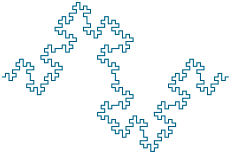
HLIN403 – Programmation Applicative

Fonctions récursives simples

Christophe Dony – Annie Chateau
Université Montpellier – Faculté des Sciences



Introduction

Définition inductive

Une définition inductive d'une partie X d'un ensemble consiste à fournir :

- la donnée explicite de certains éléments de X (base) ;
- le moyen de construire de nouveaux éléments de X à partir d'éléments déjà construits.

Exemple : ensemble des valeurs de la fonction “factorielle” sur les entiers.

Définition inductive

Les valeurs de cet ensemble peuvent être calculées par un ordinateur par exemple grace à la fonction *scheme* suivante :

```
(define fact (lambda (n)
  (if (= n 0)
    1
    (* n (fact (- n 1))))))
```

Définition inductive

Une version en C

```
int fact(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

Itération et récursion

Rappel **Itérer** : répéter n fois un processus en faisant changer la valeur des variables jusqu'à obtention du résultat.

Calcul itératif de factorielle d'un nombre : $n! = \prod_{i=1}^{n} i$

Un calcul itératif se programme par une boucle (*for* ou *tant-que* ou *repeat-until*).

Itération et récursion

Exemple de fonction itérative pour le calcul de factorielle en C

Rappel : Un invariant de boucle est une propriété qui reste vraie à chaque passage dans la boucle.

Utilité : contrôler, manuellement ou automatiquement, la bonne évolution du cacul.

```
int fact(n){ // n entier
  int i = 0;
  int result = 1;
  while (i < n){
    // result = fact(i) -- invariant de boucle
    i = i + 1;
    result = result * i;
    // result = fact(i) -- invariant de boucle
  }
  // i = n
  return(result);
}
```

Itération et récursion

Inconvénient : nécessité de gérer explicitement l'évolution des variables, l'ordre des affectations et le contrôle des invariants de boucle.

Autre version condensée en C :

```
int factorielle_iterative(int n)
{
  int res = 1;
  for (; n > 1; n--)
    res *= n;
  return res;
}
```

Autres Exemples de fonction récursives simples

Multiplication : $an = a + a(n-1)$

```
(define mult (lambda (a n)
  (if (= n 0)
    0
    (+ a (mult a (- n 1))))))
```

Autres Exemples de fonction récursives simples

Puissance : $a^n = a.a^{n-1}$

```
(define exp (lambda (a n)
  (if (= n 0)
    1
    (* a (exp a (- n 1))))))
```

Autres Exemples de fonction récursives simples

Inverse d'une chaîne : $inverse(n) = stringAppend(dernier(n), inverse(saufDernier(n)))$

```
(define inverse (lambda (s)
  (let ((l (string-length s)))
    (if (= 0 l)
        s
        (string-append (substring s (- l 1) l)
                       (inverse (substring s 0 (- l 1))))))))
```

Calcul des termes de suites récurrentes

Suite : ensemble d'éléments indexés par N ou une partie de N.

Suite récurrente : suite dont le calcul de la valeur d'un terme peut être exprimé en fonction de la valeur des termes précédents

Suite arithmétique : $u_{n+1} = u_n + r$ ou $u_n = u_0 + nr$ suite de premier terme u_0 et de raison r.

Suite géométrique de raison q : $u_{n+1} = q.u_n$

Suite Arithmético-géométrique (ou récurrente linéaire d'ordre 1) $u_{n+1} = \alpha u_n + \beta$

Calcul des termes de suites récurrentes

Suite récurrente linéaire d'ordre p

Toute suite à valeurs dans un corps K (généralement $\mathbb{C}$ ou $\mathbb{R}$) définie pour tout $n \geq n_0$ par la relation de récurrence suivante :

$a_0, a_1, \dots a_p$ étant p scalaires fixés de K (a_0 non nul), pour tout $n \geq n_0$, on a

$$u_{n+p} = a_0 u_n + a_1 u_{n+1} + \cdots + a_{p-1} u_{n+p-1}$$

Calcul des termes de suites récurrentes

Toute valeur d'une suite récurrente de la forme :

$u_0 = initial$ et pour $n > 1, u_n = \Phi(u_{n-1}, n)$

peut être calculée par une fonction (de n'importe quel langage de programmation autorisant la définition de fonctions récursives) similaire à la fonction *Scheme* suivante :

```
(define u (lambda (n)
  (if (= n 0)
    initial
    (PHI (u (- n 1)) n))))
```

Factorielle

Par exemple calcul de factorielle de 5 :

```
(define initial 1)
(define PHI *)
(u 5) --> 120
```

Suite arithmétique

Tout terme d'une suite arithmétique de raison r de la forme :

$u_0 = \mathit{initial}$ et pour $n > 1, u_n = u_{n-1} + r$ peut être calculée par la fonction

```
(define ua (lambda (n r)
  (if (= n 0)
    initial
    (+ (ua (- n 1) r) r))))
```

Suite arithmétique

Exemple : Multiplication de n par a,

```
(define initial 0)
(ua 3 4)
```

Suite arithmétique

A noter que le code suivant ne fonctionne pas (voir cours No 3, liaison lexicale) :

```
(let ((initial 0)) (ua 3 4))
```

Pour éviter de passer par une variable globale et de rajouter un paramètre inutile, on peut utiliser la structure de contrôle `letrec`

Aparté sur `letrec`

```
(letrec <bindings> <body>)
Syntax: <Bindings> should have the form ((<variable1>
<init1>) ...),
and <body> should be a sequence of one or more
expressions. It is an error for a <variable> to appear
more than once in the list of variables being bound.
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.
```

Retour à la suite arithmétique

```
(define ua (lambda (n r initial)
  (letrec ((f (lambda (n)
    (if (= n 0)
      initial
      (+ r (f (- n 1)))))))
    (f n))))
(ua 3 4 0)
= 12
```

Suite géométrique

Tout terme d'une suite géométrique de raison q de la forme :

$u_0 = initial$ et pour $n > 1, u_n = q.u_{n-1}$ peut être calculée par la fonction **ug** suivante :

```
(define ug (lambda (q n initial)
  (letrec ((f (lambda (n)
    (if (= n 0)
      initial
      (* q (f (- n 1)))))))
    (f n)))
```

Puissance

Exemple : 4 puissance 3,

```
(ug 4 3 1)
= 64
```

Calcul de la somme des termes d'une suite

Exemple historique : La flêche de Zénon (ou histoire d'Achille et la tortue) n'arrive jamais à sa cible située à une distance D car la distance effectivement parcourue d est d'abord la moitié de la distance ($(1/2)D$), puis la moitié de ce qui reste $(1/4)D$ puis encore la moitié de ce qui reste, etc. Elle a parcouru à l'étape 1, $(1/2^1).D$, à l'étape 2, $(1/2^1 + 1/2^2)D$, puis à l'étape n, $(\sum_{i=1}^{n} 1/2^i)D$.

La distance parcourue par la tortue à l'étape `n` est toujours inférieure à D, quelque soit `n`.

Calcul de la somme des termes d'une suite

```
(define sz
  (lambda (n)
    (if (= n 1)
      (/ 1 (expt 2 n))
      (+ (sz (- n 1)) (/ 1 (expt 2 n))))))
```

Calcul de la somme des termes d'une suite

Plus élégant, utiliser une fonction anonyme interne pour calculer `(/1 (expt 2 n))`

```
(define sz
  (let ((f (lambda (n) (/ 1 (expt 2 n)))))
    (lambda (n)
      (if (= n 1)
        (f 1)
        (+ (f n) (sz (- n 1)))))))
```

Calcul de la somme des termes d'une suite

Généralisation au calcul de la somme des termes de toute suite u.

```
(define sommeSuite (lambda (n)
  (if (= n 0)
    (u 0)
    (+ (u n) (sommeSuite (- n 1))))))
```

A essayer avec : `(define u fact)`

Calcul de la somme des termes d'une suite

Optionnel : même fonctionnalité en n'écrivant qu'une seule fonction récursive, à condition de passer la fonction du calcul d'un terme en argument. La fonction somme devient une fonctionnelle ou fonction d'ordre supérieur.

```
(define sommeSuite (lambda (n u)
  (if (= n 1)
    (u 1)
    (+ (sommeSuite (- n 1) u) (u n)))))
```

Calcul de la somme des termes d'une suite

On peut par exemple écrire :

```
(somme 10 (lambda (n) (/ 1 (exp 2 n))))
```

Interprétation d'un appel à une fonction récursive

Appel récursif : un appel récursif est un appel réalisé alors que l'interprétation d'un appel précédent de la même fonction n'est pas achevé.

L'interprétation d'une fonction récursive passe par une **phase d'expansion** dans laquelle les appels récursifs sont "empilés" jusqu'à arriver à un appel de la fonction pour lequel une **condition d'arrêt** sera vérifiée, puis par une **phase de contraction** dans laquelle les résultats des appels précédemments empilés sont utilisés.

Interprétation d'un appel à une fonction récursive

Première comparaison de l'interprétation des versions itératives et récursives de factorielle.

La version itérative nécessite de la part du programmeur une gestion explicite de la mémoire alors que dans la version récursive, la mémoire est gérée par l'interpréteur (usuellement via une pile).

(il faut mémoriser le fait qu'après avoir calculé `fact(4)` il faut multiplier le résultat par 5 pour obtenir `fact(5)`)

Découverte d'une solution récursive à des problèmes

Disposer d'une solution récursive à un problème permet d'écrire simplement un programme résolvant (calculant quelque chose de relatif à) ce problème.

La découverte de telles solutions est parfois complexe mais rentable en terme de simplicité d'expression des programmes.

Découverte d'une solution récursive à des problèmes

Exemple : Algorithme récursif de calcul du pgcd de deux nombres non nuls :

`SI b divise a`
  `ALORS` $pgcd(a, b) = b$
  `SINON` $pgcd(a, b) = pgcd(b, modulo(a, b))$

Découverte d'une solution récursive à des problèmes

Implantation en Scheme :

```
(define pgcd (lambda (a b)
  (if (= b 0)
    (error "b doit être non nul")
    (let ((m (modulo a b)))
      (if (= m 0)
        b
        (pgcd b m))))))
```

Récursivité terminale et non terminale

Appel récursif non terminal : appel récursif argument d'un calcul englobant.

Exemple : l'appel récursif dans la définition de factorielle est non terminal car sa valeur est ensuite multipliée par n.

Appel récursif terminal appel récursif dont le résultat est celui rendu par la fonction contenant cet appel.

Exemple : appel récursif à `pgcd` dans la fonction précédente.

Propriété : l'interprétation d'un appel récursif terminal peut être réalisée sans consommer de pile.

Il est possible, en terme de mémoire, d'interpréter une fonction récursive terminale comme une fonction itérative car la gestion de la mémoire se déduit trivialement des transformations sur les paramètres.

Récursivité croisée

Exemple canonique “pair-impair” sur les entiers naturels

```
(define pair (lambda (n)
  (or (= n 0) (impair (- n 1)))))
(define impair (lambda (n)
  (and (not (= n 0)) (pair (- n 1)))))
```

Exercice : utiliser “letrec”.

Application au dessin de figures fractales

Voir, `http://classes.yale.edu/fractals/`.

Vidéo : "Fractales à la recherche de la dimension cachée", Michel Schwarz et Bill Jersey, 2010.

Autre cours : "Les images fractales en Scheme, Une exploration des algorithmes récursifs" - Tom Mens - University de Mons-Hainaut (U.M.H.).

Application au dessin de figures fractales

Programmation avec effets de bord (impressions à l'écran) :

```
(require (lib "graphics.ss" "graphics"))
(open-graphics)
(define mywin (open-viewport "Dessin" 800 800))
```

Application au dessin de figures fractales

Fonction de dessin des triangles de *Sierpinski*

```
(define s-carre (lambda (n x y cote)
  (let ((c2 (/ cote 2)))
    (if (= 0 n)
      ((draw-solid-rectangle mywin) (make-posn x y)
cote cote)
      (begin
        (s-carre (- n 1) (+ x (/ cote 4)) y c2)
        (s-carre (- n 1) x (+ y c2) c2)
        (s-carre (- n 1) (+ x c2) (+ y c2) c2))))))
```

Application au dessin de figures fractales

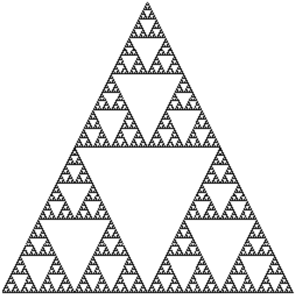


Figure – (s-carre 9 50 50 700)