Scheme - TP n° 1

Université Paris I. Maîtrise de Logique. Année 2004-2005.
Nicolas Stroppa - `stroppa@enst.fr`

1 Introduction

Le langage Scheme est un des représentants du paradigme de la *programmation fonctionnelle*, dont le langage le plus connu est Lisp. Scheme est un « descendant » de LISp (LISt Processing).

Il se fonde sur le formalisme du lambda-calcul. En Scheme, on va représenter les termes du lambda-calcul (notamment par des listes), et l'interpréteur Scheme aura pour fonction d'*évaluer* ces termes. Une des particularité du langage, liée à la nature du lambda-calcul, est de pas différencier les données et les programmes. En d'autres termes, une liste en Scheme va pouvoir représenter à la fois une donnée (e.g. la suite des 4 premiers nombres premiers: `(2 3 5 7)`), ou une fonction (e.g. la fonction ajouter2: `(define (ajouter2 x) (+ x 2))`).

En Scheme, (presque) tout est représenté sous forme de listes: ce sont les structures fondamentales du langage. Elle s'écrivent à l'aide de parenthèses, comme dans `(2 3 5 7)`. L'objectif du TP est de se familiariser avec la nature fonctionnelle de Scheme, de manipuler la structure de liste et de définir quelques fonctions agissant sur ces structures.

2 Manipulation de fonctions

2.1 Écrire des termes du lambda-calcul

Le lambda-calcul fournit un cadre formel pour exprimer les fonctions et les calculs. Par exemple, la fonction qui calcule le carré d'un nombre réel s'écrira $\lambda x.x^2$. En Scheme, l'écriture reste très proche de celle du lambda-calcul: cette fonction s'écrira `(lambda (x) (* x x))` (notez la position infixée de l'opérateur). Cette liste comprend trois termes: (i) la *forme spéciale* `lambda`, (ii) la liste de *paramètres* `(x)` et (iii) le *corps de la fonction* `(* x x)`, qui permet de définir le « résultat » du calcul.

Définissez les fonctions dont les correspondants en lambda-calcul sont:

1. $\lambda x.x - 9$.
2. $\lambda xy.x \times (y + 2)$.
3. $\lambda xyz.(x - z) \times ((y - x) \times z)$.

Appliquer une fonction `f` à un paramètre `x` s'écrit `(f x)`. Appliquez les fonctions précédemment définies à quelques paramètres et vérifiez qu'elles donnent le bon résultat.

2.2 Réutiliser les termes

Il peut être utile d'associer un nom à une valeur (terme) de façon à pouvoir la réutiliser par la suite. Ceci s'effectue à l'aide de la forme spéciale `define`. On peut ainsi définir des variables simples, comme `(define annee 2005)`, mais aussi des fonctions, comme `(define carre (lambda (x) (* x x)))`. Cette dernière écriture peut être replacée par `(define (carre x) (* x x))`.

Donnez un nom et redéfinissez de cette manière les fonctions précedemment définies.

2.3 Les fonctions: des termes comme les autres

En Scheme (comme en lambda-calcul), une fonction est un terme comme un autre. Cela veut dire en particulier qu'une fonction peut être donnée en argument à une autre fonction et que le résultat d'un calcul peut être une fonction.

Définissez les fonctions suivantes.

1. La fonction qui prend en argument deux fonctions et qui retourne la fonction qui est la composition des deux fonctions.
2. La fonction qui retourne la somme de deux fonctions. ($\forall x,\ f(x) = f_1(x) + f_2(x)$.
3. La fonction qui retourne la version currifiée d'une fonction à 2 arguments.
4. La fonction qui effectue l'opération inverse.

2.4 Procédure d'évaluation

L'interpréteur Scheme suit la procédure suivante d'évaluation. Si l'élément à évaluer est une valeur ou un nom atomique, l'évaluateur renvoie sa valeur. Si c'est une liste, deux cas se présente. Si le premier élément est une forme spéciale (`lambda`, `define`, `if`, etc.), alors les règles spécifiques à la forme sont appliquée. Sinon, on évalue les éléments de la liste, puis on applique le résultat de l'évaluation du premier élément (considéré comme une fonction) et on l'applique aux évaluations des autres éléments de la liste (considéré comme les arguments de la fonction).

2.5 Récursivité

La récursivité est au cœur des processus de calcul. En Scheme, une fonction que l'on définit avec `define` peut s'appeler elle-même et la récursivité est alors gérée de façon transparente.

Définissez les fonctions récursives suivantes.

1. La fonction factoriel.
2. La fonction de Finonacci.
3. La fonction itération (composition d'une même fonction n fois).
4. Les fonctions qui déterminent la parité d'un entier.

3 La structure de liste

Avant d'étudier la structure de liste, nous passons par la définition d'un couple. Un couple permet de considérer un groupement de 2 valeurs. Un couple peut-être construit à l'aide de la fonction `cons`: `(cons 1 2)` est évalué à `(1 . 2)`. (Ici, le point peut être considéré comme un foncteur, comme en Prolog). Le premier (resp. second) élément du couple est récupéré à l'aide la fonction `car` (resp. `cdr`). `(car (cons 1 2))` (resp. `(cdr (cons 1 2))`) est évalué à `1` (resp. `2`). À partir de la notion de couple, il est maintenant possible de définir les listes: une liste est une couple dont le premier élément représente la *tête* de la liste, et le second le *reste* de la liste, ce reste étant lui même une liste. (Cela était déjà le cas en Prolog.) La liste vide est notée `()`. Ainsi la liste `(2 4 6 8 10)` est en réalité un racourci qui signifie `(cons 2 (cons 4 (cons 6 (cons 8 (cons 10 '()))))`. Les fonctions `car` et `cdr` permettent donc de récupérer la tête et le reste d'une liste.

3.1 Manipulation de listes

Définir les fonctions suivantes :

1. `(der l)` retourne le dernier élément de la liste `x`.
2. `(nieme l n)` retourne le n-ième élément de la liste `l`.
3. `(elem l e)` retourne vrai si `e` est un élément de la liste `l`.
4. `(concat x y)` retourne la concaténation des listes `x` et `y`.
5. `(long l)` retourne la longueur de la liste `l`.

4 Éléments de corrigé

```
;; partie 2.3
(define (compose f g)
  (lambda (x) (f (g x))))
(define (ajoute_fonctions f g)
  (lambda (x) (+ (f x) (g x))))

(define (curryfie f)
  (lambda (x)
    (lambda (y) (f x y))))

(define (decurryfie f)
  (lambda (x y)
    ((f x) y)))



;; partie 2.5
(define (fact n)
  (if (<= n 1) 1 (* n (fact (- n 1)))))
(define (fibo n)
  (if (<= n 1) 1 (+ (fibo (- n 1)) (fibo (- n 2)))))
(define (id x)
  x)
(define (iter f n)
  (cond ((= n 0) id) ((= n 1) f) (#t (compose f (iter f (- n 1))))))
(define (pair n)
  (if (= n 0) #t (impair (- n 1))))
(define (impair n)
  (cond ((= n 0) #f) ((= n 1) #t) (#t (pair (- n 1)))))



;; partie 3
(define (der l)
  (let ((cdrl (cdr l)))
    (if (null? cdrl) (car l) (der cdrl))))
(define (nieme l n)
  (if (= n 1) (car l) (nieme (cdr l) (- n 1))))
(define (elem l e)
  (cond
   ((null? l) #f)
```

```
   ((= (car l) e) #t)
   (#t (elem (cdr l) e))))
(define (concat x y)
  (if (null? x) y (cons (car x) (concat (cdr x) y))))
(define (longueur l)
  (if (null? l) 0 (+ 1 (longueur (cdr l)))))
```