

```

1 ; Cours 05 : Les variables
2
3
4
5 ;;;;;;;;;;
6 ; Macro ;
7 ;;;;;;;;;;
8
9 (define-syntax-rule (with [(v-id sto-id) call] body)
10   (type-case Result call
11     [(v*s v-id sto-id) body]))
12
13 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
14 ; Définition des types ;
15 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16
17 ; Représentation des expressions
18 (define-type Exp
19   [numE (n : Number)]
20   [idE (s : Symbol)]
21   [plusE (l : Exp) (r : Exp)]
22   [multE (l : Exp) (r : Exp)]
23   [lamE (par : Symbol) (body : Exp)]
24   [appE (fun : Exp) (arg : Exp)]
25   [letE (s : Symbol) (rhs : Exp) (body : Exp)]
26   [setE (s : Symbol) (val : Exp)]
27   [beginE (l : Exp) (r : Exp)]
28   [addressE (s : Symbol)]
29   [contentE (e : Exp)]
30   [set-content!E (e : Exp) (e2 : Exp)]
31   [mallocE (e : Exp)]
32   [freeE (e : Exp)]
33   )
34
35 ; Représentation des valeurs
36 (define-type Value
37   [numV (n : Number)]
38   [closV (par : Symbol) (body : Exp) (env : Env)]
39   )
40
41 ; Représentation du résultat d'une évaluation
42 (define-type Result
43   [v*s (v : Value) (s : Store)])
44
45 ; Représentation des liaisons
46 (define-type Binding
47   [bind (name : Symbol) (location : Location)])
48
49 ; Manipulation de l'environnement
50 (define-type-alias Env (Listof Binding))
51 (define mt-env empty)
52 (define extend-env cons)

```

```

53
54 ; Représentation des adresses mémoire
55 (define-type-alias Location Number)
56
57 ; Représentation d'un enregistrement
58 (define-type Storage
59   [cell (location : Location) (val : Value)])
60
61 ; Manipulation de la mémoire
62 (define-type Pointer
63   [pointer (loc : Location) (size : Number)])
64 (define-type Store
65   [store (storages : (Listof Storage))
66         (pointers : (Listof Pointer))])
67
68 (define mt-store (store empty empty))
69 (define (override-store c l)
70   (store (override-store2 c (store-storages l)) (store-pointers
70 l))); (cons c (store-storages l)) (store-pointers l)))
71
72 (define (override-store2 c l)
73   (if (empty? l)
74       (cons c empty)
75       (let ([c2 (first l)])
76         (if (equal? (cell-location c) (cell-location c2))
77             (cons c (rest l))
78             (cons (first l) (override-store2 c (rest l)))))))
79
80 (define (override-pointers p l)
81   (store (store-storages l) (cons p (store-pointers l))))
82
83 ; Integer
84 (define (integer? n) (= n (floor n)))
85
86 ;;;;;;;;;;;;;;
87 ; Analyse syntaxique ;
88 ;;;;;;;;;;;;;;
89
90 (define (parse [s : S-Exp]) : Exp
91   (cond
92     [(s-exp-match? `NUMBER s) (numE (s-exp->number s))]
93     [(s-exp-match? `SYMBOL s) (idE (s-exp->symbol s))]
94     [(s-exp-match? `{+ ANY ANY} s)
95      (let ([sl (s-exp->list s)])
96        (plusE (parse (second sl)) (parse (third sl))))])
97     [(s-exp-match? `{* ANY ANY} s)
98      (let ([sl (s-exp->list s)])
99        (multE (parse (second sl)) (parse (third sl))))])
100    [(s-exp-match? `{lambda {SYMBOL} ANY} s)
101     (let ([sl (s-exp->list s)])
102       (lamE (s-exp->symbol (first (s-exp->list (second sl)))) (parse
102 (third sl))))])

```

```

103 [(s-exp-match? `{let [{SYMBOL ANY}] ANY} s)
104   (let ([sl (s-exp->list s)])
105     (let ([subst (s-exp->list (first (s-exp->list (second sl))))])
106       (letE (s-exp->symbol (first subst))
107         (parse (second subst))
108         (parse (third sl))))))])
109 [(s-exp-match? `{set! SYMBOL ANY} s)
110   (let ([sl (s-exp->list s)])
111     (setE (s-exp->symbol (second sl)) (parse (third sl))))])
112 [(s-exp-match? `{begin ANY ANY} s)
113   (let ([sl (s-exp->list s)])
114     (beginE (parse (second sl)) (parse (third sl))))])
115
116 [(s-exp-match? `{address SYMBOL} s)
117   (let ([sl (s-exp->list s)])
118     (addressE (s-exp->symbol (second sl))))])
119 [(s-exp-match? `{content ANY} s)
120   (let ([sl (s-exp->list s)])
121     (contentE (parse (second sl))))])
122 [(s-exp-match? `{set-content! ANY ANY} s)
123   (let ([sl (s-exp->list s)])
124     (set-content!E (parse (second sl)) (parse (third sl))))])
125
126 [(s-exp-match? `{malloc ANY} s)
127   (let [(sl (s-exp->list s))]
128     (mallocE (parse (second sl))))])
129 [(s-exp-match? `{free ANY} s)
130   (let [(sl (s-exp->list s))]
131     (freeE (parse (second sl))))])
132
133
134 [(s-exp-match? `{ANY ANY} s)
135   (let ([sl (s-exp->list s)])
136     (appE (parse (first sl)) (parse (second sl))))])
137 [else (error 'parse "invalid input")]]
138
139 ;;;;;;;;;;;;;;;;;;
140 ; Interprétation ;
141 ;;;;;;;;;;;;;;;;;;
142
143 ; Interpréteur
144 (define (interp [e : Exp] [env : Env] [sto : Store]) : Result
145   (type-case Exp e
146     [(numE n) (v*s (numV n) sto)]
147     [(idE s) (v*s (fetch (lookup s env) sto) sto)]
148     [(plusE l r)
149      (with [(v-l sto-l) (interp l env sto)]
150        (with [(v-r sto-r) (interp r env sto-l)]
151          (v*s (num+ v-l v-r) sto-r)))]
152     [(multE l r)
153      (with [(v-l sto-l) (interp l env sto)]
154        (with [(v-r sto-r) (interp r env sto-l)]

```

```

155         (v*s (num* v-l v-r) sto-r))))]
156 [(lamE par body) (v*s (closV par body env) sto)]
157 [(appE f arg)
158   (with [(v-f sto-f) (interp f env sto)]
159     (type-case Value v-f
160       [(closV par body c-env)
161         (type-case Exp arg
162           [(idE s) (interp body
163             (extend-env (bind par (lookup s env))
164               c-env)
165             sto-f)]
166           [else (with [(v-arg sto-arg) (interp arg env sto-f)]
167             (let ([l (new-loc sto-arg)])
168               (interp body
169                 (extend-env (bind par l) c-env)
170                 (override-store (cell l v-arg)
171                   sto-arg))))))]
172         [else (error 'interp "not a function")]]))]
171 [(letE s rhs body)
172   (with [(v-rhs sto-rhs) (interp rhs env sto)]
173     (let ([l (new-loc sto-rhs)])
174       (interp body
175         (extend-env (bind s l) env)
176         (override-store (cell l v-rhs) sto-rhs)))))]
177 [(setE var val)
178   (let ([l (lookup var env)])
179     (with [(v-val sto-val) (interp val env sto)]
180       (v*s v-val (override-store (cell l v-val) sto-val)))))]
181 [(beginE l r)
182   (with [(v-l sto-l) (interp l env sto)]
183     (interp r env sto-l))]
184 [(adresseE s) (v*s (numV (lookup s env)) sto)]
185 [(contentE e) (content e env sto)]
186 [(set-content!E e1 e2) (setcontent e1 e2 env sto)]
187 [(mallocE e) (malloc e env sto)]
188 [(freeE e) (free e env sto)]
189 ))
190
191 ; Fonctions utilitaires pour l'arithmétique
192 (define (free e env sto)
193   (let ([debut (interp e env sto)])
194     (let ([taille (findtaille (numV-n (v*s-v debut)) (store-pointers
194 (v*s-s debut))))]
195       (if (and (integer? taille) (> taille 0))
196         (libere (numV-n (v*s-v debut)) (numV-n (v*s-v debut)) taille
196 taille env (store-storages (v*s-s debut)) (store-pointers (v*s-s
196 debut))))
197       (error 'interp "not an allocated pointer")))))
198
199
200 (define (findtaille debut point)
201   (if (empty? point)

```

```

202         (error 'interp "not an allocated pointer")
203
204         (if (equal? (pointer-loc (first point)) debut)
205             (pointer-size (first point))
206
207             (findtaille debut (rest point))))
208     ))
209
210 (define (libere d debut t taille env stor point)
211     (if (equal? taille 0)
212         (v*s (numV 0) (store stor (supprimepoint d t point empty)))
213
214         (libere d (+ debut 1) t (- taille 1) env (recherchesto debut stor
215 empty) point)))
216
217 (define (recherchesto debut stor s)
218     (if (empty? stor)
219         (error 'interp "not an allocated pointer")
220
221         (if (equal? (cell-location (first stor)) debut)
222             (let ([fin (append s (rest stor))])
223                 fin)
224             (recherchesto debut (rest stor) (append s (list (first
224 stor))))))
225     ))
226
227 (define (supprimepoint debut taille point p)
228     (if (empty? point)
229         (error 'interp "not an allocated pointer")
230
231         (if (and (equal? (pointer-loc (first point)) debut) (equal?
231 (pointer-size (first point)) taille))
232             (let ([fin (append p (rest point))])
233                 fin)
234
235             (supprimepoint debut taille (rest point) (append p (list
235 (first point))))))
236     ))
237
238 (define (malloc e env sto)
239     (let ([n (interp e env sto)])
240
241         (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))
242
243             (allocation (numV-n (v*s-v n)) (numV-n (v*s-v n)) (new-loc sto)
243 env sto)
244             (error 'interp "not a size"))
245     ))
246
247 (define (allocation n i fstloc env sto)
248     (if (equal? 0 n)

```

```

249     (let ([dernieresto (override-pointers (pointer fstloc i) sto)])
250       (v*s (numV fstloc) dernieresto))
251     (let ([newsto (override-store (cell (new-loc sto) (numV 0)) sto)])
252       (allocation (- n 1) i fstloc env newsto)
253       )))
254
255 (define (setcontent l e env sto)
256   (let ([loc (location l env sto)])
257     (if (and (integer? (numV-n (v*s-v loc))) (> (numV-n (v*s-v loc))
257 0)))
258     (with [(v-l sto-l) (interp e env (v*s-s loc))]
259       (let ([derniersto (override-store (cell (numV-n (v*s-v loc))
259 v-l) sto-l)])
260         (v*s v-l derniersto)))
261     (error 'interp "segmentation fault")))
262   ))
263
264 (define (location l env sto)
265   (let ([n (interp l env sto)])
266     (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))
267       (v*s (v*s-v n) (v*s-s n))
268       (error 'interp "segmentation fault")))
269   ))
270
271 (define (content e env sto)
272   (let ([n (interp e env sto)])
273     (if (and (integer? (numV-n (v*s-v n))) (> (numV-n (v*s-v n)) 0))
274       (v*s (fetch (numV-n (v*s-v n)) (v*s-s n)) (v*s-s n))
275       (error 'interp "segmentation fault")))
276   ))
277
278
279 (define (num-op [op : (Number Number -> Number)]
280               [l : Value] [r : Value]) : Value
281   (if (and (numV? l) (numV? r))
282       (numV (op (numV-n l) (numV-n r)))
283       (error 'interp "not a number")))
284
285 (define (num+ [l : Value] [r : Value]) : Value
286   (num-op + l r))
287
288 (define (num* [l : Value] [r : Value]) : Value
289   (num-op * l r))
290
291 ; Recherche d'un identificateur dans l'environnement
292 (define (lookup [n : Symbol] [env : Env]) : Location
293   (cond
294     [(empty? env) (error 'lookup "free identifieur")]
295     [(equal? n (bind-name (first env))) (bind-location (first env))]
296     [else (lookup n (rest env))]))
297
298 ; Renvoie une adresse mémoire libre

```

```

299 (define (new-loc [sto : Store]) : Location
300   (+ (max-address sto) 1))
301
302 ; Le maximum des adresses mémoires utilisés
303 (define (max-address [sto : Store]) : Location
304   (max-address2 (store-storages sto))
305   )
306
307 (define (max-address2 sto) : Location
308   (if (empty? sto)
309       0
310       (max (cell-location (first sto)) (max-address2 (rest sto)))))
311
312 ; Accès à un emplacement mémoire
313 (define (fetch [l : Location] [sto : Store]) : Value
314   (fetch2 l (store-storages sto)))
315
316 (define (fetch2 [l : Location] sto) : Value
317   (cond
318     [(empty? sto) (error 'interp "segmentation fault")]
319     [(equal? l (cell-location (first sto))) (cell-val (first sto))]
320     [else (fetch2 l (rest sto))]))
321
322 ;;;;;;;;;;
323 ; Tests ;
324 ;;;;;;;;;;
325
326 (define (interp-expr [e : S-Exp]) : Value
327   (v*s-v (interp (parse e) mt-env mt-store)))
328
329
330
331

```