```
; TP 04

#lang plait

;;;;;;;;;
; 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)]
  [boxE (val : Exp)]
  [unboxE (b : Exp)]
  [setboxE (b : Exp) (val : Exp)]
  [beginE (exps : (Listof Exp))]
  [recordE (fields : (Listof Symbol)) (args : (Listof Exp))]
  [getE (record : Exp) (field : Symbol)]
  [setE (record : Exp) (field : Symbol) (arg : Exp)])

; Représentation des valeurs
(define-type Value
  [numV (n : Number)]
  [closV (par : Symbol) (body : Exp) (env : Env)]
  [boxV (l : Location)]
  [recV (fields : (Listof Symbol)) (locs : (Listof Location))])

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

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

; 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-alias Store (Listof Storage))
(define mt-store empty)

(define (override-store [c : Storage] [sto : Store]) ; Hypothèse supplémentaire : les cellules apparaissent pas ordre d'adresses décroissantes
  (cond
    [(or (empty? sto) (> (cell-location c) (cell-location (first sto)))) (cons c sto)]
    [(= (cell-location c) (cell-location (first sto))) (cons c (rest sto))]
    [else (cons (first sto) (override-store c (rest sto)))]))

;;;;;;;;;;;;;;;;;;;;;;;
; 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? `{box ANY} s)
     (let ([sl (s-exp->list s)])
       (boxE (parse (second sl))))]
    [(s-exp-match? `{unbox ANY} s)
     (let ([sl (s-exp->list s)])
       (unboxE (parse (second sl))))]
    [(s-exp-match? `{set-box! ANY ANY} s)
     (let ([sl (s-exp->list s)])
       (setboxE (parse (second sl)) (parse (third sl))))]
    [(s-exp-match? `{begin ANY ANY ...} s)
     (let ([sl (s-exp->list s)])
       (beginE (map parse (rest sl))))]
    [(s-exp-match? `{record [SYMBOL ANY] ...} s)
     (let ([sl (s-exp->list s)])
       (recordE (map (lambda (l) (s-exp->symbol (first (s-exp->list l)))) (rest sl))
                (map (lambda (l) (parse (second (s-exp->list l)))) (rest sl))))]
    [(s-exp-match? `{get ANY SYMBOL} s)
     (let ([sl (s-exp->list s)])
       (getE (parse (second sl)) (s-exp->symbol (third sl))))]
    [(s-exp-match? `{set! ANY SYMBOL ANY} s)
```

```
     (let ([sl (s-exp->list s)])
       (setE (parse (second sl)) (s-exp->symbol (third sl)) (parse (fourth 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 (lookup s env) 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)
              (with [(v-arg sto-arg) (interp arg env sto-f)]
                    (interp body (extend-env (bind par v-arg) c-env) sto-arg))]
             [else (error 'interp "not a function")]))]
    [(letE s rhs body)
     (with [(v-rhs sto-rhs) (interp rhs env sto)]
           (interp body (extend-env (bind s v-rhs) env) sto-rhs))]
    [(boxE val)
     (with [(v-val sto-val) (interp val env sto)]
           (let ([l (new-loc sto-val)])
             (v*s (boxV l) (override-store (cell l v-val) sto-val))))]
    [(unboxE b)
     (with [(v-b sto-b) (interp b env sto)]
           (type-case Value v-b
             [(boxV l) (v*s (fetch l sto-b) sto-b)]
             [else (error 'interp "not a box")]))]
    [(setboxE b val)
     (with [(v-b sto-b) (interp b env sto)]
           (type-case Value v-b
             [(boxV l)
              (with [(v-val sto-val) (interp val env sto-b)]
                    (v*s v-val (override-store (cell l v-val) sto-val)))]
             [else (error 'interp "not a box")]))]
    [(beginE exps)
     (if (cons? (rest exps))
         (with [(v sto2) (interp (first exps) env sto)]
               (interp (beginE (rest exps)) env sto2))
         (interp (first exps) env sto))]
    [(recordE fds args)
     (let* ([locs (new-locs (length args) sto)]
            [sto-rec (foldl override-store sto (map (lambda (loc) (cell loc (numV 0))) locs))]
            [sto-fin (init-fields locs args env sto)])
       (v*s (recV fds locs) sto-fin))]
    [(getE rec fd)
     (with [(v-rec sto-rec) (interp rec env sto)]
           (type-case Value v-rec
             [(recV fds locs) (v*s (fetch (find fd fds locs) sto-rec) sto-rec)]
             [else (error 'interp "not a record")]))]
    [(setE rec fd arg)
     (with [(v-rec sto-rec) (interp rec env sto)]
           (type-case Value v-rec
             [(recV fds locs)
              (with [(v-arg sto-arg) (interp arg env sto)]
                    (let ([loc (find fd fds locs)])
                      (v*s v-arg (override-store (cell loc v-arg) sto-arg))))]
             [else (error 'interp "not a record")]))]))

(define (init-fields [locs : (Listof Location)] [args : [Listof Exp]]
                     [env : Env] [sto : Store]) : Store
  (if (empty? locs)
      sto
      (with [(v-arg sto-arg) (interp (first args) env sto)]
            (init-fields (rest locs) (rest args) env
                         (override-store (cell (first locs) v-arg) sto-arg)))))

; Fonctions utilitaires pour l'arithmétique
(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]) : Value
  (cond
    [(empty? env) (error 'lookup "free identifier")]
    [(equal? n (bind-name (first env))) (bind-val (first env))]
    [else (lookup n (rest env))]))

; Renvoie une adresse mémoire libre
(define (new-loc [sto : Store]) : Location
  (+ (max-address sto) 1))

; Renvoie des adresses mémoires libres successives
(define (new-locs [n : Number] [sto : Store]) : (Listof Location)
  (let ([l (new-loc sto)])
    (build-list n (lambda (i) (+ l i)))))

; Le maximum des adresses mémoires utilisés
(define (max-address [sto : Store]) : Location
  (if (empty? sto)
      0
      (max (cell-location (first sto)) (max-address (rest sto)))))

; Accès à un emplacement mémoire
(define (fetch [l : Location] [sto : Store]) : Value
  (cond
    [(empty? sto) (error 'interp "segmentation fault")]
    [(equal? l (cell-location (first sto))) (cell-val (first sto))]
    [else (fetch l (rest sto))]))

; Recherche un symbole dans une liste de symboles et renvoie la valeur
associée
(define (find [fd : Symbol] [fds : (Listof Symbol)] [locs : (Listof
Location)]) : Location
  (cond
    [(empty? fds) (error 'interp "no such field")]
    [(equal? fd (first fds)) (first locs)]
    [else (find fd (rest fds) (rest locs))]))

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

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

(test (interp (parse `{set-box! {box 2} 3}) mt-env mt-store)
      (v*s (numV 3) (list (cell 1 (numV 3)) (cell 1 (numV 2)))))

(test (interp-expr `{let {[b {box 0}]}
                      {begin
                        {set-box! b {+ 1 {unbox b}}}
                        {set-box! b {* 2 {unbox b}}}
                        {set-box! b {+ 3 {unbox b}}}}})
      (numV 5))

(test (interp-expr `{let {[a {box 1}]}
                      {let {[r {record
                                [a {set-box! a {* 2 {unbox a}}}]
                                [b {set-box! a {* 2 {unbox a}}}]}]}
                        {+ {unbox a} {+ {get r a} {get r b}}}}})
      (numV 10))

(test (interp-expr `{let {[r {record [a 1]}]}
                      {begin {set! r a 2} {get r a}}})
      (numV 2))

(test (interp-expr `{let {[r {record [a 1] [b 2]}]}
                      {begin
                        {set! r a {+ {get r b} 3}}
                        {set! r b {* {get r a} 4}}
                        {+ {get r a} {get r b}}}})
      (numV 25))
```