```
; Cours 05 : Les variables



;;;;;;;;;
; Macro ;
;;;;;;;;;

(define-syntax-rule (with [(v-id sto-id) call] body)
  (type-case Result call
    [(v*s v-id sto-id) body]))

;;;;;;;;;;;;;;;;;;;;;;;;
; Définition des types ;
;;;;;;;;;;;;;;;;;;;;;;;;

; Représentation des expressions
(define-type Exp
  [numE (n : Number)]
  [idE (s : Symbol)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [lamE (par : Symbol) (body : Exp)]
  [appE (fun : Exp) (arg : Exp)]
  [letE (s : Symbol) (rhs : Exp) (body : Exp)]
  [setE (s : Symbol) (val : Exp)]
  [beginE (l : Exp) (r : Exp)]
  [addressE (s : Symbol)]
  [contentE (e : Exp)]
  [set-content!E (e : Exp) (e2 : Exp)]
  [mallocE (e : Exp)]
  [freeE (e : Exp)]
  )

; Représentation des valeurs
(define-type Value
  [numV (n : Number)]
  [closV (par : Symbol) (body : Exp) (env : Env)]
  )

; Représentation du résultat d'une évaluation
(define-type Result
  [v*s (v : Value) (s : Store)])

; Représentation des liaisons
(define-type Binding
  [bind (name : Symbol) (location : Location)])

; Manipulation de l'environnement
(define-type-alias Env (Listof Binding))
(define mt-env empty)
(define extend-env cons)
```

```
; Représentation des adresses mémoire
(define-type-alias Location Number)

; Représentation d'un enregistrement
(define-type Storage
  [cell (location : Location) (val : Value)])

; Manipulation de la mémoire
(define-type Pointer
  [pointer (loc : Location) (size : Number)])
(define-type Store
  [store (storages : (Listof Storage))
         (pointers : (Listof Pointer))])

(define mt-store (store empty empty))
(define (override-store c l)
  (store (override-store2 c (store-storages l)) (store-pointers l)));(cons c (store-storages l)) (store-pointers l)))

(define (override-store2 c l)
  (if (empty? l)
      (cons c empty)
      (let ([c2 (first l)])
        (if (equal? (cell-location c) (cell-location c2))
            (cons c (rest l))
            (cons (first l) (override-store2 c (rest l)))))))

(define (override-pointers p l)
  (store (store-storages l) (cons p (store-pointers l))))

; Integer
(define (integer? n) (= n (floor n)))

;;;;;;;;;;;;;;;;;;;;;;
; Analyse syntaxique ;
;;;;;;;;;;;;;;;;;;;;;;

(define (parse [s : S-Exp]) : Exp
  (cond
    [(s-exp-match? `NUMBER s) (numE (s-exp->number s))]
    [(s-exp-match? `SYMBOL s) (idE (s-exp->symbol s))]
    [(s-exp-match? `{+ ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (plusE (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? `{* ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (multE (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? `{lambda {SYMBOL} ANY} s)
     (let ([sl (s-exp->list s)])
       (lamE (s-exp->symbol (first (s-exp->list (second sl)))) (parse (third sl))))]
    [(s-exp-match? `{let [{SYMBOL ANY}] ANY} s)
     (let ([sl (s-exp->list s)])
       (let ([subst (s-exp->list (first (s-exp->list (second sl))))])
         (letE (s-exp->symbol (first subst))
               (parse (second subst))
               (parse (third sl)))))]
    [(s-exp-match? `{set! SYMBOL ANY} s)
     (let ([sl (s-exp->list s)])
       (setE (s-exp->symbol (second sl)) (parse (third sl))))]
    [(s-exp-match? `{begin ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (beginE (parse (second sl)) (parse (third sl))))]

    [(s-exp-match? `{address SYMBOL} s)
     (let ([sl (s-exp->list s)])
       (addressE (s-exp->symbol (second sl))))]
    [(s-exp-match? `{content ANY} s)
     (let ([sl (s-exp->list s)])
       (contentE (parse (second sl))))]
    [(s-exp-match? `{set-content! ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (set-content!E (parse (second sl)) (parse (third sl))))]

    [(s-exp-match? `{malloc ANY} s)
     (let [(sl (s-exp->list s))]
       (mallocE (parse (second sl))))]
    [(s-exp-match? `{free ANY} s)
     (let [(sl (s-exp->list s))]
       (freeE (parse (second sl))))]


    [(s-exp-match? `{ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (appE (parse (first sl)) (parse (second sl))))]
    [else (error 'parse "invalid input")]))

;;;;;;;;;;;;;;;;;;
; Interprétation ;
;;;;;;;;;;;;;;;;;;

; Interpréteur
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    [(numE n) (v*s (numV n) sto)]
    [(idE s) (v*s (fetch (lookup s env) sto) sto)]
    [(plusE l r)
     (with [(v-l sto-l) (interp l env sto)]
           (with [(v-r sto-r) (interp r env sto-l)]
                 (v*s (num+ v-l v-r) sto-r)))]
    [(multE l r)
     (with [(v-l sto-l) (interp l env sto)]
           (with [(v-r sto-r) (interp r env sto-l)]
                 (v*s (num* v-l v-r) sto-r)))]
    [(lamE par body) (v*s (closV par body env) sto)]
    [(appE f arg)
     (with [(v-f sto-f) (interp f env sto)]
           (type-case Value v-f
             [(closV par body c-env)
              (type-case Exp arg
                [(idE s) (interp body
                                 (extend-env (bind par (lookup s env)) c-env)
                                 sto-f)]
                [else (with [(v-arg sto-arg) (interp arg env sto-f)]
                            (let ([l (new-loc sto-arg)])
                              (interp body
                                      (extend-env (bind par l) c-env)
                                      (override-store (cell l v-arg) sto-arg)))))])]
             [else (error 'interp "not a function")]))]
    [(letE s rhs body)
     (with [(v-rhs sto-rhs) (interp rhs env sto)]
           (let ([l (new-loc sto-rhs)])
             (interp body
                     (extend-env (bind s l) env)
                     (override-store (cell l v-rhs) sto-rhs))))]
    [(setE var val)
     (let ([l (lookup var env)])
       (with [(v-val sto-val) (interp val env sto)]
             (v*s v-val (override-store (cell l v-val) sto-val))))]
    [(beginE l r)
     (with [(v-l sto-l) (interp l env sto)]
           (interp r env sto-l))]
    [(addressE s) (v*s (numV (lookup s env)) sto)]
    [(contentE e) (content e env sto)]
    [(set-content!E e1 e2) (setcontent e1 e2 env sto)]
    [(mallocE e) (malloc e env sto)]
    [(freeE e) (free e env sto)]
    ))

; Fonctions utilitaires pour l'arithmétique
(define (free e env sto)
  (let ([debut (interp e env sto)])
    (let ([taille (findtaille (numV-n (v*s-v debut)) (store-pointers (v*s-s debut)))])
      (if (and (integer? taille) (> taille 0))
          (libere (numV-n (v*s-v debut)) (numV-n (v*s-v debut)) taille taille env (store-storages (v*s-s debut)) (store-pointers (v*s-s debut)))
          (error 'interp "not an allocated pointer")))))


(define (findtaille debut point)
  (if (empty? point)
      (error 'interp "not an allocated pointer")

      (if (equal? (pointer-loc (first point)) debut)
          (pointer-size (first point))

          (findtaille debut (rest point)))
      ))

(define (libere d debut t taille env stor point)

  (if (equal? taille 0)
      (v*s (numV 0) (store stor (supprimepoint d t point empty)))

      (libere d (+ debut 1) t (- taille 1) env (recherchesto debut stor empty) point)))

(define (recherchesto debut stor s)
  (if (empty? stor)
      (error 'interp "not an allocated pointer")

      (if (equal? (cell-location (first stor)) debut)
          (let ([fin (append s (rest stor))])
            fin)
          (recherchesto debut (rest stor) (append s (list (first stor)))))
      ))

(define (supprimepoint debut taille point p)
  (if (empty? point)
      (error 'interp "not an allocated pointer")

      (if (and (equal? (pointer-loc (first point)) debut) (equal? (pointer-size (first point)) taille))
          (let ([fin (append p (rest point))])
            fin)

          (supprimepoint debut taille (rest point) (append p (list (first point)))))
      ))

(define (malloc e env sto)
  (let ([n (interp e env sto)])

    (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))

        (allocation (numV-n (v*s-v n)) (numV-n (v*s-v n)) (new-loc sto) env sto)
        (error 'interp "not a size"))
        ))

(define (allocation n i fstloc env sto)
  (if (equal? 0 n)
      (let ([dernieresto (override-pointers (pointer fstloc i) sto)])
        (v*s (numV fstloc) dernieresto))
      (let ([newsto (override-store (cell (new-loc sto) (numV 0)) sto)])
        (allocation (- n 1) i fstloc env newsto)
        )))

(define (setcontent l e env sto)
  (let ([loc (location l env sto)])
    (if (and (integer? (numV-n (v*s-v loc))) (> (numV-n (v*s-v loc)) 0))
    (with [(v-l sto-l) (interp e env (v*s-s loc))]
          (let ([derniersto (override-store (cell (numV-n (v*s-v loc)) v-l) sto-l)])
            (v*s v-l derniersto)))
    (error 'interp "segmentation fault"))
    ))

(define (location l env sto)
  (let ([n (interp l env sto)])
    (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))
        (v*s (v*s-v n) (v*s-s n))
        (error 'interp "segmentation fault"))
    ))

(define (content e env sto)
  (let ([n (interp e env sto)])
    (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))
        (v*s (fetch (numV-n (v*s-v n)) (v*s-s n)) (v*s-s n))
        (error 'interp "segmentation fault"))
    ))


(define (num-op [op : (Number Number -> Number)]
                [l : Value] [r : Value]) : Value
  (if (and (numV? l) (numV? r))
      (numV (op (numV-n l) (numV-n r)))
      (error 'interp "not a number")))

(define (num+ [l : Value] [r : Value]) : Value
  (num-op + l r))

(define (num* [l : Value] [r : Value]) : Value
  (num-op * l r))

; Recherche d'un identificateur dans l'environnement
(define (lookup [n : Symbol] [env : Env]) : Location
  (cond
    [(empty? env) (error 'lookup "free identifier")]
    [(equal? n (bind-name (first env))) (bind-location (first env))]
    [else (lookup n (rest env))]))

; Renvoie une adresse mémoire libre
```

```
(define (new-loc [sto : Store]) : Location
  (+ (max-address sto) 1))

; Le maximum des adresses mémoires utilisés
(define (max-address [sto : Store]) : Location
  (max-address2 (store-storages sto))
  )

(define (max-address2 sto) : Location
  (if (empty? sto)
      0
      (max (cell-location (first sto)) (max-address2 (rest sto)))))

; Accès à un emplacement mémoire
(define (fetch [l : Location] [sto : Store]) : Value
  (fetch2 l (store-storages sto)))

(define (fetch2 [l : Location] sto) : Value
  (cond
    [(empty? sto) (error 'interp "segmentation fault")]
    [(equal? l (cell-location (first sto))) (cell-val (first sto))]
    [else (fetch2 l (rest sto))]))

;;;;;;;;;
; Tests ;
;;;;;;;;;

(define (interp-expr [e : S-Exp]) : Value
  (v*s-v (interp (parse e) mt-env mt-store)))
```