1 Exercice 1

1.1 filtre

```
(define filtre
  (lambda (L P)
    (if (null? L)
  '()
  (if (P (car L))
      (cons (car L) (filtre (cdr L) P))
      (filtre (cdr L) P)))))

(filtre '(1 -3 8 -6 5 -9) (lambda (x) (>= x 0)))
```

```
;Value: (1 8 5)
```

1.2 Suppression des doublons

```
(define supprimerDoublon
  (lambda (L)
    (if (null? L)
  '()
  (cons (car L)
        (supprimerDoublon
         (filtre (cdr L) (lambda (x) (not (= x (car L))))))))))

(supprimerDoublon '( 1 2 3 4 1 5 2 6 3 7 6 8 3 9))
```

```
;Value: (1 2 3 4 5 6 7 8 9)
```

1.3 Image

ATTENTION: un ensemble ne contient pas de doublons.

```
(define image
  (lambda (D f)
    (supprimerDoublon (map f D))))

(image '(1 2 3 4 5 6 7 8 9) (lambda (x) (* 2 x)))
(image '(-4 -3 -2 -1 0 1 2 3 4) (lambda (x) (* x x)))
```

```
;Value: (2 4 6 8 10 12 14 16 18)
;Value: (16 9 4 1 0)
```

1.4 Avec schema récursif

IMPORTANT: il faut obligatoirement savoir refaire les schémas récursifs du cours, il y a une question dessus chaque année.

```
(define SR
  (lambda (L I C)
    (if (null? L)
  I
  (C (car L) (SR (cdr L) I C)))))

(define image2
  (lambda (D f)
    (SR D '() (lambda (x l)
    (cons (f x) (filtre l (lambda (y) (not (= y (f x))))))))))

(image2 '(1 2 3 4 5 6 7 8 9) (lambda (x) (* 2 x)))
(image2 '(-4 -3 -2 -1 0 1 2 3 4) (lambda (x) (* x x)))
```

```
;Value: (2 4 6 8 10 12 14 16 18)
;Value: (16 9 4 1 0)
```

2 Exercice 2

2.1 Part

Il faut distinguer 3 cas:

- n = 0 => retourner la liste vide
- n = k => on ne peut plus mettre de 0
- n > k => on peut mettre des 0 **et** des 1

```
(define part
  (lambda (k n)
    (if (= n 0)
  '(())
  (if (= n k)
      (map (lambda (x) (cons 1 x)) (part (- k 1) (- n 1)))
      (if (= k 0)
    (map (lambda (x) (cons 0 x)) (part 0 (- n 1)))
    (append (map (lambda (x) (cons 0 x)) (part k (- n 1)))
      (map (lambda (x) (cons 1 x)) (part (- k 1) (- n 1)))))))))

(part 2 4)
```

```
;Value: ((0 0 1 1) (0 1 0 1) (0 1 1 0) (1 0 0 1) (1 0 1 0) (1 1 0 0))
```

2.2 MP

2.2.1 Solution 1

```
(define MP
  (lambda (k n)
    (if (= k 0)
  (part 0 n)
  (append (MP (- k 1) n) (part k n)))))

(MP 2 4)
```

```
;Value: ((0 0 0 0) (0 0 0 1) (0 0 1 0) (0 1 0 0) (1 0 0 0) (0 0 1 1)
        (0 1 0 1) (0 1 1 0) (1 0 0 1) (1 0 1 0) (1 1 0 0))
```

2.2.2 Solution 2

```
(define MP2
  (lambda (k n)
    (append-map (lambda (x) (part x n)) (iota k))))

(MP2 2 4)
```

```
;Value: ((0 0 1 1) (0 1 0 1) (0 1 1 0) (1 0 0 1) (1 0 1 0) (1 1 0 0)
        (0 0 0 1) (0 0 1 0) (0 1 0 0) (1 0 0 0) (0 0 0 0))
```

2.3 DH

```
(define DH
  (lambda (L K)
    (if (null? L) ;; et null? K
  0
  (if (= (car L) (car K))
      (DH (cdr L) (cdr K))
      (+ 1 (DH (cdr L) (cdr K)))))))

(DH '(1 2 3 4) '(1 2 7 4))
```

```
;Value: 1
```

2.4 BF

2.4.1 Solution 1

```
(define taille
  (lambda (l)
    (SR l 0 (lambda (x r) (+ 1 r)))))

(define echange
  (lambda (x)
    (if (= x 0) 1 0)))

(define BF ;; on suppose que la taille de la liste est sup rieur ou egale a k
  (lambda (L k)
    (if (null? L) ;; ou k == 0
  '(())
  (if (= k (taille L))
      (list (map echange L))
      (if (> k 0)
    (append
     (map (lambda (x) (cons (echange (car L)) x)) (BF (cdr L) (- k 1)))
     (map (lambda (x) (cons (car L) x)) (BF (cdr L) k)))
    (list L))))))

(BF '(0 0 0) 0)
(BF '(0 0 0) 3)
(BF '(0 0 0) 2)
(BF '(0 0 0 0) 2)
```

```
;Value: ((0 0 0))
;Value: ((1 1 1))
;Value: ((1 1 0) (1 0 1) (0 1 1))
;Value: ((1 1 0 0) (1 0 1 0) (1 0 0 1) (0 1 1 0) (0 1 0 1) (0 0 1 1))
```

2.4.2 Solution 2

Ici, ce qu'il fallait remarquer c'est que l'on peut utiliser la fonction `part` pour générer une liste de 0 et de 1, où les 1 indiques les emplacement à modifier dans L. On demande `k` modification à `part` et il génère toutes les listes de modifications.

```
(define BF2
  (lambda (L k changer)
    (let ((mods (part k (taille L))))
      (map (lambda (x) ;; x => liste qui indique l'emplacement des modifications
       (map (lambda (y z) ;; y dans L et z dans [0,1]
        (if (= 1 z)
      (changer y)
      y))
      L x)) mods))))

(BF2 '(0 0 0) 0 (lambda (x) (if (= x 1) 0 1)))
(BF2 '(0 0 0) 3 (lambda (x) (if (= x 1) 0 1)))
(BF2 '(0 0 0) 2 (lambda (x) (if (= x 1) 0 1)))
(BF2 '(0 0 0 0) 2 (lambda (x) (if (= x 1) 0 1)))

;; marche pour tout :D
(BF2 '(A B B A) 2 (lambda (x) (if (equal? x 'A) 'B 'A)))
```

```
;Value: ((0 0 0))
;Value: ((1 1 1))
;Value: ((0 1 1) (1 0 1) (1 1 0))
;Value: ((0 0 1 1) (0 1 0 1) (0 1 1 0) (1 0 0 1) (1 0 1 0) (1 1 0 0))
;Value: ((a b a b) (a a b b) (a a a a) (b b b b) (b b a a) (b a b a))
```

3 Exercice 3

IMPORTANT: Il y a souvent des exo avec des existes et qqs !

3.1 Existe2

```
(define exist2
  (lambda (L P)
    (if (null? L)
  '()
  (if (P (car L))
      (list (car L))
      (exist2 (cdr L) P)))))

(exist2 '(11 2 3 0 9) (lambda (x) (= (modulo x 3) 1)))
(exist2 '(11 2 10 0 9) (lambda (x) (= (modulo x 3) 1)))
```

```
;Value: ()
;Value: (10)
```

3.2 Exist

```
(define exist
  (lambda (L P)
    (not (null? (exist2 L P)))))

(exist '(11 2 3 0 9) (lambda (x) (= (modulo x 3) 1)))
(exist '(11 2 10 0 9) (lambda (x) (= (modulo x 3) 1)))
```

```
;Value: #f
;Value: #t
```

3.3 QQS

```
(define qqs
  (lambda (L P)
    (not (exist L (lambda (x) (not (P x)))))))

(qqs '(11 2 3 0 9) (lambda (x) (not (= (modulo x 3) 1))))
(qqs '(11 2 10 0 9) (lambda (x) (not (= (modulo x 3) 1))))
```

```
;Value: #t
;Value: #f
```

3.4 Neutre

```
(define neutre?
  (lambda (E f)
    (if (null? E)
  '()
  (let ((r (exist2 E
       (lambda (x)
         (= (f x (car E)) (f (car E) x) x)))))
    (if (null? r)
        (neutre? (cdr E) f)
        r)))))

(neutre? '(1 2 3 4 5 6 7 8 9) (lambda (x y) (+ x y)))
(neutre? '(0 1 2 3 4 5 6 7 8 9) (lambda (x y) (+ x y)))
```

```
;Value: ()
;Value: (0)
```

3.5 Inversible

Conclusion classique de ce genre d'exo, il faut juste recopier l'énoncé en remplacer les symboles de math par les noms des fonctions.

```
(define inversible?
  (lambda (E f)
    (let ((r (neutre? E f)))
      (if (null? r)
    #f
    (qqs E (lambda (x) ;; pour tout x dans E
       (exist E (lambda (y) (= (f x y) (car r)))))))))) ;; il existe un y dans E tq
    ...

(inversible? '(0 1 2 3 4 5 6 7 8 9) (lambda (x y) (+ x y)))
```

```
;Value: #f
```