Dougy Hugo P2023

Notes: Devoir final + TP

Partie 1:

Pour exécuter les instructions: ghci Calc.hs puis rentrer dans le module main. On remarque que quand l'expression est incorrecte, le programme retourne la dernière valeur saisie. Ex: 1 1 1 - 133321 -> Résultat: 133321

```
3) Pour construire la fonction montre_exp_postfix on fait comme pour
montre_exp. On définie d'abord le cas où la chaîne entrée est juste un Int.
Ensuite le cas où il y a plusieurs opérateurs et opérandes.
On utilise la récursivité car si l'expression comporte plus de 2 opérandes et
1 opérateur il faut découper les valeurs de e1 et e2.
```

```
4) montre_exp_prefix: on utilise la même méthode que pour la question
précédente. on doit bien mettre les parenthèses pour toute l'opération et
entre l'opérateur.
```

Partie 2:

1) Plusieurs tests de la fonction lis_entier:
avec l ="2": lis_entier l -> ("2","")
avec l ="+": lis_entier l -> ("","+")
avec l ="2+": lis_entier l -> ("2","+")
avec l ="2+2": lis_entier l -> ("2","+2")
avec l ="-+*2*+-" *lis_entier l -> ("*","-+2+")
La fonction est écrite en utilisant deux fonctions. Une de la librairie standard de Haskell: span et l'autre de la librairie Data.Char, isNumber.
D'après ghci: span:: (a->Bool) -> [a] -> ([a],[a]) cette fonction applique un prédicat à une liste et retourne un couple de liste dont le 1er élément vérifie la condition. Si le premier élément ne vérifie pas la condition, l'ensemble de la liste est placée dans le 2nd élément.
Ensuite, isNumber:: Char -> Bool prend en entrée un Char et renvoie un Bool. Elle renvoie True si le Char est un nombre (ne fonctionne qu'avec des positifs) et False sinon.

Donc dans lire_entier: span prend en argument isNumber pour le prédicat ainsi que la liste s d'entier.

```
2) A première vue, la fonction mange_blancs:: String -> String supprime les
espaces en début de liste.
par exemple:
    avec s = "LOL": mange_blancs s -> "LOL"
    avec s = " LOL": mange_blancs s -> "LOL"
    avec s = "    L   O   L   ": mange_blancs s -> "L   O   L"
```

Elle est écrite de manière récursive. On définie le cas où l'entrée est une liste sans espace. On renvoie donc cette même liste.
Si elle contient un espace au début, on applique la même formule avec la liste sans le premier espace … la condition d'arrêt sera qu'il n'y a plus d'espace au début.

3) La fonction separe_gauche:: Char -> String -> [String] prend en entrée un char (c) et une liste (x:xs).
Elle est définie de manière récursive. Si la liste d'entrée est vide, on renvoie la liste vide.

Ensuite, si le premier élément de la liste est c, on concatène (ie remplacer ici) c par "\n" dans la liste et on recommence.
Si le premier élément est différent de c, on utilise la fonction ajoute_au_premier:: a -> [[a]] -> [[a]].
Cette fonction permet de concaténer un élément à la liste en première position.

Ainsi, dans le cas où x != c, on ajoute le premier élément à la liste que l'on va renvoyer etc... donc à chaque fois que l'on va avoir x == c on va remplace c par "\n" et créer une nouvelle liste.

Exemple: avec c = 'c' separe_gauche c "abcdcf" -> ["ab\n","d\n"]

4) fait dans la question précédente

5) La fonction ajoute_au_premier est utilisé pour écrire la liste dans l'ordre sans perdre d'information.

Partie 3:

1) Pour utiliser Oper: On fait une disjonction de cas en fonction de l'opérateur (Plus,Moins,Fois). Ainsi on créer la fonction eval de manière récursive premièrement si l'expression est seulement une valeur int, ensuite en fonction de l'opérateur (Plus,Moins,Fois), on calcule l'expression 1 et l'expression 2 jusqu'à que l'on ait qu'une valeur Int.

2) On importe le module Evaluation dans Main et on ajoute la ligne 'print (eval e)'. e est bien de type expression d'après la fonction entrer_expression_entier.
Ainsi on retourne bien la valeur de l'expression entrée.

FIN V1.

Partie 4:

1) On peut voir seulement avec un exemple que la version 1 ne peut pas manipuler de grands entiers. Par exemple avec le test suivant: 10000000000000000000 10000000000000000000 + -> Résultat: (7766279631452241920+7766279631452241920) -2914184810805067776 On voit bien que le calcul est totalement faux. La taille des Int dans Haskell est de 64bits et 10^20 > (2^64)-1

2) En modifiant le type de Exp en Integer plutôt que Int et de la fonction eval, on peut maintenant faire autant de calculs que l'on souhaite car le type Interger n'est pas borné comparé à Int.

FIN V2.

Partie 5:

1) Pour la fonction filtre_flottant, on utilise les gardes ainsi que les fonctions de la libraire standard elem et de la librairie Data.Char isNumber. De plus on créer la liste de Char [‘.’,‘,’,‘e’,‘E’,‘-’]. On teste donc si le Char en entrée est un chiffre avec isNumber et on teste si il fait partie de la liste grâce à la fonction elem.

2) Pour la fonction lis_flottant :: String -> (String,String) on s'inspire de la fonction lis_entier. Sauf que cette fois ci, on utilise le prédicat filtre_flottant.

3) Pour que V3 calcule les flottants, il a fallu modifier le type dans Type.hs dans la définition de Exp. De même pour le module Evaluation.
De plus, il fallait modifier tous les lis_entier présents dans la fonction parseur du module du même nom. Si on fait maintenant un test avec les flottant, on a par exemple:
-5.3e2 10.666e3 + -> 10136.0

4) Pour ajouter l'opérateur division, on le définit dans Type.hs. Ensuite on ajoute le cas Div à notre fonction eval. Enfin, on doit modifier Parseur.hs pour y ajouter également le cas Div. Ainsi on peut faire des opérations avec l'opérateur division.
    Exemple: 12 4 / -> 3.0

5) J'ai essayé de simplifier et de réduire le code le plus possible en gardant les fichiers importants. Premièrement, les modules expression et exemples sont inutiles pour calculer. je l'ai donc supprimés. De plus que le main avec montre_exp …

Dans parseur, j'ai également supprimé la fonction lis_entier puisqu’on a lis_flottant et import takewhile car on ne l'utilise pas
De même de que la fonction montre_oper avec show.

En testant avec la commande wc -l, s’obtient que la sommes des lignes des fichiers est égale à 66.

FIN DM

TP3: Exercice 1:

1) On réécrit cumule sans type de manière à pourvoir l'utiliser généralement. On modifie donc le prototype de la fonction.

2) On définie somme2 comme dans le TP2.

3) On définie somme_num avec le prototype suivant: somme_num :: Num => p -> [p] -> p.

4) On créer un module Agreable contenant la classe Agreable contenant neutre et operation.

5) Après la classe, on créer l'instance Int pour la classe Agreable pour l'addition. On choisit donc le neutre = 0 et l'opération +.

6) On déclare cumule_agreable de type Agreable => [a] -> a
cette fonction ne calcule que la somme d'une liste de Int.

7) Pour tester, on définit dans Agreable.hs une liste l :: [Int]
et on lui applique une valeur, ici l = [1,2]
Ensuite, dans l'interpréteur, on saisie cumule_agreable l -> 3

8) On créer une instance pour bool dans la classe Agreable. On prend False pour le neutre et l'opérateur || pour l'opération.

9) On créer une instance pour [a] dans la classe Agreable. On prend [] pour le neutre et ++ comme opérateur.

10) On créer simplement un nouveau fichier appelé IE et on import le module Agreable.

11) Pour créer le type Int_ex, on utilise data. data Int_ex = Moins_inf | I Int deriving Show. Où I est le constructeur de type, et deriving Show permet d'en faire une instance de la classe Show.

12) On créer la fonction max_ie définissant le max avec le type Int_ex. On fait une disjonction de cas pour savoir quel résultat renvoyer.

13) On créer l'instance Agreable pour Int_ex. Ainsi, on définie l'élément neutre comme Moins_inf et l'opération avec max_ie.

14) D'après l'interpréteur Haskell, la fonction map I a pour prototype, [Int] -> [Int_ex] donc map I transforme une liste d'entier passée en paramètre en une liste de type Int_ex. Par exemple, (map I) [1,2,3] -> [I 1, I 2, I 3]

15) Pour créer max_liste, on se sert des fonctions cumule_agreable et (map I)

```
max_liste [] = I 0
max_liste l = cumule_agreable (map I) l
```

FIN TP
FIN NOTES