```
; Cours 05 : Les variables

#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)]
  [setE (s : Symbol) (val : Exp)]
  [beginE (l : Exp) (r : Exp)]
  [addressE (var : Symbol)]
  [contentE (ptr : Exp)]
  [setcontentE (ptr : Exp) (val : Exp)]
  [mallocE (size : Exp)]
  [freeE (ptr : 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 Store
```

```
  [store (storages : (Listof Storage)) (pointers : (Listof Pointer))])
(define mt-store (store empty empty))
(define (override-store c sto)
  (store (cons c (store-storages sto)) (store-pointers sto)))
(define-type Pointer
  [pointer (loc : Location) (size : Number)])

;;;;;;;;;;;;;;;;;;;;;
; 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)])
       (setcontentE (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 ;
;;;;;;;;;;;;;;;;;
```

```
120
121 ; Interpréteur
122 (define (interp [e : Exp] [env : Env] [sto : Store]) : Result
123   (type-case Exp e
124     [(numE n) (v*s (numV n) sto)]
125     [(idE s) (v*s (fetch (lookup s env) sto) sto)]
126     [(plusE l r)
127      (with [(v-l sto-l) (interp l env sto)]
128            (with [(v-r sto-r) (interp r env sto-l)]
129                  (v*s (num+ v-l v-r) sto-r)))]
130     [(multE l r)
131      (with [(v-l sto-l) (interp l env sto)]
132            (with [(v-r sto-r) (interp r env sto-l)]
133                  (v*s (num* v-l v-r) sto-r)))]
134     [(lamE par body) (v*s (closV par body env) sto)]
135     [(appE f arg)
136      (with [(v-f sto-f) (interp f env sto)]
137            (type-case Value v-f
138              [(closV par body c-env)
139               (type-case Exp arg
140                 [(idE s) (interp body
141                                  (extend-env (bind par (lookup s env))
141 c-env)
142                                  sto-f)]
143                 [else (with [(v-arg sto-arg) (interp arg env sto-f)]
144                             (let ([l (new-loc sto-arg)])
145                               (interp body
146                                       (extend-env (bind par l) c-env)
147                                       (override-store (cell l v-arg)
147 sto-arg))))])]
148              [else (error 'interp "not a function")]))]
149     [(letE s rhs body)
150      (with [(v-rhs sto-rhs) (interp rhs env sto)]
151            (let ([l (new-loc sto-rhs)])
152              (interp body
153                      (extend-env (bind s l) env)
154                      (override-store (cell l v-rhs) sto-rhs))))]
155     [(setE var val)
156      (let ([l (lookup var env)])
157        (with [(v-val sto-val) (interp val env sto)]
158              (v*s v-val (override-store (cell l v-val) sto-val))))]
159     [(beginE l r)
160      (with [(v-l sto-l) (interp l env sto)]
161            (interp r env sto-l))]
162     [(addressE var) (v*s (numV (lookup var env)) sto)]
163     [(contentE ptr)
164      (with [(v-ptr sto-ptr) (interp ptr env sto)]
165            (type-case Value v-ptr
166              [(numV loc) (v*s (fetch loc sto-ptr) sto-ptr)]
167              [else (error 'interp "segmentation fault")]))]
168     [(setcontentE ptr val)
169      (with [(v-ptr sto-ptr) (interp ptr env sto)]
170            (type-case Value v-ptr
171              [(numV loc)
172               (if (positive-integer? loc)
173                   (with [(v-val sto-val) (interp val env sto-ptr)]
174                         (v*s v-val (override-store (cell loc v-val)
174 sto-val)))
175                   (error 'interp "segmentation fault"))]
176              [else (error 'interp "segmentation fault")]))]
```

```
177     [(mallocE size)
178      (with [(v-size sto-size) (interp size env sto)]
179            (type-case Value v-size
180              [(numV n) (if (positive-integer? n)
181                            (let ([loc (new-loc sto-size)])
182                              (v*s (numV loc) (add-pointer loc n
182 (init-cells n loc sto-size))))
183                            (error 'interp "not a size"))]
184              [else (error 'interp "not a size")])))]
185     [(freeE ptr)
186      (with [(v-ptr sto-ptr) (interp ptr env sto)]
187            (type-case Value v-ptr
188              [(numV loc) (v*s (numV 0) (remove-pointer loc sto-ptr))]
189              [else (error 'interp "not an allocated pointer")])))]))
190
191 ; Prédicat pour les entiers strictement positifs
192 (define (positive-integer? [n : Number]) : Boolean
193   (and (> n 0) (= n (floor n))))
194
195 ; Renvoie un état de la mémoire où l'on a ajouté nb cellule à partir de
195 l'adresse loc
196 (define (init-cells [n : Number] [loc : Location] [sto : Store]) : Store
197   (if (= n 0)
198       sto
199       (init-cells (- n 1) loc (override-store (cell (new-loc sto) (numV
199 0)) sto))))
200
201 ; Ajoute un pointeur à la liste des pointeurs de sto
202 (define (add-pointer [loc : Location] [size : Number] [sto : Store]) :
202 Store
203   (store (store-storages sto) (cons (pointer loc size) (store-pointers
203 sto))))
204
205 (define (remove [e : 'a] [l : (Listof 'a)]) : (Listof 'a)
206   (cond
207     [(empty? l) l]
208     [(equal? (first l) e) (rest l)]
209     [else (cons (first l) (remove e (rest l)))]))
210
211 ; Renvoie un pointeur associé à une adresse
212 (define (find-pointer [loc : Location] [pointers : (Listof Pointer)]) :
212 Pointer
213   (cond
214     [(empty? pointers) (error 'interp "not an allocated pointer")]
215     [(= (pointer-loc (first pointers)) loc) (first pointers)]
216     [else (find-pointer loc (rest pointers))]))
217
218 ; Retire un pointeur de la liste des pointeurs et supprime toutes les
218 cellules associées
219 (define (remove-pointer [loc : Location] [sto : Store]) : Store
220   (let ([ptr (find-pointer loc (store-pointers sto))])
221     (type-case Pointer ptr
222       [(pointer loc size)
223        (store (filter (lambda (c)
224                         (or (< (cell-location c) loc)
225                             (>= (cell-location c) (+ loc size))))
226                       (store-storages sto))
227               (remove ptr (store-pointers sto)))])))
228
229 ; 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]) : 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 (store-storages sto)) 1))

; Le maximum des adresses mémoires utilisés
(define (max-address [sto : (Listof Storage)]) : 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
  (local [(define (fetch-aux [storages : (Listof Storage)]) : Value
            (cond
              [(empty? storages) (error 'interp "segmentation fault")]
              [(equal? l (cell-location (first storages))) (cell-val
(first storages))]
              [else (fetch-aux (rest storages))]))]
    (fetch-aux (store-storages sto))))

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

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

(test (interp (parse `{let {[p {malloc 3}]} p}) mt-env mt-store)
      (v*s (numV 1) (store (list (cell 4 (numV 1))
                                 (cell 3 (numV 0))
                                 (cell 2 (numV 0))
                                 (cell 1 (numV 0)))
                           (list (pointer 1 3)))))

(test (interp (parse `{let {[p {malloc 3}]} {free p}})
              mt-env
              mt-store)
      (v*s (numV 0) (store (list (cell 4 (numV 1)))
                           empty)))

(test (interp (parse `{let {[p {malloc 3}]} p}) mt-env mt-store)
```

```
289         (v*s (numV 1) (store (list (cell 4 (numV 1)) ; addresse de p
290                                    (cell 3 (numV 0))
291                                    (cell 2 (numV 0))
292                                    (cell 1 (numV 0)))
293                              (list (pointer 1 3)))))
```