Informatique Option Listes, récursion

Sujet (l' Victor Nicollet 2006, 2007)

12 mars 2008

1 Récursion

On dit qu'une fonction est *récursive* lorsqu'elle apparaît dans sa propre définition. Il s'agit de la traduction dans le langage des suites définies par récurrence. Par exemple:

$$u_n = u_{n-1} \cdot n \qquad u_0 = 1$$

Se traduit en par la fonction récursive:

```
let rec u(n) =  if n = 0 then 1
else u(n-1) * n;;
```

Le symbole `rec` indique que la fonction est récursive: le nom `u` est disponible à l'intérieur de la définition (et pas uniquement après la fin de celle-ci: essayez de retirer le `rec`, pour voir).

Lorsque `u(n)` est exécutée, elle va appeler `u(n-1)`, qui va appeler `u(n-2)` et ainsi de suite jusqu'à 0. Il va donc y avoir $n+1$ appels à `u` pour calculer `u(n)`.

Question 1 *Écrire une fonction récursive simple qui calcule le terme n de la suite de Fibonnaci:*

$$u_n = u_{n-1} + u_{n-2} \qquad u_0 = u_1 = 1$$

On note A_n le nombre d'appels à `u` qui sont effectués lorsqu'on calcule `u(n)`. Que peut-on dire sur la suite A_n ? Comparer avec le nombre d'opérations effectuées si on avait utilisé une boucle `for`.

Question 2 *On remarque que:*

$$x^0 = 1 \qquad x^{2k} = (x^2)^k \qquad x^{2k+1} = x(x^2)^k$$

Si l'on dispose d'une fonction qui est capable de calculer efficacement x^k pour tout x et pour tout $k < n$, comment calculer efficacement x^n ?

Écrire la fonction récursive utilisant cette methode (appelée exponentiation rapide*) pour calculer x^k pour x et k donnés.*

Un problème peut être résolu récursivement s'il vérifie deux conditions:

- Il exite une notion de *taille*: par exemple, le problème de calculer u_6 est plus grand que celui de calculer u_5. Plus généralement, il faut arriver à définir une notion de taille à partir des arguments: taille d'un intervalle de recherche, somme en valeur absolue des arguments, taille de l'argument ou autres propriétés du problème.
- Il est possible d'exprimer un problème à partir de problèmes strictement plus petits (et, pour les plus petits problèmes, la solution est triviale).

Ces deux propriétés assurent qu'une fonction récursive pourrait résoudre un problème en le réduisant à des problèmes de plus en plus petits, jusqu'à atteindre un problème si petit qu'il devient trivial à résoudre.

Question 3 *On s'intéresse pour la troisième fois à la suite de Syracuse, définie par son premier terme et par la récurrence:*

$$s_{n+1} = \begin{cases} s_n/2 & \text{si } s_n \text{ est pair} \\ 3s_n + 1 & \text{sinon} \end{cases}$$

Cette fois, on s'intéresse aux antécédents de rang n d'une valeur donnée[1].

Écrire une fonction `ante(y)(n)` qui affiche tous les antécédents de `y` de rang inferieur ou égal à `n`, séparés par des espaces. Pour simplifier, l'ordre ne sera pas important, et un même nombre pourra apparaître plusieurs fois.

2 Listes

Une valeur `v` est une liste dans les deux cas suivants:
- si elle est la liste vide, notée `v = []`
- ou bien si elle contient une valeur `h` (*head*, la tête de `v`) et une liste `t` (*tail*, la queue de `v`). On note alors `v = h :: t`

Les listes sont donc une structure récursive, puisque toutes les listes (sauf la liste vide) sont définies à partir d'une liste plus petite (leur queue). Par exemple, une liste des entiers de 1 à 3 est `1:: (2:: (3:: []))`, qu'il est possible pour des raisons de lisibilité, d'écrire de manière abrégée `[ 1; 2; 3 ]`

[1] On considèrera que x est un antécédent de rang n de y si et seulement si, lorsque l'on fixe le premier terme $s_0 = x$, $s_n = y$.

Contrairement aux tableaux, les listes ne peuvent pas être modifiées. La manipulation des listes ne comporte donc que deux possibilités: parcourir la liste pour en extraire des informations, ou créer une nouvelle liste.

2.1 Créer une liste

Créer une liste est très simple, et se fait grâce à une fonction récursive. La queue est la valeur de la fonction sur un problème plus petit, et la tête est le résultat d'un calcul supplémentaire. Dans les problèmes triviaux, la fonction renvoie souvent la liste vide. Par exemple, la liste des carrés des entiers inférieurs à n est donnée par:

```
let rec carres (n) =
if n = 0 then []
else (n*n):: (carres (n-1));;
```

Question 4 *Écrire une fonction qui calcule la liste de diviseurs premiers d'un entier n, dans l'ordre croissant et apparaissant autant de fois dans la liste que dans n. Le résultat attendu est donc, par exemple:*

```
\# diviseurs(18);;
- : int list = [ 2; 3; 3 ]
```

On pourra, pour s'aider, écrire une fonction qui calcule le plus petit diviseur premier d'un nombre.

2.2 Parcourir une liste

Pour parcourir une liste, on utilise également une fonction récursive. Celle-ci utilise le puissant système de motifs de (le *pattern matching*) pour détecter si l'on a atteint la fin de la liste (un problème trivial) ou si l'on peut réduire le problème à un problème plus petit (la queue de la liste).

Par exemple, la fonction ci-dessous donne la longueur d'une liste:

```
let rec length = function
  | [] → 0
  | h:: t → length t + 1;;
```

parcourt dans l'ordre les motifs (qui commencent après chaque `|`). Si un motif correspond à l'objet examiné, alors le code qui est juste à droite du `- >` qui suit est utilisé pour calculer la valeur. Les noms qui ont été donnés aux sous-parties de l'objet dans le motif sont disponibles dans ce code (comme `t` dans l'exemple ci-dessus).

Question 5 *Écrire un programme qui calcule la somme des éléments d'une liste d'entiers.*

Écrire un programme qui calcule le produit des éléments d'une liste de réels.

2.3 Transformation de listes

On ne peut pas transformer les listes, puisqu'on ne peut pas les modifier ! En revanche, on peut créer une liste à partir des données lues dans une autre. Pour cela, il suffit en général d'écrire une fonction qui lit des données dans une liste, et qui renvoie une liste.

Question 6 *La fonction `list_map` prend en argument une liste `[ x1; x2 \ldots xn]` et une fonction `f` et renvoie la liste `[ f(x1); f(x2) \ldots f(xn)]`.*

Réecrivez vous-même cette fonction.

Question 7 *Écrire une fonction `insert` qui, étant donnés une liste triée `l` et un élément `e`, renvoie la liste triée contenant `e` ainsi que les éléments de `l` (on parle d'insérer un élément dans une liste triée).*

En déduire (et écrire) une fonction qui applique le tri par insertion pour trier une liste.

3 Autres structures récursives

La définition récursive de la liste est: `[]` ou `h:: t`.

Il est possible en de définir des types suivant ce modèle, en utilisant le mot-clé `type` et une syntaxe proche de celle du pattern matching. Par exemple, on peut définir une liste d'entiers simple suivant le modèle des vraies listes:

```
type autreListe =
| ListeVide | TeteQueue of int * autreListe;;

let rec longueur = function
| ListeVide → 0 | TeteQueue (h,t) → longueur t + 1;;

longueur (TeteQueue(10,TeteQueue(3,TeteQueue(7,ListeVide))));;
```

Question 8 *On utilise le type suivant:*

```
type expression =
| Var | Const of float
| Mul of expression * expression
| Add of expression * expression
                | Sub of expression * expression;;
```

Ce type permet de représenter des expressions mathématiques. Par exemple, on représente $(1 + x) * 2$ *par* `Mul (Add (Const 1.,Var),Const 2.)`.

- *Écrire une fonction qui affiche l'expression sous une forme lisible. Attention au parenthésage !*
- *Écrire une fonction récursive qui renvoie l'expression dans laquelle on a remplacé toutes les occurences de la variable une valeur donnée. Par exemple, remplacer la variable par* 1 *dans* $(x + 1) * x$ *donne* $(1 + 1) * 1$
- *Écrire une fonction qui récursive lit une expression sans variables et qui calcule son résultat. Par exemple,* $(1 + 1) * 2$ *donne* 1.
- *Écrire une fonction qui calcule la dérivée d'une expression par rapport à une variable donnée.*

4 Sujets divers

4.1 Reformulations

Parfois, il est utile de reformuler le problème, de manière à ce qu'il comporte une notion de taille. Par exemple, chercher l'élément minimal d'un tableau est un problème qui ne fait pas intervenir de taille. En revanche, chercher l'élément minimal entre les indices 0 et n d'un tableau fait intervenir explicitement la taille n du problème.

Lorsqu'on reformule un problème, on écrit d'abord une fonction qui résout le problème modifié. Puis on écrit une fonction qui traduit le problème original en le problème modifié, utilise la fonction précédente, et renvoie le résultat. Par exemple:

```
 let min_tableau (tabl) = (* Probleme initial *)
let rec min_rec_tableau (n) = (* Probleme modifie *)
 if n = 0 then tabl.(0)
 else min (tabl.(n)) (min_rec_tableau (n-1))
in min_rec_tableau (vect_length tabl - 1);;
```

Question 9 *Écrire une fonction `second(liste)` qui calcule récursivement le deuxième plus grand élément d'une liste (celui qui serait en deuxième position si la liste était triée). La complexité doit être linéaire en la taille de la liste (on ne peut donc pas la trier).*

4.2 Récursion terminale

On parle de récursion terminale (*tail recursion*) lorsque la dernière chose que renvoie une fonction récursive est la fonction elle-même. Par exemple:

```
let facto (n) =
let rec tail (n) (m) =
if n = 0 then m  else tail (n-1) (n*m)
in tail (n) (1);;
```

Chaque appel consomme un peu de mémoire de ce qu'on appelle la *pile*, que le programme utilise pour se souvenir des opérations qui restent à faire dans chaque appel. La factorielle normale ne peut fonctionner pour n grand, car elle doit retenir l'opération `n*` pour tout `n`. Au contraire, la récursion terminale ne coûte pas de mémoire (il n'y a rien à retenir), et fonctionne donc pour n'importe quelle valeur de n.

Une telle fonction transmet la valeur finale en argument à chaque appel, et renvoie cet argument lorsque la récursion est terminée. C'est le principe de l'argument `m` ci-dessus.

Question 10 *Écrire une fonction en récursion terminale `rev` qui inverse une liste (au sens que `[1; 3; 2; 4]` devient `[4; 2; 3; 1]`).*

En déduire une fonction en récursion terminale qui prend en argument une liste `l` et un élément `e`, et renvoie une liste qui contient tous les éléments de la liste initiale sauf `e` (et dans le même ordre).

4.3 Pattern matching

Le *pattern matching* est très puissant: il permet de reconnaître à peu près n'importe quelle forme d'objets, en appliquant dessus des contraintes arbitraires et en donnant des noms aux différentes parties et sous-parties de la forme reconnue. Voici un exemple un peu plus avancé pour compter le nombre d'élements consécutifs égaux dans une liste:

```
let rec doublons = function
| [] | [_ ]  → 0
| a::(b::_ as t) when a = b → doublons t + 1
| _ :: t → doublons t;;
```

- La deuxième ligne dit que si la liste est vide, ou contient un seul élément, alors il n'y a pas de doublons. Le `_` signifie qu'on ne donne pas à l'élément en question un nom.
- La troisième dit que si la liste contient au moins deux éléments, on appelle `a` le premier et `b` le second, et on donne le nom `t` à la liste `b::_` avec `as`. Si la contrainte `a = b` introduite par `when` est vérifiée, alors on calcule le nombre de doublons de `t` et on y ajoute 1.
- La quatrième ligne gère les cas restants, où le premier et deuxième élément sont différents, et calcule le nombre de doublons de la queue.

Question 11 (Très difficile, seulement si vous avez le temps)

On travaille sur le même type que l'exercice 8. Notre objectif est d'intégrer une expression. Malheureusement, même sur des expressions aussi simples (juste des opérations de base) l'intégration est très difficile. Il faut donc commencer par pré-traiter l'expression pour la mettre sous une forme que nous sommes capables d'intégrer.

Étape 1 *Utiliser le pattern matching pour écrire une fonction qui remplace dans une expression toutes les soustractions $a - b$ par des additions $a + (-1 * b)$.*

Étape 2 *Utiliser le pattern matching pour écrire une fonction qui lit une expression et développe tous les produits de sommes en sommes de produits. Par exemple:*

$$(1 + X) \cdot (1 + 2 \cdot (1 + X))$$

est transformée en

$$1 \cdot 1 + 1 \cdot 2 \cdot 1 + 1 \cdot 2 \cdot X + X \cdot 1 + X \cdot 2 \cdot 1 + X \cdot 2 \cdot X$$

Étape 3 *On va maintenant utiliser l'associativité de la multiplication pour transformer un produit de constantes et de variables en un couple (c, d), tel que l'expression soit égale à cX^d. Par exemple:*

$$X \cdot 2. \cdot 1.$$

devient

$$(2., 1)$$

Étape 4 *On a obtenu une expression qui est une somme de termes polynômiaux. Parcourir cette expression et construire une liste de couples (c, d) correspondant aux termes de la somme. Ainsi,*

$$1 \cdot 1 + 1 \cdot 2 \cdot 1 + 1 \cdot 2 \cdot X + X \cdot 1 + X \cdot 2 \cdot 1 + X \cdot 2 \cdot X$$

correspond à (pas forcément dans cet ordre)

```
\ \ [1.,0; 2.,0; 2.,1; 1.,1; 2.,1; 2.,2]
```

Étape 5 *Construire une fonction qui à un couple associe sa primitive (sous la forme d'une expression mathématique) en remarquant que la primitive de (c, d) est $cX^{d+1}/(d+1)$. On pensera d'ailleurs à l'exponentiation rapide pour construire X^{d+1}. Ainsi,*

```
\ \ 2.,2
```

devient:

```
\ \ Mul(2./.3.,Mul(Var,Mul(Var,Var)))
```

Étape 6 *Écrire une fonction qui utilise* `list_map` *pour créer une liste des primitives, et une fonction qui prend cette liste et construit la somme de ses éléments. Par exemple,*

```
\ \ [1.,0; 2.,0; 2.,1; 1.,1; 2.,1; 2.,2]
```

devient:

$$1X/1 + 2X/1 + 2X \cdot X/2 + 1X \cdot X/2 + 2X \cdot X/2 + 2X \cdot X \cdot X/3$$

Le problème est résolu: on a réussi à trouver une primitive de l'expression initiale. S'en convaincre en intégrant puis dérivant un polynôme et en calculant sa valeur en assez de points.