```scheme
;;;;
;;;; Corrige-TD6.stk    -- Corrigé de la feuille 6
;;;;
;;;;           Author: Erick Gallesio [eg@essi.fr]
;;;;    Creation date: 28-May-2003 13:18 (eg)
;;;; Last file update: 28-May-2003 15:03 (eg)
;;;;

;==============================================================================
;                                    EXO 1
;==============================================================================
(define-macro (++ var . val)
  `(set! ,var (+ ,var ,(if (null? val) 1 (car val)))))

;==============================================================================
;                                    EXO 2
;==============================================================================
(define-macro (prog1 prems . reste)
  `(let ((tmp ,prems))
     ,@reste
     tmp))

;; Si on veut éviter les problème d'hygiène, voilà une version plus propre...
(define-macro (prog1 prems . reste)
  (let ((tmp (gensym)))
    `(let ((,tmp ,prems))
       ,@reste
       ,tmp)))

;==============================================================================
;                                    EXO 3
;==============================================================================
(define-macro (push! l v)
  `(set! ,l (cons ,v ,l)))

(define-macro (pop! l)
  `(prog1 (car ,l) (set! ,l (cdr ,l))))

#| TEST
        (define s '())
        (push! s 1)
        (push! s 2)
        (pop!  s)    ; => 2
        (pop!  s)    ; => 1
|#

;==============================================================================
;                                    EXO 4
;==============================================================================

;; while avec un let nommé
(define-macro (while expr . body)
  `(let Loop ()
     (when ,expr
       ,@body
       (Loop))))

;; Version tenant compte de l'hygiène
(define-macro (while expr . body)
  (let ((etiq (gensym "Etiquette")))
    `(let ,etiq ()
          (when ,expr
```



```scheme
            ,@body
            (,etiq)))))

;==============================================================================
;                                    EXO 5
;==============================================================================

;; For -- sans contrôle

(define-macro (for bindings . body)
  (let ((var     (car bindings))
        (depart  (cadr bindings))
        (fin     (caddr bindings))
        (pas     (cadddr bindings))
        (etiq    (gensym "Etiquette")))
    `(let ,etiq ((,var ,depart))
        (unless (= ,var ,fin)
          ,@body
          (,etiq (+ ,var ,pas))))))
#|
   (for (i 5 0 -1) (display i) (newline))
|#

;==============================================================================
;                                    EXO 6
;==============================================================================

;; fonction avec contrôle des paramètres

(define-macro (define-func args . body)
  `(define ,(car args) (lambda ,(map car (cdr args))
                         ,@(map (lambda (x)
                                  `(unless ,(cdr x)
                                     (error "~S est du mauvais type" ,(car x))))
                                (cdr args))
                         ,@body)))

#|
;; Test
(macro-expand '(define-func (puissance (x number?) (y number?))
                 (let loop ((p 1) (y y))
                   (if (> y 0)
                       (loop (* p x) (- y 1))
                       p))))

; ==>   (define puissance (lambda (x y)
;                           (unless (number?)
;                             (error "~S est du mauvais type" x))
;                           (unless (number?)
;                             (error "~S est du mauvais type" y))
;                           (let loop ((p 1) (y y))
;                             (if (> y 0)
;                                 (loop (* p x) (- y 1))
;                                 p))))

|#
;==============================================================================
;                                    EXO 7
;==============================================================================

(define *traced-functions* (make-hash-table))
```

```
(define-macro (trace f)
  `(set! ,f (trace-function ',f ,f)))

(define (trace-function name proc)
  (let ((res (let ((n 0))
               (lambda l
                 (let ((res #f))
                   (format #t "~A Appel de ~A\n" (make-string n #\.) `(, name ,@l))
                   (set! n   (+ n 3))
                   (set! res (apply proc l))
                   (set! n   (- n 3))
                   (format #t "~A Résultat: ~A\n" (make-string n #\.) res)
                   res)))))
    (hash-table-put! *traced-functions* name proc)
    res))

(define-macro (untrace f)
  `(set! ,f (untrace-function ',f ,f)))

(define (untrace-function name proc)
  (let ((res (hash-table-get *traced-functions* name #f)))
    (or res proc)))

;=============================================================================
;                                       EXO 8
;=============================================================================

;; let nommé -- et un let par la même occasion

(define-macro (my-let . body)
  (if (pair? (car body))
      ;; c'est un let standard
      (let ((bindings (car body))
            (body     (cadr body)))
        `((lambda ,(map car bindings) ,body) ,@(map cadr bindings)))
      ;; C'est un let nommé
      (let ((etiq     (car body))
            (bindings (cadr body))
            (body     (caddr body)))
        `(letrec ((,etiq (lambda ,(map car bindings)
                           ,body)))
           (,etiq ,@(map cadr bindings))))))

#|
;; Tests
(macro-expand '(my-let loop ((a 1000))
                       (when (> a 0)
                           (display a)
                           (newline)
                           (loop (- a 1)))))  ;;(letrec ((loop (lambda (a)
                                              ;;                  (when (> a 0)
                                              ;;                    (display a)
                                              ;;                    (newline)
                                              ;;                    (loop (- a 1))
))))
                                              ;;     (loop 1000))
(macro-expand '(my-let ((a 1) (b 2))
                  (cons b a)))                ;; ((lambda (a b) (cons b a)) 1 2)

|#
```