LI101 : Programmation Récursive

Université Pierre et Marie Curie

Semestre : Automne 2013

Cours 3 : Récursion sur nombres entiers

Plan du cours

1. Récursion sur les entiers naturels
   - Exemples : $n!$, x^n
   - Principe de la récursion
   - Définition d'une fonction récursive
   - Évaluation par substitution
2. Nommage de valeurs : formes spéciales `let` et `let*`

Définitions récursives

- Factorielle
  Définition **informelle** de la factorielle : $n! = 1 * 2 * 3 * \ldots * n$
  Définition **récursive** de factorielle n :

$$\begin{array}{lcll} n! & = & 1 & \text{pour} \quad n = 0 \\ n! & = & n * (n-1)! & \text{pour} \quad n \geq 1 \end{array}$$

- Puissance
  Définition **informelle** de la puissance : $x^n = x * x * x * \ldots * x$
  Définition **récursive** de x puissance n :

$$\begin{array}{lcll} x^n & = & 1 & \text{pour} \quad n = 0 \\ x^n & = & x * x^{n-1} & \text{pour} \quad n \geq 1 \end{array}$$

Principe

- **Décomposition** : $f(n) = \ldots f(n-1) \ldots$
  Exprimer $f(n)$ en fonction de $f(p)$ avec $\mathbf{p} < \mathbf{n}$
- **Cas de base** : $f(0) = \ldots$
  donner la(les) valeur(s) de f pour la(les) valeur(s) de base

Une autre définition de la puissance

$$\begin{array}{lcll} x^n & = & 1 & \text{pour} \quad n = 0 \\ x^n & = & x^{n \div 2} * x^{n \div 2} & \text{pour} \quad n \text{ pair} \geq 1 \\ x^n & = & x^{n \div 2} * x^{n \div 2} * x & \text{pour} \quad n \text{ impair} \geq 1 \end{array}$$

Les nombres de Fibonacci

$$\begin{array}{lcll} fib(n) & = & 1 & \text{pour} \quad n = 0 \\ fib(n) & = & 1 & \text{pour} \quad n = 1 \\ fib(n) & = & fib(n-1) + fib(n-2) & \text{pour} \quad n \geq 2 \end{array}$$

Récursion en Scheme

Schéma général d'une définition récursive en scheme :

```
(define (f n)
 (if (> n 0)
     expression fonction de f(n-1)
     cas de base
  ))
```

Définition en Scheme de n !

```
;;; fact : nat -> nat
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ) )
```

Repérer

- le cas de base
- l'appel récursif

Évaluation d'une application

```
(f 3)  ⇝  (* 3 (f 2))
       ⇝  (* 3 (* 2 (f 1)))
       ⇝  (* 3 (* 2 (* 1 (f 0))))
```

`(f 0)` **cas de base : valeur → arrêt des appels récursifs**

```
       ⇝  (* 3 (* 2 (* 1 1)))
       ⇝  (* 3 (* 2 1))
       ⇝  (* 3 2)
       ⇝  6
```

Evaluation d'une application de fonction récursive

- C'est une application de fonction comme une autre : règle d'évaluation
- Évaluer `(fact 0)`, `(fact 3)`
- Faire la différence entre l'appel récursif de la définition de la fonction et les appels récursifs lors de l'évaluation.
- Que se passe-t-il lors de l'évaluation de `(fact -1)` ou `(fact 2.5)` ?
- À priori, on ne sait pas ce que cela peut donner : valeur retournée incorrecte, évaluation infinie, arrêt brutal dû à une opération interdite, etc. . .
- Dans le cas de `fact`, on peut éviter le processus infini en remplaçant la condition par `(<= n 0)`. La fonction rend alors 1 pour les nombres non entiers ou négatifs (spécification non respectée).

Définitions de la puissance

Définition **récursive** de x puissance n :

$$
\begin{array}{rcll}
x^n & = & 1 & \text{pour} \quad n = 0 \\
x^n & = & x * x^{n-1} & \text{pour} \quad n \geq 1
\end{array}
$$

Une autre définition de la puissance

$$
\begin{array}{rcll}
x^n & = & 1 & \text{pour} \quad n = 0 \\
x^n & = & x^{n \div 2} * x^{n \div 2} & \text{pour} \quad n \text{ pair} \geq 1 \\
x^n & = & x^{n \div 2} * x^{n \div 2} * x & \text{pour} \quad n \text{ impair} \geq 1
\end{array}
$$

La fonction puissance

```
;;; puissance : Nombre * nat -> Nombre
;;; (puissance x n) rend x à la puissance n
(define (puissance x n)
  (if (> n 0)
      (* x (puissance x (- n 1)))
      1 ) )
```

Repérer

- le cas de base
- l'appel récursif

Évaluer `(puissance 2 10)`, `(puissance 10 2)`, `(puissance 2 -1)`

La fonction puissance deuxième version

Traduction directe de la deuxième définition récursive :

```
;;; puissanceLent : Nombre * nat -> Nombre
;;; (puissanceLent x n) rend x à la puissance n

(define (puissanceLent x n)
  (if (= n 0)
      1
      (if (even? n)
          (* (puissanceLent x (quotient n 2))
             (puissanceLent x (quotient n 2)))
          (* x
             (puissanceLent x (quotient n 2))
             (puissanceLent x (quotient n 2))))))
```

Repérer dans la définition le cas de base et les appels récursifs.
Évaluer `(puissanceLent 2 2)`, `(puissanceLent 2 8)` et compter le nombre d'appels récursifs : regarder la trace des appels avec `trace`.

Trace d'exécution

```
|(puissanceLent 2 6)
| (puissanceLent 2 3)
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| 8
| (puissanceLent 2 3)
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| |(puissanceLent 2 1)
| | (puissanceLent 2 0)
| | 1
| | (puissanceLent 2 0)
| | 1
| |2
| 8
|64
64
```

Problème : le calcul de $x^{n \div 2}$ est fait 2 fois à chaque appel récursif, définition non efficace !
Solution ?

Mémoriser un résultat

Dans le cas où n est un entier pair non nul :

$$x^n = x^{n \div 2} * x^{n \div 2}$$

Au lieu d'effectuer 2 fois le calcul $x^{n \div 2}$, on aimerait :

1. calculer la valeur de $x^{n \div 2}$
2. donner un nom à cette valeur (P par exemple)
3. utiliser cette valeur pour le résultat final : $x^n = P * P$

La forme spéciale `let` correspond à ce mécanisme

Syntaxe du let

Règle de grammaire :

<bloc> ⟶ (**let** (<liaison>*) <corps>)

<liaison> ⟶ (<variable> <expression>)

<corps> ⟶ <expression>

L'écriture d'un `let` :

```
(let (( var1 expr1)
      ( var2 expr2)
      ...
      ( varN exprN) )
  corps )
```

Évaluation d'un let

Pour évaluer un `let` :

- **évaluation** des expressions *expr1*, ..., *exprN*
- **création** des variables, *var1*, ..., *varN*
- **enrichissement** de l'environnement courant en associant à chaque variable *varI* la valeur de *exprI*
- **évaluation** du corps *corps* dans cet environnement.

Portée des variables : corps du let

Puissance version efficace

Pour ne faire qu'un seul calcul de $x^{n \div 2}$ à chaque appel récursif, on utilise un `let` :

```
(define (puissanceRapide x n)
  (if (= n 0)
      1
      (let ((P (puissanceRapide x (quotient n  2))))
        (if (even? n)
            (* P P)
            (* x P P) ) ) ) )
```

Ici calcul de $x^{n \div 2}$ est mis en facteur.

Trace d'exécution

```
|(puissanceRapide 2 6)
| (puissanceRapide 2 3)
| |(puissanceRapide 2 1)
| | (puissanceRapide 2 0)
| | 1
| |2
| 8
|64
```

Puissance

Pour ne faire qu'un seul calcul de $x^{n \div 2}$ à chaque appel récursif, on peut aussi utiliser une fonction `carré` :

```
;;; carre : Nombre -> Nombre
;;; (carre y) rend le carré de y
(define (carre y)
  (* y y))

;; puissanceRapideBis a la même spécification
;; que puissance
(define (puissanceRapideBis x n)
  (if (= n 0)
      1
      (if (even? n)
          (carre (puissanceRapideBis x (quotient n  2)))
          (* x
             (carre (puissanceRapideBis x (quotient n 2)))))))
```

Trace d'exécution

```
|(puissanceRapideBis 2 6)
| (puissanceRapideBis 2 3)
| |(puissanceRapideBis 2 1)
| | (puissanceRapideBis 2 0)
| | 1
| | (carre 1)
| | 1
| |2
| |(carre 2)
| |4
| 8
| (carre 8)
| 64
|64
```

Retour sur le let : l'imbrication

Pour nommer des valeurs qui dépendent d'autres valeurs nommées on est obligé d'imbriquer les `let`.

```
(let ((a 2))
   (let ((b (+ a 2)))
      (let ((c (+ b 2)))
         c)))
```

La forme spéciale `let*` permet de le faire en une seule fois !

let versus let*

```
(let* ((var1 expr1)                 (let ((var1 expr1))
      ...                   ⟺          (let (...)
       (varN exprN))                       (let ((varN exprN))
  corps )                                      corps )))
```

Règle d'évaluation du `let*`

- **évaluation** de l'expression *expr1*, **création** de la variable *var1* et **enrichissement** de l'environnement en associant *expr1* à *var1*
- **évaluation** de l'expression *expr2* (dans l'environnement enrichi), **création** de la variable *var2* et **enrichissement** de l'environnement en associant *expr2* à *var2*
- ... et ainsi de suite jusqu'à associer *exprN* à *varN*
- **évaluation** du corps *corps* dans l'environnement résultant des enrichissements successifs.

Exemples avec let et let*

```
(let ((a 2)                |   (let ((a 2)
      (b (+ 3 4)))         |         (b (+ a 5)))
  (* a a b b))             |     (* a b))
--------------------------------------------------------

(let ((a 2))               |     (let* ((a 2)
  (let ((b (+ a 5)))       |            (b (+ a 5)))
    (* a b)))              |       (* a b))
--------------------------------------------------------

(let ((a 2))               |    (let ((a 2))
  (let ((a 3)              |      (let* ((a 3)
        (b (* a 2)))       |             (b (* a 2)))
    b))                    |        b))
```