

TP n°6 – Le compte est bon

Ce TP porte sur le jeu du compte est bon. En partant d'une séquence de nombres et d'un nombre cible, ce jeu consiste à trouver une expression dont la valeur égale le nombre cible en utilisant un ou plusieurs nombres de la séquence, les opérations sur les nombres (addition, soustraction, multiplication, division) et les parenthèses.

Chaque nombre de la séquence peut apparaître que zéro fois ou qu'une seule fois dans l'expression. De plus, seuls des nombres entiers strictement positifs, y compris dans les calculs intermédiaires, sont autorisés (pas de zéro, pas de nombre négatif, pas de nombre décimaux).

Exemple : avec la séquence $[1, 3, 7, 10, 25, 50]$ et la cible = 765, une solution possible est l'expression $(1+50)*(25-10)$; il y a 780 solutions dans ce cas précis.

Les opérations arithmétiques

Les opérations arithmétiques autorisées sont représentées par le type suivant :

data Op = Add | Sub | Mul | Div

- Les opérations doivent être affichées. Ecrivez l'instance *Show Op*.
Exemple : *show Add == " + "*
- Ecrivez une fonction *valid :: Op -> Int -> Int -> Bool* qui teste si l'opération donnée appliquée à deux entiers est valide pour le jeu.

Exemples : *valid Mul 1 10 == True*

valid Div 2 3 == False

valid Sub 4 5 == False

valid Sub 5 4 == True

- Ecrivez une fonction *apply :: Op -> Int -> Int -> Int* qui applique l'opération sur les deux nombres donnés. On suppose que l'opération sur les deux entiers est valide pour le jeu.

Exemple : *apply Add 2 3 == 5*

Les expressions numériques

Les expressions numériques sont représentées par le type suivant :

data Expr = Val Int | App Op Expr Expr

d) Ecrivez l'instance Show Expr.

Exemples : $Val\ 10 == "10"$

$App\ Add\ (Val\ 10)\ (Val\ 12) == "10 + 12"$

$App\ Add\ (App\ Mul\ (Val\ 10)\ (Val\ 2))\ (Val\ 12) == "(10 * 2) + 12"$

e) Ecrivez une fonction $values :: Expr \rightarrow [Int]$ qui retourne la liste des valeurs d'une expression. (Cette fonction sera utile plus tard dans le TP).

Exemple : $values\ (App\ Add\ (App\ Mul\ (Val\ 10)\ (Val\ 2))\ (Val\ 12))$
 $== [10, 2, 12]$

f) Ecrivez une fonction $eval :: Expr \rightarrow [Int]$ qui retourne une liste contenant la valeur finale de l'expression sinon une liste vide.

Exemples : $eval\ (App\ Add\ (App\ Mul\ (Val\ 10)\ (Val\ 2))\ (Val\ 12)) == [32]$
 $eval\ (App\ Sub\ (Val\ 4)\ (Val\ 5)) == []$
 $eval\ (Val\ (-5)) == []$

Fonctions combinatoires

Vous aurez besoin de différentes fonctions retournant toutes les listes possibles qui satisfont certaines propriétés.

g) Etudiez la fonction $subs$ suivante. Que fait cette fonction ?

$subs :: [a] \rightarrow [[a]]$

$subs [] = [[]]$

$subs (x:xs) = map\ (x:) (subs\ xs) ++ subs\ xs$

h) Ecrivez la fonction $interleave :: a \rightarrow [a] \rightarrow [[a]]$ qui retourne toutes les manières d'insérer un élément dans une liste. Vous pouvez appliquer les fonctions $take$ et $drop$ ou définir une récursion sur le schéma de $subs$.

Exemple : $interleave\ 1\ [2, 3, 4] == [[1, 2, 3, 4], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1]]$

i) Ecrivez la fonction $perms :: [a] \rightarrow [[a]]$ qui retourne toutes les permutations possible d'une liste en complétant le code suivant :

$perms\ (x:xs) = concat\ [interleave\ x\ ys\ | \ ys <- \dots]$

La fonction $concat$ aplatit une liste de listes en une liste.

Exemples : $concat\ [[1], [2], [3]] == [1, 2, 3]$ et $concat\ [[[1]]] == [[1]]$

Exemple : $perms\ [1, 2, 3]$

$== [[1, 2, 3], [2, 1, 3], [2, 3, 1], [1, 3, 2], [3, 1, 2], [3, 2, 1]]$

- j) Ecrivez une fonction *choices* :: *[a]* -> *[[a]]* qui retourne tous les constructions possible à partir d'une liste. Cette fonction peut être définie comme les permutations des sous-listes. C'est le même schéma que *perms*.

Exemple : *choices [1,2,3]*
== *[[1,2,3],[2,1,3],[2,3,1],[1,3,2],[3,1,2],[3,2,1],*
[1,2],[2,1],[1,3],[3,1],[1],[2,3],[3,2],[2],[3],[[]]]

Le compte est bon ?

Il s'agit d'écrire une fonction qui teste si une expression donnée, compte tenu de la séquence de nombres, donne la valeur cible attendue.

- k) Ecrivez la fonction *solution* :: *Expr* -> *[Int]* -> *Int* -> *Bool*. Il faut s'assurer que l'expression donnée contient un sous-ensemble de la séquence de nombres donnée et que l'évaluation égale la cible. Utilisez la fonction *elem* :: *a* -> *[a]* -> *Bool* qui teste si un élément fait partie ou non d'une liste.

Exemple :
e = App Mul (App Add (Val 1) (Val 50)) (App Sub (Val 25) (Val 10))
solution e [1,3,7,10,25,50] 765 => True

Trouver les solutions par force brute

Une première approche, que l'on qualifiera de force brute, consiste à générer toutes les expressions possibles et à ne retenir que les expressions égales à la cible.

- l) Commencez par écrire une fonction *split* :: *[a]* -> *[[a],[a]]* qui génère toutes les façons possibles de découper une liste en deux listes non vides. Vous pouvez appliquer les fonctions *take* et *drop* dans une liste par compréhension.

Exemple : *split [1,2,3,4] == [[1], [2,3,4]], ([1,2],[3,4]), ([1,2,3],[4])]*

- m) Ecrivez la fonction *combine* :: *Expr* -> *Expr* -> *[Expr]* qui génère toutes les expressions possibles combinant les deux expressions données.

Pour cela vous définirez la liste *ops* suivante :

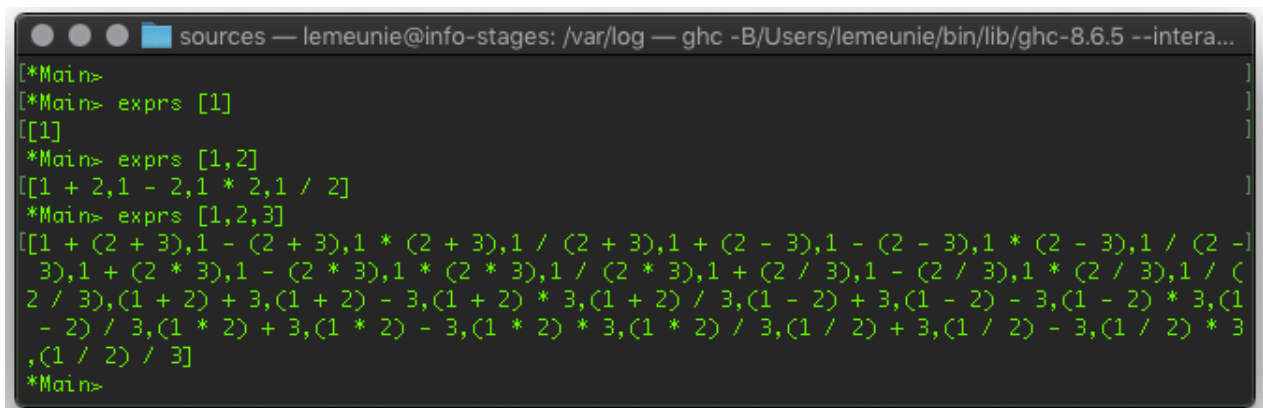
ops :: *[Op]*
ops = [Add, Sub, Mul, Div]

Exemple : *combine (Val 5) (Val 6)*
== *[App Add (Val 5) (Val 6), App Sub (Val 5) (Val 6),*
App Mul (Val 5) (Val 6), App Div (Val 5) (Val 6)]
== *[5 + 6, 5 - 6, 5 * 6, 5 / 6]* affiché dans le terminal

- n) Ecrivez la fonction `exprs :: [Int] -> [Expr]` qui génère toutes les expressions arithmétiques possibles dans le contexte d'une séquence de nombre donnée.

Complétez le code suivant :

```
exprs :: [Int] -> [Expr]
exprs [] = ...
exprs [n] = ...
exprs ns = [ e | (ls, rs) <- ... ,
                 l     <- ... ,
                 r     <- ... ,
                 e     <- combine l r ]
```



```
sources — lemeunie@info-stages: /var/log — ghc -B/Users/lemeunie/bin/lib/ghc-8.6.5 --intera...
[*Main>
[*Main> exprs [1]
[1]
[*Main> exprs [1,2]
[1 + 2,1 - 2,1 * 2,1 / 2]
[*Main> exprs [1,2,3]
[1 + (2 + 3),1 - (2 + 3),1 * (2 + 3),1 / (2 + 3),1 + (2 - 3),1 - (2 - 3),1 * (2 - 3),1 / (2 - 3),1 + (2 * 3),1 - (2 * 3),1 * (2 * 3),1 / (2 * 3),1 + (2 / 3),1 - (2 / 3),1 * (2 / 3),1 / (2 / 3),
(1 + 2) + 3,(1 + 2) - 3,(1 + 2) * 3,(1 + 2) / 3,(1 - 2) + 3,(1 - 2) - 3,(1 - 2) * 3,(1 - 2) / 3,(1 * 2) + 3,(1 * 2) - 3,(1 * 2) * 3,(1 * 2) / 3,(1 / 2) + 3,(1 / 2) - 3,(1 / 2) * 3,(1 / 2) / 3]
[*Main>
```

- o) Enfin écrivez la fonction `solutions :: [Int] -> Int -> [Expr]` qui retourne la liste de toutes les expressions valides qui égalent le nombre cible. Utilisez `choices` pour générer tous les sous-ensembles possibles de séquence de nombres, `exprs` pour générer toutes les expressions sur les séquences possibles et `solution` pour vérifier la validité de chaque expression.

Testez avec l'expression : `print (solutions [1,3,7,10,25,50] 765)`. Afficher toutes les solutions peut prendre plusieurs dizaines de secondes.

Remarques finales

Pour améliorer le temps du traitement, bien que la performance du programme dépasse déjà sans problème la performance d'un humain moyen, il est possible de faire deux optimisations substantielles (de l'ordre d'un facteur 140) :

- ne pas générer toutes les expressions possibles mais générer et vérifier en même temps si l'expression est valide (par exemple sur une séquence de 6 nombres il y a uniquement 14% d'expressions valides) ;
- ne pas valider certaines expressions déjà générées du fait des propriétés algébriques (par exemple $2 + 3 = 3 + 2$), ce qui réduit aussi très fortement le nombre d'expressions à traiter.

Références web

Site officiel Haskell

<https://www.haskell.org>

Site des modules officiels Haskell

<https://downloads.haskell.org/~ghc/latest/docs/html/libraries/>