Colle 10 : Le compte est bon

1 février 2022

Commentaires sur les exceptions :

1. Pour définir une exception bla, il suffit d'écrire : `Exception bla;;`
2. Pour lever une exception bla dans une code, il suffit d'écrire `raise bla`
3. Pour utiliser une fonction qui peut éventuellement lever une exception bla à l'intérieur d'une autre fonction et ignorer un appel avec exception on utilise la commande suivante :
   `try ( "vous faites l'appel a votre fonction ici" ) with bla->()`

1 Générer les arbres binaires et les permutations

Dans ce TP, nous n'étiquetterons pas les arbres, nous nous limitons donc aux squelettes d'arbres binaires définis par :

```
type tree = Nil | Node of tree* tree;;
```

▷ **Question 1.** Soit $n > 0$, expliquez comment générer tous les arbres binaires à n noeuds si l'on sait générer tous les arbres à $0 \leq k < n$ noeuds. ◁

▷ **Question 2.** Ecrire une fonction `sous_arbre_gauche : tree->tree list->tree list` qui prend en entrée un arbre et une liste d'arbres et qui construit une nouvelle liste d'arbres dont chacun des élément est un arbre dont le sous arbre gauche est `g` et le sous-arbre droit un élément de la liste passée en argument. ◁

▷ **Question 3.** Ecrire une fonction `construire : tree list->tree liste->tree list` qui prend en entrée deux listes `l1` et `l2` d'arbres et qui renvoie la liste de tous les arbres que l'on peut composer avec comme sous arbre gauche un élément de `l1` et comme sous arbre droit un élément de `l2`. ◁

▷ **Question 4.** Ecrire une fonction `genere : int -> tree list` qui génère la liste de tous les arbres à `n` noeuds (on remplira un tableau de listes tel que la case k contiendra la liste des arbres à k noeuds).

Pour tester votre fonction vous pouvez vérifier que vous obtenez le bon nombre d'éléments sachant que les premiers nombres de Catalan (la $n$eme nombre de Catalan est le nombre d'arbres à n noeuds) valent : 1, 1, 2, 5, 14, 42, 132, 429, 1 430 et 4 862. ◁

▷ **Question 5.** Ecrire une fonction `insere_tete : 'a->'a list list->'a list list` qui insére un élément en tête de chacune des listes de la liste argument.

Par exemple, `insere_tete 2 [[1;2;3];[3;3];[5;4;3;2;1]]` renvoie +[[2;1;2;3];[2;3;3];[2;5;4;3;2;1]]+ ◁

▷ **Question 6.** Ecrire une fonction `insere_toute_place : 'a -> 'a list->'a lis list` qui prend en entrée un élément et une liste `l` et renvoie la liste des listes où on a inséré l'élement à toutes les places possible de `l`.

Par exemple, `insere_toute_place 4 [1;2;3]` renvoie `[[4;1;2;3];[1;4;2;3];[1;2;4;3];[1;2;3;4]]`. ◁

▷ **Question 7.** Ecrire une fonction `insere_toute_place_toute_liste : 'a->'a list list ->'a list list` qui insère un élément à toutes les places possibles de chacune des listes contenues dans la liste argument. La fonction renverra la liste de toutes les listes résultats. ◁

▷ **Question 8.** Ecrire une fonction `gen_perm : 'a list -> 'a list list` qui génère toutes les permutations d'une liste. ◁

2 Arbre pour "Le compte est bon"

Dans "Le compte est bon", les candidats doivent trouver une séquence d'opérations à réaliser sur un ensemble de nombres donnés afin de s'approcher le plus possible d'un objectif. Pour simplifier, nous allons considérer que les candidats doivent utiliser tous les nombres donnés. On peut alors modéliser un calcul par un arbre binaire dont les noeuds sont des opérations $(+, -, /, \times)$ et les feuilles (les Nil dans nos squelettes) sont les nombres donnés.

Dans un premier temps, nous considérons que les nombres de l'ensemble sont donnés dans un certain ordre à respecter, c'est-à-dire qu'ils forment une liste. La figure donne un exemple d'arbre de calcul pour la liste de nombre `[6;25;50;75;100]`. Comme l'on peut le constater, la liste feuilles de gauche à droite est bien `[6;25;50;75;100]`. De plus, la liste des opérations dans l'ordre infixe est $[\times; +; \times; +]$. On définit cet arbre comme étant l'arbre de calcul pour le squelette d'arbre binaire T de la figure, pour les valeurs aux feuilles `[6;25;50;75;100]` et pour les opérations aux noeuds $[\times; +; \times; +]$ et que l'on note $A(T, [6; 25; 50; 75; 100], [\times; +; \times; +])$.

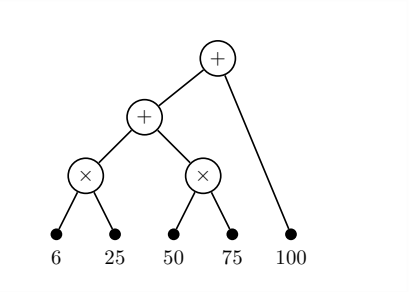

Figure 1 – Exemple d'arbre de calcul dont les feuilles forment la liste `[6;25;50;75;100]`

▷ **Question 9.**
Définir une nouvelle exception `Invalid_div`.
Définissez quatre opérations binaires `add, sub, mul et div` sur les entiers. On impose que `div` doit lever une exception `Invalid_div` si le résultat de la division n'est pas un entier. ◁

▷ **Question 10.** Ecrire une fonction `eval_tree : tree -> 'a list -> 'a bin_op list -> 'a` qui étant donnés t, $values$ et ops donne le résultat de l'évaluation de l'arbre de calcul $A(t, values, ops)$. On pourra commencer par écrire une fonction récursive qui renverra le triplet : valeur calculée, liste des valeurs restantes, liste des opérateurs restants. Testez votre fonction sur l'exemple de la figure (vous devez trouver 4000) :

```
let t =Node(Node(Node(Nil,Nil),Node(Nil,Nil)),Nil);;
let val=[6;25;50;75;100];;
let op=[mul;add;mul;add];;
```

◁

▷ **Question 11.** Si l'arbre possède n feuilles alors combien a-t-il de noeuds? ◁

▷ **Question 12.** Dans l'exercice précédent, l'arbre de calcul était entièrement défini. Nous relâchons maintenant cette contrainte et nous donnons seulement le squelette de l'arbre et les valeurs aux feuilles. Nous voulons alors évaluer l'arbre selon toutes les combinaisons d'opérations aux noeuds parmi $\{add, sub, div, mul\}$. De plus, nous définissons le type d'une fonction de résultat ainsi :
`type 'a res_f = 'a -> tree -> 'a list -> 'a bin_op list -> unit;;` (Le type `bin_op` n'étant pas vraiment défini, il est inutile de déclarer ce type, on se contentera d'utiliser de tels fonctions).
Ecrire une fonction `eval_tree_with_all_ops : tree -> int list -> int res_f -> unit` qui étant donnés `t,values` et `f`, appelle la fonction `f` pour tous les arbres de calcul `A(t, values,_)` possibles en lui fournissant en entrée le résultat de l'évaluation ainsi que les différents paramètres de l'arbre. On prendra garde au fait que l'opération `div` peut lancer deux exception `Invalid_div` et `Division_by_zero`. Si une exception est lancée lors de l'évaluation d'un arbre de calcul, on poursuivra l'algorithme en ignorant cet arbre. ◁

▷ **Question 13.** Dans l'exercice précédent, les nombres aux feuilles avaient un ordre fixé. Nous relâchons maintenant cette contrainte et voulons évaluer l'arbre selon toutes les combinaisons possibles d'opérations aux noeuds et toutes les permutations possibles des valeurs aux feuilles. Ecrivez une fonction
`eval_tree_with_all_perms : tree -> int list -> int res_f -> unit` qui réalise cette tâche. ◁

▷ **Question 14.** Dans l'exercice précédent, le squelette de l'arbre de calcul était fixé. Nous relâchons maintenant cette dernière contrainte et nous voulons évaluer tous les arbres de calculs possibles avec toutes les variations décrites dans l'exercice précédent. Ecrivez une fonction
`eval_game_with_all_trees : int list -> int res_f -> unit` qui réalise cette tâche. ◁

3 "Le compte est bon"

Nous allons maintenant utiliser les fonctions précédentes pour résoudre le problème du "compte est bon".

▷ **Question 15.** Ecrire une fonction :
`find_game_best_solution : int list -> int -> int * (int list) * tree * ( bin_op list)` qui étant donné une instance du problème du compte est bon, renvoie la valeur la plus proche de la valeur objective avec la liste des valeurs dans l'ordre choisi, l'arbre de calcul et la liste des opérations correpondant à une énumération infixe des noeuds de l'arbre. Pour cela, on utilisera la fonction `eval_game_with_all_trees` avec la fonction `better res arbre perm op target` définie de la manière suivante :

```
let better res arbre perm op target sol t values ope =
   if(abs(sol-target)<abs(!res-target)) then
      begin
      res:=sol;
      arbre:=t;
      perm:=values;
      op:=ope;
      end;;
```

Résoudre l'instance suivante : 25, 50, 75, 100, 3, 6.
But à atteindre : 952.
Vérifiez que votre algorithme trouve une solution valide à cette instance.
On utilisera l'affichage suivant des opérations :

```
let convert_op ops=
  let identify_op op =
        if op == add then "+" else if op == subb then "-"
        else if op == mul then "*" else "/" in
        List.map (identify_op) ops;;
```

◁