'interpréteur, cela ne signifie pas pour autant qu'elle calcule ce que vous souhaitez. Vous devez pour vous en assurer la tester en l'appelant avec différentes valeurs de paramètres pour être sûr que ce qu'elle vous renvoie est correct.

1 Les grands entiers

Comme vous l'avez peut-être déjà remarqué (et comme cela a été évoqué en CM), les entiers de type `int` sont limités en OCaml, comme dans la plupart des langages de programmation d'ailleurs.

Le plus grand entier possible a un nom, `max_int`, et la plus petit entier est `min_int`. Voilà leurs valeurs respectives :

```
# max_int;;
- : int = 2147483647
# min_int;;
- : int = -2147483648
```

Si on tente d'ajouter 1 au plus grand des entiers, le résultat qu'on obtient est assez aléatoire. Dans certains langages de programmation, et dans certaines versions d'OCaml, cela provoque une erreur. Dans la version qu'on utilise en TP, OCaml retourne le plus petit des entiers, c'est-à-dire `min_int`. Cela peut surprendre, mais c'est en fait lié à la représentation des entiers en machine, et à l'algorithme qu'OCaml utilise pour calculer la somme de deux entiers. On peut faire la même remarque si on soustrait 1 au plus petit des entiers : OCaml retourne le plus grand des entiers, c'est-à-dire `max_int`.

```
# max_int + 1;;
- : int = -2147483648
# min_int - 1;;
- : int = 2147483647
```

Dans ce TP, nous allons nous affranchir de cette limitation d'OCaml en permettant à un utilisateur de travailler avec des entiers positifs ou nuls de taille quelconque. Pour cela, on commence par appeler `chiffre` tout `int` compris entre 0 et 9, et on appelle `grand` toute liste d'éléments de type `chiffre` telle que :

— `[]` est un `grand` représentant l'entier 0 ;
— si `x` est un `chiffre` représentant le chiffre x et `n` est un `grand` représentant l'entier n, alors `x::n` est un `grand` représentant l'entier $x + 10 \times n$.

Par exemple, l'entier 1 est codé par la liste `1::[]` (soit `[1]`) et représente l'entier $1 + 10 \times 0$; l'entier 10 est codé par la liste `0::[1]` (soit `[0;1]`) et représente l'entier $0 + 10 \times 1$; l'entier 11223344556677889900 est codé par la liste `[0;0;9;9;8;8;7;7;6;6;5;5;4;4;3;3;2;2;1;1]`...

2 Des `string` aux `grand`, et vice versa

1. Écrire une fonction `chiffre_of_char : char -> int` qui prend un caractère et retourne l'entier correspondant (donc le `chiffre`) quand ce caractère est compris entre `'0'` et `'9'`, et fait une erreur sinon. Ainsi, `chiffre_of_char '2'` renvoie 2.
2. Écrire une fonction `grand_of_string : string -> int list` qui renvoie le `grand` correspondant à un entier codé sous forme de chaîne de caractères. Votre fonction doit se comporter comme sur l'exemple ci-dessous. À vous de définir, au besoin, des fonctions auxiliaires.

```
# grand_of_string "423100" ;;
- : int list = [0; 0; 1; 3; 2; 4]
1# grand_of_string "0" ;;
- : int list = []
# grand_of_string "-612" ;;
Exception: Failure "erreur chiffre_of_char".
# grand_of_string " 6 1 2 " ;;
Exception: Failure "erreur chiffre_of_char".
# grand_of_string "12345678901234567890" ;;
- : int list = [0; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0; 9; 8; 7; 6; 5; 4; 3; 2; 1]
# grand_of_string "0000000000000000021" ;;
- : int list = [1;2]
```

   Selon la façon dont vous programmerez, vous pourrez avoir besoin de :

```
# String.sub "le chat mange" 0 4;;
- : string = "le c"
# String.sub "le chat mange" 3 6;;
- : string = "chat m"
```

3. Écrire une fonction `string_of_chiffre : int -> string` qui prend un entier (donc un `chiffre`) et retourne la chaîne correspondante quand cet entier est compris entre 0 et 9, et fait une erreur sinon. Ainsi, `string_of_chiffre 2` renvoie `"2"`.

4. Écrire une fonction `string_of_grand : int list -> string` qui renvoie la chaîne de caractères correspondant à un `grand` (une liste d'entiers). Cette fonction doit se comporter comme sur l'exemple ci-dessous :

```
# string_of_grand [0;9;8;7;6;5;4;3;2;1;0;9;8;7;6;5;4;3;2;1] ;;
- : string = "12345678901234567890"
# string_of_grand (grand_of_string "12345678901234567890") ;;
- : string = "12345678901234567890"
# string_of_grand (grand_of_string "0000000000000000001") ;;
- : string = "1"
```

À partir de maintenant, on supposera que tous les `grand` ont été correctement définis. En particulier, on partira du principe qu'ils ne contiennent aucun 0 superflu, et on n'en introduira pas de nous-mêmes.

3 Opérations de comparaison

Certaines fonctions sur les listes que vous avez vues en TD sont si fréquemment utilisées qu'elles sont prédéfinies en OCaml. C'est le cas du calcul de la longueur d'une liste, de son inversion, ou des opérations de comparaison de listes :

```
# List.length [] ;;
- : int = 0
# List.length [1;2;3] ;;
- : int = 3
# List.rev [1;2;3] ;;
- : int list = [3; 2; 1]
# [1;2] = [1;2] ;;
- : bool = true
# [1;2] = [2;1] ;;
- : bool = false
# [1;1;1] <= [1;2] ;;
- : bool = true
# [1;1;1] <= [1;1] ;;
- : bool = false
# [1;2;3] <= [1;3;2] ;;
- : bool = true
```

Notez que `(<=)` compare les deux premiers éléments des listes, puis les deux suivants, etc, jusqu'à trouver deux éléments différents dans la première et la seconde liste, ou bien jusqu'à ce que l'une des deux listes soit épuisée (c'est donc l'ordre du dictionnaire appliqué aux listes). Bien entendu, vous pouvez utiliser les quatre fonctions ci-dessus pour répondre aux questions suivantes :

1. Écrire une fonction `inf_strict : 'a list -> 'a list -> bool` qui prend deux `grand` et retourne vrai si le premier des deux entiers correspondants est strictement inférieur au second, et faux sinon. On rappelle que dans les `grand`, l'ordre des chiffres dans la liste est inversée. Par exemple :

```
# inf_strict [1;2] [2;3];;
- : bool = true
# inf_strict [1;2] [2;3;4];;
- : bool = true
# inf_strict [1;2] [1;2];;
- : bool = false
# inf_strict [1;2;3] [1;2];;
- : bool = false
```

2. En déduire une fonction `inf_egal : 'a list -> 'a list -> bool` qui prend deux `grand` et retourne vrai si le premier des deux entiers correspondants est inférieur ou égal au second, et faux sinon.
3. En déduire une fonction `sup_strict : 'a list -> 'a list -> bool` qui prend deux `grand` et retourne vrai si le premier des deux entiers correspondants est strictement supérieur au second, et faux sinon.
4. Écrire une fonction `sup_egal : 'a list -> 'a list -> bool` qui prend deux `grand` et retourne vrai si le premier des deux entiers correspondants est supérieur ou égal au second, et faux sinon.

4 Arithmétique des grand

1. Écrire une fonction `successeur : int list -> int list` qui prend un `grand` et renvoie son successeur. Par exemple :

```
# successeur [1;2;4];;
- : int list = [2; 2; 4]
# successeur [9;2;4];;
- : int list = [0; 3; 4]
# successeur [9;9;4];;
- : int list = [0; 0; 5]
# successeur [9;9;9];;
- : int list = [0; 0; 0; 1]
```

2. Écrire une fonction `additionne : int list -> int list -> int list` qui additionne deux `grand`. Notons que si deux entiers n_1 et n_2 s'écrivent sous la forme $x_1 + 10 \times m_1$ et $x_2 + 10 \times m_2$ avec $x \in [0, 9]$, alors leur somme $n_1 + n_2$ s'écrit aussi sous la forme $x + 10 \times m$ avec $x \in [0, 9]$. La valeur de x dépend de la valeur de $x_1 + x_2$. De plus, si cette dernière somme est inférieure ou égale à 9, alors $m = m_1 + m_2$, sinon $m = (m_1 + m_2) + 1$. Par exemple :

```
# additionne [1;2] [3;1];;
- : int list = [4; 3]
# additionne [1;2] [9;1];;
- : int list = [0; 4]
# additionne [1;2] [1];;
- : int list = [2; 2]
# additionne [2] [3;1];;
- : int list = [5; 1]
```

3. Écrire une fonction `predecesseur : int list -> int list` qui prend un `grand` et renvoie son prédécesseur. Par exemple :

```
# predecesseur [2];;
- : int list = [1]
# predecesseur [1];;
- : int list = []
# predecesseur [];;
Exception: Failure "erreur predecesseur".
# predecesseur [9;9;9];;
- : int list = [8; 9; 9]
# predecesseur [0;0;0;1];;
- : int list = [9; 9; 9]
# predecesseur [0;0;0;1;1];;
- : int list = [9; 9; 9; 0; 1]
```

4. Écrire une fonction `soustrait : int list -> int list -> int list` qui soustrait deux `grand`. Par exemple :

```
# soustrait [5;1] [3;1];;
- : int list = [2]
# soustrait [3;1] [5;1];;
Exception: Failure "erreur predecesseur".
# soustrait [5;1] [1];;
- : int list = [4; 1]
# soustrait [5;1] [6];;
- : int list = [9]
# soustrait [2;5] [1;5];;
- : int list = [1]
```