```
; Cours 04 : Les boîtes avec macro simplificatrice



;;;;;;;;;
; 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 (l : Exp) (r : (Listof Exp))]
  [recordE (s : (Listof Symbol)) (e : (Listof Exp))]
  [getE (e : Exp) (s : Symbol)]
  [setE (e : Exp) (s : Symbol) (e2 : Exp)])


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


;;;;;;;;;;;;;;;;;;;;;;
; 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)
```

```
104       (let ([sl (s-exp->list s)])
105         (setboxE (parse (second sl)) (parse (third sl)))))]
106      [(s-exp-match? `{begin ANY ANY ...} s)
107       (let ([sl (s-exp->list s)])
108         (beginE (parse (second sl)) (map parse (rest (rest sl)))))]
109      [(s-exp-match? `{set! ANY SYMBOL ANY} s)
110       (let ([sl (s-exp->list s)])
111         (setE (parse (second sl)) (s-exp->symbol (third sl)) (parse
111  (fourth sl))))]
112
113
114      [(s-exp-match? `{record [SYMBOL ANY] ...} s)
115       (let ([sl (s-exp->list s)])
116         (recordE (map (lambda (l) (s-exp->symbol (first (s-exp->list
116  l)))) (rest sl))
117                  (map (lambda (l) (parse (second (s-exp->list l))))
117  (rest sl))))]
118       [(s-exp-match? `{get ANY SYMBOL} s)
119       (let ([sl (s-exp->list s)])
120         (getE (parse (second sl)) (s-exp->symbol (third sl))))]
121
122
123      [(s-exp-match? `{ANY ANY} s)
124       (let ([sl (s-exp->list s)])
125         (appE (parse (first sl)) (parse (second sl))))]
126
127      [else (error 'parse "invalid input")]))
128
129  ;;;;;;;;;;;;;;;;;;
130  ; Interprétation ;
131  ;;;;;;;;;;;;;;;;;;
132
133  ; Interpréteur
134  (define (interp [e : Exp] [env : Env] [sto : Store]) : Result
135    (type-case Exp e
136      [(numE n) (v*s (numV n) sto)]
137      [(idE s) (v*s (lookup s env) sto)]
138      [(plusE l r)
139       (with [(v-l sto-l) (interp l env sto)]
140             (with [(v-r sto-r) (interp r env sto-l)]
141                   (v*s (num+ v-l v-r) sto-r)))]
142      [(multE l r)
143       (with [(v-l sto-l) (interp l env sto)]
144             (with [(v-r sto-r) (interp r env sto-l)]
145                   (v*s (num* v-l v-r) sto-r)))]
146
147
148      [(lamE par body) (v*s (closV par body env) sto)]
149      [(appE f arg)
150       (with [(v-f sto-f) (interp f env sto)]
151             (type-case Value v-f
152               [(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))))]
    [(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)))]
             [else (error 'interp "not a box")]))]
    [(beginE l r) (beg l r env sto)]
    [(recordE s e) (let ([a (rec e empty env sto)]) (v*s (recV s (fst
a)) (snd a)))]


    [(getE e s)
     (let ([a (interp e env sto)])
     (type-case Value (v*s-v a)
       [(recV fds vs) (v*s (fetch (find s fds vs) (v*s-s a)) (v*s-s
a))]
       [else (error 'interp "not a record")]))]

    [(setE e s e2)
     (let ([a (v*s-v (interp e env sto))])
     (type-case Value a
       [(recV fds vs) (let ([b (interp e2 env sto)]) (v*s (v*s-v b)
(override-store (cell (find s fds vs) (v*s-v b)) (v*s-s b))))]
       [else (error 'interp "not a record")]))]
    ))

; Fonctions utilitaires pour l'arithmétique

(define (update [fd : Symbol] [new-val : Value]
                [fds : (Listof Symbol)] [vs : (Listof Value)] sto)
  (cond
    [(empty? fds) (error 'interp "no such field")]
    [(equal? fd (first fds)) (cons new-val (rest vs))]
    [else (cons (first vs) (update fd new-val (rest fds) (rest vs)
sto))]))
```

```
200
201
202 (define (rec e l env sto)
203   (if (empty? e)
204       (pair l sto)
205       (with [(v-l sto-l) (interp (first e) env sto)]
206             (let ([loc (new-loc sto-l)])
207                   (rec (rest e) (append l (list loc)) env
207 (override-store (cell loc v-l) sto-l)))))
208   )
209
210 (define (find [fd : Symbol] [fds : (Listof Symbol)] [vs : (Listof
210 Location)]) : Location
211   (cond
212     [(empty? fds) (error 'interp "no such field")]
213     [(equal? fd (first fds)) (first vs)]
214     [else (find fd (rest fds) (rest vs))]))
215
216 (define (beg l r env sto)
217   (if (empty? r)
218       (interp l env sto)
219       (with [(v-l sto-l) (interp l env sto)]
220             (beg (first r) (rest r) env sto-l))))
221
222
223
224 (define (num-op [op : (Number Number -> Number)]
225                 [l : Value] [r : Value]) : Value
226   (if (and (numV? l) (numV? r))
227       (numV (op (numV-n l) (numV-n r)))
228       (error 'interp "not a number")))
229
230 (define (num+ [l : Value] [r : Value]) : Value
231   (num-op + l r))
232
233 (define (num* [l : Value] [r : Value]) : Value
234   (num-op * l r))
235
236 ; Recherche d'un identificateur dans l'environnement
237 (define (lookup [n : Symbol] [env : Env]) : Value
238   (cond
239     [(empty? env) (error 'lookup "free identifier")]
240     [(equal? n (bind-name (first env))) (bind-val (first env))]
241     [else (lookup n (rest env))]))
242
243 ; Renvoie une adresse mémoire libre
244 (define (new-loc [sto : Store]) : Location
245   (+ (max-address sto) 1))
246
247 ; Le maximum des adresses mémoires utilisés
248 (define (max-address [sto : Store]) : Location
249   (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))]))

;;;;;;;;;
; 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)))))



(test (interp (parse `{let {[b { box 0}]}

                        {let {[c {box 1}]}

                          {let {[a {box 2}]}

                            {set-box! b 8}}}}) mt-env mt-store )

      (v*s (numV 8) (list (cell 1 (numV 8)) (cell 2 (numV 1)) (cell 3
(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} } }

                           { set-box! b {+ 8 { unbox b} } }

                           { set-box! b {* 3 { unbox b} } }}}})

       ( numV 39))
```

```
301
302
303
304 (test (interp-expr `{let {[b {box 0}]}
305
306                       {begin
307
308                         {set-box! b {+ 1 {unbox b}}}
309
310                         }})
311
312       (numV 1))
313
314
315
316 (test (interp-expr `{let {[r {record [x 1] [y 2]}]}
317
318                       {get r x}}) (numV 1))
319
320
321
322 ( test ( interp-expr `{ let {[a { box 1}]}
323
324                          { let {[r { record
325
326                                      [a { set-box! a {* 2 { unbox a} }
326 }]
327
328                                      [b { set-box! a {* 2 { unbox a} }
328 }]}]}
329
330                             {+ { unbox a} {+ { get r a} { get r b} } }
330 } })
331
332        ( numV 10))
333
334 (test (interp-expr `{let {[r {record [x 1] [y 2]}]}
335
336                       {get r y}}) (numV 2))
337
338 (test (interp-expr `{let {[r {record [x 1] [y {+ 2 3}]}]}
339
340                       {get r y}}) (numV 5))
341
342 (test (interp-expr `{let {[r {record [x 1]}]}
343
344                       {get r x}}) (numV 1))
345
346 (test/exn (interp-expr `{{record [x 0]} 1}) "not a function")
347
348 (test/exn (interp-expr `{+ {record [x 0]} 1}) "not a number")
349
```

```
(test (interp-expr `{let {[b1 {box 1}]}

                      {let {[b2 {box 2}]}

                        {let {[v {set-box! b1 3}]}

                          {unbox b2}}}})

      (numV 2))



(test (interp-expr `{let {[b1 {box 1}]}

                      {let {[b2 {box 2}]}

                        {let {[v {set-box! b2 3}]}

                          {unbox b1}}}})

      (numV 1))

( 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))

( interp-expr `{ let {[r { record [a 1] [b 2] [c 5]}]}
                  { begin
                     { set! r a {+ { get r b} 3} }
                     { set! r c {* { get r c} 4} }
                     { set! r b {* { get r c} 4} }}})

(interp-expr `{ let {[r { record [a 1] [b 2] [c 5]}]}

                      { set! r b 3} })

(test (interp (parse `{begin
                        {box 2}
```

```
                          {set-box! {box 5} 6}}) mt-env  mt-store)
       (v*s (numV 6) (list (cell 1 (numV 2)) (cell 2 (numV 6)))))

(test (interp-expr `{get {record [a 0]} a})
       (numV 0))
```