Option PF2

TP 1

« Programmation Scheme Avancée »

***Nous n'utiliserons plus le teachpack valrose.rkt ni le niveau de langage « Etudiant avancé »**, et travaillerons exclusivement dans des « modules ».*

Exercice 1.1 **a)** On s'intéresse aux expressions arithmétiques Scheme n-aires sur les quatre opérations `+ - * /`. Ces expressions ne contiendront que des constantes. Par exemple `(+ (* 2 3 4) 10 (- 6 (/ 8 2 2) 5))` dont la valeur est `33`. Pour rester propre, nous programmons un petit type abstrait du pauvre sur ces expressions dont une grammaire est donnée à gauche :

`<expr>  ::= NOMBRE | <noeud>`
`<noeud> ::= (<op> <expr> <expr> ...)`
`<op>    ::= +  |  -  |  *  |  /`
	`(define noeud list)`
	`(define (feuille? expr) (not (pair? expr)))`
	`(define operator car)`
	`(define arguments cdr)`

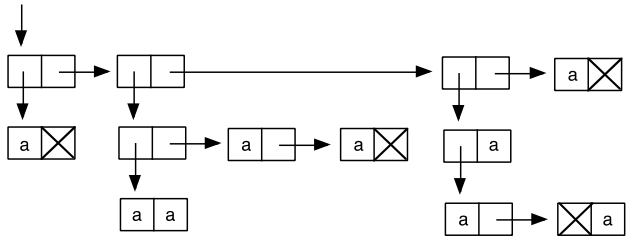
Un noeud est donc vu comme la donnée d'un opérateur à la racine et de la *liste des fils* (une liste des arguments de gauche à droite). Sauvegardez cet ADT de 4 lignes dans un module `adt-expr.rkt`. Attention, vous devez être dans le langage « déterminé par le source » débutant par `#lang racket`. N'oubliez pas le *provide*.

b) Fermez votre fichier `adt-expr.rkt`, et ouvrez un nouveau fichier `tp1.rkt`. Ce fichier sera aussi un module. Dans ce fichier, vous allez *requérir* le type abstrait `adt-expr`, et programmer une fonction `(valeur expr)` prenant une expression arithmétique et retournant sa valeur. Exemple : `(valeur '(+ (* 2 3 4) 10 (- 6 (/ 8 2 2) 5)))` `--> 33`. *Oui, je sais, je n'ai pas utilisé le constructeur de noeud, mais je ne le ferai pas dans une fonction manipulant ce type abstrait, promis juré, juste pour le test...*

• *Vous restez dans le fichier* `tp1.rkt` *pour les exercices qui suivent.*

Exercice 1.2 Avec les vraies fonctions Scheme sur les listes (`car`, `cdr`…), programmez une fonction `(beg L)` qui prend une liste `L` et dédouble tous ses éléments : `(beg '(il fait beau))` → `(il il fait fait beau beau)`. Inversement programmez `(debeg LB)` telle que `(debeg (beg L))` ≡ `L`.

Exercice 1.3 **a)** Linéarisez le diagramme ci-dessous de ***boîtes et pointeurs*** en une expression parenthésée. On prendra bien soin de livrer *l'expression unique totalement simplifiée* [sans point ou parenthèse superflus] :



b) Réciproquement, quel est l'unique diagramme de « boîtes et pointeurs » représentant l'implantation en mémoire de la liste :

`(((a b) . c) (d ((e . f))) g () h)`

*Au fait, à quoi voyez-vous que c'est une **liste** [dans l'écriture parenthésée ET dans le diagramme de boîtes et pointeurs] ? Il existe un endroit bien précis dans le dessin où cela se voit tout de suite...*

Exercice 1.4 **a)** Ecrire une fonction `log-base` d'arité 1 ou 2. Avec un argument, `(log-base x)` retourne le logarithme népérien usuel de `x`, tandis que `(log-base x a)` retourne le logarithme $\log_a(x)$ de `x` en base `a`. Maths : $\log_a(x) = \ln(x)/\ln(a)$. *Il vous faudra donc tester le nombre d'arguments.*

`(log-base 10)` → `2.3025...` `(log-base 10 2)` → `3.3219...`

b) *Plus difficile*. Programmez une fonction `(minimax L1 L2 …)` prenant un nombre quelconque de listes `L1`, `L2`… et retournant le plus petit maximum de toutes ces listes. Exemple :

`(minimax '(5 2 6 8 1) '(3 2 7 0) '(6 2 9 10 4))` → `7`

Exercice 1.5 Programmez la **macro** `(show expr)` prenant une expression SCHEME `expr`, et jouant le rôle de la fonction `show` de l'ancien teachpack `valrose.rkt`. Lors de l'exécution de `(show (+ 2 3))` dans l'éditeur, on souhaite voir au toplevel :

```
? (+ 2 3)
--> 5
```

• *Vous verrez parfois dans les docs de Racket des* ***crochets*** `[` *et* `]` *à la place de certaines parenthèses, souvent pour dénoter des* ***couples****. Ils ne sont pas obligatoires, on peut mettre des parenthèses à la place, c'est juste une histoire de style chez certains schemeurs. Vérifiez dans les docs de* `let` *ou de* `cond` *par exemple...*

```
(let ([x 2) [y 3])
  (+ x y 10))
```

```
(cond [(> x 0) (* x 2)]
      [(< x 0) (/ x 2)]
      [else 8])
```

```
(for ([i (in-range 10 20)])
  (printf "~a " i))
```

Exercice 1.6 **IMPORTANT.** La **boucle** `for` est disponible en Racket, dans diverses formes s'inspirant plus ou moins de celle de Python[1]. Elles est bien entendu programmée en Racket sous la forme d'une macro. Sans l'utiliser, programmez-en une version réduite nommée `For` avec un `F` majuscule. Vous allez uniquement simuler les deux formes possibles ci-dessous :

```
> (for ([i (in-range 10 20)])                              ; for i in range(10,20) :
    (printf "~a " i))                                      ;     print(i,sep=' ',end='')
10 11 12 13 14 15 16 17 18 19
> (for ([i (in-range 10 20)] #:when (= 0 (modulo i 3)))    ; en Python ????
    (printf "~a " i))
12 15 18
```

```
(define-syntax For                    ; avec un F majuscule pour ne pas confondre !
   (syntax-rules (in-range when)      ; mots réservés (#:when est réservé à Racket)
      ((For ([x (in-range a b)]) e ...)           (<une construction équivalente>))
      ((For ([x (in-range a b)] when test) e ...) (<une construction équivalente>))))
```

N.B. Les ***compréhensions de listes*** *de Python sont des boucles* `for/list` *en Racket :*

```
> (for/list ([i '(4 2 5 1 3)] #:when (odd? i))    ; [i*i for i in [4,2,5,1,3] if i%2==1]
    (sqr i))
(25 1 9)
```

[1] C'est un peu la philosophie de Scheme. Si une construction linguistique d'un autre langage est intéressante, il est de bon ton de l'incorporer à Racket. Oui, mais en la programmant en Scheme (*Racket is a "programmable programming language"*)...

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.

POUR LES TP ULTERIEURS :

Il était bien pratique, ce teachpack `valrose.rkt`, au sens où nos petits utilitaires de programmation [et surtout les types abstraits] étaient automatiquement chargés dès l'entrée dans Scheme. Vous allez le remplacer par un module `utils.rkt` qui contiendra au fur et à mesure toutes les fonctions ou macros qui vous sembleront **ré-utilisables**. Aujourd'hui, vous allez le créer avec seulement la macro `show`.

Les deux premières lignes de votre prochain fichier `tp2.rkt` seront :

```
#lang racket
;;; fichier tp2.rkt
(require "utils.rkt")              ; qui sera dans le même répertoire, c'est plus simple
```

Vous pourrez alors tester votre fonction `foo` préférée comme avant avec `(show (foo 5))` !… Mais n'oubliez pas de rajouter dans `utils.rkt` tout ce qui vous semblera intéressant au fur et à mesure des séances.