Option PF2, Automne 2016

« Programmation Scheme Avancée »

TP 2

Attention, danger ! Vous pénétrez dans le monde impératif, domaine du traitement séquentiel et des mutations irréversibles !

*Vous travaillez dans un fichier `tp2.rkt` qui est un **module**. Il débutera par les lignes :*

```
#lang racket
;;; tp2.rkt
(require "utils.rkt")      ; ce module exportera uniquement while et show
```

Créez `utils.rkt`, et au fur et à mesure des TP, vous placez dans `utils.rkt` les fonctions que vous pensez ré-utilisables…

L'AFFECTATION

Exercice 2.1 Oubliez la machine. Complétez les réponses manquantes sur la feuille, puis vérifiez vos réponses au toplevel une fois que vous avez tout complété :

```
> (define (empiler x L)
    (set! L (cons x L))
    L))
> (define L '(a b c))
> (empiler 1 L)
```

```
> L
```

```
> (set! L (cons 1 L))
> L
```

Exercice 2.2 **a)** Avec une boucle `while`, *en une seule expression*, et *sans écrire de nouvelle fonction*, faites afficher avec `printf` la somme des carrés des entiers impairs de [1,100]. Réponse : 166650

b) Avec une boucle `while`, programmez impérativement la fonction `(reverse L)` retournant une copie de l'inverse d'une liste `L`.

Exercice 2.3 **a)** Pour remédier à l'exercice 1 [*appel par valeur*], programmez la **macro** `(empiler! x L)` empilant la valeur de `x` en tête de la liste `L`. On souhaite que `L` reste modifiée en sortie. Aucun résultat.

b) Programmez de même la macro `(depiler! L)` retournant le premier élément de la liste `L`, mais en le supprimant de `L`, qui reste modifiée en sortie.

Exercice 2.4 **a) Dans le fichier `utils.rkt`,** programmez [avec, puis sans les boucles `for` de Racket] une version très simple de la **macro** `loop` du logiciel musical *Grace*, limitée aux 3 motifs possibles ci-dessous [j'ai Capitalisé les mots réservés pour ne pas entrer en conflit avec ceux de Racket] :

```
(loop For i From a To b By s Sum expr)
(loop For x In L When test Collect expr)
```

dont des exemples pourraient être :

```
> (loop For i From 1 To 10 By 2 Sum i)       ; 1+3+5+7+9
25
> (loop For x In '(3 2 5 4 9 8) When (even? x) Collect (sqr x))
(4 16 64)
```

b) Testez-la dans le fichier `tp2.rkt` sur la somme des carrés des entiers impairs de [1,100] qui vaut 166650.

LES GÉNÉRATEURS

Exercice 2.5 **a)** Ecrivez une fonction `(make-gen-fac)` retournant un **générateur de factorielles** : 1, 1, 2, 6, 24, 120,... [en O(1) à chaque génération]

b) Ecrivez un générateur de nombres entiers aléatoires distincts, de signes quelconques (disons entre -99 et 99 inclus).

LES MÉMO-FONCTIONS *(exercice fait en L1-Python)*

Exercice 2.6 **a)** On rappelle la définition additive de la fonction récursive `(binomial n p)` retournant le coefficient C_n^p du binôme de Newton [0 ≤ p ≤ n], basée sur la formule de récurrence $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$:

```
(define (binomial n p)
  (if (or (= p 0) (= n p))      ; si je suis sur le bord du triangle de Pascal
      1
      (+ (binomial (- n 1) p) (binomial (- n 1) (- p 1)))))      ; sinon récurrence !
```

qui permet de construire additivement le triangle de Pascal. Vérifiez que le temps de calcul de `(bin 50 20)` n'est pas vraiment rapide…

b) Programmez une autre version `(memo-bin n p)` qui se souvienne des calculs déjà effectués, en les stockant au fur et à mesure dans une *table* qui aura la structure d'une A-liste. Dans cette table, le neurone `((n p) r)` mémorisera le résultat du calcul $r = C_n^p$. Quel est alors le temps de calcul de `(memo-bin 50 20)` ? Chronométrez ce même calcul deux fois de suite…

c) Avec une boucle `for`, faites afficher à l'horizontale la ligne numéro 10 du triangle de Pascal : `1 10 45 120 ... 10 1`.

EXERCICES SUPPLÉMENTAIRES OPTIONNELS POUR LA **FORCE**

=> Il serait bon (mais non obligatoire) que vous vous familiarisiez avec ***la vraie boucle `for` de Racket*** *(ou plutôt la famille des boucles `for`...), car elle est de plus en plus utilisée dans le code Racket qui circule, même si elle ne fait pas partie du Scheme normalisé. Elle est un peu inspirée de ce qui se fait avec les* ***itérateurs*** *qui vous seront enseignés en Python au S3.*

Exercice 2.7 **a)** Ecrire une version itérative impérative de la fonction de Fibonacci, de complexité linéaire en nombre d'opérations, donc assez rapide, et cette fois O(1) en espace [ceci signifie « en espace constant »].

b) La question b) n'existe pas.

Exercice 2.8 Documentez-vous sur la fonction primitive (**`for-each proc L`**), l'analogue impératif du `(map f L)` fonctionnel mais sans résultat. Utilisez-la pour programmer impérativement la fonction `($reverse L)`.

Exercice 2.9 Sur le modèle de l'Exercice 2.5, programmez une fonction `(make-gen-fib)` retournant un *générateur* de nombres de Fibonacci : 0, 1, 1, 2, 3, 5, 8,... [O(1) à chaque génération]

Exercice 2.10 Un canvas vertical 80 x 300. Une balle de rayon 30 est lâchée sans vitesse initiale du milieu du tube, et soumise à la gravitation [g = 1], avec une coeff. de friction [F = 0.95]. Modélisez cette animation d'un MONDE MUTABLE (brrr).

N.B. Le monde étant mutable, on fait muter directement les variables locales de l'animation. On est obligé de le passer en paramètre mais au fond il ne sert à rien. Pauvre monde...

Exercice 2.11 **a)** Reprenez la solution de l'exercice 2.6 en remplaçant les A-listes par des tables hash-code mutables. Programmez la fonction `(H1-binomial n p)` qui gèrera une table de hash en variable privée.

b) Reprenez la question **a)** en utilisant cette fois des tables de hash-code fonctionnelles. Ne pouvant les muter, il faudra passer la table de hash en paramètre et la rendre dans le résultat de l'appel récursif ! Programmez une fonction `(H2-binomial n p H)` qui prendra aussi en argument une table de hash `H`, et rendra une liste à deux éléments `(res new-H)`, où `res` est le résultat du calcul et `new-H` la table de hash finale qui aura été construite lors de son calcul. Bien entendu, si le neurone `(n p)` est déjà dans `H`, le résultat est immédiat...

c) Reprenez l'exercice 2.6 et les questions **a)** et **b)** précédentes. En remarquant qu'une liste à deux éléments `(a b)` coûte plus cher en mémoire (pourquoi ?) qu'un doublet `(a . b)`, remplacez partout les listes à deux éléments par des doublets.

Exercice 2.12 Considérez la fonction Python suivante `foo` et prévoyez le résultat de `[foo(),foo(),foo()]`. Chantez les louanges des langages *simples*...

```
def foo(L=[]) :
    L.append(1)
    return L
```