TP Ocaml 1

Dans ce TP aucune importation de module comme `List` n'est autorisé.
Les *récursions terminales* sont encouragées.

Exercice 1. Écrire la fonction `factorielle : int -> int` qui renvoie la factorielle de l'argument (en récursion terminale).

Exercice 2. Écrire une fonction `long l : 'a list -> int` qui retourne la longueur d'une liste (en récursion terminale).

Exercice 3. Ecrire une fonction `rev l : 'a list -> 'a list` qui inverse une liste.
Donner une version en récursion terminale.

Exercice 4. Ecrire une fonction `copy l` qui renvoie la copie d'une liste. `copy : 'a list -> 'a list`.

Exercice 5. Écrire la fonction `emballer : 'a list -> 'a list list` qui transforme une liste d'éléments en une liste de listes à un seul élément chacune.

```
# emballer [1;2;3];;
- : int list list = [[1]; [2]; [3]]
```

Exercice 6. Écrire une fonction `loop : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a` telle que `loop p f x` applique la fonction `f` de façon répétée à l'argument `x` tant que la valeur obtenue ne vérifie pas le prédicat `p`.

```
# loop (fun x -> x > 5) (fun x -> x + 3) 1;;
- : int = 7
# loop (fun x -> x > 5) (fun x -> x + 3) 6;;
- : int = 6
```

Exercice 7. Écrire une fonction `sqrt : int -> int` qui calcule la racine carrée (arrondie à l'entier inférieur) d'un entier positif donné en argument.
Un code d'un ligne sera valorisé.

```
# sqrt 46;;
- : int = 6
# sqrt 49;;
- : int = 7
# sqrt 51;;
- : int = 7
```

Exercice 8. Écrire l fonction `separer : 'a list -> 'a list * 'a list` qui décompose une liste `l` en deux listes de tailles égales à 1 près et dont la réunion des contenus est celui de `l`.

```
# separer [1;2;3;4];;
- : int list * int list = ([1; 3], [2; 4])
# separer [1;2;3;4];;
- : int list * int list = ([1; 3], [2; 4])
# separer [1;2;3;4;5];;
- : int list * int list = ([1; 3; 5], [2; 4])
```

Exercice 9.

1. Écrire la fonction `exists : ('a -> bool) -> 'a list -> bool` qui prend en paramètres un prédicat `p` et une liste `l` et qui retourne `true` si un des éléments de la liste vérifie le prédicat `p` ; `false` sinon.

```
# exists (fun x -> x mod 2 = 1) [10;20;30];;
- : bool = false
# exists (fun x -> x mod 2 = 1) [10;21;30];;
- : bool = true
# exists (fun x -> x mod 2 = 1) [];;
- : bool = false
```

2. Application : écrire une fonction `mem 'a-> 'a list -> bool` telle que `mem x l` renvoie vrai si `x` est dans `l` et faux sinon.
3. Écrire une fonction `for_all: ('a -> bool) -> 'a list -> bool` qui détermine si tous les éléments d'une liste vérifient une propriété donnée.

```
# for_all (fun x -> x > 2) [];;
- : bool = true
# for_all (fun x -> x > 2) [5;6];;
- : bool = true
# for_all (fun x -> x > 2) [5;1;6];;
- : bool = false
```

Exercice 10. Tri par insertion.

1. Ecrire la fonction `insere` de type `'a -> 'a list -> 'a list` qui prend en paramètre un objet d'un type totalement ordonné et une liste triée par ordre croissant. La fonction insère le nombre à sa bonne place dans la liste.

```
# insere 25 [10;20;30];;
- : int list = [10; 20; 25; 30]
```

   Analyser la complexité temporelle dans le pire cas et le meilleur cas.

2. Ecrire la fonction `tri_insertion l` de type `'a list -> 'a list` qui prend une liste d'objets (d'un type totalement ordonné) en paramètre et retourne une version triée par ordre croissant.

```
# tri_insertion [10;2;9;6;3;11;8];;
- : int list = [2; 3; 6; 8; 9; 10; 11]
```

   Analyser la complexité temporelle dans le pire cas et le meilleur cas.

Exercice 11. Ecrire la fonction `kieme k l` qui prend en paramètre une liste `l` et un entier `k` et retourne l'élément numéro `k` de la liste `l`. Si `k` est trop grand, une exception `failwith "k trop grand"` doit être soulevée (sans calcul préalable de la longueur de liste). Signature : `int -> 'a list -> 'a`.

```
# kieme 3 [2;6;9];;
Exception: Failure "k trop grand".
# kieme 2 [2;6;9];;
- : int = 9
```

Exercice 12.

1. Écrire le fonction `mem : 'a -> 'a list -> bool` qui teste si un objet est dans une liste.
2. Écrire une fonction `test : 'a list -> bool` qui teste si il y a des doublons dans une liste et renvoie donc un bouléen.
3. Ecrire `elimine : 'a list -> 'a list` qui élimine les doublons d'une liste et renvoie donc une nouvelle liste sans doublon.

Exercice 13. Ecrire la fonction `prefixe : 'a list -> 'a list list` qui retourne la liste des préfixes non vide d'une liste. On n'impose pas de récursion terminale.

```
# prefixe [1;2;3;4;5];;
- : int list list = [[1; 2; 3; 4; 5]; [1; 2; 3; 4]; [1; 2; 3]; [1; 2]; [1]]
# prefixe [];;
Exception: Failure "liste vide".
```

Exercice 14. Ecrire la fonction `produit_cartesien :'a list -> 'b list -> ('a * 'b) list list` qui retourne le produit cartésien de deux listes

```
# produit_cartesien [1;2] ["a"; "b"; "c"];;
- : (int * string) list list =
[(1, "a"); (1, "b"); (1, "c"); (2, "a"); (2, "b"); (2, "c")]
```

Exercice 15. Ecrire la fonction `assoc` de signature `'a -> ('a * 'b) list -> 'b` telle que `assoc x l` retourne le second membre du premier tuple dont le premier membre est égal à x.

```
# assoc 3 [(1, "a"); (10, "bcv"); (15, "c"); (2, "a"); (20, "b"); (2, "c")];;
Exception: Failure "pas d'image".
# assoc 20 [(1, "a"); (10, "bcv"); (15, "c"); (2, "a"); (20, "b"); (2, "c")];;
- : string = "b"
```

Pour préparer l'exercice suivant, voici un petit tutoriel sur les chaînes de caractères et l'ordre sur les caractères :

```
# let s = "abcbd-12e";;
val s : string = "abcbd-12e"
# s.[0], s.[3];;
- : char * char = ('a', 'b')
# String.length s;;
- : int = 9
# s.[8];;
- : char = 'e'
# s.[9];;
```

```
Exception: Invalid_argument "index out of bounds".
# '3';;
- : char = '3'
# 'a' < 'k';;
- : bool = true
# 'm' <= 'k';;
- : bool = false
# '0' < '3';;
- : bool = true
# '3' < '2';;
- : bool = false
# '9' < 'a';;
- : bool = true
# '-' < '0';;
- : bool = true
```

Exercice 16. Dans le langage Ada les identifieurs (de variables, de fonctions...) sont des mots qui commencent pas une lettre minuscule, se poursuivent avec des lettres minuscules, des chiffres ou des symboles de soustraction `-` et se terminent par une lettre minuscule ou un chiffre. S'y ajoute la contrainte que jamais deux symboles `-` ne peuvent être successifs.

Ecrire la fonction `identifier : string -> bool` qui vérifie si une chaîne de caractère est un identifieur correct ou non.

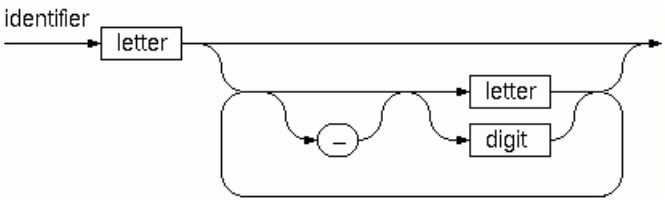

Figure 1 – Diagramme de syntaxe Ada

Voici quelques exemples :

```
# identifier "this-is-a-long-identifier";;
- : bool = true
# identifier "this-ends-in-";;
- : bool = false
# identifier "two--hyphens";;
- : bool = false
# identifier "-dash-first";;
- : bool = false
```

Exercice 17. Un *mot de Dyck* est un mot formé de 0 et de 1 en même quantité avec la caractéristique que pour tout préfixe du mot, le nombre de 0 est supérieur au nombre de 1.

Dans cet exercice OCaml, on représente les mots de Dyck par des listes d'entiers.

Ecrire la fonction `dyck : int list -> bool` tel que `dyck l` renvoie un bouléen indiquant si `l` est un mot de Dyck.

```
# dyck [0;0;1;0;0;1;1;1];;
```

```
- : bool = true
# dyck [0;0;1;0;0;1;1;1;1;0];;
- : bool = false
# dyck [0;0;1;0;1;1;0;1];;
- : bool = true
# dyck [1;0];;
- : bool = false
# dyck [0;1;0;0;1];;
- : bool = false
```