

## IAP1 - des exercices de soutien

Ecrire les interfaces de chaque fonction demandée (en particulier pour chaque fonction demandée, deviner le type le plus général. Testez les fonctions.

### 1 Session

Compléter la session suivante. Il n'y pas d'erreur syntaxique mais des erreurs de typage peuvent exister. Dans ce cas, indiquez la réponse "erreur de typage" et expliquez en quelques mots pourquoi..

```
1 #let f x = x + x;;
```

```
val f :  = <fun>
```

```
2 #let g x y = y && (x>0);;
```

```
3 #g 2 true;;
```

```
- :  = 
```

```
4 #g 1;;
```

```
5 #let h (x,y) = g y x;;
```

```
6 #h true;;
```

```
7 #let j f = f (f 0);;
```

```
8 #let succ x = x +1 in j succ;;
```

```
9 #let k f x = f (f x);;
```

```
10 #k (function x -> x @ x) [1];;
```

```

    #exception Pb;;
exception Pb

12 #let m c e = match (c,e) with (0,_) -> 2
    | (x, true) -> x
    | _ -> raise Pb;;

```

```
13 #m 1 false;;
```

```
14 #m (0,true);;
```

```
15 #m (1, false) ;;
```

```
16 #(* Interface rr
```

```
type :
```

```
arguments : p l
```

```
precondition : rien
```

```
postcondition :
```

```
*)
```

```
let rec rr p l = match l with
```

```
  [] -> true
```

```
  | x::r -> (p x) && (rr p r);;
```

```
17 #(* Interface vv
```

```
type :
```

```
arguments : y l
```

```
precondition : rien
```

```
postcondition :
```

```
*)
```

```
let vv y l = rr (function x -> x=y) l;;
```

```
18 #vv 1 [1;2;1;3;1;1];;
```

## 2 Des fonctions sur les listes

1. Ecrire une fonction `supprimer` qui prend 2 arguments `l`, `e` et qui produit une liste où toutes les occurrences de `e` ont été supprimées.
2. Ecrire une fonction `remplace` qui prend 3 arguments `l`, `e` et `v` et qui produit une liste où la 1ère occurrence de `v` est remplacée par `x`.
3. Ecrire une fonction `remplace` qui prend 3 arguments `l`, `e` et `v` et qui produit une liste où toutes les occurrences de `v` sont remplacées par `x`.
4. Ecrire la fonction `dernier` qui retourne le dernier élément d'une liste.
5. Ecrire une fonction `begaie` qui prend en argument une liste `l` et qui produit une liste où tous les éléments sont répétés deux fois. Par exemple `repetier [1;2;1;3]` produit `[1;1;2;2;1;1;3;3]`.
6. Ecrire une fonction `insérer` qui prend en argument une liste `l` et un élément `e` et qui produit une liste où tous les éléments de `l` sont précédés de `e`. Par exemple `insérer [1;2;1;3] 10` produit `[10;1;10;2;10;1;10;3]`. LA liste vide reste vide.
7. Trier une liste de couples selon l'ordre lexicographique. Tri par insertion. Puis tri fusion. Puis tri quicksort.

## 3 Des arbres

On travaille sur le type des arbres binaires défini par

```
type 'a arbreb = Vide | Racine of 'a*'a arbreb *'a arbreb;;
```

1. Que signifie le `'a` dans la définition précédente ?
2. Ecrire une fonction qui récupère la liste des valeurs rangées dans un arbre binaire. Ecrire une première version où la liste aura éventuellement des doublons, puis une seconde version où la liste n'a pas de doublons.
3. Ecrire une fonction qui fait la somme des éléments d'un arbre binaire d'entiers.
4. Ecrire la fonction qui calcule la valeur minimale contenue dans un arbre ? Quel est son type ?
5. On considère la nouvelle définition suivante pour les arbres :

```
type 'a arbreb = Feuille of 'a | Racine of 'a*'a arbreb *'a arbreb;;
```

Un arbre est soit une feuille, soit un arbre binaire non vide avec une valeur à la racine, un sous-arbre gauche et un sous-arbre droit.

Avec cette définition un arbre peut-il être vide ?

Définir quelques arbres avec ce nouveau type.

Reprendre les fonctions précédentes pour cette nouvelle définition.

## 4 Une session avec de l'ordre sup à compléter

```
let coller_1 x y = y ^ x;;  
let coller_2 (x,y) = y ^ x;;  
coller_1 "a" "b";;  
coller_2 "a" "b";;  
let F = function f -> f (0,0);;  
let G f = function x -> f (x,0);;
```

```

let H f x = f (x,x);;
let perm f = function (x,y) -> f (y,x);;
F coller_2;;
H coller_2 "a";;
perm coller_2 ("jour","bon");;

```

## 5 Exercice

On définit le type des entiers naturels de la façon suivante

```
type nat = Z | S of nat;;
```

1. Ecrire l'entier 5 de type `nat`.
2. Ecrire le prédicat `inf` qui teste si un entier naturel est inférieur ou égal à un autre entier naturel.

## 6 Fonctionnelle

Réécrire les fonctions `remplacer`, `supprimer`, `insérer`, `begaie` de l'exercice 1 avec des fonctionnelles.

## 7 Terminaison

```

let rec f (x,y) = match (x,y) with
  (0, _) -> 1
| (_, 0) -> 2
| _ _ -> if x mod 2 = 0 then f (x/2, 2*y)
          else f (x, y/2);;

```

Montrer que la fonction `f` termine quand elle est utilisée avec des couples d'entiers naturels.

## 8 Compaction

On s'intéresse ici à la représentation compactée de fichiers qui contiennent beaucoup de caractères répétés.

Considérons par exemple la séquence `aaaaa++++++bchhhhhhhh`. Il est possible de la représenter de manière plus compacte sous la forme d'une suite de couples dont la première composante est le caractère représenté et la deuxième est le nombre de ses occurrences successives. On dit alors que la séquence est sous forme compactée.

Notre exemple est représenté par : (a,5) (+,7) (b,1) (c,1) (h,8). On décide de représenter ces séquences compactées par des listes de couples (*chaîne de caractères, entier*). L'exemple compacté appelé `ex` est alors codé en Caml de la façon suivante :

```
[("a",5); ("+",7); ("b",1); ("c",1); ("h",8)].
```

1. Définir la fonction `compte` qui prend en argument une séquence compactée et calcule le nombre de caractères contenus dans la séquence compactée. Par exemple : `compte ex` s'évalue à 22. Donner le type de la fonction `compte`.
2. Écrire une fonction `expanse` qui prend en argument un couple `(c,n)` où `c` est une chaîne de caractères et `n` un entier et qui construit la chaîne de caractères contenant `n` fois `c`. Vous en donnerez une version récursive et une version définie à partir de la fonctionnelle `fold_left` ou `fold_right`. Donner le type de la fonction `expanse`.
3. Écrire une fonction `expanse_tout` qui prend en argument une séquence compactée et construit la chaîne de caractères représentant la séquence d'origine. Par exemple : `expanse_tout ex` retourne la chaîne `"aaaaa++++++bchhhhhhhh"`.

## 9 Expressions arithmétiques

Dans cet exercice, on manipule des expressions arithmétiques construites selon la définition suivante (vu en cours) :

- Une variable est une expression arithmétique,
  - Un entier est une expression arithmétique,
  - Si  $e_1$  et  $e_2$  sont deux expressions arithmétiques alors  $e_1 + e_2$  est une expression arithmétique,
  - Si  $e_1$  et  $e_2$  sont deux expressions arithmétiques alors  $e_1 * e_2$  est une expression arithmétique.
1. Définir le type des expressions arithmétiques. Les variables seront représentées par des chaînes de caractères.
  2. Représenter en Caml l'expression  $a + (b * (-3) * c)$  où  $a$ ,  $b$  et  $c$  sont des variables.
  3. Ecrire une fonction `vars` qui calcule la liste des variables qui apparaissent dans une expression. On prendra garde à ne faire figurer dans la liste qu'une seule fois chacune des variables (liste sans doublons). Si une expression ne contient pas de variable, la fonction retourne la liste vide. Les noms des variables seront ordonnés.  
La solution réclame éventuellement l'écriture de fonctions auxiliaires.
  4. On définit maintenant le type `signe` :

```
type signe = Positif | Negatif | SaisPas;;
```

Ecrire une fonction `CalcSigne` qui prend en argument une expression et tente de déterminer son signe sans calculer la valeur de l'expression. Par exemple elle retourne `Positif` pour l'expression  $(-3) * ((-3) + (-6))$  en raison des règles suivantes : *la somme de 2 nombres négatifs est un nombre négatif et le produit de 2 nombres négatifs est un nombre positif*. Elle retourne la valeur `SaisPas` dès qu'une variable apparaît ou que l'on ne peut déterminer le signe (cas de la somme de 2 expressions de signe différent).