Programmation Scheme Avancée

option L2-Info+Math

UNS, Janvier 2017

LE POLYCOPIE DU COURS ET LE LIVRE DE COURS SONT LES SEULS DOCUMENTS AUTORISES. Soignez l'indentation, le parenthésage et la limpidité du code. Le correcteur est davantage intéressé par un code qui fonctionne que par de vagues idées. Répondez aux questions sur la copie d'examen qui vous est fournie. Ne joignez aucune autre feuille ni intercalaire. Et ne perdez pas de temps !

1 Une petite fonction dans divers styles [6 pts]

On s'intéresse à la fonction `(zip L1 L2)` prenant deux listes `L1` et `L2` de même longueur [ceci est garanti], et retournant la liste des couples de leurs éléments de même rang :

`(zip '(1 2 3) '(a b c))` $\rightsquigarrow$ `((1 a) (2 b) (3 c))`

a) Programmez `(zip L1 L2)` de manière **fonctionnelle récursive enveloppée**.
b) Programmez `zip` de manière **fonctionnelle récursive terminale** (donc itérative).
c) Programmez `zip` *en une ligne* de manière **fonctionnelle à l'ordre supérieur**.
d) Programmez `zip` en **impératif**, avec une boucle `(while test e1 e2 ...)` supposée exister.
e) Faites en sorte que le résultat de `(zip L1 L2)` soit un **générateur** des couples attendus. Le générateur retournera le symbole `'*echec*` lorsqu'il sera épuisé.

```
> (define foo (zip '(1 2 3) '(a b c)))
; foo est un générateur
> foo
#<procedure>
> (foo)
(1 a)
```

```
> (foo)
(2 b)
> (foo)
(3 c)
> (foo)
*echec*
```

2 Une nouvelle structure de boucle [2 pts]

On se propose de programmer la structure de boucle `(repeat n e1 e2 ...)` qui commence par évaluer `n`, puis évalue en séquence `n` fois la suite d'expressions `e1, e2,...` On suppose que `n` $\geq$ `0`. Aucun résultat. Exemple :

```
> (define x 0)
> (repeat 5 (set! x (+ x 1)) (printf "~a " x))
1 2 3 4 5
> x
5
```

a) Pourquoi `repeat` ne peut-elle *pas* être une fonction ?
b) Programmez `repeat`.

3 Un transformateur syntaxique [4 pts]

Dans l'interprète μSCHEME écrit en PYTHON (cours 11), nous avions traité la conditionnelle `(if p q r)` mais pas la forme `(and t1 t2 ...)`. Pour cela, il suffit de transformer cette expression en une forme `if` ne contenant plus aucun `and`, quelque chose comme une macro-expansion du source. Pour vous faire la main, vous allez programmer en SCHEME une fonction `(and->if expr)` prenant une expression SCHEME `expr` pouvant ou non contenir des appels à `and`, et retournant une expression *équivalente*[1] mais dans laquelle chaque appel à `and` aura été remplacé par un appel à `if`. On pourra supposer que tous les appels à `and` sont **binaires**. Il s'agit essentiellement d'un parcours en profondeur d'une liste. Exemples :

```
> (and->if '(if (and (> x 0) (> y 0)) (+ x 1) (- x 1)))
(if (if (> x 0) (> y 0) #f) (+ x 1) (- x 1))
> (and->if '(or (if (and (> x 0) (and (> y 0) (= x y)))
                    (> x z)
                    (and (> x z) (= z 2)))))
(or (if (if (> x 0) (if (> y 0) (= x y) #f) #f) (> x z) (if (> x z) (= z 2) #f)))
```

4 Une classe de listes comme en PYTHON [8 pts]

On souhaite implémenter en SCHEME une classe `plist%` analogue à la classe `list` de PYTHON, en utilisant la couche-objet de RACKET. Nous nommerons *p-liste* un objet de la classe `plist%` : éléments mutables, accès à tout élément en $\mathcal{O}(1)$ et ajout possible d'un élément en queue de liste en $\mathcal{O}(1)$ amorti. L'un des champs d'un objet `L` de la classe `plist%` sera un vecteur `v`, dont la longueur sera initialement 3 à la construction de l'objet. Les *éléments* de `L` sont dans les cases numérotées $0,\ldots,k-1$ de `v` et l'on dira que la *taille* de `L` est égale à k. La case numéro k est donc la première case libre du vecteur sous-jacent. On ne confondra pas la *taille* de `L`, et la *longueur* de `v` (qui se nomme aussi la *capacité* de `L`).

Programmez la classe complète `plist%`.

- Un objet de cette classe possède deux champs **privés**. Le premier est le vecteur `v`, initialement de longueur 3. Le second est la taille `k` initialisée à 0, indice de la première case libre de `v`. Le constructeur construit *uniquement* une p-liste vide de capacité 3.
- Une méthode publique `len` retourne en résultat la taille de la p-liste courante.
- Une méthode `get` retourne un élément de la p-liste courante dont on précise la position `i`. Elle déclenche une erreur *Indice incorrect* si la position `i` est illégale.
- Une méthode publique `append` permet de rajouter un élément en queue de la p-liste courante. Si le vecteur est plein, un nouveau vecteur de taille double est automatiquement alloué, l'ancien vecteur est recopié dans le nouveau, et l'élément est rajouté. Aucun résultat. *Le Garbage Collector s'occupera de recycler l'ancien vecteur.*
- Une méthode publique `pop` permet de supprimer l'élément de la p-liste courante dont on précise la position `i`. On supposera ici que la position `i` est légale, et la méthode procèdera par décalage à gauche des cases utilisées à partir de l'indice `i+1`. Elle retournera en résultat l'élément supprimé.
- Une méthode publique `dump`, à des fins de *debug*, fait afficher la p-liste courante, les éléments non utilisés du vecteur étant affichés sous la forme d'un `'?'`. Aucun résultat.

1. *Equivalente* signifiant ici : faisant exactement les mêmes calculs.

```
> (define L (new plist%))
> L
(object:plist% ...)
> (send L append 'a)
> (send L append 'b)
> (send L append 'c)
> (send L dump)
a b c
> (send L append 'd)
> (send L len)
4
```

```
> (send L dump)
a b c d ? ?
> (send L pop 2)
c
> (send L dump)
a b d ? ? ?
> (send L get 2)
d
> (send L get 3)
ERROR : Indice incorrect
```

FIN