TDTP donné en devoir maison noté

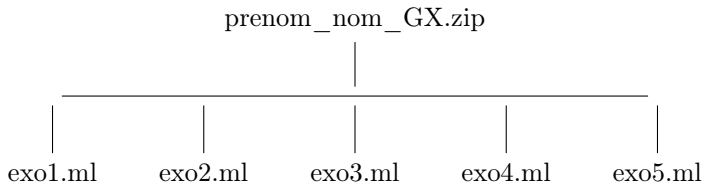
Dernière version du sujet ici.

2021

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.zip` 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 (*même si vous n'avez pas fait tous les exercices*), nommés `exoN.ml` où `N` correspond au numéro de l'exercice.



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. Tout problème de formattage entrainera une pénalité.

Le TDTP est à rendre sur eCampus dans la rubrique TDTP > 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 / Exécution

Question 1.1 *On considérera le code suivant.*

```
let x z y =
let x =
let z =
y - z in
let x z y =
z - y in
```

```
7  x y z in
8  x + y
```

(a) *Pour chaque occurrence des noms x, z et y, indiquer où se trouve son lieur — c'est-à-dire la déclaration qui correspond à cette occurrence — en commentant à la ligne où apparaît la variable, le numéro de la ligne de son lieur.*

(b) *Dire ce que la fonction calcule.*

Question 1.2 *Trouver deux parenthésages du code suivant tel que dans un des cas la fonction calcule la somme des deux arguments, alors que dans l'autre cas le résultat est le premier argument moins le deuxième.*

```
let plus_moins x y =
let g = fun u -> - u in
match x with
| 0 -> g 0 + y
| x -> g 0 + g y - x
```

Question 1.3 *On considérera le code suivant.*

```
exception E

let f y l =
  let rec h = fun l ->
    match l with
    | [] -> raise E
    | x :: [] -> (x, [y])
    | x :: l' -> let g = h l' in
                 let s = min (fst g) x in
                 (fst g, s :: (snd g))
  in
  snd (h l)
```

(a) *Remplir le tableau suivant, qui doit détailler le calcul de* `f 9 [5;1;7;3;4]`*. Chaque ligne doit représenter les valeurs des déclarations locales lors de l'appel à la fonction* `h` *sur l'argument* `l` *et quand* `y=9`*. Une fois le tableau rempli, donner le résultat du calcul de* `f 9 [5;1;7;3;4]`*.*

`l =`	`g =`	`s =`	`h l =`
`[4]`	pas de valeur	pas de valeur	
`[3;4]`			
`[7;3;4]`			
`[1;7;3;4]`			
`[5;1;7;3;4]`			

(b) *Expliquer ce que la fonction* `f` *calcule. Indication : la fonction fait deux choses au même temps.*

2 Typage

Question 2.1 *Donner le type de chacune des variables de ces deux fonctions et de l'objet que chacune renvoie. Justifier votre réponse.*

```
let f1 x (y, z) = match x with
  | 0 -> y = z
  | _ -> fst y;;

let f2 x y = if x then [x]::y else [];;
```

Question 2.2 *Qu'est-ce ce que la currying ? (répondez en une ou deux phrases)*
Utiliser la currying pour créer une fonction `not_list` *qui prend en argument une liste de booléens et renvoie la liste des booléens inverses (par exemple* `not_list [ true; false; false ]` *doit renvoyer* `[ false; true; true ]`*).*

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 `iter_arbre` *qui prend en entrée un arbre binaire représenté par ce type et une fonction* `f` *de type* `'a -> unit`*, et qui applique* `f` *à chacune des étiquettes de l'arbre dans un ordre quelconque.*
Écrire une fonction `compte_arbre` *qui prend en entrée un arbre binaire et un prédicat (fonction de type* `'a -> bool`*), et renvoie le nombre d'étiquettes de l'arbre qui vérifient ce prédicat.*
Quels sont les types de ces deux fonctions ?

3 Listes

Question 3.1 *La fonction* `flatten` *prend en paramètre une liste de listes, et les concatène. Par exemple,* `flatten [ [1 ; 1; 2]; [3 ; 5]; [8 ; 13; 21]]` *renvoie la liste* `[1 ; 1 ; 2 ; 3 ; 5 ; 8 ; 13 ; 21]`*.*

Ecrire une première version de `flatten` *à l'aide de* `List.fold_right`*.*

Puis, écrire une seconde version dénommée `flatten2` *en n'utilisant ni* `List.fold_right` *ni aucun* `let rec`*. On a cependant droit aux autres fonctions du module* `List`*, en particulier,* `fold_left`*,* `rev`*,* `map`*,* `iter`*,* `exists`*,* `forall`*...*

Question 3.2

1. *Ecrire une fonction* `sumcompose` *qui prend en paramètre trois variables* `f`*,* `x` *et* `n` *et renvoie* `[x ; x + f x ; x + f x + f^2 x ; ... ; x + f x + ... + f^n x ]` *où* `f^k x` *désigne le résultat qu'on obtient en appliquant* `f` *à* `x` *à* `k` *reprises.*
   Ecrire une deuxième version `sumcompose2` *qui soit* ***récursive terminale****.*

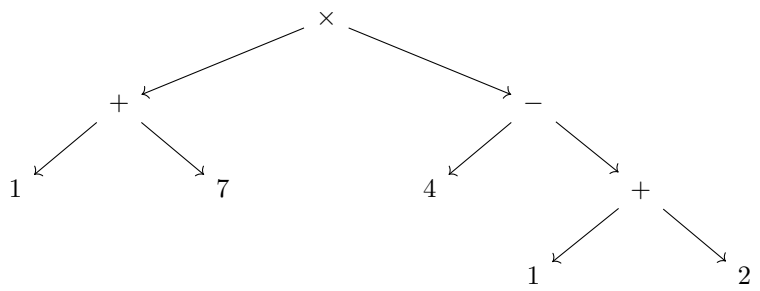
2. *Justifier (et non pas seulement recopier) en commentaire le type de* `sumcompose`*.*

4 Arbres

Cet exercice a pour but de vous faire utiliser un Arbre de Syntaxe Abstrait (AST pour Abstract Syntax Tree en anglais) pour représenter des expressions arithmétiques simples et pour les évaluer.

Un AST est un arbre dont chaque noeud représente une opération et ses fils représentent les opérandes. Les feuilles de l'arbre représentent simplement des valeurs.

Par exemple, l'expression $(1 + 7) \times (4 - (1 + 2))$ va être représentée par l'AST :



On va s'intéresser dans un premier temps au cas où les opérations sont : l'addition, la soustraction, la multiplication et la division. Les valeurs possibles sont les entiers.

Question 4.1 *Ecrivez le type* `op` *représentant les 4 opérations possibles, puis le type* `arbre` *représentant nos AST.*

Question 4.2 *Déclarez trois variables* `e1, e2, e3` *de type* `arbre` *représentant respectivement les expressions :*

- $(1 + 2) + 3$
- $(12 \times 7) - ((2/4) + 7)$
- $(((3 \times 4) + ((18/3) \times 4)) + (10 + 4)) - 8$

Question 4.3 *Ecrivez une fonction* `to_string : arbre -> string` *qui prend un AST et donne sous la forme d'une chaîne de caractère l'expression qu'il représente. Ecrivez une deuxième fonction* `print_tree : arbre -> unit` *qui affiche à l'écran l'expression représentée par un AST.*

Faites attention à obtenir le bon parenthésage.

Question 4.4 *Ecrivez une fonction* `eval : arbre -> int` *qui va évaluer l'expression représentée par l'arbre.*

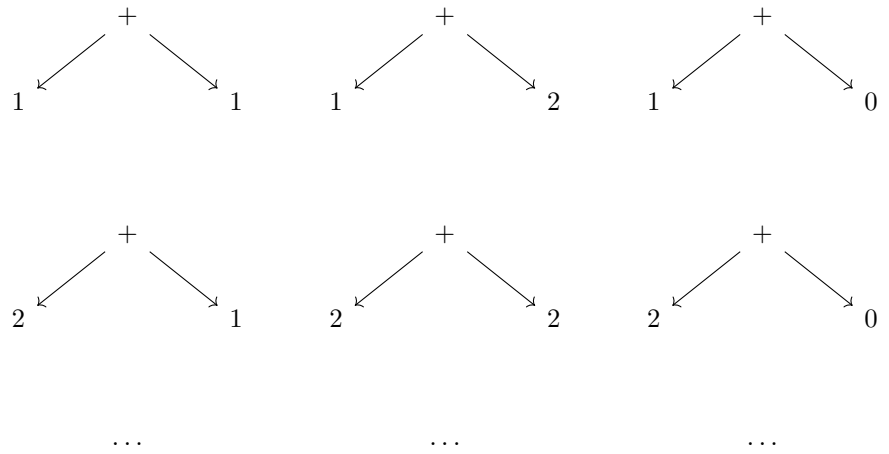
On va maintenant s'intéresser à un sous-ensemble des AST représentés par le type `arbre`. Les seules opérations possibles seront l'addition et la multiplication. Les valeurs possibles ne seront plus tous les entiers possibles, mais uniquement les valeurs $0, 1, 2$, car nous effectuerons de l'arithmétique modulo 3.

Question 4.5 *Ecrivez une fonction* `check3: arbre-> bool` *qui vérifie que l'arbre donné en entrée correspond bien à la restriction décrite.*

Question 4.6 *Adaptez votre fonction* `eval` *en* `eval3` *pour que l'évaluation se produise modulo* 3*.*

Question 4.7 *Ecrivez une fonction* `tree_of_height_n : int -> arbre list` *qui va renvoyer tous les arbres possibles de hauteur au plus* n*.*

Par exemple `tree_of_height_n 2` va nous donner les arbres :



Question 4.8 *Ecrivez une fonction qui prend en entrée un entier n, et un entier k compris entre 0 et 2, et vous renvoie tous les arbres t de profondeur au plus n tels que* `eval3 t = k`.

5 Problème

Question 5.1 *Ecrire une fonction* `is_perm: 'a list -> 'a list -> bool` *qui détermine si deux listes sont équivalentes à permutation près. Par exemple* `[2; 3; 4; 3]` *est équivalente à* `[3; 3; 4; 2]` *mais pas équivalente à* `[2 ; 4; 3]`.

On considère le type d'arbre n-aire suivant :
```
type tree = Leaf of int | Sum of tree list | Product of tree list.
```

Question 5.2 *Soit deux arbres* `t1` *et* `t2`*, écrivez une fonction* `equivalent : arbre -> arbre -> bool` *qui détermine si les deux arbres sont équivalents à associativité et commutativité, des sommes et des produits, près.*
Par exemple, les arbres représentants $1+2+5$ et $5+2+1$ sont équivalents, mais ils ne sont pas équivalents à celui représentant $4*2$).
Attention : les arbres représentant $(1+2+(2+2+1))$ et $(1+2+2+2+1)$ sont équivalents.