PLAN DU COURS 3

« Programmation récursive »

Définitions récursives : récursion sur les nombres

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

DÉFINITIONS RÉCURSIVES

- Factorielle

  Spécification **informelle** de la factorielle : $n! = 1 * 2 * 3 * \ldots * n$

  Définition **récursive** de factorielle n :

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

- Puissance

  Spécification **informelle** de la puissance : $x^n = x * x * x * \ldots * x$

  Définition **récursive** de x puissance n :

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

PRINCIPES

- **Décomposition** : $f(n) = \ldots f(n-1) \ldots$

  Exprimer $f(n)$ en fonction de $f(p)$ avec **p<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

$$\begin{aligned} x^n &= 1 \quad \text{pour} \quad n = 0 \\ x^n &= x^{n \div 2} * x^{n \div 2} * x^{n \bmod 2} \quad \text{pour} \quad n \geq 1 \end{aligned}$$

Les nombres de Fibonacci

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

EN SCHEME

- Définition d'une fonction

```
(define (f n)
  (if (>= n 0)
      EXPRESSION2 : CALCUL EN FONCTION DE f(n-1) ...
      EXPRESSION1 : QUOI RENVOYER POUR CAS DE BASE
  ))
```

- Évaluation d'une application

`(f 5)` → `(f 4)` → `(f 3)` → `(f 2)` → `(f 1)` → `(f 0)`

`(f 0)` *cas de base : ARRÊT des appels récursifs*

Définition en Scheme de n !

```
;;; fact : nat/>=0/ -> 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

Évaluer `(fact 0)`, `(fact 3)`, `(fact -1)` : pour éviter le processus infini (!) il faut remplacer la condition par `(>= n 0)`

DEM

Autre définition

```
;;; fact : nat -> nat
;;; ERREUR si n n'est pas positif ou nul
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (or (negative? n) (not (integer? n)))
      (erreur 'fact "attend un argument entier >= 0")
      (if (= n 0)
          1
          (* n (fact (- n 1))) ) ) )
```

Évaluer `(fact 5)`, `(fact -5)`, `(fact 2.5)`

Inconvénient : le test supplémentaire est fait à chaque appel récursif

Tester la positivité en dehors de la fonction récursive

DEM

```
;;; factorielle : nat  -> nat
;;; ERREUR si n n'est pas positif ou nul
;;; (factorielle n) rend la factorielle de n
(define (factorielle n)
  (if (negative? n)
      (erreur 'factorielle "attend un arg. entier >= 0")
      (fact n) ))

;;; HYPOTHÈSE n est un entier positif ou nul
;;; (fact n) rend la factorielle de n
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ) )
```


Définitions de la puissance

Définition **récursive** de x puissance n :

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

Une autre définition de la puissance

$$\begin{aligned} x^n &= 1 \quad \text{pour} \quad n = 0 \\ x^n &= x^{n \div 2} * x^{n \div 2} * x^{n \bmod 2} \quad \text{pour} \quad n \geq 1 \end{aligned}$$

La fonction Puissance

```
;;; puissance : Nombre/>=0/ * nat -> nat
;;; (puissance x n) rend x a 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)`

DEM

Puissance

Une autre définition de la puissance :

```
;;; puissance : Nombre * nat -> Nombre
;;; HYPOTHESE n est un entier positif ou nul
;;; (puissance x n) rend x a la puissance n
(define (puissance x n)
  (if (= n 0)
      1
      (let (( P (puissance x (quotient n  2))))
         (if (even? n)
             (* P P)
             (* x P P)))))
```


PB : calcul de $x^{n \div 2}$ est fait 2 fois à chaque appel récursif

```
|(puissance 2 6)
| (puissance 2 3)
| |(puissance 2 1)
| | (puissance 2 0)
| | 1
| | (puissance 2 0)
| | 1
| |2
| |(puissance 2 1)
| | (puissance 2 0)
| | 1
| | (puissance 2 0)
| | 1
| |2
| 8
```



```
| (puissance 2 3)
| |(puissance 2 1)
| | (puissance 2 0)
| | 1
| | (puissance 2 0)
| | 1
| |2
| |(puissance 2 1)
| | (puissance 2 0)
| | 1
| | (puissance 2 0)
| | 1
| |2
| 8
|64
```

Un autre essai de définition de la puissance (mais mêmes calculs)

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

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



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


Avec une fonction interne

```
(define (puissanceBisBis x n)

  (define (carre y)
    (* y y) )

  (if (= n 0)
      1
      (if (even? n)
          (carre (puissanceBisBis x (quotient n  2)))
          (* x (carre (puissanceBisBis x (quotient n 2)) ) ))))
```

Évaluer `(puissanceBisBis 100 2)`, `(carre 5)`

Notion d'environnement GLOBAL vs COURANT

Et avec les spécifications :

```
;;; puissanceBisBis : Nombre * nat ->  Nombre
;;; (puissanceBisBis x n) rend x a la puissance n
(define (puissanceBisBis x n)
  ;; carre : Nombre -> Nombre
  ;; (carre y) rend le carre de y
  (define (carre y)
    (* y y) )
  (if (= n 0)
      1
      (if (even? n)
          (carre (puissanceBisBis x (quotient n  2)))
          (* x (carre (puissanceBisBis x (quotient n 2)))) ) ) )
```

Puissance

Encore une autre définition

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

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


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


La syntaxe d'un `let` (Rappel)

<bloc> → `(let (` *<liaison>** `)` *<corps>* `)`

<liaison> → `(` *<variable>* *<expression>* `)`

L'écriture d'un `let` :

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


Évaluation d'un `let` (rappel)

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

Syntaxe et évaluation d'un `let*`

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

- **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)))   NON !!!
  (* 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))
```


Travail avant le prochain TD/TP

- Reprendre vos notes ce soir
- Lire le cours
  - Écriture d'algorithmes récursifs
  - Nommage de valeurs (formes spéciales LET et LET*)
- Faire les exercices « d'assouplissement » de cette partie de cours
- Etre capable d'écrire, sans l'aide des notes et du cours, la spécification et une définition de :
  - factorielle
  - puissance (différentes versions)
- Faire tourner sur Drscheme la factorielle ou la puissance en pas à pas

- Faire tourner sur Drscheme la factorielle et la puissance avec des traces des appels récursifs