

# LIFPF – Programmation fonctionnelle

## TD4 – Fonctions d'ordre supérieur

Licence informatique UCBL – Printemps 2025

### Exercice 1 : Décorateurs

Un *décorateur* est une fonction  $d$  qui va prendre (entre autres) une fonction  $f$  en argument et renvoyer une nouvelle fonction dont le comportement sera une "modification" de celui de  $f$ . Pour les décorateurs suivants, donner le type puis le code du décorateur. Dans la description de ces décorateurs, la fonction passée au décorateur sera nommée  $f$  et la fonction obtenue  $g$ .

1. *lift\_option* :  $g$  pourra prendre une `option` en argument. Si cette option est `None`, alors le résultat sera également `None`, sinon cela sera `Some` avec le résultat de l'application de  $f$  à la valeur de l'option.
2. *default* :  $f$  renvoie une option et  $g$  renvoie une valeur par défaut  $r$  si  $f$  renvoie `None`, ou la valeur de l'option sinon.  $r$  doit être un paramètre de *default*.
3. *except* :  $g$  renverra le même résultat que  $f$  sauf pour une valeur particulière  $a$  pour laquelle le résultat sera  $r$ . *except* devra être paramétrée par  $a$  et  $r$ .
4. *power* :  $g$  applique  $f$   $n$  fois à son argument (c'est-à-dire  $g = f^n$ ).

### Exercice 2 : Portée de variable, fermetures et applications partielles

Soit le code suivant :

```
let f =  
  let x = 5 in  
  fun y -> x + y  
in  
f 2
```

1. Ce code s'évalue-t-il sans erreur ? Si oui quel est le résultat ?
2. Détailler l'évaluation de l'expression en la traduisant auparavant en  $\lambda$ -calcul. On pourra traduire `let a =  $e_1$  in  $e_2$`  par  $(\lambda a.e_2 \ e_1)$ . On effectuera cette évaluation avec une stratégie avide en s'interdisant de réduire à l'intérieur d'un  $\lambda$  (ce qui correspond au mode d'évaluation de OCaml). Indiquer quelle(s) réduction(s) correspond(ent) à un calcul de fermeture.
3. On considère maintenant le code suivant :

```
let f x =  
  let y = g x in  
  fun z -> y + z  
  
let f' x z =  
  let y = g x in  
  y + z
```

```
let v =
  let h = f 3 in
  [ h 10; h 12; h 14; h 16 ]
```

```
let v' =
  let h' = f' 3 in
  [ h' 10; h' 12; h' 14; h' 16 ]
```

En supposant qu'un appel à `g` prenne 1 min de calcul, combien de temps prend le calcul de `v`? Combien de temps prend le calcul de `v'`?

4. On souhaite, en utilisant `List.map`, implémenter une fonction qui prend une liste de liste d'`int` et renvoie la liste de liste d'`int` contenant des listes où les int de départ ont été élevés au carré. Aurait-on pu facilement coder cette fonction si `List.map` avait pris en une fois une paire (fonction de transformation, liste à transformer)?

### Exercice 3 : Parcours génériques d'arbres

On souhaite implémenter une structure d'arbre binaire de recherche pour implémenter une structure d'association clé-valeur, une clé pouvant être associée à plusieurs valeurs. Les données stockées aux nœuds seront donc de la forme (clé, liste de valeurs).

1. Définir le type de cette structure.
2. Définir la fonction `insere` qui ajoute une valeur en fonction de la clé et d'une fonction de comparaison des clés. On utilisera le type `type cmp_result = Lt | Eq | Gt` (plus petit, égal, supérieur) pour le résultat des comparaisons.
3. Définir une fonction `map_valeurs` qui prend en argument une fonction et transforme toutes les valeurs de l'arbre en utilisant cette fonction. On utilisera `List.map` pour transformer les listes le cas échéant.
4. Définir une fonction `fold_valeurs` qui prend en argument une fonction `f` et une valeur initiale `v`. `f` effectuera un calcul sur un accumulateur et une valeur pour renvoyer une nouvelle valeur d'accumulateur. `fold_valeurs` renverra la valeur de  $f(f(\dots(f\ v\ x_n)\dots)x_2)x_1$  si  $\{x_1, x_2, \dots, x_n\}$  sont les valeurs stockées dans l'arbre. On utilisera `List.fold_left` le cas échéant. Est-il possible d'avoir plusieurs implémentations différentes de cette fonction produisant des résultats différents?
5. Quelles difficultés vont se poser si on souhaite implémenter une fonction `map_cles` qui transforme les clés? Proposer une implémentation de `map_cles` qui respecte la contrainte que le résultat doit être un arbre binaire de recherche.
6. Définir une fonction `fold_arbre` qui va travailler sur les données des nœuds et pas simplement sur les valeurs. La fonction passée en argument à `fold_arbre` prendra deux valeurs d'accumulateur (pourquoi?). Recoder `map_valeur` à partir de cette fonction.