Récursivité suite - Fonctions récursives sur les listes. Récursions arborescentes

Université Montpellier-II - FLIN304 - Programmation Applicative et Récursive - Cours No 6

13 Fonctions récursives - suite

13.1 Fonction récursive terminale sur les listes

Exemple : Recherche d'un élément

```
(define (member? x l)
  (cond ((null? l) nil) ; ou #f
        ((equal? x (car l)) l) ; ou #t
        (#t (member? x (cdr l)))))
```

Recherche du nième cdr.

```
(define (nthcdr n l)
  (cond ((null? l) nil)
        ((= n 0) l)
        (#t (nthcdr (- n 1) (cdr l)))))
```

13.2 Apparté : Letrec

```
(letrec <bindings> <body>)

Syntax: <Bindings> should have the form ((<variable1> <init1>) ...),
and <body> should be a sequence of one or more expressions.

Semantics: The <variable>s are bound to fresh locations holding
undefined values, the <init>s are evaluated in the resulting
environment (in some unspecified order), each <variable> is assigned
to the result of the corresponding <init>, the <body> is evaluated in
the resulting environment, and the value(s) of the last expression in
<body> is(are) returned. Each binding of a <variable> has the entire
letrec expression as its region, making it possible to define mutually
recursive procedures.
```

13.3 Réalisation d'un programme avec des listes

Réalisation d'un dictionnaire et d'une fonction de recherche d'une définition.

```
(define dico '((hugo lavoisier einstein joliot knuth)
               (écrivain chimiste physicien mathématicien informaticien)))

(define (donneDef nom)
```

```
  (letrec ((chercheDef (lambda (noms definitions)
     (cond ((null? noms) 'non-défini)
           ((eq? (car noms) nom) (car definitions))
           (#t (chercheDef (cdr noms) (cdr definitions)))))))

    (chercheDef (car dico) (cadr dico))))

(donneDef 'joliot)
= mathématicien
```

13.4 fonction récursive enveloppée sur les listes : schéma 1

```
(define (recListe l)
  (if (null? l)
      (traitementValeurArrêt)
      (enveloppe (traitement (car l))
                 (recListe (cdr l)))))
```

13.4.1 Exemple 1

```
(define (longueur l)
    (if (null l)
        0
        (1+ (longueur (cdr l)))))
```

- traitementValeurArret : rendre 0
- traitement (car l) : ne rien faire
- enveloppe : +1

```
(longueur '(1 3 4))
--> (+ 1 (longueur '(3 4)))
  --> (+ 1 (+ 1 (longueur '(3))))
    --> (+ 1 (+ 1 (+ 1 (longueur ()))))
      --> (+ 1 (+ 1 (+ 1 0)))
...
3
```

13.4.2 Exemple 2

```
(define (append l1 l2)
  (if (null l1)
      l2
      (cons (car l1)
            (append (cdr l1) l2))))
```

- traitement traitementValeurArret : rendre l2
- traitement (car l) : rien à faire
- enveloppe : cons

```
--> (append '(1 2) '(3 4))
  --> (cons 1 (append '(2) '(3 4)))
    --> (cons 1 (cons 2 (append () '(3 4))))
      --> (cons 1 (cons 2 '(3 4)))
...
(1 2 3 4)
```

Consommation mémoire : taille de l1 doublets.

13.4.3 Exemple 3

```
(define (add1 l)
    (if (null l)
        ()
        (cons (+ (car l) 1) (add1 (cdr l)))))

(add1 '(1 2 3))
= (2 3 4)
```

- traitementValeurArret : rendre ()
- traitement (car l) : +1
- enveloppe : cons

13.4.4 Exemple 4 : tri par insertion

```
(define (tri-insertion l)
    (if (null l)
        ()
        (insérer (car l) (tri-insertion (cdr l)))))

(define inserer (x l2)
    (cond ((null l2) (list x))
          ((< x (car l2)) (cons x l2))
          (t (cons (car l2) (inserer x (cdr l2))))))
```

Consommation mémoire : chaque insertion consomme en moyenne "taille(l2)/2" doublets ; il y a "taille(l)" insertions. La consommation mémoire est donc en $O(l^2/2)$

Consommation pile : pour `tri-insertion`, autant que d'appels récursifs donc que de nombres dans la liste à trier. Pour insérer, en moyenne "taille(l2)/2"

13.5 fonction récursive enveloppée - second schéma

```
(define (recListe2 l)
  (if (null? l)
      (traitement ())
      (enveloppe (recListe2 (cdr l))
                 (traitement (car l)))))
```

Exemple.

```
(define (reverse l)
    (if (or (null l) (null (cdr l)))
        l
        (append (reverse (cdr l)) (list (car l)))))
```

Consommation mémoire : 'taille de l' doublets.

13.6 Récursivité arborescente

Fonction récursive arborescente : fonctions récursives contenant plusieurs appels récursifs, éventuellement enveloppés, ou dont l'enveloppe est elle-même un appel récursif

Fonction dont l'interprétation nécessite un arbre de mémorisation.

13.6.1 L'exemple des suites récurrentes à deux termes

Exemple du calcul des nombres de fibonacci.

"Si l'on possède initialement un couple de lapins, combien de couples obtient-on en n mois si chaque couple engendre tous les mois un nouveau couple à compter de son deuxième mois d'existence".

Ce nombre est défini par la suite récurrente linéaire de degré 2 suivante :

$F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$

Les nombres de Fibonacci sont célèbres en arithmétique et en géométrie pour leur relation avec le nombre d'or, solution de l'équation $x^2 = x + 1$. La suite des quotients de deux nombres de Fibonacci successifs a pour limite le nombre d'or.

Les valeurs de cette suite peuvent être calculées par la fonction scheme suivante.

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (#t (+ (fib (- n 1)) (fib (- n 2))))))
```

Complexité :

L'interprétation de cette fonction génére un arbre de calcul (récursivité arborescente). L'**arbre de calcul** de `(fib n)` possède $fib(n+1)$ feuilles. Exemple, fib(4) : 5 feuilles.

Ceci signifie que l'on calcule à $fib(n+1)$ fois fib(0) ou fib(1), et que l'on effectue $fib(n+1)-1$ additions.

Par exemple, pour calculer $fib(30) = 842040$, cette fonction effectue $fib(31) - 1$ soit 1346268 additions.

13.6.2 Exemple2 : listes généralisées

Recherche d'un élément dans une liste généralisée (une liste contenant des listes)

```
(define (genmember? x lgen)
  (cond ((null? lgen) #f)
        ((equal x (car lgen)) #t)
        ((list? (car lgen))
```

```
          (or (genmember? x (car lgen))
              (genmember? x (cdr lgen))))
         (#t (genmember? x (cdr lgen)))))
```

Exercice : écrire une fonction qui "applatit" une liste généralisée.

```
(flat '((1 2) 3 (4 (5 6) 7)))
= (1 2 3 4 5 6 7)
```

13.7 Un exemple plus complexe

13.7.1 Problème

L'exemple suivant, tiré de "structure and interpretation of computer programs" montre la réduction d'un problème par récursivité.

Problème : soit à calculer le nombre `N` de façon qu'il y a de rendre une somme `S` en utilisant `n` types de pièces.

Il existe une solution basée sur une réduction du problème jusqu'à des conditions d'arrêt qui correspondent au cas ou la somme est nulle ou au cas où il n'y plus de types de pièces à notre disposition.

13.7.2 Réduction du problème

Soit V l'ensemble ordonné des valeurs des types de pièces, n le nombre de types de pièces (égal au cardinal de V), n_i le ième type de pièce, v_i la valeur du ième type de pièce, (par exemple, avec l'euro il y a 8 types de pièces de valeurs respectives 1, 2, 5, 10, 20, 50, 100 et 200 cts). Le nombre de façon de rendre une somme S est donné par l'algorithme suivant.

Réduction du problème : $N(S, n) = N(S - v_1, n) + N(S, n - 1)$

Soit :

(le nombre de façon de changer la somme S - la valeur de la première pièce, avec tous les types de pièces)

+

(le nombre de façon de changer la même somme en utilisant tous les types de pièces sauf le premier)

13.7.3 Tests et valeurs d'arrêt

si $S = 0$, 1
si $S < 0$, 0 (on ne peut pas rendre une somme négative - ce cas ne peut se produire que pour des monnaies qui ne possèdent pas la pièce de 1 centime)
si $S > 0, n = 0$, 0 (aucune solution pour rendre une somme non nulle sans pièce)

13.7.4 Exemple

Exemple :

```
Rendre 5 centimes avec (1 2 5) =
       Rendre 4 centimes avec (1 2 5)
       + Rendre 5 centimes avec (2 5)
```

```
soit en développant le dernier,

= Rendre 4 centimes avec (1 2 5)
  + rendre 3 centimes avec (2 5)
  + Rendre 5 centimes avec (5)

soit en développant le dernier,

=  Rendre 4 centimes avec (1 2 5)
   + rendre 3 centimes avec (2 5)
   + rendre 0 centimes avec (5)
   + Rendre 5 centimes avec ()

soit en appliquant les tests d'arrêt,

= Rendre 4 centimes avec (1 2 5)
  + 0
  + 1
  + 0

...

= 4 ((1 1 1 1 1) (1 1 1 2) (1 2 2) (5))
```

13.7.5 Algorithme

Soit V l'ensemble ordonné des valeurs des types de pièces, n le nombre de types de pièces, soit le cardinal de V, v_i la valeur du ième type de pièce, (par exemple, avec l'euro il y a 8 types de pièces de valeurs respectives 1, 2, 5, 10, 20, 50, 100 et 200 cts). Le nombre de façon de rendre une somme S est donné par l'algorithme suivant.

Rendre(S, n, V) =
si $S = 0$, 1
si $S < 0$, 0 (on ne peut pas rendre une somme négative)
si $S > 0, n = 0$, (0 solution pour rendre une somme non nulle sans pièce)
Sinon,
$Rendre(S - v_1, n, V)$ (le nombre de façon de changer la somme - la valeur de la première pièce, avec tous les types de pièces) + $Rendre(S, n-1, V\ v_1)$ (le nombre de façon de changer la même somme en utilisant tous les types de pièces sauf le premier)

13.7.6 Implantation

```
(define (rendreMonnaie somme nbSortesPieces valeursPieces)

  (letrec ((rendre (lambda (somme nbSPieces)
    ;; à calculer le nombre de façon de donner une somme avec un certain nombre de pièces de monnaies
    ;; d'après SICP
    (cond ((= somme 0)
           1)
          ((or (< somme 0) (= nbPieces 0))
           ;; aucune façon de donner une somme négative
```

```
          ;; ou de changer une somme non nulle (le cas nul a été traité
          ;; dans la clause précédente) sans pièce
          0)
         (#t (+ (rendre somme (- nbSPieces 1))
                (rendre (- somme (valeurPiece nbSPieces)) nbSPieces)))))
    (rendre somme nbSortesPieces))

(define (valeurPiecesEuro piece)
  ;; fonction à réécrire pour chaque monnaie.
  ;; liste la valeur en centime des différentes pièces.
  (cond ((= piece 1) 1) ;; la piece no 1 vaut 1 centime, etc
        ((= piece 2) 2)
        ((= piece 3) 5)
        ((= piece 4) 10)
        ((= piece 5) 20)
        ((= piece 6 ) 50)
        ((= piece 7 ) 100)
        ((= piece 8 ) 200)
        ))

(define (valeurPiecesDollar piece)
  ;; fonction à réécrire pour chaque monnaie.
  ;; liste la valeur en centime des différentes pièces.
  (cond ((= piece 1) 1) ;; la piece no 1 vaut 1 centime, etc
        ((= piece 2) 5)
        ((= piece 3) 10)
        ((= piece 4) 25)
        ((= piece 5) 50)
        ))

(define (rendreEuro somme)
  (rendreMonnaie somme 8 valeurPiecesEuro))

(define (rendreDollar somme)
  (rendreMonnaie somme 5 valeurPiecesDollar))

(rendreEuro somme valeurPiecesEuro)
```

13.7.7 Exécution

- 4 façons de rendre 5 centimes ((1 1 1 1 1) (1 1 1 2) (1 2 2) (5)),
- 11 façons de rendre 10 centimes,
- 4112 façons de rendre 100 centimes,
- 73682 façons de rendre 200 centimes.

Exercice : Ecrivez une variante du programme qui rend la liste des solutions.

La complexité est du même ordre que celle de la fonction fibonacci mais pour ce problème il est beaucoup plus difficile d'écrire un algorithme itératif.