4.4 Annexe GEN & DU

4.4.1 GEN (Générateur programmes et données)

contient next-programme, next-data

bijection entre ω et LISTE : NEXT
fonction next avec choix des atomes (voir initialisation)

```
(defun big-jump-a (l)
 (cond ((last-atom? l)(cons (first-atom) nil))
       ((equal l (cons (last-atom) nil))
          (cons (cons (first-atom) nil)nil))
       (t (cons (big-jump-a (cdr l)) nil))
 ))

(defun jump-aux-a (l aux)
 (cond ((equal (car l)(last-atom))
          (jump-aux-a (cdr l) (cons (first-atom) aux)))
       ((null (cdr l)) (cons (caar (big-jump-a (car l)))
                             (cons (first-atom) aux)))
       (t (append (cons (caar (big-jump-a (car l))) aux)
                  (cons (big-jump-a (cadr l)) (cddr l))))
 ))

(defun jump-a (l)
 (cond ((null l) (cons (first-atom) nil))
       ((list-last-atom l) (big-jump-a l))
       (t (jump-aux-a l nil))
 ))

(defun list-last-atom (l)
 (cond ((null l) t)
       ((atom l) nil)
       ((last-atom? (car l))(list-last-atom (cdr l)))
 ))

(defun complet-a (l)
 (or (last-atom? l) (list-last-atom l))
 )

(defun list-complet-a (l)
 (cond ((null l) t)
       ((complet-a (car l)) (list-complet-a (cdr l)))
 ))

(defun faire-premier-a (listatom)
 (car (big-jump-a listatom))
 )

(defun next-i-a (l)
 (cond ((not (complet-a (car l)))
          (cons (next-a (car l)) (cdr l)))
       (t (cons (faire-premier-a (car l))
                (next-i-a (cdr l)) ))
 ))
```

```
(defun next-a (l)
 (cond ((null l)(first-atom))
       ((atom l)(next-atom l))
       ((list-complet-a l)(jump-a l))
       (t (next-i-a l))
 ))
```

initialisation 1 (deux atomes)

```
(defun first-atom ()
  'a)

(defun last-atom? (l)
 (equal l 'b)
 )

(defun last-atom ()
 'b)

(defun next-atom (l)
  (cond
    ((equal l 'a) 'b)
    ((equal l 'b) '(a))
 ))
```

initialisation 2 (petits programmes LISP)

```
(defun first-atom ()
   'equal)

(defun last-atom? (l)
 (equal l 'null)
 )

(defun last-atom ()
 'null)

(defun next-atom (l)
  (cond
    ((equal l 'equal) 'car)
    ((equal l 'car) 'cdr)
    ((equal l 'cdr) 'x)
    ((equal l 'x) 'quote)
    ((equal l 'quote) 'lambda)
    ((equal l 'lambda) 'k)
    ((equal l 'k) 'cons)
    ((equal l 'cons) 'cond)
    ((equal l 'cond) 'null)
    ((equal l 'null) '(equal))   nil = (equal 'null 'cond)
 ))
```

filtre brut pour les petits programmes

```
(defun var (x)
 (equal x 'x)
 )
```

```
(defun sf (x n)
 (or (and (equal x 'quote)(equal n 1))
     (and (equal x 'car)(equal n 1))
     (and (equal x 'cdr)(equal n 1))
      (and (equal x 'k)(equal n 1))
     (and (equal x 'cons)(equal n 2))
 ))

(defun sr (x n)
 (or (and (equal x 'null)(equal n 1))
     (and (equal x 'equal)(equal n 2)))
)

(defun terme (x)
 (cond ((atom x)(var x))
       (t (or (and (equal (car x) 'quote)
                   (not (null (cdr x)))
                   (null (cddr x)))
              (and (sf (car x)
                   (length (cdr x)))
                   (list-terme (cdr x))
              )))
 ))
```

dans DU 2 on tient compte du terme constant

```
(defun list-terme (x)
 (cond ((null x) t)
       ((terme (car x)) (list-terme (cdr x)))
 ))

(defun prop (x)
 (or (equal x nil)
     (equal x 't)
     (and (listp x)
          (sr (car x)
              (length (cdr x)) )
          (list-terme (cdr x)))
 ))

(defun exp-cond (x)
 (and (listp x)
      (equal (car x) 'cond)
      (test (cadr x))
      (list-test (cddr x))
 ))


(defun test (x)
 (and (listp x)
      (null (cddr x))
      (prop (car x))
      (terme (cadr x))
 ))

(defun list-test (x)
 (cond ((null x) t)
       ((test (car x)) (list-test (cdr x)))
 ))

(defun bod (x)
 (or (terme x)
     (prop x)
     (exp-cond x)
 ))
```

boucles génératives

```
(defun boum (x)
 (prog (a)
       (setq a x)
     b (setq a (next-a a))
       (print a)
       (go b))
 )

(defun brol (x)
 (prog (a c)
       (setq a x)
     b (setq a (next-a a))
       (setq c (read-char))
       (cond ((equal c #\Space )(print a))
             (t (print '(barre d'espacement!...))))
j'ai trouvé "#\Space" en éxécutant (print (read-char)) !
       (go b))
 )

(defun gen (x)
 (prog (a)
       (setq a x)
     b (setq a (next-a a))
       (cond ((bod a )(print (list 'lambda '(x) a))))
       (go b))
 )
```

<u>éxécution :</u>

```
? (gen nil)       il s'agit de gen le vieux

(LAMBDA (X) X)
(LAMBDA (X) (CAR X))
(LAMBDA (X) (CDR X))
(LAMBDA (X) 'X)
(LAMBDA (X) (NULL X))
(LAMBDA (X) (EQUAL X X))
(LAMBDA (X) (CONS X X))
(LAMBDA (X) (CAR (CAR X)))
(LAMBDA (X) (CDR (CAR X)))
(LAMBDA (X) '(CAR X))
(LAMBDA (X) (NULL (CAR X)))
(LAMBDA (X) (CAR (CDR X)))
(LAMBDA (X) (CDR (CDR X)))
(LAMBDA (X) '(CDR X))
(LAMBDA (X) (NULL (CDR X)))
(LAMBDA (X) (CAR 'X))
(LAMBDA (X) (CDR 'X))
(LAMBDA (X) ''X)
(LAMBDA (X) (NULL 'X))
(LAMBDA (X) (CAR (CONS X X)))
(LAMBDA (X) (CDR (CONS X X)))
(LAMBDA (X) '(CONS X X))
(LAMBDA (X) (NULL (CONS X X)))
(LAMBDA (X) (EQUAL X (CAR X)))
(LAMBDA (X) (CONS X (CAR X)))
(LAMBDA (X) (EQUAL X (CDR X)))
(LAMBDA (X) (CONS X (CDR X)))
(LAMBDA (X) (EQUAL X 'X))
(LAMBDA (X) (CONS X 'X))
Aborted
?
```

voilà le dernier gen (celui de gen prog data)

```
? (gen nil)

(LAMBDA (X) X)
(LAMBDA (X) 'EQUAL)
(LAMBDA (X) 'CAR)
(LAMBDA (X) 'CDR)
(LAMBDA (X) (CAR X))
(LAMBDA (X) (CDR X))
(LAMBDA (X) 'X)
(LAMBDA (X) (K X))
(LAMBDA (X) (NULL X))
(LAMBDA (X) 'QUOTE)
(LAMBDA (X) 'LAMBDA)
(LAMBDA (X) 'K)
(LAMBDA (X) 'CONS)
(LAMBDA (X) 'COND)
(LAMBDA (X) 'NULL)
(LAMBDA (X) '(EQUAL))
(LAMBDA (X) '(CAR))
(LAMBDA (X) '(CDR))
(LAMBDA (X) '(X))
(LAMBDA (X) '(QUOTE))
(LAMBDA (X) '(LAMBDA))
(LAMBDA (X) '(K))
(LAMBDA (X) '(CONS))
(LAMBDA (X) '(COND))
(LAMBDA (X) '(NULL))
(LAMBDA (X) (EQUAL X X))
(LAMBDA (X) (CONS X X))
```

next-programme & next-data (sur le mode de la conversation)

```
(defun next-programme (p)
  (let ((b (next-a (body p)) ))
    (cond ((bod b) (list 'lambda (list 'x) b))
          (t (next-programme (list 'lambda (list 'x) b)))
       )
  )
 )  cela aurait été plus joli avec un next-body ! (passons)

(defun next-data (d)
  (list 'quote (next-a (cadr d)))
 )

(defun next-programme-avec-nouvelle-var (p)
 (let ((b (next-a (body p)) )
       (new-var (gensym)))
   (cond ((bod b) (list 'lambda (list new-var) (subst new-var 'x b)))
         (t (next-programme (list 'lambda (list 'x) b)))
      )
  )
 )
```

ne marche pas : E-step devra appeler gensym lui-même après avoir effectué next-programme.
ça ne marche pas car x joue un rôle privilégié (pour aller plus vite dans la de programme.

a moins d'appeler next-programme et de substituer les variables avant et après:

```
(defun next-programme-avec-nouvelle-var (p)
 (let ((px (remettre-x p))
       (new-var (gensym)))
   (subst new-var 'x (next-programme px))
  )
 )
```

pas besoin de let :

```
(defun next-panv (p)
  (cond ((null p) '(lambda (x) x))
        (T (next-programme-avec-nouvelle-var p))
       )
  ) pour partir de nil.

(defun next-programme-avec-nouvelle-var (p)
  (subst (gentemp) 'x (next-programme (remettre-x p)))
  )


(defun remettre-x (p)
  (subst 'x (car (argu p)) p)
  )
```

test

?

```
(defun test-data ()
 (prog (a)
       (setq a nil)
     b (setq a (next-data a))
       (print a)
       (go b)))


'EQUAL
'CAR
'CDR
'X
'QUOTE
'LAMBDA
'K
'CONS
'COND
'NULL
'(EQUAL)
'(CAR)
'(CDR)
'(X)
'(QUOTE)
'(LAMBDA)
'(K)
'(CONS)
'(COND)
'(NULL)
'((EQUAL))
'((CAR))
'((CDR))
'((X))
'((QUOTE))
'((LAMBDA))
'((K))
'((CONS))
'((COND))
'((NULL))
'(EQUAL EQUAL)
'(CAR EQUAL)              ON N'A PAS LA DONNEE "NIL"!!
'(CDR EQUAL)
```

```
'(X EQUAL)

?
(defun test-npanv ()
 (prog (a)
       (setq a NIL)
     b (setq a (next-programme-avec-nouvelle-var a))
       (print a)
       (go b)))


Compiler warnings for function TEST-NPANV :       (???)
  Undeclared free variable (I)
TEST-NPANV
? (test-npanv)                ne marche pas à partir de nil !

(LAMBDA (#:G313) 'EQUAL)
(LAMBDA (#:G314) 'CAR)
(LAMBDA (#:G315) 'CDR)
(LAMBDA (#:G316) (CAR #:G316))
(LAMBDA (#:G317) (CDR #:G317))
(LAMBDA (#:G318) '#:G318)
(LAMBDA (#:G319) (K #:G319))
(LAMBDA (#:G320) (NULL #:G320))
(LAMBDA (#:G321) 'QUOTE)
(LAMBDA (#:G322) 'LAMBDA)
(LAMBDA (#:G323) 'K)
(LAMBDA (#:G324) 'CONS)
(LAMBDA (#:G325) 'COND)
(LAMBDA (#:G326) 'NULL)
(LAMBDA (#:G327) '(EQUAL))
(LAMBDA (#:G328) '(CAR))
(LAMBDA (#:G329) '(CDR))
(LAMBDA (#:G330) '(#:G330))
(LAMBDA (#:G331) '(QUOTE))
(LAMBDA (#:G332) '(LAMBDA))
(LAMBDA (#:G333) '(K))
(LAMBDA (#:G334) '(CONS))
(LAMBDA (#:G335) '(COND))
(LAMBDA (#:G336) '(NULL))
(LAMBDA (#:G337) (EQUAL #:G337 #:G337))
(LAMBDA (#:G338) (CONS #:G338 #:G338))
(LAMBDA (#:G339) '((EQUAL)))


encore un problème

? ACL ne comprend pas :
? ((LAMBDA (#:G313) #:G313) 'a)
> Error: Unbound variable: #:G313 .
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
1 >

plus étrange Φ-lisp ne comprend pas non plus :(avec même type de message)

(e '((LAMBDA (#:G313) #:G313) 'a) en)
(ERREUR #:G313 C-EST-QUOI-?)
? (e '((lambda (b) b) 'a) en)
A
? (atom #:G313)
> Error: Unbound variable: #:G313 .
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
1 > (atom '#:G313)
T
alors que Φ-lisp accepte n'importe qu'elle atome :
1 > (e '((lambda (3) 3) 'a) en)
A
```

à la différence de ACL qui refuse une lambda expression du type ((lambda (3) 3) 'a).

```
1 > ((lambda (#:G313) (car #:G313)) '(a b c))
> Error: Unbound variable: #:G313 .
> While executing: SYMBOL-VALUE
> Type Command-/ to continue, Command-. to abort.
2 > ((lambda (x) (car x)) '(a b c))
A
2 >
```

encore plus bizarre :

```
2 > (remettre-x '(lambda (#:G313) (car #:G313)))
(LAMBDA (X) (CAR #:G313))
2 > (remettre-x '(lambda (#:G262) (car (cdr #:G262))))
(LAMBDA (X) (CAR (CDR #:G262)))
2 >
```

plus bizarre car test-npanv a fonctionné ! (et fonctionne encore)

et ceci !?! :

```
? (equal '(#:G262) '(#:G262))
NIL
```

encore des tests :

```
(defun ?1 ()
  (let ((x (gensym)))
    (equal x x)
   )
 )
? (?1)
T
```

```
(defun next-programme-avec-nouvelle-var (p)
  (subst (gentemp) 'x (next-programme (remettre-x p)))
 )
? (e '((LAMBDA (T25) (EQUAL T25 T25)) 'a) en)
T
```

```
(defun next-programme-avec-nouvelle-var (p)
  (subst (gensym) 'x (next-programme (remettre-x p)))
 )
? (e '((LAMBDA (#:G234) (EQUAL #:G234 #:G234)) 'a) en)

(ERREUR #:G234 C-EST-QUOI-?)
```

LA MORALE : ON N'A PAS LE PROBLEME AVEC GENTEMP :

```
?
(defun test-npanv ()
 (prog (a)
       (setq a '(lambda (x) x))
     b (setq a (next-panv a))
       (print a)
       (go b)))
TEST-NPANV
```

```
? (test-npanv)

(LAMBDA (T2) 'EQUAL)
(LAMBDA (T3) 'CAR)
(LAMBDA (T4) 'CDR)
(LAMBDA (T5) (CAR T5))
(LAMBDA (T6) (CDR T6))
(LAMBDA (T7) 'T7)
(LAMBDA (T8) (K T8))
(LAMBDA (T9) (NULL T9))
(LAMBDA (T10) 'QUOTE)
(LAMBDA (T11) 'LAMBDA)
(LAMBDA (T12) 'K)
(LAMBDA (T13) 'CONS)
(LAMBDA (T14) 'COND)
(LAMBDA (T15) 'NULL)
(LAMBDA (T16) '(EQUAL))
(LAMBDA (T17) '(CAR))
(LAMBDA (T18) '(CDR))
(LAMBDA (T19) '(T19))
(LAMBDA (T20) '(QUOTE))
(LAMBDA (T21) '(LAMBDA))
(LAMBDA (T22) '(K))
(LAMBDA (T23) '(CONS))
(LAMBDA (T24) '(COND))
(LAMBDA (T25) '(NULL))
(LAMBDA (T26) (EQUAL T26 T26))
(LAMBDA (T27) (CONS T27 T27))
Aborted
? ((LAMBDA (T26) (EQUAL T26 T26)) 'a)
T
? (e '((LAMBDA (T5) (CAR T5)) '(ef gr)) en)
EF


j'ai modifié : next-panv part de nil :
? (defun test-npanv ()
 (prog (a)
       (setq a NIL)
     b (setq a (next-panv a))
       (print a)
       (go b)))
TEST-NPANV
? (test-npanv)

(LAMBDA (X) X)        correct : la première variable ne doit pas être renommée
(LAMBDA (T24) 'EQUAL)  correct gentemp a déjà été utilisé lors de la pre-
(LAMBDA (T25) 'CAR)    mière session.
(LAMBDA (T26) 'CDR)
(LAMBDA (T27) (CAR T27))
(LAMBDA (T28) (CDR T28))
(LAMBDA (T29) 'T29)
(LAMBDA (T30) (K T30))
(LAMBDA (T31) (NULL T31))
(LAMBDA (T32) 'QUOTE)
(LAMBDA (T33) 'LAMBDA)
(LAMBDA (T34) 'K)
(LAMBDA (T35) 'CONS)
(LAMBDA (T36) 'COND)
(LAMBDA (T37) 'NULL)
(LAMBDA (T38) '(EQUAL))
(LAMBDA (T39) '(CAR))
(LAMBDA (T40) '(CDR))
(LAMBDA (T41) '(T41))
(LAMBDA (T42) '(QUOTE))
(LAMBDA (T43) '(LAMBDA))
(LAMBDA (T44) '(K))
(LAMBDA (T45) '(CONS))
(LAMBDA (T46) '(COND))
```

```
(LAMBDA (T47) '(NULL))
(LAMBDA (T48) (EQUAL T48 T48))
(LAMBDA (T49) (CONS T49 T49))
Aborted

? (next-triple nil)
(1 1 1)
? (next-panv nil)
(LAMBDA (X) X)
? (next-panv '(LAMBDA (X) X))
(LAMBDA (T51) 'EQUAL)
? (next-data nil)
'EQUAL
```

4.4.2 DU (dovetelleur universel)

bijection entre ω et ω^3 : NEXT TRIPLE (19 2 91)

need Φ-lisp & Φ-dove (UT LI DO IS)

Next-triple (triple de nombres)

```
(defun next-triple (triple)
  (cond
   ((null triple) '(1 1 1))
   ((and (equal (cadr triple) 1)
         (equal (caddr triple) 1))
    (list 1 1 (+ 1 (car triple))))
   ((equal (caddr triple) 1)
    (list (+ 1 (car triple))
          1
          (- (cadr triple) 1)))
   (t (list (car triple)
            (+ 1 (cadr triple))
            (- (caddr triple) 1))
    )
  )
 )


? (next-triple nil)
(1 1 1)
? (next-triple '(1 1 1))
(1 1 2)
? (next-triple '(1 1 2))
(1 2 1)
? (next-triple '(1 2 1))
(2 1 1)
? (next-triple '(2 1 1))
(1 1 3)
? (next-triple '(1 1 3))
(1 2 2)
?
```

next-mémoire

```
(defun next-mémoire (mémoire)
  (let ((x (next-triple (car mémoire))))
    (cond
     ((equal x '(1 1 1)) (cons x nil))     pour éviter NIL dans la mémoire
                                           depuis que next-triple part de nil
     ((nouvelle-execution x) (cons x mémoire))
     (t (cons x (del-derniere-éxécution x mémoire)))
    )
```

```
  )
 )
```

plus tard enlevé les éxécutions arrêtées (EEA) : (t (EEA (rempl ...)

```
(defun nouvelle-éxécution (triple)    plus tard avec triplet
 (equal (caddr triple) 1)
 )

(defun del-derniere-éxécution (x mémoire)  x = la derniere éxécution
 (cond
  ((null mémoire) nil)
  ((equal (list (car x) (cadr x))
          (list (caar mémoire) (cadar mémoire)))
   (cdr mémoire))
  (t (cons (car mémoire) (del-derniere-éxécution x (cdr mémoire)))
   )
 )
)
```

test

```
(defun ttt ()    teste la suite infinie des triples (Test Tous les Triples)
 (prog (a)
       (setq a '())
       (print a)
     b (setq a (next-triple a))
       (print a)
       (go b))
)

(defun ttm () teste la suite infinie des mémoires (Test Toutes les Mémoires)
 (prog (a)
       (setq a '(()))
       (pprint a)  (print (length a)) (terpri)
     b (setq a (next-mémoire a))
       (pprint a) (print (length a)) (terpri)
       (go b))      )
```

éxécution (détail)

```
TTM
? (ttm)

((1 1 1))
1

((1 1 2))
1

((1 2 1) (1 1 2))
2

((2 1 1) (1 2 1) (1 1 2))
3

((1 1 3) (2 1 1) (1 2 1))
3

((1 2 2) (1 1 3) (2 1 1))
3

((1 3 1) (1 2 2) (1 1 3) (2 1 1))
```

4

((2 1 2) (1 3 1) (1 2 2) (1 1 3))
4

((2 2 1) (2 1 2) (1 3 1) (1 2 2) (1 1 3))
5

((3 1 1) (2 2 1) (2 1 2) (1 3 1) (1 2 2) (1 1 3))
6

((1 1 4) (3 1 1) (2 2 1) (2 1 2) (1 3 1) (1 2 2))
6

((1 2 3) (1 1 4) (3 1 1) (2 2 1) (2 1 2) (1 3 1))
6

((1 3 2) (1 2 3) (1 1 4) (3 1 1) (2 2 1) (2 1 2))
6

((1 4 1) (1 3 2) (1 2 3) (1 1 4) (3 1 1) (2 2 1) (2 1 2))
7

((2 1 3) (1 4 1) (1 3 2) (1 2 3) (1 1 4) (3 1 1) (2 2 1))
7

((2 2 2) (2 1 3) (1 4 1) (1 3 2) (1 2 3) (1 1 4) (3 1 1))
7

((2 3 1) (2 2 2) (2 1 3) (1 4 1) (1 3 2) (1 2 3) (1 1 4) (3 1 1))
8

((3 1 2) (2 3 1) (2 2 2) (2 1 3) (1 4 1) (1 3 2) (1 2 3) (1 1 4))
8

((3 2 1) (3 1 2) (2 3 1) (2 2 2) (2 1 3) (1 4 1) (1 3 2) (1 2 3) (1 1 4))
9

…

((9 5 3) (9 4 4) (9 3 5) (9 2 6) (9 1 7) (8 8 1) (8 7 2) (8 6 3) (8 5 4) (8 4 5)
(8 3 6) (8 2 7) (8 1 8) (7 9 1) (7 8 2) (7 7 3) (7 6 4) (7 5 5) (7 4 6) (7 3 7)
(7 2 8) (7 1 9) (6 10 1) (6 9 2) (6 8 3) (6 7 4) (6 6 5) (6 5 6) (6 4 7)
(6 3 8) (6 2 9) (6 1 10) (5 11 1) (5 10 2) (5 9 3) (5 8 4) (5 7 5) (5 6 6)
(5 5 7) (5 4 8) (5 3 9) (5 2 10) (5 1 11) (4 12 1) (4 11 2) (4 10 3) (4 9 4)
(4 8 5) (4 7 6) (4 6 7) (4 5 8) (4 4 9) (4 3 10) (4 2 11) (4 1 12) (3 13 1)
(3 12 2) (3 11 3) (3 10 4) (3 9 5) (3 8 6) (3 7 7) (3 6 8) (3 5 9) (3 4 10)
(3 3 11) (3 2 12) (3 1 13) (2 14 1) (2 13 2) (2 12 3) (2 11 4) (2 10 5) (2 9 6)
(2 8 7) (2 7 8) (2 6 9) (2 5 10) (2 4 11) (2 3 12) (2 2 13) (2 1 14) (1 15 1)
(1 14 2) (1 13 3) (1 12 4) (1 11 5) (1 10 6) (1 9 7) (1 8 8) (1 7 9) (1 6 10)
(1 5 11) (1 4 12) (1 3 13) (1 2 14) (1 1 15) (14 1 1) (13 2 1) (13 1 2)
(12 3 1) (12 2 2) (12 1 3) (11 4 1) (11 3 2) (11 2 3) (11 1 4) (10 5 1)
(10 4 2) (10 3 3) (10 2 4) (10 1 5) (9 6 1))
113

Remarquons que next-mémoire ne sera jamais utilisé tel quel, il sert de caricature à next-me-mp-md.

Recherche ou création des programmes (donnée) lors du dovetellage : ROC

Le dovetelleur trimbale la mémoire des programmes créés, ainsi que des données créées, ainsi que des éxécutions déjà effectuées a fin de ne pas calculer deuxfois la même chose.

ME = mémoire des éxécutions, c'est essentiellement la MEMOIRE de plus haut dans
laquelle on inclu le résultat des éxécutions (partielles) obtenues.
MP = la mémoire des programmes générés.
MD = la mémoire des données générées.
L'argument de NEXT-ME-MP-MD est la liste <ME,MP,MD> donc :

```
(defun select-me (me-mp-md)
  (car me-mp-md)
  )

(defun select-mp (me-mp-md)
  (cadr me-mp-md)
  )

(defun select-md (me-mp-md)
  (caddr me-mp-md)
  )
```

A présent il faut une routine capable de chercher un programme (resp. une donnée) en mémoire et de le (resp. la) créer s'il ne trouve pas ce dernier (resp. cette dernière).

la forme de MP est du genre : ((5 P5) (1 P1) (3 P3) (2 P2)) sans ordre
MD ((1 D1) (7 D7)) ...

(la forme de ME : (<triple> <éxéc> <triple> <éxéc> ...)).

A fin de pouvoir utiliser directement le résultat d'un ROC, ce dernier place en tête son résultat dans MP (resp. MD).

ainsi (ROC 5 ((2 P2)(5 P5)(1 P1))) donne ((5 P5)(2 P2)(1 P1))
(ROC 6 ((2 P2)(5 P5)(1 P1))) donne ((6 P6)(5 P5)(2 P2)(1 P1)).

où Pi est le ième programme.
ROC utilise des mémoires auxiliaires à fin d'éviter de parcourir plusieur fois la mémoire.

```
(defun roc (n m pvd) n = le numéro (donné par le triple généré)
                     m = la mémoire des programmes (MP) ou des données (MD), le
                     même ROC cherche les deux (=> appel de EVAL)
                     pvd = NEXT-PANV ou NEXT-DATA. (dans GEN (P&D)).
 (cond ((null m)
        (cons (list 1 (eval (list pvd nil)))
              nil))
       ((equal n (caar m)) m) facilite la vie de ROC-AUX
       (T (roc-aux n nil m nil pvd))
      )
 )
```

dans (roc-aux n nil m nil pvd) le premier nil = ce qui à déjà été parcouru dans la mémoire, c'est le AUX de roc-aux, le second NIL est un emplacement pour
retenir le programme N-1 au cas où on ne trouve pas le programme N, à fin de
ne pas devoir reparcourir la mémoire pour trouver ce PN-1 à partir duquel on va créer PN.(ou DN-1)

```
(defun roc-aux (n aux m PvD-N-1 pvd)
  (cond
   ((null m)
```

```
 (cons (list n
            (eval (list pvd (list 'quote PvD-N-1)))) pas sûr
       aux)) m a été entièrement parcouru, PvD-N-1 a du être instancier.
            attention on elève les éxécutions arrêtées, mais cela ne touche
          ni MP, ni MD.
 ((equal n (caar m))
  (cons (list n (cadar m))
        (append aux (cdr m)))) on évite le parcours inutil d'un del-assoc
 ((equal (- n 1) (caar m))
  (roc-aux n
           (cons (car m) aux)
          (cdr m)
           (cadar m) on retient en passant le PvD-N-1
          pvd))
 (T (roc-aux n (cons (car m) aux) (cdr m) PvD-N-1 pvd))
 )
)
```

```
essai (next-panv et next-data proviennent de GEN (P&D).)
? (roc 1 nil 'next-panv)
((1 (LAMBDA (X) X)))
? (roc 1 nil 'next-data)
((1 'EQUAL))
? (roc 2 '((1 (LAMBDA (X) X))) 'next-panv)
((2 (LAMBDA (T56) 'EQUAL)) (1 (LAMBDA (X) X)))
? (roc 1 '((1 (LAMBDA (X) X))) 'next-panv)
((1 (LAMBDA (X) X)))
? (roc 1 '((2 (LAMBDA (T54) 'EQUAL)) (1 (LAMBDA (X) X))) 'next-panv)
((1 (LAMBDA (X) X)) (2 (LAMBDA (T54) 'EQUAL)))

notons que (roc 3 nil "pvd") n'est pas censé fonctionner on utilise le fait
que NEXT-ME-MP-MD part de (roc 1 nil "pvd") si bien qu'il n'est nécessaire
de
créer un Pn seulement lorsque Pn-1 a déjà été créé.


remarquons que ROC fonctionne avec next-data, next-programme (next-
panv), mais
DU va fonctionner avec ROC parce que next-panv gère le processus de
renommage
des variables. Si on utilise DU sur un autre générateur de programmes, il
convient de s'assurer que ce générateur renomme les variables. On peut
aussi
modifier le dovetaileur de telle façon qu'il gère lui-même le renommage.
```

next-mémoire-programme-donnée
NEXT-ME-MP-MD

Next-me-mp-md doit utilise next-triple et s'inspire de next-memoire pour mémoriser les éxécutions de façon parsimonieuse. Il utilise ROC pour la gestion des mémoires de programmes et de données respectivement.

need GEN (P&D) (notamment next-programme (next-panv) et next-data)
need E-step, donc dove (et donc E, donc tout UT LI DO IS)

le me-mp-md initial sera ((nil) nil nil).

```
(defun next-me-mp-md (me-mp-md)
 (let* (                    n- et a- = nouveau et ancien resp.
       (a-me (select-me me-mp-md))
       (x (next-triple (car (select-me me-mp-md))))
       (n-mp (roc (car x) (select-mp me-mp-md) 'next-panv)) (mi à jo de mp)
       (n-md (roc (cadr x) (select-md me-mp-md) 'next-data)) (mi à jo de md)
      )
   mise à jour de me (qui est plus délicate) s'inspire de next-mémoire
   (cond ((nouvelle-éxécution x)
         (list (cons x (cons (a-step (cadar n-mp)
                                     (list (cadar n-md)) en) (car me-mp-md)))
               n-mp
               n-md))
        (T
          (list (cons x       ajout du triple
                  (cons              ajout de sa réalisation
                    (e-step (recherche-éxéc x a-me) en)
                       (del-dernière-éxécution x
                        cette dernière routine doit être modifier
                        à cause de la présence explicite des éxéc.
                                    a-me)))
               n-mp
               n-md))
        )
  )
 )
```

EEA inutile puisqu'on enlève le chemin, et il faut garder la toute dernière éxécution (de terminaison) de façon telle que lorsque le triple suivant est généré next-me-mp-md ne cherche pas l'éxécution suivante.

```
(defun recherche-éxéc (triple me)
  (cond ((null me) (print 'erreur))
        ((equal (list (car triple) (cadr triple))
                (list (caar me) (cadar me)))
        (cadr me))
        (t (recherche-éxéc triple (cddr me)))
      )
 )
```

remarquons que recherche-éxéc ne cherche pas la dernière éxécution, c-à-d il ne compare pas le 3ième élément du triple. Il tient compte du fait que next-me-mp-md nettoie me pour ne laisser que les dernières éxécutions (avec del-dernière-éxécution).

Ce programme est le même que celui utilisé dans next-mémoire à deux détails près : son nom (bien sûr) l'accent est mis sur dernière. (diff. de derniere) sa fonction ? enlève le triple ET le résultat (qui suit le triple),mais se plantera si une éxécution n'est pas une liste.
Rappelons que me est la même chose que mémoire avec les éxécutions placées derriere chaque triples. (à tester)

```
(defun del-dernière-éxécution (x me)  x = la dernière éxécution
                                      me = mémoire éxécution
  (cond
   ((null me) nil)
   ((equal (list (car x) (cadr x))
           (list (caar me) (cadar me)))
    (cddr me))
```

```
 (t (cons (car me) (cons (cadr me) (del-dernière-éxécution x (cddr me))))
   )
 )
)
```

dans EEA je suppose aussi que toutes les éxécutions sont des listes : il doit aussi enlever le triple précédent ! (ce n'est pas encore fait).

```
(defun EEA (me)
 (cond ((null me) nil)
       ((equal (caar me) 'val) (EEA (cdr me))) cf e-step
       (T (cons (car me) (EEA (cdr me))))
      )
 )
```

un dovetelleur universel : DU

<ne pas oublier renomme-var> : ici le générateur (next-panv) de programmes s'en occupe.

gestion des sorties

```
(defun print-me-mp-md (m) sans détails
  (print-triple (caar m))
  (print (cadar m))
  (terpri)
)


(defun print-triple (tr)
  (print (list 'programme (car tr)))
  (print (list 'donnee (cadr tr)))
  (print (list 'etape (caddr tr)))
  )


(defun print-me-mp-md-avec-détail (m)
  (print '-----------------------------------------------------)
  (print (list 'programme (car (caar m)) '******* (cadar (select-mp m))))
  (print (list 'donnee (cadr (caar m)) '******* (cadar (select-md m))))
  (print (list 'execution (caddr (caar m)) '******* (cadr (select-me m))))
  (print '-----------------------------------------------------)
  (terpri)
)


(defun print-me-mp-md-avec-tous-les-détails (m)
  (print '-----------------------------------------------------)
  (pprint m) (terpri)
  (print (list 'programme (car (caar m)) '******* (cadar (select-mp m))))
  (print (list 'donnee (cadr (caar m)) '******* (cadar (select-md m))))
  (print (list 'execution (caddr (caar m)) '******* (cadr (select-me m))))
  (print '-----------------------------------------------------)
  (terpri) (terpri)
)
```

DUs

```
(defun du ()
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (print-me-mp-md-avec-détail a)
       (go b))
)


(defun du-rapide ()
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (print-me-mp-md a)
       (go b))
)


(defun du-lent ()
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (print-me-mp-md-avec-tous-les-détails a)
       (go b))
)


(defun du-très-rapide ()
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (let ((me (select-me a)))
         (print (list (car me) '+++++++ (cadr me))))
       (go b))
)


(defun du-express ()
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (cond ((equal (caadar a) 'val) (go b)))
       (let ((me (select-me a)))
         (print (list (car me) '+++++++ (cadr me))))
       (go b))
)


(defun du-muet ()          ne sort rien (mais rêve-t-il ?)
 (prog (a)
       (setq a '((nil)()())) le me-mp-md initial ne sera pas imprimé
     b (setq a (next-me-mp-md a))
       (go b))
)
```

Programme à corrigé :

```
(defun du-omega ()
 (prog (a NdPL LdPT)
   (setq a '((nil)()())) ;le me-mp-md initial ne sera pas imprimé
   (setq NdPL 0)
   (setq LdPT nil)
  b (setq a (next-me-mp-md a))
   (cond ((nouvelle-éxécution (car (select-me a)))
      ;  (terpri)
        (setq NdPL (+ 1 NdPL))
       ; (print (list 'nd-d'executions-lancees '= NdPL))
        (go c)))
                  ;compte le Nombre de Programme Lancé.
   (cond ((equal (caadar a) 'val)
         (setq LdPT (gère-terminaison (select-me a) nil LdPT NdPL))
         (go b)))
                  ; Liste des Programmes Terminés
  c (print '------------------------------------------------------------)
   (print (list 'programme (car (caar a)) '******* (cadar (select-mp a))))
   (print (list 'donnee (cadr (caar a)) '******* (cadar (select-md a))))
   (print (list 'execution (caddr (caar a)) '******* (cadr (select-me a))))
   (print '------------------------------------------------------------)

   (go b)
 )
)

(defun gère-terminaison (me L-aux L NdPL)
 (cond ((null L)

     ;  (print (list 'approximation 'omega '= (+ 0.0 (/ (length L-aux) NdPL))))
       (print (list 'approximation 'omega '= (/ (length L-aux) NdPL)))
    (terpri)
   (terpri)
      (cons  (print (list (caar me) (cadar me)
                    'arrete 'en (- (caddar me) 1) 'etapes))  ;print ?
          L-aux))

     ((equal (list (caar me) (cadar me))
           (list (caar L) (cadar L)))
      (append L-aux L))
     (T (gère-terminaison me
                   (cons (list (caar L) (cadar L)) L-aux)
                  (cdr L)
                  NdPL))
    )
)
```

<u>éxécution</u>

```
? (du)

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 2 ******* (VAL EQUAL))
```

```
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 2 ******* 'CAR)
(EXECUTION 1 ******* 'CAR)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T0) 'EQUAL))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 3 ******* (VAL EQUAL))
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 2 ******* 'CAR)
(EXECUTION 2 ******* (VAL CAR))
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 3 ******* 'CDR)
(EXECUTION 1 ******* 'CDR)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T0) 'EQUAL))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 2 ******* (VAL EQUAL))
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T0) 'EQUAL))
(DONNEE 2 ******* 'CAR)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 3 ******* (LAMBDA (T1) 'CAR))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 1 ******* 'CAR)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 4 ******* (VAL EQUAL))
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 2 ******* 'CAR)
(EXECUTION 3 ******* (VAL CAR))
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 3 ******* 'CDR)
(EXECUTION 2 ******* (VAL CDR))
------------------------------------------------------------

------------------------------------------------------------
```

```
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 4 ******* 'X)
(EXECUTION 1 ******* 'X)
------------------------------------------------------------

------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T0) 'EQUAL))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 3 ******* (VAL EQUAL))
------------------------------------------------------------


Aborted
? (du-express)

((1 1 1) +++++++ 'EQUAL)
((1 2 1) +++++++ 'CAR)
((2 1 1) +++++++ 'EQUAL)
((1 3 1) +++++++ 'CDR)
((2 2 1) +++++++ 'EQUAL)
((3 1 1) +++++++ 'CAR)
((1 4 1) +++++++ 'X)
((2 3 1) +++++++ 'EQUAL)
((3 2 1) +++++++ 'CAR)
((4 1 1) +++++++ 'CDR)
((1 5 1) +++++++ 'QUOTE)
((2 4 1) +++++++ 'EQUAL)
((3 3 1) +++++++ 'CAR)
((4 2 1) +++++++ 'CDR)
((5 1 1) +++++++ (CAR 'EQUAL))
((1 6 1) +++++++ 'LAMBDA)
((2 5 1) +++++++ 'EQUAL)
((3 4 1) +++++++ 'CAR)
((4 3 1) +++++++ 'CDR)
((5 1 2) +++++++ ((VAL P1) 'EQUAL))
((5 2 1) +++++++ (CAR 'CAR))
((6 1 1) +++++++ (CDR 'EQUAL))
((1 7 1) +++++++ 'K)
((2 6 1) +++++++ 'EQUAL)
((3 5 1) +++++++ 'CAR)
((4 4 1) +++++++ 'CDR)
((5 1 3) +++++++ ((VAL P1) (VAL EQUAL)))
((5 2 2) +++++++ ((VAL P1) 'CAR))
((5 3 1) +++++++ (CAR 'CDR))
((6 1 2) +++++++ ((VAL P2) 'EQUAL))
((6 2 1) +++++++ (CDR 'CAR))
((7 1 1) +++++++ 'T7)
((1 8 1) +++++++ 'CONS)
((2 7 1) +++++++ 'EQUAL)

Aborted
? (du-omega)

------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------
(APPROXIMATION OMEGA = 0)
(NPRO-ARRETE 0)
(1 1 ARRETE EN 1 ETAPES)
------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 2 ******* 'CAR)
(EXECUTION 1 ******* 'CAR)
------------------------------------------------------------
------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T11) 'EQUAL))
(DONNEE 1 ******* 'EQUAL)
```

```
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------
(APPROXIMATION OMEGA = 1/3)
(NPRO-ARRETE 1)
(1 2 ARRETE EN 1 ETAPES)
------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 3 ******* 'CDR)
(EXECUTION 1 ******* 'CDR)
------------------------------------------------------------
(APPROXIMATION OMEGA = 1/2)
(NPRO-ARRETE 2)
(2 1 ARRETE EN 1 ETAPES)
------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T11) 'EQUAL))
(DONNEE 2 ******* 'CAR)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------
------------------------------------------------------------
(PROGRAMME 3 ******* (LAMBDA (T12) 'CAR))
(DONNEE 1 ******* 'EQUAL)
(EXECUTION 1 ******* 'CAR)
------------------------------------------------------------
(APPROXIMATION OMEGA = 1/2)
(NPRO-ARRETE 3)
(1 3 ARRETE EN 1 ETAPES)
------------------------------------------------------------
(PROGRAMME 1 ******* (LAMBDA (X) X))
(DONNEE 4 ******* 'X)
(EXECUTION 1 ******* 'X)
------------------------------------------------------------
(APPROXIMATION OMEGA = 4/7)
(NPRO-ARRETE 4)
(2 2 ARRETE EN 1 ETAPES)
------------------------------------------------------------
(PROGRAMME 2 ******* (LAMBDA (T11) 'EQUAL))
(DONNEE 3 ******* 'CDR)
(EXECUTION 1 ******* 'EQUAL)
------------------------------------------------------------
(APPROXIMATION OMEGA = 5/8)
(NPRO-ARRETE 5)
(3 1 ARRETE EN 1 ETAPES)
------------------------------------------------------------
```