Série d'exercices #10

IFT-2035

June 9, 2025

10.1 Compteurs de références

Soit une libraire de gestion de listes simplement chaînées en C:

```
typedef struct list list;
struct list {
  int refcount;
  void *value;
  list *next;
}
list *list_new    (void *car, list *cdr);
void *list_car    (list *l);
list *list_cdr    (list *l);
/* Maintenance des compteurs de référence.  */
void  list_incr   (list *l);
void  list_decr   (list *l);
```

Écrire le code des fonctions proposées.

Compléter en ajoutant une opération `list_map`.

Justifiez pourquoi les incréments et décréments que vous avez judicieusement placés sont suffisants pour garantir que le comportement sera toujours correct. En extraire une convention spécifiant où doivent être ajoutés les incr/décr, principalement clarifier qui de l'appelant ou de l'appelé est en charge de quelles incrémentations et quelles décrémentations et pourquoi vous avez fait ces choix; décrire aussi si ces règles s'appliquent uniformément à toutes les fonctions, ou si certaines des fonctions ci-dessus sont spéciales.

Que se passe-t-il si vous voulez manipuler des listes de listes?

10.2 Transfert de *ownership* en Rust

Soit le code Rust ci-dessous:

```
fn main() {
   let vec0 = Vec::new();

   let mut vec1 = fill_vec(vec0);

   // Ne pas changer l'instruction suivante!
   println!("{} has length {} content `{:?}`",
            "vec0", vec0.len(), vec0);

   vec1.push(88);

   println!("{} has length {} content `{:?}`",
             "vec1", vec1.len(), vec1);
}

fn fill_vec(vec: Vec<i32>) -> Vec<i32> {
   let mut vec = vec;
   vec.push(22);
   vec.push(44);
   vec.push(66);
   vec
}
```

Le compilateur signale une erreur sur le premier *println!* indiquant que *vec0* ne peut pas être utilisé.

1. Décrire le problème.
2. Proposer une manière de changer le code qui évite ce problème.
3. Discuter des alternatives possibles.

Note: Le {?:} dans le *println!* sert à imprimer le contenu interne des structures de données dans un format destiné seulement au debugging.

10.3 Structure de données fonctionelle

Soit le type suivant en Haskell qui définit un arbre binaire que l'on peut utiliser pour représenter une table associative (qui associe des *clés* de type Int à des valeurs de type β):

```
data TreeMap b = Empty | Node Int b (TreeMap b) (TreeMap b)
```

L'exercice est de définir les opérations typiques sur une telle structure de donnée. Bien sûr, pour être utile l'arbre doit être maintenu dans l'ordre: toutes les clés dans la branche de gauche d'un Node doivent être plus petites que la clé du noeud, et vice versa pour la branche de droite.

Il y a trois opérations:

- *tmLookup*: rechercher la valeur associée à une clé passée en paramètre.
- *tmInsert*: ajouter une entrée (donnée sous la forme d'une clé et de sa valeur) dans la table.
- *tmRemove*: enlever une entrée (dont la clé est passée en paramètre).

Ces fonctions doivent être totales (elles terminent toujours et ne doivent jamais signaler d'erreur).

1. Donner un type acceptable pour chacune de ces trois fonctions.
2. Donner le code des deux premières fonctions (points de karma en bonus pour *tmRemove* qui est plus pénible et moins souvent nécessaire).
3. Pour un arbre de profondeur 10 contenant $\sim$1000 éléments, quel est le coût approximatif de chacune de ces fonctions.

Pour rendre l'exercice plus utile, il est important de faire ces étapes dans l'ordre: i.e. ne pas écrire le code avant d'avoir décidé du type des fonctions.