

TDTP donné en devoir maison noté

Dernière version du sujet [ici](#).

2022

Consignes

Avertissement : ce TDTP est à faire de manière individuelle. Tout plagiat et toute triche seront sanctionnés : nous appliquerons des algorithmes de détection de “similitudes” entre les copies d’étudiants pour cela. Tous les documents du cours sont autorisés cependant, ainsi bien sûr que la [documentation OCaml](#).

Vous devez produire une archive de la forme `prenom_nom_GX.2.x` où `prenom` et `nom` seront orthographiés exactement comme dans votre adresse email `prenom.nom@u-psud.fr`, et où `X` est le numéro de votre groupe de TD. Cette archive contiendra 5 fichiers nommés `exoN.ml` où `N` correspond au numéro de l’exercice.

<https://powcoder.com>

Add WeChat powcoder

exo1.ml exo2.ml exo3.ml exo4.ml exo5.ml

Chaque fichier `.ml` doit impérativement compiler. Vous pouvez ajouter des lignes de tests. Mais si vous voulez rajouter du code qui ne compile pas, alors mettez-le en commentaire. Si les réponses sont à rédiger en français, mettez-les également en commentaires dans le fichiers correspondant, en omettez les accents, afin d’éviter les problèmes de codage de fichiers.

Le TDTP est à rendre sur eCampus dans la rubrique [TD/TP](#) > TP Noté, avant la date indiquée. Les exercices sont indépendants et peuvent être faits dans n’importe quel ordre. Ne restez pas bloqué sur une question.

1 Syntaxe

Question 1.1 *Trouver deux manières de parenthéser la fonction `ajout_deux` pour qu’elle fasse bien ce que l’on souhaite : ajouter deux à un nombre entier `g`.*

```
let f a b = a + b
```

```
let ajout_deux g = if g = 0 then f 2 0 else f 2 + 1 g - 1
```

Question 1.2 Trouver les erreurs de syntaxe dans la fonction suivante, qui parcourt deux listes d'entiers jusqu'à leurs derniers éléments, pour sommer ces deux derniers éléments et renvoyer cette somme. Il y a quatre erreurs à corriger.

```
let somme_derniers l1 l2 = match l1 with
| [] -> match l2 with
| [] -> 0
| x2 :: [] -> x2
| hd2 :: tl2 -> somme_derniers [] tl2
| x1 :: [] -> match l2 with
| [] -> x1
| x2 :: [] -> x1 + x2
| hd2 @ tl2 -> somme_derniers [x1] tl2
| hd1 :: tl1 -> somme_derniers tl1 l2
```

Question 1.3 Trouver les erreurs de syntaxe dans le code suivant. Il y a trois erreurs.

```
let x a b = a +. b
let f y z = let v = y + 5
if z > y then Printf.printf "%d" z ; z
else x v 0
```

Question 1.4 Pour chaque occurrence des variables x , z et y , indiquer où se trouve leur lieu, c'est-à-dire la déclaration qui correspond à cette occurrence. Pour cela indiquer, en commentaire de ligne où apparaît la variable, le numéro de la ligne de son lieu.

```
1 let y = 5.3 ;;
2 let u z =
3 z -. y ;;
4 let y =
5 let y =
6 y +. 2.5
7 in let z = u y
8 in y *. z
9 in let u = 2.0 in
10 y +. y +. u ;;
```

2 Typage

Question 2.1 Donner le type de chacune des variables de la fonction et de l'objet renvoyé. Justifier votre réponse.

```
let f1 (x,y) z =
  let g a b = a < b in
  if z then g x else g y ;;
```

Question 2.2 Dans le code suivant, que font les fonctions `list_sum` et `list_or`? Quels sont leur types?

Le code des lignes 4-6 est-il bien typé? Quel est le type de `a` à chaque étape?

Mêmes questions pour les lignes 8-10.

```
1 let list_sum p = List.fold_left (fun x y -> if p y then x + 1
    else x) 0;;
2 let list_or = List.fold_left (fun x y -> x || y) false;;
3
4 let a = [];;
5 list_sum a;;
6 list_or a;;
7
8 let a = ref [];;
9 list_sum !a;;
10 list_or !a;;
```

Question 2.3 On définit le type suivant :

```
type 'a arbre_binaire =
  | Feuille
  | Noeud of 'a * 'a arbre_binaire * 'a arbre_binaire;;
```

Écrire une fonction `map_arbre` qui prend en entrée un arbre binaire représenté par ce type et une fonction `f`, et qui renvoie l'arbre obtenu en transformant chacune des valeurs portées en chaque noeud, par `f`.

Écrire une fonction `forall_arbre` qui prend en entrée un arbre binaire et un prédicat (fonction de type `'a -> bool`), et renvoie `true` ssi toutes les étiquettes vérifient le prédicat.

Quels sont les types de ces deux fonctions?

3 Recherche linéaire d'un chemin optimal

Dans cette section on se propose d'implémenter un algorithme de recherche d'un chemin de plus grande somme dans une grille d'entiers, de complexité linéaire en le nombre de cases de cette grille. La grille suivante à 4 lignes et 3 colonnes est la grille-exemple.

-2	7	6
0	2	-1
1	-3	3
4	-4	5

Plus de détails sur les chemins seront donnés dans la suite. Intéressons-nous pour le moment aux grilles d'entiers, celles-ci sont représentées en OCaml par le type suivant :

```
type grid = int list list
```

Une grille est donc représentée par une liste de listes, où chaque liste représente une colonne de la grille. Par exemple, la grille

0	2
1	3

est représentée par `[[0;1];[2;3]]` de type `grid`.

Question 3.1 Définir un élément `g_example : grid` représentant la grille-exemple.

Question 3.2 Définir la fonction `height : grid -> int` qui, pour une grille `g` donnée en argument, renvoie la longueur de la première liste de `g`, ou une exception si `g` est vide. Par exemple on doit avoir `height g_example = 4`.

Un élément `g` est dit *bien formé* si `g` n'est pas la liste vide et si chaque liste qui compose `g` a une même taille non-nulle.

Question 3.3 Définir la fonction `wf_grid_exn : grid -> unit` qui renvoie une exception si et seulement si l'argument donné n'est pas un grille bien formée.

On suppose dorénavant que nous ne travaillons qu'avec des grilles bien formées.

Un *chemin* dans une grille commence par une case dans la première colonne puis continue avec une case dans la deuxième colonne, et ainsi de suite pour arriver à une case dans la dernière colonne. En plus, si le chemin passe par la case i de la colonne j , il ne peut continuer à la colonne $j + 1$ que par les cases des lignes $i - 1$, i ou $i + 1$. Par exemple, dans la grille-exemple, un chemin contenant la case 2 doit continuer soit par 6, -1 ou 3.

Nous considérons de plus que **la première ligne est en-dessous de la dernière ligne et que la dernière ligne est au-dessus de la première ligne** – vous pouvez imaginer la grille comme un cylindre.

La valeur d'un chemin est définie de la façon la plus naturelle possible : c'est la somme des cases par lesquelles il passe.

Question 3.4 Écrire en commentaire le nombre maximal que l'on peut obtenir en faisant la somme des cases d'un chemin valide dans la grille-exemple.

Question 3.5 Écrire une fonction `rotate_up : 'a list -> 'a list` qui, appliquée à une liste non-vide $[i_0; i_1; \dots; i_n]$ donne la liste $[i_1; \dots; i_n; i_0]$.

Question 3.6 Écrire une fonction `rotate_down : 'a list -> 'a list` qui, appliquée à une liste non-vide $[i_0; i_1; \dots; i_n]$ donne la liste $[i_n; i_0; i_1; \dots; i_{n-1}]$.

Étant donné une grille $[i_0; \dots; i_n]$, un *chemin partiel* dans cette grille est un chemin dans $[i_0; \dots; i_k]$, pour un certain $k \leq n$. Autrement dit, c'est un chemin qui commence par la première colonne mais qui n'a pas besoin d'aller jusqu'à la dernière colonne.

Question 3.7 À l'aide de `rotate_up` et `rotate_down`, définir la fonction `best_option : int list -> int list`. Considérons une colonne $j > 0$ quelconque. Cette fonction prend en argument une liste dont la position i indique la plus grande valeur d'un chemin partiel qui termine à la ligne i de la colonne $j - 1$. Elle doit calculer une liste dont la position i indique la plus grande valeur d'un chemin qui termine à la colonne $j - 1$ et qui peut être étendu pour passer par la ligne i de la colonne j .¹

Question 3.8 En utilisant `best_option`, définir une fonction `sums : grid -> int list` qui, étant donnée une grille, renvoie une liste dont la position i contient la plus grande valeur d'un chemin qui termine à la ligne i .

Question 3.9 Écrire une fonction `max_list : int list -> int` qui renvoie le plus grand élément d'une liste, ou une exception si la liste est vide.

Question 3.10 Écrire une fonction `solve : grid -> int` qui, étant donnée une grille, renvoie la valeur maximale d'un chemin dans cette grille.

1. Remarquez que ce calcul ne dépend pas des valeurs dans la colonne j

4 Fibonacci et mémoïsation

Cet exercice portera votre attention sur un problème d'efficacité lié au fait de devoir combiner plusieurs appels récursifs. Il proposera une solution dite de mémoïsation.

Question 4.1 Programmer `fibonacci : int -> int` de manière récursive et naïve, c'est-à-dire en retranscrivant directement la définition mathématique suivante : rappelons que $F_0 = 0$, $F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$. On utilisera `failwith` en cas d'argument négatif.

Question 4.2 Copier et renommer `fibonacci` en `fibonacci_count : int -> int`, ainsi que les appels récursifs bien sûr, et ajouter une ligne en début de fonction pour incrémenter un compteur global `count` (que vous aurez pris soin de déclarer au préalable en tant que `ref`). Ce compteur vous permet de traquer le nombre d'appel à `fibonacci_count`. Ayant initialisé `count` à 0, puis exécuté `fibonacci_count n` pour différentes valeurs de n , et répété l'expérience quelques fois, commenter sur la loi de croissance de `count` en fonction de n .

Question 4.3 Déclarer un tableau `f` de taille 1024, initialisé à -1 partout. Copier et renommer `fibonacci_count` en `memo_fibonacci_count` en remplaçant tous les appels récursifs à `fibonacci_count i`, par un appel à une fonction `fibonacci_check i`.

Votre `fibonacci_check i` vérifie dans le tableau pour voir si `f.(i)` y a déjà été calculé. Si `f.(i) = -1` ce n'est pas le cas, et donc `fibonacci_check` appelle `memo_fibonacci_count i` pour obtenir cette valeur, et elle la met ensuite dans `f.(i)`. Dans tous les cas, `fibonacci_check i` finit par renvoyer le `f.(i)`.

Attention, `memo_fibonacci_count` et `fibonacci_check` sont donc mutuellement récursives.

Question 4.4 Ayant initialisé `count` à 0, puis exécuté `memo_fibonacci_count n` pour différentes valeurs de $n < 1024$, et répété l'expérience quelques fois, commenter sur la loi de croissance de `count` en fonction de n . Donner une courte explication de la différence de comportement par rapport à 4.1.

5 Tortue.

Nous allons programmer une tortue dans un langage de programmation très simple, représenté en ocaml par le type :

```
type command = Up | Down | Left | Right | Seq of command list
```

On suppose que la tortue démarre en coordonnées $(0, 0)$, où la première coordonnée est celle de l'abscisse et la seconde celle de l'ordonnée. Le terrain est carré, le case inférieure gauche étant de coordonnées $(-100, -100)$, et la case supérieure droite de coordonnées $(100, 100)$.

Question 5.1 Écrire une fonction `eval_pos c : command -> int * int` qui, étant donné un jeu de commandes, calcule la coordonnée finale de la tortue. Lever une exception si la tortue sort du terrain.

Question 5.2 Une version 2 de la tortue dispose de la possibilité de répéter n fois une commande. Donner le code des modifications à apporter au type `command` et à `eval_pos` pour prendre cette évolution en compte.

Question 5.3 Une version 3 de la tortue dispose de la possibilité de répéter à l'infini une commande. Donner les modifications à apporter au type `command` pour prendre cette évolution en compte. Écrire une fonction `safety c : command -> bool` qui renvoie `true` si et seulement si la tortue ne sortira jamais du terrain au cours de cette exécution infinie.