Correction TD n°1

Les nombres dans Dr Scheme

I Différents types de nombres

```
> 5
5
> -5
-5
> pi
#i3.141592653589793
> 2/3
0.6
> 2/3
2/3
> 6/9
0.6
> 6/9
2/3
> 17/3
5.6
> 17/3
17/3
> 17/3
5 2/3
> (exact->inexact 6/9)
#i0.6666666666666666
> (exact->inexact 17/3)
#i5.666666666666667
```

Ecriture préfixée, infixée, postfixée

II Ecrire en notation usuelle les expressions arithmétiques SCHEME suivantes :

En utilisant une fonction (à écrire plus tard !) qui transforme une expression préfixée en expression infixée

```
a)          >>>(pre->in2 '( + 5 (- 8 x) (* x y)))
            (5 + (8 - x) + (x * y))
b)          >>> (pre->in2 '(- (* 2 a) (* b b)))
            ((2 * a) - (b * b))
c)          >>> (pre->in2 '(* (- (/ a b) c) d))
            (((a / b) - c) * d)
d)          >>> (pre->in2 '(/ ( * a (- b 1) c) (+ d 2)))
            ((a * (b - 1) * c) / (d + 2))
```

III Ecrire en notation SCHEME les expressions suivantes :

En utilisant une fonction (à écrire plus tard !) qui transforme une expression infixée en expression préfixée

```
a)          >>>(in->pre2 '(x - y - z))
            (- x (- y z))
b)          >>>(in->pre2 '(x - (y - z)))
            (- x (- y z))
c)          >>> (in->pre2 '((2 * u) + v))
            (+ (* 2 u) v)
d)          >>> (in->pre2 '(2 * ((x + 1) f (g y))))
            (* 2 (f (+ x 1) (g y)))
e)          >>> (in->pre2 '((b * b) - (4 * a * c)))
            (- (* b b) (* 4 (* a c)))
```

```
f)      >>> (in->pre2 '(1 + 2 + 3 + 4 + 5))
        (+ 1  2  3  4  5) ; fonction n_aire
g)      >>> (in->pre2 '(5  *  4  *  3  *  2  *  1))
        (* 5  4  3  2  1) ; fonction n_aire
h)      >>> (in->pre2 '(sin (a *  x + b)))
        (sin (+  (*  a  x) b))
i)      >>> (in->pre2 '(tan ( (a *  x + b)/(a * x **2 + b * x + c))))
        (tan (/(+ (*  a  x) b) (+ (+ (* a (** x 2)) (* b x)) c)))
```

Correction TD n°2

Variable et valeur de variable, affectation, échange de valeurs

I Echange de valeur de variables

Faire tout d'abord cet exercice sur papier !

Soient les intialisations des deux variables x et y (← *pour le caractère Entrée*)
(define x 9) ← (define y 5) ←

Que répond Scheme si on tape ?

Saisie dans l'évaluateur	x	y	z
Réponse ?	9	5	Variable indéfinie

On saisit ensuite :
(define x 12) ← (define y 10) ← (define z x) ← (set! x y) ← (set! y z) ←

Que répond ensuite Scheme si on tape ?

Saisie dans l'évaluateur	x	y	z
Réponse ?	10	12	12

Qu'a-t-on fait dans cette succession d'affectations? Echange de valeurs de variables

Cliquer sur ***Exécute***r
On saisit ensuite :
(define x 11) ← (define y 7) ←
puis (set! x (+ x y)) ← (set! y (- x y)) ← (set! x (- x y)) ←

Que répond ensuite Scheme si on tape ?

Saisie dans l'évaluateur	x	y	z
Réponse ?	7	11	Variable indéfinie

Conclusion ?

Quelle est la limitation de cette méthode ? manipuler des variables qui sont des nombres (pour pouvoir addition et soustraction !)

Définitions de fonction
II Définir en SCHEME les fonctions suivantes :

```
; Exercice 2
; vus en cours
(define (carre x)
  (* x x))

(define (moycar x y)
  (/ (+ (carre x) (carre y)) 2))

(define (prix-ttc pht taux)
  (* pht (+ 1 (/ taux 100))))

; rappel  1$ = 1.35 €
(define (convert_euro->dollar x)
  (/ x 1.35))

(define (convert_dollar->euro x)
  (* x 1.35))
; que l'on peut améliorer en passant le taux de change en argument avec
; 1$ = taux 1,35
(define (convert_euro->dollar2 x taux)
  (/ x taux))

(define (convert_dollar->euro2 x taux)
  (* x taux))

(define (trinome a b c x)
  (+ (* a (carre x)) (* b x) c))

(define (surface_cercle r)
  (* pi (carre r)))

(define (volume_sphere r)
  (* 4/3 pi (* r r r)))

(define (volume_cylindre r h)
  (* (surface_cercle r) h))

(define (hypothenuse a b)
  (sqrt (+ (carre a) (carre b))))

(define (rectangle? a b c)
  (= c (hypothenuse a b)))
```

(*on admettra que c > a et c > b*)

A propos de la représentation des nombres

IV Que répond SCHEME si l'on essaie d'évaluer les expressions suivantes :

```
Langage: Etudiant niveau avancé; memory limit: 128 MB.
> (= 0.7 (* 7  0.1))
true
> (= 0.7 (* 7.0  0.1))
true
> (= (/ 0.7 7) 0.1)
true
> (+ 0.1 0.1 0.1 0.1 0.1 0.1 0.1)
0.7
> (inexact->exact (+ 0.1 0.1 0.1 0.1 0.1 0.1 0.1))
7/10
```

Correction TD n°3

Opérations sur les booléens

I Evaluation

```
> (define A #t)
> (define B true)
> (define C true)
> (define D false)
> (define E false)
> (define e false)
e: this name was defined previously and cannot be re-defined
; il faudrait donc redéfinir e  par set! et pas define
> (and A (or B E))
true
> (or E (and (not E) A C))
true
> (not (or (not A) (not B)))
true
> (and (or A E) (not (or B E)))
False
```

II Evaluation de AND et OR dans DrScheme

Dans chaque cas, essayer de prévoir la réponse de DrScheme avant de taper les expressions suivantes

Tout d'abord l'évaluation de (zero ? (/ 5 0)) *entraîne un message d'erreur car on ne peut diviser par zéro*

a) évaluation de OR

(or (zero ? (/ 5 0)) (< 2 3)) ; *Dr Scheme retourne le message d'erreur*

(or (< 2 3)(zero ? (/ 5 0))) ; *DrScheme retourne #t .....et (zero ? (/ 5 0)) n'est pas évalué*

Conclusion : OR est évalué de la gauche vers la droite et il retourne #t dès qu'un argument est #t

b) évaluation de AND

(and (zero ? (/ 5 0)) (< 2 3)) ; *Dr Scheme retourne le message d'erreur*

(and (< 2 3)(zero ? (/ 5 0))) ; *DrScheme retourne #f .....et (zero ? (/ 5 0)) n'est pas évalué*

Conclusion : and est évalué de la gauche vers la droite et il retourne #f dès qu'un argument est #f

III Evaluation des opérateurs booléens d'inégalité

(*ne pas oublier lors de la nouvelle affectation de x d'utiliser set! et pas define*)

	x = 4	x = 2	x = 7/2
(> x 3)	#t	#f	#t
(and (> x 3) (< x 4))	#f	#f	#t
(= (* x x) (* 2 x))	#f	#t	#f

Fonctions utilisant les opérateurs booléens

IV Ecrire les fonctions correspondant aux tests suivants :

a)
```
(define (intervalle1? n) ;Bien mettre le point d'interrogation
     (and (< 5 n) (< n 6)))
```

b)
```
(define (intervalle2? n)
     (or (intervalle1? n) (>= n 10)))
```

Les fonctions IF et COND

V Passage de IF à COND

Ecrire un analogue des expressions suivantes :

a) avec COND

```
(if (zero? x) (+ y 1) (* y 2))          est équivalent à :

       (cond ((zero? x) (+ y 1))
             (else (* y 2)))
(if (zero? x) y (if (odd? y) (- y 1) y)) est équivalent à :

       (cond ((zero? x) y)
             ((odd? y) (- y 1))
             (else  y))
```

b) avec IF

```
            (cond
                 ((> a 0) (+ a 1))
                 ((zero? a) b)
                 (else (+ a b)))          est équivalent à :
       (if (zero? a) b (if (> a 0) (+ a 1) (+ a b)))
```

VI AND, OR, NOT avec IF

```
(and p q)    ≡    (if p q #f)  ou  (if p q p)
(or p q)     ≡    (if p #t q)  ou  (if p p q)
(not p)      ≡    (if p #f #t)
```

VII Que pensez vous de la fonction suivante ?

```
            (cond
                 ((> a 0) (+ a 2))
                 ((> a 2) (* a 2))
                 (else (- a 1)))
```

La deuxième clause n'est JAMAIS évaluée puisque si `(> a 2) ≡ #t` a fortiori `(> a 0) ≡ #t`

Donc attention à l'écriture de l'ordre des clauses dans un COND !

VIII Sur le trinôme $ax^2 + bx + c$:

```
1) (define (racine_trinome? a b c x)
   ; N x N x N x N => Booleen
   (zero? (trinome a b c x)))

 2)(define (racine? a b c)
   ; N x N x N  => Booleen
   (>= (- (carre b) (* 4 a c)) 0))

 3)(define (racine a b c)
   ; N x N x N  => N x N  ou N
   (let ((delta (- (carre b) (* 4 a c))))
     (cond ((negative? delta)  "pas de racine")
           ((zero? delta) (/ (- b) (* 2 a)))
           (else (cons (- (/ (- b) (* 2 a)) (sqrt delta))
                       (+ (/ (- b) (* 2 a)) (sqrt delta)) )))))
```

4) Soient a et b les valeurs des deux côtés du rectangle : on doit avoir $a + b = L/2$ et $a \,.\, b = A$ donc a et b doivent être racines du trinôme $x^2 - (L/2)\,x + A = 0$

a) Pour L = 150 m et A = 1435 m^2, le trinôme n'a pas de racine - `(racine? 1 -75 1435) retourne #f` - donc madame DrScheme s'est manifestement trompée !

b) Par contre pour L = 152 m et A = 1435 m^2, le trinôme a deux racines :

```
(racine? 1 -76 1435) retourne #t
et
> (racine 1 -76 1435)
(35 . 41)
```

Correction TD n°4

Algorithmique : de la récursivité

```
; TD 4
; quelques fonctions utiles
(define (carre n)
  (* n n))
(define (cube n)
  (* n n n))

; exercice 1
(define (mul x y)
  (if (zero? y)
      0
      (+ x (mul x (sub1 y)))))
(define (puis x n)
  (if (zero? n)
      1
      (* x (puis x (sub1 n)))))
Rem : la fonction sub1 est la décrémentation  équivalente à  1-
On pourrait définir la fonction 1-  par  (define (1- x) (- x 1))
; exercice 2
(define (somme n)
  (if (zero? n)
      0
      (+ n (somme (sub1 n)))))
(define (somme_carre n)
  (if (zero? n)
      0
      (+ (carre n) (somme_carre (sub1 n)))))

(define (carre_somme n)
  ; il faut remarquer que Sn - Sn-1 = n * n * n (différence de deux carrés)
  (if (zero? n)
      0
      (+ (cube n) (carre_somme (sub1 n)))))
```

```
(define (somme_gene f n)
  (if (zero? n)
      0
      (+ (f n) (somme_gene f (sub1 n)))))

; la fonction Leibniz
; la fonction de calcul intermédiaire
(define (f n) (/ 1 (+ (* 2 n) 1)))

;première version
(define (leibniz n)
 (cond ((zero? n) 1)
       ((even? n) (+ (f n) (leibniz (- n 1))))
       (else (+ (- (f n)) (leibniz (- n 1))))))

; deuxième façon de procéder, en faisant gérer le signe par la fonction f
(define (f2 n)
 (if (even? n) (/ 1 (+ (* 2 n) 1)) (- (/ 1 (+ (* 2 n) 1)))))

(define (leibniz2 n)
 (if (zero? n) 1 (+ (f2 n) (leibniz2 (- n 1)))))

; troisième version : gestion du signe dans leibniz
(define (leibniz3 n)
 (cond ((zero? n) 1)
       (else ((if (even? n) + -) (leibniz3 (- n 1)) (f n)))))

; fonction somme_racine
(define (somme_racine_partielle p n)
 (cond ((= p n) (sqrt n))
      (else (sqrt (+ p (somme_racine_partielle (+ p 1) n))))))

(define (somme_racine n)
  (somme_racine_partielle 1 n))
```

Correction TD n°5

Algorithmique : de la dichotomie

```
(require (lib "trace.ss"))
; TD 5
; les fonctions de base :
(define (1+   x) (+ x 1))
(define (1-   x) (- x 1))     ; equivalent à sub1
 ; pour la dichotomie
(define (mul2 x) (* 2 x))

(define (div2 x) (quotient x 2)) , attention, il s'agit de la division
entière

(define (carre x)
  (* x x))

; exercice 1
(define (mul x y)
  (if (zero? y)
      0
      (+ x (mul x (sub1 y)))))

;version rigoureusement conforme à l'algorithme
```

```
(define (mul_dicho x y)      ; x,y entiers naturels
  (cond ((zero? y) 0)
        ((even? y) (mul_dicho (mul2 x) (div2 y )))
        (else (+ x (mul_dicho x (sub1 y ))))))

; une version qui fait gagner les appels récursifs quand y est impair
(define (mul_dicho2 x y)      ; x,y entiers naturels
  (cond ((zero? y) 0)
        ((even? y) (mul_dicho2 (mul2 x) (div2 y )))
        (else (+ x (mul_dicho2 (mul2 x) (div2 y ))))))

(define (mul_dicho3 x y)     ; x,y entiers naturels
  (if (zero? y)
      0          ; le cas de base sur lequel le calcul converge
      (+ (if (even? y) 0 x)
         (mul_dicho3 (mul2 x) (div2 y )))))
```

; Bien comprendre le passage entre la version mul_dicho, à mul_dicho2 puis mul_dicho3 qui correspond non pas à une modification de l'algorithme (c'est le même dans les trois cas !) mais à une simplification de l'écriture

```
;exercice 2
(define (puis x n)
  (if (zero? n)
      1
      (* x (puis x (sub1 n)))))

;rigoureusement la même démarche que pour mul =>_dicho
(define (puis_dicho x n)      ; n entier naturel
  (cond ((zero? n) 1)
        ((even? n) (puis_dicho (carre x) (div2 n )))
        (else (* x (puis_dicho x          (sub1 n ))))))

(define (puis_dicho2 x n)
  (cond ((zero? n) 1)
        ((even? n) (puis_dicho2 (carre x) (div2 n )))
        (else (* x (puis_dicho2 (carre x) (div2 n ))))))

(define (puis_dicho3 x n)      ; n entier naturel
  (if (zero? n)
      1          ; le cas de base sur lequel le calcul converge
      (* (if (even? n) 1 x)
         (puis_dicho3 (carre x) (div2 n)))))
```

; la même fonction qui travaille aussi bien pour la multiplication ou la puissance dichotomique … puisque ces deux fonctions (voir ci-dessus) ont exactement la même structure, seule la fonction enveloppante (+ pour la multiplication, * pour la puissance) de l'appel récursif change…

```
(define (travaille x y op elem_neutre)
; la function enveloppante est passée en argument : op
  (if (zero? y)
      elem_neutre
      (op (if (even? y) elem_neutre x)
         (travaille (op x x) (div2 y) op elem_neutre))))

(define (mul_dicho4 x y)
  (travaille x y + 0))
(define (puis_dicho4 x y)
  (travaille x y * 1))

;exercice 3
(define (somme n)
  (if (zero? n)
      0
      (+ n (somme (sub1 n)))))

(define (somme_dicho n)
  (if (zero? n)
      0
      (+ (mul2 (somme_dicho (div2 n)))
         (somme_impair (if (even? n)(sub1 n) n)))))

(define (somme_impair n)
  ; n est nécessairement impair
  (if (= n 1)
      1
      (+ n (somme_impair (sub1 (sub1 n))))))
```