TD6 : Dérécursivation

TOP : Techniques and tOols for Programming – 1A

★ Exercice 1: Dérécursivation de fonctions sur les chaînes de caractères.

L'objectif de cet exercice est de revenir sur les fonctions sur les chaînes vues lors du TD2 afin de les dérécursiver. On rappelle les opérateurs de base du type chaîne :

$$\begin{cases} \texttt{Nil} \mapsto \text{La liste vide} & \\ \mathit{list}.\texttt{head} \mapsto \text{Premier caractère de la liste } \mathit{list} & \text{(défini ssi } \mathit{list} \text{ n'est pas vide)} \\ \mathit{list}.\texttt{tail} \mapsto \mathit{list} \text{ privée du premier élément} & \text{(défini ssi } \mathit{list} \text{ n'est pas vide)} \\ \mathit{entier} :: \mathit{list} \mapsto \text{Concaténation de l'entier } \mathit{entier} \text{ et de la liste } \mathit{list} & \end{cases}$$

▷ **Question 1:** *est_membre* : $\begin{cases} List \times Int \mapsto Bool \\ \text{retourne VRAI ssi l'entier fait partie de la liste} \end{cases}$

Réponse

est_membre(ch,c)

```
si list = Nil alors FAUX
              sinon si list.head == i alors VRAI
                                        sinon est_membre(list.tail ,i)
```

Attention, il ne faut pas trouver un algorithme itératif répondant à la question (ce qui est possible vu comment la question est dure), mais bien appliquer la méthode du cours sur un cas simple pour se faire la main avant plus compliqué.

Ici, tout va bien, c'est récursif terminal.

est_membre(ch,c)

```
    list_tmp = list
    while (list_tmp != Nil) {
      si ch.head = i alors return VRAI
                     sinon ch_tmp = ch.tail
    }
    return FAUX
```

Fin réponse

▷ **Question 2:** *occurence* : $\begin{cases} List \times Int \mapsto \mathbb{N} \\ \text{retourne le nombre d'occurences de la valeur dans la liste} \end{cases}$

Réponse

occurences(list,c)

```
si list = Nil alors 0
              sinon si list.head = i alors 1 + occurence(list.tail,i)
                                       sinon     occurence(list.tail,i)
```

Pour dérécursiver, il faut passer sous forme terminale. Encore une fois, on peut aller tout droit, mais ce n'est pas l'objectif du TD. Donc, on prend le temps de faire une fonction avec plus d'arguments, très exactement un argument supplémentaire par opération réalisée lors de la remontée. Ici, juste une addition, donc juste un arg supplémentaire.

Il faut également rappeller qu'on a le droit de le faire car l'addition est associative et commutative. On initialise l'arg supplémentaire avec l'élément neutre de l'opération, j'ai nommé 0.

occurences(list,c) terminale

```
occurences(list, c) = occurences_terminale(list, c, 0)

occurences_terminale(list, c, n)
  si list==Nil alors n
               sinon si list.head == c alors occurences_terminale(reste(list),c,n+1)
                                       sinon occurences_terminale(reste(list),c,n)
```

Si on déplie une séquence d'appel, on voit bien que les additions qui avaient lieu à la remontée ont maintenant lieu lors de la descente.

On est prets pour appliquer la recette de cuisine pour dérécursiver, qui est :

– Copie locale des arguments portant la récursion
– Tant que la condition du cas terminal n'est pas atteinte

  - Faire les opérations réalisées dans le cas général à la descente
  - Modifier l'argument portant la récursion
- Faire le traitement du cas terminal

occurences(list,c) itérative
```
occurences(list, c) =
  list_tmp = list // c'est lui qui porte la récursion => on copie
  n = 0 // pas besoin de copier ca, mais faut le créer pour émuler
        //  l'initialisation faite lors du lancement de helper
  while (list!=Nil ) // while (!cond_finale)
     si premier(list) = c alors n=n+1
     list_tmp = reste(list_tmp)
  // y'a rien a faire dans le cas terminal pour occurence
```

Fin réponse

▷ **Question 3:** *retourne* : $\begin{cases} List \mapsto List \\ \text{retourne la liste lue en sens inverse} \end{cases}$

Réponse

Là, à première vue, c'est encore plus compliquée car la version de base est assez difficile à dérécursiver car l'opération à la remontée est la concaténation (::), qui n'est pas commutative (ca veut dire A::B != B::A ;).

retourne(list) -- version bourinne
```
si list = Nil alors Nil
            sinon dernier(list) :: retourne1(saufdernier(list))
```

L'idée est de changer l'algorithme, ne serait-ce que pour passer de $O(n^2)$ à $O(n)$. **Comment leur faire trouver mieux :** Demandez leur de réfléchir à comment ils inversent l'ordre d'une pile de cartes : on prend une pile supplémentaire, on passe le premier de la pile de départ sur l'autre, et on recommence avec la deuxieme de la pile de départ. Si ca aide pas, faut détailler un exemple :

A trier ABC ⇝

ABC	∅
BC	A
C	BA
∅	CBA

⇝ résulat = CBA

retourne(list) -- avec helper
```
retourne2(list):
  retourne2_helper(list,Nil)

retourne2_helper(list_todo,list_done):
  si list_todo = Nil alors list_done
                   sinon retourne2_helper(suite(list_todo),
                                          premier(list_todo) :: list_done)
```

La concaténation (::) n'est pas commutative, mais on a remplacé les dernier() en premier() en meme temps qu'on changait l'ordre, alors ca va. L'idée est de prendre un accumulateur initialisé à ce que la récursion trouve au début de la remontée (la chaine vide), et faire les opérations dans l'ordre que la remontée l'aurait faite. Je saurais pas vous dire ca en terme technique (dernier() n'est pas l'inverse de premier(), ni l'opposé, et pas vraiment l'opération symétrique non plus), mais ca marche bien.

Ce qui est bô, c'est que du coup c'est récursive terminal, et on va donc pouvoir appliquer notre recette de cuisine.

retourne(list) -- itérative
```
retourne_iter(list):
  list_tmp = list
  res = Nil // ce qui était l'accumulateur de la remontée
  while (list_tmp != Nil)
    premier(list_tmp) :: res
    list_tmp=suite(list_tmp)
  // tjs rien à faire au moment du cas terminal
  return res
```

Fin réponse

▷ **Question 4:** *concat* : $\begin{cases} List \times List \mapsto List \\ \text{le résultat est la concaténation des deux listes} \end{cases}$

Réponse

version brutale: $O(n^2)$
```
concat1(list1,list2):
  si list1 = Nil alors list2
               sinon concat1(saufdernier(list1),
                             dernier(list1) :: list2)
```

Celle-ci est rigolote puisque la version inefficace est terminale, et donc on peut faire une version itérative toute pourrie :

version brutale: $O(n^2)$
```
concat_brute_iter(list1,list2):
  list1_tmp = list1
  while (! list1_tmp = Nil)
    dernier(list1_tmp) :: list2
    list1_tmp = saufdernier(list1)
  return list2
```

Pour aller plus vite ($O(n^2) \to O(n)$), il faut mettre list1 à l'envers une bonne fois pour toute au lieu d'aller piocher le dernier à tout bout de champ. Exemple à donner pour qu'ils trouvent :

ABC	DEF	en donnée
CBA	DEF	on inverse list1 avant d'appeller helper
BA	CDEF	récursion dans helper
A	BCDEF	récursion dans helper
$\emptyset$	ABCFED	Cas terminal de la récursion dans helper

$\rightsquigarrow$ résultat = ABCDEF

version récursive avec helper: $O(n)$
```
concat2_helper(list1,list2):
  si list1 = Nil alors list2
               sinon concat2_helper(list1.tail,
                                    list1.head :: list2)

concat2(list1,list2):
  concat2_helper(retourne(list1),list2)
```

c'est encore terminal, on peut y aller.

version itérative efficace: $O(n)$
```
concat_iter_good(list1,list2):
  list1_tmp = retourne(list1)
  while(list1_tmp != Nil)
     list1_tmp.head :: list2
     list1_tmp = list1_tmp.tail
  // tjs rien au cas terminal
  retourne list2
```

Fin réponse

▷ **Question 5:** $difference : \begin{cases} List \times List \mapsto List \\ \text{Le résultat est la liste de tous les éléments de list1 ne faisant pas partie de list2} \end{cases}$

Réponse

Normalement, vous avez pas eu le temps de la faire, celle là, lors du TD2 vu qu'elle était en dernière position. C'est l'occasion ;)

difference(list1,list2)
```
si list2 = Nil alors
  list1
sinon
  si list1 = Nil alors
    Nil
  sinon
    si est_membre(list1.head,list2) alors
      difference(list1.tail,list2)
    sinon
      list1.head :: difference(list1.tail, list2)
    finsi
  finsi
finsi
```

Ce n'est pas terminal, il faut modifier. Mais si on applique la méthode de retourne(), ie si on introduit un accumulateur, il faut changer les premier() en dernier() pour faire les opérations dans le même ordre.

version terminale

```
difference_term(list1, list2)=
 si list2=Nil alors list1 // comme ca, plus besoin de tester à chaque coup
            sinon difference_term_help(list1,list2,Nil)

difference_term_help(list1,list2,acc):
  si list1 = Nil alors
    acc
  sinon
    si est_membre(dernier(list1),list2) alors
      difference_term_help(saufdernier(list1),list2,acc)
    sinon
      difference_term_help(saufdernier(list1),list2,dernier(list1) :: acc)
    finsi
  finsi
finsi
```

On peut dérécursiver ça, mais c'est dommage de dégrader ainsi les perfs (on vient de passer $O(n)$ à $O(n^2)$). On peut avoir l'idée de mettre list1 à l'envers avant de commencer afin d'obtenir facilement les derniers, qui se retrouvent devant.

version terminale efficace

```
difference_term_good(list1, list2)=
 si list2=Nil alors list1 // comme ca, plus besoin de tester à chaque coup
            sinon difference_term_help(retourne(list1),list2,Nil)

difference_term_good_help(list1,list2,acc):
  si list1 = Nil alors
    acc
  sinon
    si est_membre(list1.head, list2) alors
      difference_term_help(list1.tail, list2,acc)
    sinon
      difference_term_help(list1.tail,list2,  list1.head :: acc)
    finsi
  finsi
finsi
```

Voilà qui est mieux. On peut dérécursiver.

version terminale efficace

```
difference_iter(list1, list2)=
 si list2=Nil alors retourne list1
 list1_tmp = retourne(list1)
 accu = Nil
 while(list1_tmp != Nil)
    si ! est_membre(list1_tmp.head, list2) alors
      list1_tmp.head :: accu
      list1_tmp = list1_tmp.tail
 // je n'arrive pas à trouver de fonction récursive faisant qqch de non trivial
 // au cas terminal.. Tant pis
 retourne accu
```

Une autre idée est de se dire simplement que l'énoncé n'impose pas d'ordre sur les lettres qu'on retourne. Peut-être bien qu'on peut alors se permettre de renvoyer les lettres dans le desordre (càd, utiliser list1.head sans prendre la peine de mettre list1 à l'envers). Dans la vraie vie, il faudrait vérifier, renégocier le cahier des charges avec le client, mais là, on va pas s'embetter.

Fin réponse

▷ **Question 6:** $nnaturels : \begin{cases} \mathbb{N} \mapsto List \\ \text{résultat : une liste formée des n premiers entiers naturels} \end{cases}$

Réponse

Question optionnelle a faire si vous avez décidément pas envie de dérécursiver Hanoi. Ce qui suit est juste la correction du TD2. Elle est amusante à dérécursiver, car avant, on avait du mal à faire la liste à l'endroit. Ici, elle va se construire tout naturellement dans l'accumulateur, pile comme il faut.

Version simple qui donne la liste à l'envers

```
nnaturels1(n):
  si n = 0 alors Nil
           sinon n :: nnaturels1(n-1)
```

Version trichée qui donne la chaine à l'endroit:

```
nnaturels2(n):
  retourne(nnaturels1(list))
```

Pour faire la série dans l'ordre sans tricher, il faut une fonction d'aide. Pour le faire trouver, on peut écrire au tableau les différents arguments pris par cette fonction d'aide.

Version avec helper:

```
nnaturels3(n):
  nnaturels3_helper(1,n-1)

nnaturels3_helper(n, todo):
  si todo = 0 alors Nil
              sinon n :: nnaturels3_helper(n+1,todo-1)
```

Fin réponse

★ **Exercice 2: Dérécursivation de l'exponentiation rapide.**

Réponse

Le principal intéret de cet exo est d'avoir un traitement sur le cas terminal. Il faudrait le retoucher (et surtout refaire la correction) pour expliciter ca. Peut-etre l'an prochain.

Fin réponse

On souhaite calculer (rapidement) x^n (x et n étant entiers).

- Si n est pair alors $x^n = \left(x^2\right)^{\frac{n}{2}}$. Il suffit alors de calculer $y^{n/2}$ avec $y = x^2$.
- Si n est impair et $n > 1$, alors $x^n = x \times \left(x^2\right)^{\frac{n-1}{2}}$. Il suffit de calculer $y^{\frac{n-1}{2}}$ avec $y = x^2$ et de multiplier le résultat par x.

Cela nous amène à l'algorithme récursif suivant qui calcule x^n pour un entier strictement positif n :

$$puissance(x, n) = \begin{cases} x, & \text{si } n = 1 \\ puissance(x^2, \frac{n}{2}), & \text{si } n \text{ pair} \\ x \times puissance(x^2, \frac{n-1}{2}), & \text{si } n \text{ impair } (n \neq 1) \end{cases}$$

▷ **Question 1:** Écrivez une fonction récursive calculant l'exponentiel d'un entier avec cet algorithme.

Réponse

```
public int puissance(int x, int n) {
    if (n==1)                  /* cas terminal */
        return x;
    if (n % 2 == 0)            /* pair */
        return puissance(x*x,n/2);
                               /* impair */
    return x*puissance(x*x,n/2);
}
```

Fin réponse

▷ **Question 2:** Quelle est la complexité de cet algorithme ?

Réponse

On divise le travail restant par deux à chaque étape. La complexité est donc $O(log_2 n)$.

Fin réponse

▷ **Question 3:** Transformez cette fonction en une fonction récursive terminale.

Réponse

- On l'écrit sous forme fonctionnelle :

puiss(x, n)=
si $n = 0$ **alors** 1
sinon si pair(n)
alors puiss($x \times x$, n div 2)
sinon $x \times$puiss($x \times x$, n div 2)

- Pour passer en récursivité terminale, il faut créer une autre fonction avec plus de paramètres. Les paramètres supplémentaires servent d'accumulateurs pour faire lors de la descente les calculs qui auraient du avoir lieu lors de la remontée.
- Ici, lors de la remontée, on multiplie par un nombre (pas toujous le même).
- On pose donc $puissTerm(x, n, s) = s \times puiss(x, n)$
- On transforme :
  $puissTerm(x, n, s) = s \times puiss(x, n)$
  $= s\times$ (**si** $n = 0$ **alors** 1
  sinon si pair(n)
  alors puiss($x \times x$, n div 2)
  sinon $x \times$puiss($x \times x$, n div 2)
)
  - On rentre le $s\times$
  $=$ **si** $n = 0$ **alors** $s \times 1$
  sinon si pair(n)
  alors $s\times$ puiss($x \times x$, n div 2)
  sinon $s \times x \times$puiss($x \times x$, n div 2)
  - On replie (utilisant en chemin l'associativité du $\times$). Ca donne la définition de $puissTerm$.
  $=$ **si** $n = 0$ **alors** s
  sinon si pair(n)
  alors $puissTerm(x \times x, n \text{ div } 2, s)$
  sinon $puissTerm(x \times x, n \text{ div } 2, s \times x)$
  - Il suffit bien entendu d'initialiser $s = 1$ au démarrage de la récursion.

Fin réponse

▷ **Question 4:** Transformez la fonction obtenue en fonction itérative.

Réponse

```
int puiss(int u, int r) {
    int x = u;
    int n = r;
    int s = 1;
    while (n != 0) {
        if (n % 2 == 0) { /* pair */
            s = s * x;
        }
        x = x * x;
        n = n / 2;
    }
    return s;
}
```

Fin réponse

★ **Exercice 3: Dérécursivation des tours de Hanoï.**

HANOI(n,a,b) :
si n = 1 **alors** deplacer(a,b)
sinon hanoi(n-1, a, c)
deplacer(a, b)
hanoi(n-1, c, b)
finsi

▷ **Question 1:** Dérécursivez cet algorithme. Comme cet algorithme n'est pas récursif terminal, il faut utiliser une pile. On y conservera l'état courant du programme, constitué des paramètres de la fonction récursive auxquels on ajoute un marqueur entier indiquant le numéro de l'appel récursif simulé (puisqu'il y en a 2).

Réponse

CORRECTION

- Cette dérécursivation a été vue en cours, mais c'est pas simple à comprendre sans écouter ;) Je vous invite à re-regarder les transparents 87 et 88 du cours pour voir ce qu'ils ont vu (et ce qu'il faut leur expliquer, donc).
- Pour guider les élèves pour qu'ils le trouvent, je pense qu'il faut tourner la version récursive à la main et leur montrer ce que fait l'ordinateur. Il utilise une pile pour se souvenir ? Ben on va faire pareil.
- Remarquons que cet algo est dans le cours, hein.

```
hanoiIter(int n, int dep, int arr, int aux) {
    int appel; // le numéro de l'appel récursif
    empiler(n,dep,arr,aux,1)
    while (pile non vide) {
        (n, dep, arr, aux, appel) ← depiler()
        if (n > 0) {
            if (appel == 1) {
                empiler(n, dep, arr, aux, 2)
                empiler(n-1, dep, aux, arr, 1)
            } else { // deux valeurs possibles seulement
                deplace(dep, arr)
                empiler(n-1, aux, arr, dep, 1)
            }
        }
    }
}
```

Fin réponse

▷ **Question 2:** Dessinez les états successifs de la pile lors de Hanoi(3,a,b)

Réponse

Il est impossible de comprendre l'algo si on le fait pas tourner une fois à la main.

Fin réponse