Programmation Fonctionnelle I, Printemps 2017 – TD6

http://deptinfo.unice.fr/~roy

SI VOUS CONNAISSEZ PYTHON : Ne pas confondre les listes Scheme et les listes Python dont les opérations n'ont pas du tout la même complexité ! Les listes Scheme sont ce que l'on nomme classiquement les "listes chaînées" [linked lists], et celles de Python les "tableaux extensibles" [expandable arrays]...

Exercice 6.1 *Fermez vos documents, ne regardez pas le cours.* Reprogrammez le **tri par insertion**. Ce genre de gamme doit être jouée sans l'ombre d'une hésitation si vous avez appris votre cours !

Exercice 6.2 Programmez par récurrence les fonctions suivantes :

a) La primitive `(range start end step)` que la documentation en ligne de Racket explique de la manière suivante : *"*Constructs a list of numbers by *step*ping from *start* to *end"*. On supposera `step > 0` dans cet exercice. Exemple :

```
(range 10 20 2) → (10 12 14 16 18)      ; elle travaille donc dans [start,end[
```

b) La fonction `(nomul3 n)` prenant un entier `n ≥ 0` et retournant la liste des entiers de `[0,n]` non multiples de 3. Quelle est la complexité de votre fonction, si l'on mesure le nombre d'appels à `cons` ? Si elle est *quadratique* [$O(n^2)$], vous êtes priés d'en produire une à complexité *linéaire* [$O(n)$].

Exercice 6.3 a) Programmez par récurrence la fonction `(card L)` prenant une liste L et retournant le nombre d'éléments distincts de L :

```
(card '(a b c b b a d c)) → 4        ; prolongement dans l'exo 6.6
```

b) Programmez par récurrence la fonction `(compacter L)` prenant une liste `L` contenant des répétitions et retournant une liste de listes à deux éléments `((x n) ...)` où `n` est le nombre d'apparitions de `x`. L'ordre n'a pas d'importance. Exemple :

```
(compacter '(a b c d b b a d)) --> ((d 2) (b 3) (c 1) (a 2))
```

Annexe : Python vs Scheme - Séance 6

Insistons bien sur la différence de traitement des **listes Scheme** qui sont classiquement nommées *listes chaînées* (leurs éléments sont éparpillés dans la mémoire et non stockés de manière linéaire dans la mémoire de la machine), contrairement aux **listes Python** qui sont des *tableaux* (ensembles de cellules mémoire contigües) extensibles. **Nous traiterons ici les listes Scheme de manière exclusivement récursive** - ce qui comprend aussi l'itération comme nous le verrons plus tard - alors qu'on utilisera surtout des boucles en Python. Les expressions `(first L)` et `(rest L)` de Scheme pourraient naïvement se traduire `L[0]` et `L[1:]` en Python mais du coup la fonction `rest` aurait une complexité $O(n)$, ce qui serait proprement catastrophique dans des algorithmes par récurrence.

Notez que les listes Racket ne sont **pas mutables** par défaut (il existe des listes mutables, la tendance actuelle est à l'élimination des mutations), mais ceci ne nous concerne pas puisque la mutation en Scheme n'est étudiée que dans l'option de *Programmation Avancée* (PF2) au semestre 3 de L2-Info...

Il reste bien entendu que l'on peut simuler en Scheme les listes Python et leurs opérations, et inversement (vous le ferez en L2-Info) ! Tous les langages sont réputés avoir la même puissance, mais on doit s'adapter aux propriétés du langage utilisé. La bonne possession de 2 ou 3 langages bien distincts en License Informatique est une force majeure pour le futur programmeur, qui est à mille lieux de se douter du langage qu'il devra utiliser mais aussi de l'évolution des langages de programmation.

Programmation Fonctionnelle I, Printemps 2017 – TP6

http://deptinfo.unice.fr/~roy

Exercice 6.1 a) Programmez une fonction `(somme L)` prenant une liste de nombres `L`, et retournant la somme de ces nombres.

`(somme '(6 3 1 8 2)) → 20`

b) En déduire une fonction `(moyenne L)` retournant la moyenne de la liste de nombres `L` :

`(moyenne '(6 3 1 8 2)) → 4`

c) Sauriez-vous programmer `(moyenne L)` en un seul passage récursif sur la liste `L`, donc en calculant la longueur en même temps que la somme sans utiliser `length` ? *Indication : programmez une fonction retournant deux résultats !*

Exercice 6.2 En utilisant la primitive `build-list` [donc sans récurrence], programmez les fonctions suivantes :

a) La fonction `(entiers n)` prenant un entier `n ≥ 0` et retournant la liste des entiers de `[0,n]` :

`(entiers 6) → (0 1 2 3 4 5 6)`

b) La fonction `(intervalle a b)` prenant deux entiers `a` et `b`, et retournant la liste des entiers de `[a,b]`. Exemple :

`(intervalle -2 5) → (-2 -1 0 1 2 3 4 5)`

Exercice 6.3 Récupérez dans le cours 6 la fonction `(tri-ins L rel?)` qui trie une liste `L` de nombres avec l'algorithme du **tri par insertion**, par-rapport à la relation d'ordre `rel?`. Testez-la sur une petite liste d'entiers. Nous avons vu que la complexité de ce tri était quadratique, en $O(n^2)$. Nous nous proposons de vérifier expérimentalement ce résultat théorique, pour n grand.

a) Programmez la fonction `(Lrandom n max)` prenant deux entiers `n` et `max > 0`, et retournant une liste de longueur `n` comportant des entiers aléatoires de `[0,max]`. Exemple : `(Lrandom 10 100) → (84 11 25 91 97 65 11 78 24 9).`

b) Définissez deux listes `L2000` et `L4000` constituées respectivement de `2000` et `4000` entiers aléatoires de `[0,100]`.

c) Comparez les **temps de calcul** du tri par insertion de ces deux listes [utilisez la primitive `time`]. Si l'algorithme est vraiment quadratique $O(n^2)$, vous devriez trouver un temps environ 4 fois plus long pour `L4000` que pour `L2000`. Que dit le chrono ?

d) Vérifiez au chronomètre que le tri prédéfini `(sort L rel)` de Racket semble bien en *O(n log n)*, donc plus rapide.

N.B. i) *N'oubliez pas de jeter la grosse liste triée à la poubelle en enveloppant le tri par un appel à la fonction `void` [on ne s'intéresse pas au résultat, seulement à la mesure du temps].*

ii) *Dans l'affichage de `time`, on ne retient que le premier temps diminué du troisième : cpu - gc, en millisecondes.*

Exercice 6.4 a) Programmez la fonction `(que-les-impairs L)` prenant une liste d'entiers `L`, et retournant une liste formée des mêmes éléments que `L`, dans le même ordre, mais en ne conservant que les entiers impairs. Exemple :

`(que-les-impairs '(7 4 6 3 9 12 21 8 1)) → (7 3 9 21 1)`

b) Programmez une fonction `(hasard L)` retournant un élément au hasard d'une liste `L`, chaque élément avec la même probabilité. Indication : utilisez `list-ref`.

Exercice 6.5 Convenons d'appeler **ensemble** une liste sans répétitions et dont l'ordre n'a pas d'importance. Par exemple `(a b c)` et `(c a b)` sont les mêmes ensembles tandis que `(a b a)` n'est pas un ensemble.

a) Programmer une fonction `(liste->ens L)` prenant une liste `L` et retournant l'ensemble de ses éléments :

`(liste->ens '(a b b c a d b b f a)) → (c d b f a)`

b) Programmer une fonction `(union E1 E2)` retournant l'ensemble réunion des ensembles `E1` et `E2`.

c) Programmez la fonction `(difference E1 E2)` retournant l'ensemble `E1 − E2` des éléments de `E1` qui ne sont pas dans `E2`. Exemple : `(difference '(a b c d e) '(j e f c)) → (a b d)`

d) Programmez une fonction `(produit E1 E2)` retournant l'ensemble produit des ensembles `E1` et `E2`, c'est à dire l'ensemble des couples `(x y)` avec `x ∈ E1` et `y ∈ E2`. Exemple :

`(produit '(a b c) '(1 2)) → ((c 1) (c 2) (b 1) (b 2) (a 1) (a 2))`

Exercice 6.6 Dans le même ordre d'idées que l'exercice 6.1c, programmez une fonction `(parité L)` retournant **deux résultats** sous la forme d'une liste `(L1 L2)` où `L1` est la liste des pairs et `L2` celle des impairs, dans le même ordre. Exemple :

`(parité '(7 4 6 3 9 12 21 8 1)) → ((4 6 12 8) (7 3 9 21 1))`