```
;;;; EXERCICE 1 ;;;;;;;;;;;;;;;;;;;;;;;;
(defun sigma (f n p)
  (if (> n p)
      0
      (+ (funcall f n)
         (sigma f (+ n 1) p))))

(defun sqr (x)
  (* x x))

(sigma (function sqr) 1 10)

(sigma #'sqr 1 10)

(sigma #'(lambda (x) (/ 1 (+ 1 (sqrt x)))) 1 10)

;;;; EXERCICE 2 ;;;;;;;;;;;;;;;;;;;;;;;;
(defun valeur-en-un (f)
  (funcall f 1))

(defun polynome (x) (+ (* 3 x x) 4.7))

(valeur-en-un #'polynome)

;;;; EXERCICE 3 ;;;;;;;;;;;;;;;;;;;;;;;;
(defun multiplicateur (n)
  #'(lambda (x) (* n x)))

(multiplicateur 3)

(funcall (multiplicateur 3) 5)

(defvar *double* (multiplicateur 2))

(funcall *double* 3)

(setf (symbol-function 'doublebis)
      (multiplicateur 2))

(doublebis 5)

;;;;à ne pas faire (car liaison inutile):
(defun multiplicateur_faux (n)
  (defun multiplicateur_n (x)
    (* n x)))

;;;; EXERCICE 4 ;;;;;;;;;;;;;;;;;;;;;;;;
(defun derivee-approx (f precision)
  #'(lambda (x)
    (/ (- (funcall f (+ x precision))
          (funcall f (- x precision)))
       (* 2 precision))))

(derivee-approx #'sin 0.001d0)
(derivee-approx #'polynome 0.001d0)

(funcall (derivee-approx #'sin 0.001d0) 1)
(funcall (derivee-approx #'polynome 0.0001d0) 1)
```



```
(defun derivee-n-approx (f ordre precision)
  (if (zerop ordre)
      (lambda (x) (funcall f x))
      (derivee-n-approx (derivee-approx f precision) (1- ordre) precision)))

(funcall (derivee-n-approx #'polynome 0 1e-5) 1)
(funcall (derivee-n-approx #'polynome 1 1e-5) 1)
(funcall (derivee-n-approx #'polynome 2 1e-5) 1)

;;;; EXERCICE 5 ;;;;;;;;;;;;;;;;;;;;;;;;
;; version avec associativité à droite
(defun op_prod (f operation valeur-initiale n p)
  (if (> n p)
      valeur-initiale
      (funcall operation
               (funcall f p)
               (op_prod f operation valeur-initiale n (1- p)))))

(defun fact (n)
  (op_prod #'(lambda (x) x) #'* 1 1 n))

(fact 6)

;;; version itérative
(defun op_prod2 (f operation valeur-initiale n p)
  (do ((p p (1- p))
       (acc valeur-initiale
            (funcall operation
                     (funcall f p)
                     acc)))
      ((> n p) acc)))

(defun fact2 (n)
  (op_prod2 #'(lambda (x) x) #'* 1 1 n))

(fact2 6)

;;; approximation du nombre e par n itérations
(defun e-approx (n)
  (accumuler #'(lambda (n) (/ 1 (fact n)))
             #'+
             0.0
             0
             n))

(e-approx 10)
;;; remarque: beaucoup de duplication de calculs , lesquels ?

;;;; somme des f(i)x^i/i! jusqu'à n
(defun serie (f x n)
  (do* ((i 0 (+ i 1))
        (terme 1 (/ (* terme  x) i))
        (acc (funcall f 0) (+ (* (funcall f i) terme) acc)))
      ((>= i n) acc)))

;;; série exponentielle
(serie #'(lambda (i) 1) 1 3)

;;; cosinus : 1 - x^2! + x^4/4! - x^6/6! ...
```

```
(defun cos-approx (x nb-pas)
  (serie #'(lambda (i)
             (cond ((oddp i) 0)
                   ((oddp (/ i 2)) -1)
                   (t 1)))
         x
         nb-pas))

(cos-approx (/ pi 4) 100)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;BONUS TRACK ;;;;;;;;;;;;;;;;;;;;;;;;
;;; itération d'une fonction jusqu'à ce qu'un critère soit satisfait
(defun iterer-jusqu-a (f predicat valeur-initiale)
  (do ((resultat valeur-initiale (funcall f resultat)))
      ((funcall predicat resultat)
       resultat)))

(defun racine-carre-par-newton (x precision)
  (iterer-jusqu-a #'(lambda (y) (/ (+ y (/ x y)) 2.0))
                  #'(lambda (y) (< (/ (abs (- x (* y y))) x) precision))
                  (/ x 2)))

(racine-carre-par-newton 2 0.1)
(racine-carre-par-newton 2 0.01)
(racine-carre-par-newton 2 1e-6)
(racine-carre-par-newton 4 1e-6)
```