```
;*****************************************************************************
; ceci n'est pas LA correction mais UNE correction du projet sur la numération *
;*****************************************************************************
;; Des utilitaires generaux

(require (lib "trace.ss"))
; permet de tracer une fonction

 (define (error reason . args)
 ; permet d'afficher un message d'erreur et d'arreter une evaluation
      (display "Error: ")
      (display reason)
      (for-each (lambda (arg)
                  (display " ")
            (write arg))
                    args)
  (newline)
 (scheme-report-environment 5))

;;;;;;;;;;;;;;;;;;;
; PROJET NUMERATION;
;;;;;;;;;;;;;;;;;;;
; quelle que soit la base, un nombre est ecrit sous la forme (base (.......))

; la conversion de l'ecriture normale d'un nombre
; vers la structure de donnÃ©e proposee par le cahier des charges (base = 10)

; version recursive non terminale
(define (convert_ecriture nb)
  ; la fonction qui calcule la decomposition en definition locale
 (define (calcul nb)
    (if (< nb 10)
        (list nb)
        (cons (remainder nb 10)
              (calcul (quotient nb 10)))))
   (list 10 (reverse (calcul nb))))

; version recursive  terminale
(define (convert_ecriture2 nb)
  ; la fonction qui calcule la decomposition en definition locale
 (define (calcul nb acc)
    (if (< nb 10)
        (list 10 (cons nb acc))
       (calcul (quotient nb 10)
               (cons (remainder nb 10) acc))))
   (calcul nb '()))
```

```
; la conversion inverse
(define (convert_inverse l)
  (define (calcul nb acc)
    (if (null? nb)
        acc
        (calcul (cdr nb)
                (+ (* 10 acc) (car nb)))))
  (calcul (cadr l) 0))

; conversion de la base 10 vers la base b
; version recursive non terminale
; pour b < 10
(define (conv10tob nb b)
  (define (calcul nb)
    (if (< nb b)
        (list nb)
        (cons (remainder nb b)
              (calcul (quotient nb b)))))
  (list b (reverse (calcul nb))))


; version recursive terminale
(define (conv10tob2 nb b)
  (define (calcul nb acc)
    (if (< nb b)
        (list b (cons nb acc))
       (calcul (quotient nb b)
               (cons (remainder nb b) acc))))
   (calcul nb '()))

; conversion de la base b (b <= 10) vers la base 10
; version recursive terminale
(define (convbto10 l)
  (let ((base (car l)) (nb (cadr l)))
    (define (calcul nb acc)
      (if (null? nb)
          acc
          (calcul (cdr nb)
                  (+ (* base acc) (car nb)))))
  (calcul nb 0)))
```

```
; versions ameliorees pour etudier des bases superieures a  10 (jusque 16)
; conversion base 10 vers base b
(define (conv10tob_gene nb b)
  (define (calcul nb acc)
    (let ((alphabet_10tob '((10 A) (11 B)(12  C)(13  D) (14  E) (15  F))))
     (if (< nb b)
           (list b (cons (if (< nb 10)
                             nb
                             (cadr (assoc nb alphabet_10tob)))
                         acc))
           (calcul (quotient nb b)
                   (cons (let ((chiffre (remainder nb b)))
                           (if (< chiffre 10)
                               chiffre
                               (cadr (assoc chiffre alphabet_10tob))))
                         acc)))))
  (calcul nb '()))

; conversion base b vers base 10
(define (convbto10_gene l)
  (let ((base (car l))
         (nb (cadr l))
         (alphabet_bto10 '((A 10) (B 11)(C 12)(D 13) (E 14) (F 15))))
  (define (calcul nb acc)
     (if (null? nb)
           acc
           (calcul (cdr nb)
                   (+ (* base acc)
                      (if (number? (car nb))
                          (car nb)
                          (cadr (assoc (car nb) alphabet_bto10)))))))
  (calcul nb 0)))

; les operations en base b (addition soustraction
; un utilitaire qui a  partir de deux listes de longueurs differentes, ajuste
la plus courte a  la longueur de l'autre en ajoutant des zeros a  droite
; le resultat est une liste dont le car est la premiere liste, le cdr la
deuxieme
(define (meme_longueur l1 l2)
  (let ((long1 (length l1))
        (long2 (length l2)))
    (define (construit long)  ; construit une liste de longueur l contenant
des 0
      (if (zero? long)
          '()
          (cons 0 (construit (- long 1)))))
    (cond ((= long1 long2) (cons l1 l2))
          ((< long1 long2) (cons (append l1 (construit (- long2 long1))) l2))
          (else (cons l1 (append l2 (construit (- long1 long2))))))))
```

```
; L'ADDITION

; d'abord un utilitaire qui additionne trois chiffres dans la base b
; version TOP
(define (add_chiffre c1 c2 ret b)
  ; retourne une liste dans laquelle le car est la somme et le cadr la
retenue
  (let* ((alphabet_10tob '((10 A) (11 B)(12  C)(13  D) (14  E) (15  F)))
         (alphabet_bto10 '((A 10) (B 11)(C 12)(D 13) (E 14) (F 15)))
         (somme (+ ret
                   (if (number? c1) c1 (cadr (assoc c1 alphabet_bto10)))
                   (if (number? c2) c2 (cadr (assoc c2 alphabet_bto10))))))
  ; le let*  permet que somme soit evaluee alors que alphabet... a etÃ©
evalue©  -evaluations successives suivant l'ordre dans le let*-
    (if (< somme b)
        (cons (if (< somme 10) somme (cadr (assoc somme alphabet_10tob)))
(list 0))
        (cons (if (< (- somme b) 10) (- somme b) (cadr (assoc (- somme b)
alphabet_10tob))) (list 1)))))

; version VULGUM PECUS
; Remarque : au lieu d'avoir une let* recursif pour l'association chiffre<=>
digit
; on peut ecrire deux fonctions qui se charent de ce travail suivant le sens
de la conversion
; suivant l'idee de l'enonce avec une fonction ASSOCIE

(define (associe_to10 digit)
  (let ((alphabet_bto10 '((A 10) (B 11)(C 12)(D 13) (E 14) (F 15))))
    (if (number? digit) digit (cadr (assoc digit alphabet_bto10)))))
(define (associe_tob digit)
  (let ((alphabet_10tob '((10 A) (11 B)(12  C)(13  D) (14  E) (15  F))))
    (if (< digit 10) digit (cadr (assoc digit alphabet_10tob)))))

(define (add_chiffre2 c1 c2 ret b)
(let ((somme (+ ret (associe_to10 c1) (associe_to10 c2))))
    (if (< somme b)
        (cons (if (< somme 10) somme (associe_tob somme)) (list 0))
        (cons (if (< (- somme b) 10) (- somme b) (associe_tob (- somme b)))
(list 1)))))

; les choses serieuses : l'addition de deux nombres en base b
```

```
(define (add n1 n2)
  (let* ((base1 (car n1))
         (base2 (car n2))
         (pair_oper (meme_longueur (reverse (cadr n1)) (reverse (cadr n2))))
         ; met les deux listes a la meme longeur en mettant les unites A
GAUCHE
         (oper1 (car pair_oper))
         (oper2 (cdr pair_oper)))
    (define (calcul op1 op2 retenue res)
      (if (and (null? op1) (null? op2))
          (if (zero? retenue) res (append res (list 1)))
          (let ((add_digit (add_chiffre (car op1) (car op2) retenue base1)))
            (calcul (cdr op1) (cdr op2) (cadr add_digit) (append res (list
(car add_digit)))))))
    (if (not (= base1 base2))
        (error " les deux nombres ne sont pas dans la mÃªme base !")
        (list base1 (reverse (calcul oper1 oper2 0 '()))))))

; LA SOUSTRACTION

; quelques utilitaires
; la complementation Ã  b
(define (complement_b_1 L base)
  ; transforme chaque chiffre de la liste L en son complement a  (b - 1)
  (define (bosse L res) ; version directement recursive terminale
    (if (null? L)
        res
        (let ((diff (- (- base 1) (associe_to10 (car L)))))
          (bosse (cdr L) (append (list (associe_tob diff)) res)))))
  (reverse (bosse L '())))

 (define (complement_b L)
   ;determine le complement a  la base b de L
   ; soit le complement a  (b - 1)   + 1
   (let* ((base (car L))
          (pair_oper (meme_longueur (list 1) (reverse (complement_b_1 (cadr L)
base))))
          ; met les deux listes Ã  la meme longueur en mettant les unites A
GAUCHE
          (oper1 (car pair_oper))
          (oper2 (cdr pair_oper)))
     (define (calcul op1 op2 retenue res)
       (if (and (null? op1) (null? op2))
           res
           (let ((add_digit (add_chiffre (car op1) (car op2) retenue base)))
            (calcul (cdr op1) (cdr op2) (cadr add_digit) (append res (list
(car add_digit)))))))
     (list base (reverse (calcul oper1 oper2 0 '())))))

 ; et en fin la soustraction.... qui est une addition !
```

```
 (define (sous n1 n2)
   (let* ((pair_oper (meme_longueur (reverse (cadr n1)) (reverse (cadr n2))))
          ; met les deux listes a  la meme longueur en mettant les unites A
GAUCHE
          (res (add (list (car n1) (reverse (car pair_oper)))
                    (complement_b (list (car n2) (reverse (cdr
pair_oper)))))))
          ; il faut "nettoyer" les chiffres en tete du resultat de l'addition
si necessaire (voir l'enonce du projet)
     (if (<= (length (cadr res)) (max (length (cadr n1)) (length (cadr n2))))
              res
              (list (car res) (enleve_zero (cdadr res))))))

 ; un utilitaire pour enlever les "zeros a  gauche"
 (define (enleve_zero L)
   (if (and (number? (car L)) (zero? (car L)) (not(null? (cdr L))))
       (enleve_zero (cdr L))
       L))

 ; la multiplication  .... qui utilise add et sous
 (define (mul l1 l2)
  (define (bosse l1 l2 acc)
    (if (AND (number? (caadr l2))(AND (null? (cdadr l2))(= 0 (caadr l2))))
        acc
        (bosse l1 (sous l2 (list (car l1)'(1)))(add acc l1))))
  (bosse l1 l2 (list (car l1)'(0))))

 (define (div l l2)   ; ne fonctionne que pour des nombres divisibles
  (define (bosse l l2 acc)
    (if (AND (number? (caadr l))(AND (null? (cdadr l))(= 0(caadr l))))
         acc
         (bosse (sous l l2) l2 (add acc (list (car l)'(1))))))
  (bosse l l2 (list (car l)'(0))))
```