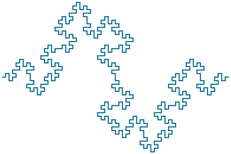
HLIN403 – Programmation Applicative

Récursivité suite - Fonctions récursives sur les listes. Récursions arborescentes

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

Fonctions récursives sur les listes

Récursivité arborescente

Fonctions récursives sur les listes

Exemple : Recherche d'un élément

```
(define member?
  (lambda (x l)
    (cond ((null? l) ())
          ((equal? x (car l)) l)
          (else (member? x (cdr l))))))
```

Exemple

Exemple : Recherche du nième cdr.

```
(define nthcdr
  (lambda (n l)
    (cond ((null? l) ())
          ((= n 0) l)
          (else (nthcdr (- n 1) (cdr l))))))
```

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
  (lambda (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
```

Fonction récursive enveloppée sur les listes : schéma 1

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

Exemple 1

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

- traitementValeurArret : rendre 0
- traitement du car : ne rien faire
- enveloppe : (lambda (x) (+ x 1))

Exemple 1

```
(longueur '(1 3 4))

--> (+ 1 (longueur '(3 4)))

  --> (+ 1 (+ 1 (longueur '(3))))

    --> (+ 1 (+ 1 (+ 1 (longueur ()))))

      --> (+ 1 (+ 1 (+ 1 0)))

...

3
```

Exemple 2

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

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

Exemple 2

```
--> (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.

Exemple 3

```
(define add1
  (lambda (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

Exemple 4 : tri par insertion

```
(define tri-insertion
  (lambda (l)
    (letrec
      ((insertion (lambda (x l2)
        (cond ((null? l2) (list x))
          ((< x (car l2)) (cons x l2))
          (#t (cons (car l2) (insertion x (cdr l2))))))))
      (if (null? l)
        ()
        (insertion (car l) (tri-insertion (cdr l)))))))
```

Exemple 4 : tri par insertion

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`, $taille(l)$. Pour `insertion`, en moyenne $taille(l2)/2$

Fonction récursive enveloppée - second schéma

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

Exemple

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

Consommation mémoire : taille de l doublets.

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.

L'exemple des suites récurrentes à deux termes

Exemple du calcul des nombres de Fibonacci.

Problème initial : "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 nombres de Fibonacci

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

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

Les nombres de Fibonacci

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 $fib(n+1)$ 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.

Exemple2 : listes généralisées

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

```
(define genmember?
  (lambda (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 “aplatit” une liste généralisée.

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

Un exemple plus complexe

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 façons 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 où la somme est nulle ou au cas où il n'y plus de types de pièces à notre disposition.

Réduction du problème

Soient

- V l'ensemble ordonné des valeurs des différentes 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 $V = \{1, 2, 5, 10, 20, 50, 100, 200\}$ et $n = 8$ (il y a 8 types de pièces de valeurs respectives 1, 2, 5, 10, 20, 50, 100 et 200 cts)

Réduction du problème

On peut réduire le problème ainsi :

$$\boxed{NFRendre(S, n) = NFRENDRE(S - v_1, n) + NFRendre(S, n - 1)}$$

Valeurs initiales

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)

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

Algorithme

Soient :

- S la somme à rendre
- V l'ensemble ordonné des valeurs des différentes pièces
- n le nombre de types de pièces (égal au cardinal de V)
- n_i le ième type de pièce et v_i la valeur de ce ième type de pièce.

Algorithme

NFRendre(S, n, V) =

si $S = 0$ alors 1

sinon si $S < 0$ alors 0 (on ne peut pas rendre une somme négative)

sinon si $n = 0$ alors 0 (on ne peut pas rendre une somme sans pièce)

sinon $NFRendre(S - v_1, n, V) + NFRendre(S, n - 1, V\ \ v_1)$

Implantation

```
(define NFRendre
  (lambda (somme nbSortesPieces valeursPieces)
    (letrec ((rendre (lambda (somme nbSPieces)
             (cond ((= somme 0) 1)
                   ((or (< somme 0) (= nbSPieces 0)) 0)
                   (else (+ (rendre somme (- nbSPieces 1))
               (rendre (- somme (valeursPieces nbSPieces))
                       nbSPieces)))))))
      (rendre somme nbSortesPieces))))
```

Implantation

```
(define (valeurPiecesEuro piece)
  (cond ((= piece 1) 1)
        ((= piece 2) 2)
        ((= piece 3) 5)
        ((= piece 4) 10)
        ((= piece 5) 20)
        ((= piece 6 ) 50)
        ((= piece 7 ) 100)
        ((= piece 8 ) 200)
        ))
```

Implantation

```
(define (valeurPiecesDollar piece)
  (cond ((= piece 1) 1)
        ((= piece 2) 5)
        ((= piece 3) 10)
        ((= piece 4) 25)
        ((= piece 5) 50)
        ))

(define rendreEuro
  (lambda (somme)
    (NFRendre somme 8 valeurPiecesEuro)))

(define rendreDollar
  (lambda (somme)
    (NFRendre somme 5 valeurPiecesDollar)))
```

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.