TD #9&10 – DEV

27 novembre 2023
Erwan Kerrien
Abdel Bourjij

Rappel : si vous avez des questions sur ce TD ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

1 Listes chaînées

L'objectif de cet exercice est de proposer une représentation via une liste chaînée des opérations vues en cours sur une liste récursive. Par souci de concision, nous appellerons Liste la sorte pour la liste récursive. On rappelle la signature de cette sorte :

```
Sorte : Liste
Utilise : Elément, booléen
Opérations :
  liste_vide  :  -> Liste
  EstVide     : Liste -> booléen
  Contenu     : Liste -> Elément
  Succ        : Liste -> Liste
  Créer       : Elément x Liste -> Liste
  Détruire    : Liste -> Liste
```

Cette signature pose des questions pratiques concernant les fonctions `Contenu` et `Succ` : elle ne spécifie pas si les accès sont en lecture simple (on veut juste récupérer les valeurs) ou en écriture (on veut changer les valeurs). Par exemple, prenons une Liste dont l'Elément est un entier. Comment faire pour incrémenter la valeur stockée dans la cellule en tête de liste ? Il faudrait employer une instruction comme `Contenu(`L`)` $\leftarrow E$. Ceci est possible avec des langages comme C++ ou Python qui offrent des mécanismes pour ce faire. Ce n'est pas possible en C (un appel de fonction ne peut pas être une L-value, c'est-à-dire se trouver à gauche d'une affectation). On va donc modifier la signature de notre liste pour la rendre plus compatible avec un langage comme C.

```
Sorte : Liste
Utilise : Elément, booléen
Opérations :
  liste_vide  :  -> Liste
  EstVide     : Liste -> booléen
  ContenuLire : Liste -> Elément
  ContenuModif: Liste x Elément ->
  SuccLire    : Liste -> Liste
  SuccModif   : Liste x Liste ->
  Créer       : Elément x Liste -> Liste
  Détruire    : Liste -> Liste
```

Les opérations `ContenuLire` et `SuccLire` sont les mêmes que dans la version précédente : elle permettent d'accéder en lecture respectivement au contenu et au successeurs de la cellule en tête de liste. Les fonction `ContenuModif` et `SuccModif` en sont les versions en écriture : la valeur à stocker est passée en deuxième argument (le premier reste toujours notre liste). Ces opérations ne renvoient rien.

Plus précisement, voici les spécifications pour ces opérations :

- liste_vide doit renvoyer une liste vide. Pour fixer les idées, en C, ce sera un pointeur NULL (adresse mémoire = 0x00000000)
- `EstVide` renvoie Vrai si la Liste passée en argument est liste_vide, et Faux sinon.
- `ContenuLire` va renvoyer l'Elément contenu dans la première cellule de la liste. element_invalide sera renvoyé si la liste est vide.

— `ContenuModif` va écrire un Elément dans la première cellule de la liste. Il ne se passe rien si la liste est vide ou si on essaie de stocker element_invalide.
— `SuccLire` va renvoyer la Liste suivante, c'est-à-dire celle qui commence avec la deuxième cellule de la liste. Cette fonction renverra liste_vide si la liste en entrée est vide.
— `SuccModif` va stocker la Liste passée en deuxième argument comme successeur de la première cellule de la liste. Cette fonction ne fait rien si la liste en premier argument est vide. En revanche, on peut bien stocker une liste vide et donc la passer en deuxième argument.
— `Créer` va en effet créer une cellule, ce qui implique, ainsi que nous l'avons vu en cours, d'allouer la place mémoire nécessaire pour une cellule, pour ensuite l'initialiser (avec le bon élément et liste_vide en successeur). La cellule (ou un pointeur vers cette cellule si on utilise le langage C) sera renvoyée. Je vous rappelle le cours : une cellule est une liste à un seul élément.
— `Détruire` va faire l'opération inverse sur la mémoire, c'est-à-dire qu'elle va libérer la mémoire allouée pour la première cellule de la liste et renvoyer le successeur de cette cellule, c'est-à-dire le reste de la liste.

Comme la sorte Liste utilise la sorte Elément, vous avez besoin de savoir quelles sont les opérations disponibles sur cette sorte. Voici comment cette sorte est définie.

```
Sorte : Elément
Utilise : booléen
Opérations :
 élément_invalide : -> Elément
 ElémentEstValide : Elément -> booléen
 ElémentAfficher : Elément ->
 ElémentComparer : Elément x Elément -> Booléen
```

Les spécifications des opérations sur un Elément sont les suivantes :

— element_invalide doit renvoyer quelque chose de type Elément qui soit bien identifié comme une valeur interdite. Par exemple, ce pourra être -1 si Elément est un entier positif, `NaN` (*Not A Number*) si Elément est un réel. Si Elément est un type composite (avec plusieurs champs rassemblés), on pourra aussi ajouter un champ dit "drapeau" (flag) qui sera un booléen indiquant si l'Elément est valide ou pas.
— `ElémentEstValide` renverra Faux si l'Elément est element_invalide, et Vrai sinon.
— `ElémentAfficher` affichera l'Elément passé en argument. L'affichage pourra par exemple gérer de manière élégante le cas où Elément n'est pas valide.
— `ElémentComparer` compare deux Elément et renvoie Vrai s'ils sont identiques, et Faux sinon.

1.1 Questions

1. Commencez par prendre le temps de bien lire les spécifications de sorte ci-dessus. L'objectif du TD est d'implémenter un certain nombre de fonctions qui opèrent sur une Liste. Vous n'aurez pas le droit d'utiliser autre chose que les opérations décrites, en plus des structures algorithmiques et types de base que nous avons déjà l'habitude de manipuler.
2. Écrire la fonction `Longueur` qui prend en entrée une Liste L et en renvoie la longueur (nombre d'éléments, identique à la fonction `Taille` du cours).

3. Écrire la fonction `Afficher` qui affiche les éléments d'une Liste L

4. Écrire la fonction `Rechercher` qui teste si un Elément E est dans une Liste L est renvoie la Liste dont il est le premier élément si c'est le cas, ou liste_vide sinon.
   Note : cela donne une implantation directe de la fonction `EstDans` *vue en cours puisque cette fonction sera équivalente à* **Renvoyer** `non(EstVide(Rechercher(`L`,`E`)))`

5. Écrire la fonction `Dernier` qui prend une Liste L, et qui renvoie la liste formée de la dernière cellule de L.

6. Écrire la fonction `Supprimer` qui prend une Liste L, ainsi qu'un entier r et supprime l'élément de rang r dans L. La fonction renvoie la liste mise à jour. *Attention à bien* `Détruire` *la cellule correspondant à l'élément supprimé.* Si le rang r est supérieur à la longueur de la Liste, rien n'est fait.

Que faudrait-il faire si la fonction `Détruire` ne renvoyait pas le reste de la liste, et était simplement une procédure ?

7. Écrire la fonction `Concaténer` qui prend entrée deux Liste $L1$ et $L2$ et les concatène, autrement dit, elle ajoute $L2$ à la suite de $L1$. La fonction renverra $L1$ ainsi mise à jour. Attention au cas où $L1$ est la liste_vide : dans ce cas, la fonction renverra $L2$

8. Écrire la fonction `Ajouter` qui prend entrée une Liste L, un Elément E, et un entier r, et qui ajoute E à L de telle manière qu'il a le rang r dans la liste mise à jour. La fonction renvoie cette liste mise à jour. Si le rang r est supérieur à la longueur de la Liste, E est ajouté à la fin.

9. Écrire la fonction `Inverser` qui recopie une Liste L dans une nouvelle Liste $Linv$, mais avec un ordre inverse pour les éléments. La fonction renvoie $Linv$.

   Que faudrait-il faire pour faire une fonction `Copier` qui recopie L dans le bon ordre ?

10. Écrire la fonction `Vider` qui vide une Liste L (à la fin L vaut liste_vide) en libérant proprement la mémoire allouée pour toutes les cellules de L