4.1 ANNEXE Φ-LISP & Φ-DOVE

4.1.1 Introduction

Cette annexe décrit un interprète LISP métacirculaire, c-à-d écrit en LISP (en *Little Lisper* plutôt, voir Friedman 1974). Cet interprète, appelé Φ-DOVE, est constitué de telle façon qu'il permet une illustration aisée des notions introduites dans le texte. (Pour LISP, voir aussi Abelson & Susman 1985, Allen 1978).

Listes

NIL est une liste (la liste vide)
Si y est une liste CONS (x, y) est une liste

x représentant pratiquemment *n'importe quoi,* par exemple

cons (nil, nil) est une liste, que j'appelle Alfred
cons (AAA, cons (BBBB, cons (cons 324, nil), nil) nil) est une liste que j'appelle Arthur.

On note traditionnellement NIl : (), et (cons a NIL) : (a)
ainsi Alfred est notée (()), et Arthur : (AAA BBBB ((324))). En pratique on utilise l'opérateur CONS avec l'axiome (CONS x NIL) donne (x)

Deux opérations primitives permettent la décomposition d'une liste, CAR et CDR.
CAR appliquée à une liste donne le premier élément. Par exemple CAR(Arthur) donne AAA.
CDR appliquée à une liste donne la liste sans son premier élément, par exemple CDR(Arthur) donne (BBBB ((324))).

On a aussi :

CAR (cons (x, y)) donne x
CDR (cons (x, y)) donne y

ni CAR, ni CDR ne sont définit sur NIL.

Φ-DOVE

Φ-DOVE est un langage de programmation dialecte de LISP. J'ai construit Φ-DOVE pour traduire facilement les preuves (pas nécessairement constructives) apparaissant en théorie de la récursion. Il permet l'identification aisée des programmes avec les nombres naturels :

$\phi_i(x)$ correspond à (i x)

sans qu'il faille passer *stricto-sensu* par les nombres naturels proprement dit. La métaprogrammation, comme dans l'expression

$\phi\phi_j(x)(y)$

correspond, en Φ-DOVE, à parenthéser à gauche :

((j x) y)

Φ-DOVE admet une double implémentation : Φ-LISP et Φ-DOVE proprement dit. Φ-LISP est l'interprète métacirculaire. On peut le regarder comme une sémantique dénotationnelle de Φ-

DOVE. La partie "dove" est, en fait, une autre implémentation de Φ-LISP permettant l'appel et l'éxécution parallèle des programmes dans différents environnements possibles.

4.1.2 Davis 56 ou Davis 57 ?

Une fonction universelle $\lambda xy\ \phi_u(x,y)$ est essentiellement une sorte d'ordinateur. *x* correspond au code (indice) d'un programme, *y* correspond à la description des données, et *u* calcul, simule, ou émule l'application du programme *x* sur la donnée *y* pour donner le résultat $\phi_x(y)$.
Comme l'ensemble des fonctions partielles récursives $\{\phi_i\}$ contient des fonctions strictement partielles, ϕ_u n'est certainement pas une fonction totale. Néanmoins dans certaines situations, il est intéressant de regarder une fonction universelle comme une fonction totale. En particulier c'est ainsi que l'on peut considérer l'évaluateur en un pas. Ceci est proche de l'esprit de la première définition de machine universelle donnée par Davis en 1956. Nous ne suivons pas Rogers qui, dans son livre de 1967 (voir la bibliographie générale), utilise systématiquement le correctif que Davis a apporté lui-même à sa définition en 1957. Néanmoins la définition métacirculaire de Φ-DOVE, c'est à dire Φ-LISP, vérifie la définition de 57. Elle est utilisée pour des raisons d'efficacité et présentée pour des raisons de lisibilité. Φ-DOVE correspond à la définition plus général de 1956. Φ-LISP et Φ-DOVE ont la même sémantique dénotationnel, et la même sémantique opérationnelle aussi longtemps que Φ-DOVE est utilisé de façon purement itérative (aux bugs près!).

4.1.3 Identification des nombres et des programmes

Nous voulons pouvoir interpréter *(i x)* comme $\phi_i(x)$ Il aurait été facile, mais inefficace, de construire une fonction universelle appliquant l'indice naturel, c-à-d le numéro d'ordre de ϕ_i dans l'énumération des programmes LISP $\{\phi_i\}$ sur la description codée aussi avec des indices naturels de la donnée *x*. Outre l'inefficacité due aux faits qu'à chaque appel de programme sur des données il faille générer les programmes et les données, le code serait devenu illisible, du genre :

(34546789443762134 556545455544456666776655099123456 78)

Ce qui facilite et repose l'esprit du théoricien et du philosophe complique considérablement la tâche du praticien et de l'illustrateur. Mais la théorie de la récursion et en particulier le théorème de récursion de Kleene marche sur des domaines variés, bien différents du domaine brut des nombres naturels. Des outils algébriques sont généralement utilisés pour établir proprement les correspondances. Ici la correspondance est capturée par une modification élémentaire de la fonction de substitution. Cela permet d'utiliser la primitive *quote* comme une indexation ou une Gödelisation. En effet une simple substitution modifie naturellement les quotes internes :

```
? (subst 'n 'a '(a a '(a a)))
(n n '(n n))
```

Comme (quote <programmme>) représente le nombre de Gödel (*liste* de Gödel dirons-nous), ses variables liées ne doivent pas être modifiées lors d'une manipulation par métaprogramme. On utilisera donc une substitution moins aveugle ne modifiant pas les quotes internes :

```
? (subst-sauf-quote 'n 'a '(a a '(a a)))
(n n '(a a))
```

Par exemple lorqu'on diagonalise un programme λxy f(s, x, y) où s représente un code, donc un programme *quoté*, une liste de Gödel, du genre "(quote (lambda ...)", on aimerait que les variables liées par lambda, et qui seraient éventuellement dénotées par "*x*" dans *s*, ne soient

pas traitées par la diagonale (ce qui aurait été automatique si *s* avait été représenté par un nombre naturel).

Voilà un code pour *subst-sauf-quote* :

```
(defun subst-sauf-quote (n a ll)
  (cond ((equal ll a) n)
        ((atom ll) ll)
        ((null ll) nil)
        ((equal (car ll) (quote quote)) ll)
        ((equal (car ll) a) (cons n
                                  (subst-sauf-quote n a (cdr ll))))
        ((atom (car ll)) (cons (car ll)
                               (subst-sauf-quote n a (cdr ll))))
        (t (cons (subst-sauf-quote n a (car ll))
                 (subst-sauf-quote n a (cdr ll))))
))
```

Subst-list-sauf-quote applique *subst-sauf-quote* sur tous les éléments d'une liste. J'aurais pu utiliser des opérateurs comme "*map*", mais pour des raisons de transportabilité et de lisibilité je persévère dans mon style *little lisper*.

```
(defun subst-list-sauf-quote (ln la ll)
  (cond ((equal ln nil) ll)
        (t (subst-sauf-quote (car ln)
                             (car la)
                             (subst-list-sauf-quote (cdr ln)
                                                    (cdr la)
                                                    ll)))
))
```

4.1.4 Sémantique de base : Φ-LISP

E(val) est l'évaluateur, c'est-à-dire ce qu'on peut regarder comme une sémantique dénotationnelle (présentée ici de façon informelle en pure LISP), à l'exception des effets de bord présents dans la gestion un peu sauvage de l'environnement.

```
(defun E (x env)
  (cond

        ((constante-atomique? x) x)
```

Les constantes sont *t*, *nil*[1], les nombres naturels et les primitives que l'on peut déclarer. *Constante-atomique?* est définie dans les utilitaires Φ-LISP plus loin.

```
        ((atom x) (valeur x env))
```

Un atome qui n'est pas une constante est considéré comme une abréviation pour ce qu'il dénote. L'environnement est une liste de couples. *Valeur* cherche la "valeur" de façon récursive.

```
? (valeur 'a '((b 3) (c 4) (a 2) (d 5)))
2
```

Le code est donné dans les utilitaires Φ-LISP.

Dans de nombreux LISP les nombres sont des atomes particuliers qui se dénotent eux-mêmes :

```
? 4
4
```

---

[1] nil dénote aussi bien la valeur booléenne FAUX que la liste vide. Un faiblesse (?) dénotationnelle qui a une valeur pragmatique et auxquel les lispers sont attachés.

de même on a

```
? (= '4 4)
t
```

En Φ-LISP, il en est de même pour deux sortes d'expression. Les lambda-expressions, c'est à-dire les programmes anonymes d'une part et les erreur-expressions qui servent à la gestion de l'erreur.

```
? (ini)
(lambda (x) (+ x y))
(ini (lambda (x) (+ x y)))
(erreur v b v x)
(ini (erreur v b v x))
```

de même :

```
(= (lambda (x) x) '(lambda (x) x))
(ini t)
```

Cependant les erreur-expressions et les lambda-expressions diffèrent profondément en regard de l'application A(pply) qui est donnée plus loin, si bien que :

```
(= (erreur x v c) '(erreur x v c))
(ini (erreur x v c))
(cond ((= (erreur x v c) (quote (erreur x v c))) (pp 'hello)))
(ini (erreur x v c))
```

En résumé

```
((constante-expression? x) x)
```

Voilà *quote,* une gödelisation particulièrement simple et efficace. Je rappelle que 'a est une abréviation pour (quote a)

```
((equal (car x) 'quote) (cadr x))
```

La conditionnelle mérite un traitement particulier

```
((equal (car x) 'cond) (E-cond (cdr x) env))
```

de même que l'itération

```
((equal (car x) 'proL) (E-proL (cdr x)))
```

Commandes utilitaires (effets de bord de haut niveau, voir leur rôle plus loin) :

```
((equal (car x) 'pp) (pprint (E (cadr x) env)))
((equal (car x) 'AE) (setq en (append (eval (cadr x)) env)))
((equal (car x) 'take) (E-list (eval (cadr x)) env))
((equal (car x) 'ENV) en)
((equal (car x) 'fprim) (setq prim (cons (cadr x) prim)))
((equal (car x) 'prim) prim)
((equal (car x) 'dprim) (setq prim (delete (cadr x) prim)))
```

Dans tous les autres cas, de la forme (x y z) on applique l'évaluation de x sur la liste des évaluations des éléments restants de la liste. C'est la ligne principale de l'évaluateur :

```
(t
    (A (E (car x) env) (E-list (cdr x) env) env))
```

On a terminé la conditionnelle que nous étions en train de définir :

```
)
```

En fait on a terminé la sémantique de Φ-LISP, à APPLY, E-COND et quelques utilitaires près :

```
)
```

Exercice. Evaluer en Φ-LISP l'expression ("""i ""i).

a) Conditionnelle Φ-LISP

Le code est plus clair que je ne pourrais l'être. Les seules difficultés présentes sont dues à la gestion de l'erreur.

```
(defun E-cond (x env)
  (cond ((null x) (list 'erreur 'cond))
        ((equal (car-atom (E (caar x) env)) 'erreur) (E
              (caar x) env))
        ((E (caar x) env) (E (cadar x) env))
        (t (E-cond (cdr x) env))
       )
  )
```

Car-atom est le car de LISP sinon qu'il donne l'atome sur lequelle on l'applique. Il est nécessaire pour gérer l'erreur :

```
(defun car-atom (x)
  (cond ((atom x) x)
        (t (car x)))
  )
```

b) Une machine universelle pour Φ-LISP, A(PPLY)

A est l'applicateur invoqué dans la ligne principale de l'évaluateur :

```
(defun A (m largev env)
  (cond


   ((member-erreur largev))
   ((numberp m) (list 'erreur m 'c-est-pas-un-nom-de-fct))
   ((atom m)
    (cond
     ((equal m 'p1)
      (cond
       ((atom (car largev))
        (list 'erreur 'p1 'refuse (car largev)))
           (t (caar largev))) )
     ((equal m 'p2)
      (cond ((atom (car largev))
             (list 'erreur 'p2 'refuse (car largev)))
            (t (cdar largev))) )
     ((equal m 'list) largev)
     ((equal m 'u) (a (car largev) (cadr largev) env))
     ((equal m 'e) (e (car largev) env))
     ((equal m 'def) (setq en (cons largev (del-assoc (car largev) en)))
                 (list 'voila (car en)))
```

```
    ((equal m 'del) (setq en (del-assoc (car largev) en)))
    ((primitive? m) (eval (cons m (mapquote largev))))

    (t (A (E m env) largev env)) ))

  (t
   (cond
    ((equal (car m) 'erreur) m)
    ((equal (car m) 'lambda)
     (E (subst-list-sauf-quote (mapquote largev)
                               (cadr m) (caddr m)) Env))
    (t (A (E m env) largev env))
    ))
  )
  )
```

c) Construction Iterative : Φ-LISP

Il s'agit essentiellement d'un interprète impératif auxiliaire, une sorte de FORTRAN qui a son charme, E-PROL :

```
(defun E-proL (L)
  (E-proL-aux L L)
)

(defun E-proL-aux (L L-global)
 (cond ((null L) nil)
       ((atom (car L)) (E-proL-aux (cdr L) L-global))
       ((equal (caar L) 'cond) (E-cond-proL (cdar L) L-global))
       ((equal (caar L) 'go)
        (E-proL-aux (à-partir-de (cadar L) L-global) L-global))
       ((equal (caar L) 'output) (E! (cadar L)))
       ((equal (car-t (E! (caar L))) 'erreur) (E! (caar L)))  ;car-t nec ?
       (t (E! (car L)) (E-proL-aux (cdr L) L-global ))
      )
 )

(defun E! (x)          ;  horrible!
  (E x en)
```

attention E-PROL *sort* toujours nil. (Il est purement itératif) :

```
(defun à-partir-de (a l)   ;après
 (cond ((null l) nil)
       ((equal (car l) a) (cdr l))
       (t (à-partir-de a (cdr l)))
)
)
```

d) La conditionnelle itérative Φ-LISP

```
(defun E-cond-proL (tests scope)
  (cond ((null tests) (print 'erreur-cond))
        ((erreur? (E! (caar tests))) (E! (caar tests)))
        ((E! (caar tests)) (E-proL-aux (cdar tests) scope))
        (t (E-cond-proL (cdr tests) scope))
       )
)
```

e) Utilitaire Φ-LISP

Voici *constante-atomique?*

```
(defun constante-atomique? (x)
 (or (numberp x)
    (equal x T)
    (equal x nil)
    (primitive? x)
 )

(defun constante-expression? (x)
 (or (equal (car x) 'lambda)
    (equal (car x) 'erreur)
 )
```

Une fonction est primitive si elle est déclarée comme telle avec la commande *fprim*, cela permet à Φ-DOVE d'utiliser les primitives où les programmes compilés qui existent sur le LISP interprétant Φ-DOVE.

```
(defun primitive? (x)
  (member x prim)
)

; primitives initiales :

(setq prim '(cons null or print list append reverse ini u listp
           A E read + * - U S K terpri p1 p2 atom equal pprint
            it-st it-st-val ini-st ini-st-val ini-st-n and subst
            ini-st-n-muette dovetelle dovetelle-muette isv isr
            ini-st-rec > <      ;defun pour monter sans problème (nécessite
                        ; un changement pour K-DOVE)
           val? E-step A-step
          is
           cadar def del
           fprim dprim)
 )
```

Attention : COMMON LISP n'est pas strict, Φ-LISP est strict (il évalue tous ses arguments), et le caractère non strict d'une primitive COMMON LISP n'est pas hérité en Φ-LISP avec l'utilisation de *fprim* (ex: (AND T (FOO)).

```
(defun valeur (x env)
   (cond ((null env)(list 'erreur x 'c-est-quoi-?))
         ((equal (caar env) x) (cadar env))
         (t (valeur x (cdr env)))
  )
  )


(defun member-erreur (l)
  (cond ((null l) nil)
        ((atom (car l)) (member-erreur (cdr l)))
        ((equal (caar l) 'erreur) (car l))
        (t (member-erreur (cdr l)))
       )
  )

(defun del-assoc (x l)
 (cond ((null l) nil)
       ((equal x (caar l)) (del-assoc x (cdr l)))
       (t (cons (car l) (del-assoc x (cdr l))))
  )
)
```

E-list évalue tous les éléments d'une liste :

```
(defun E-list (l env)
  (cond ((null l) nil)
        (t (cons (E (car l) env) (E-list (cdr l) env)))
       ) )
```

4.1.5 Sémantique de Base : Φ-DOVE

Φ-DOVE appelle itérativement un évaluateur par étape :

```
(defun E-step (x env)
  (E-step-aux  nil x env)
)

(defun E-step-aux (l f env)
  (cond ((null f) (a-step (car l) (cdr l) env))
        ((constante? f) (list 'val f))  ;ne traite pas NIL, --> A
        ((primitive? f) (list 'val f))
        ((val? f) f)
        ((atom f) (list 'val (valeur f env)))
        ((equal (car f) 'lambda) (list 'val f))
        ((equal (car f) 'quote) (list 'val (cadr f)))
        ((equal (car f) 'erreur) (list 'val f))
        ((equal (car f) 'cond) (E-step-cond l f env)) ;env à supprimer
        ((equal (car f) 'prol) (E-step-prol f))
        ((equal (car f) 'prol!) (E-step-prol! f)) ;usage interne !
        ((equal (car f) 'defun) (E f env))     ; pour aller au plus pressé
        ((equal (car f) 'prim) (E f env))      ; idem
        ((equal (car f) 'fprim) (E f env))     ; idem
        ((equal (car f) 'dprim) (E f env))     ; idem
        ((equal (car f) 'env) (list 'val en))
        ((and (listp (car f)) (equal (caar f) 'lambda)) ;(print 'jisuis)
         (subst-list-sauf-quote (cdr f) (cadar f) (caddar f)))  ;version eff.

     ;  (a-step (car f) (cdr f) env))  version pédagogique.

; attention
       ((or (constante? (car f))
            (primitive? (car f))
            (val? (car f)))
        (E-step-aux  (lcons (car f) l) (cdr f) env))
       (t (append l (list (E-step (car f) env)) (cdr f)))

      )
 )
```

a) Conditionnelle Φ-DOVE

```
(defun E-step-cond (l f env)
  (append l (E-step-cond-aux f env))
 )

(defun E-step-cond-aux (f env)
  (cond
    ((or (equal (resultat1 f) '(val T))
         (equal (resultat1 f) 'T)) (cadadr f))
    ((and (val? (resultat1 f)) (equal (caadr (resultat1 f)) 'erreur))
      ; (or  (equal (car (resultat1 f)) 'erreur) inutil car erreur appel "val"
     (resultat1 f))
      ; plus tard 2ème rôle de lambda (en lisp-i)
    ((equal (resultat1 f) '(val nil)) (cons 'cond (cddr f)))
    (t (cons 'cond (append (list (list (E-step (resultat1 f) env) (action1 f)))
                           (cddr f))))

      )
 )
```

b) Une *machine universelle* pour Φ-DOVE, A(PPLY)

```
(defun A-step (m largev env)
 (cond
  ((null m) (list 'val nil)) ; car E teste NULL avant CONSTANTE?
  ((val? m)
   (cond
    ((numberp (cadr m)) (cons 'val (list (list
                                 'erreur (cadr m) 'n-est-pas-une-fct))))
    ((primitive? (cadr m))
     (cons 'val (list
              (a (cadr m) (supprime-val largev) en))))  ;(eval (cons...
    ((atom (cadr m)) (cons (list 'val (valeur (cadr m) env))
                    largev))
    ((equal (car (cadr m)) 'quote) (cons (list 'val
                                    (cadr (cadr m)))
                                 largev))
    ((equal (car (cadr m)) 'erreur) m)
    ((lambda? (cadr m))  (subst-list-sauf-quote  largev
                                    (cadr (cadr m))
                                    (caddr (cadr m)) ))
    (t  (cons (cadr m) largev))))  ;ça y est
  (t (cond
      ((numberp m) (cons 'val (list (list
                                 'erreur m 'n-est-pas-une-fct))))
      ((primitive? m)
       (cons 'val (list (a m (supprime-val largev) en))))   ;idem
      ((atom m) (cons (list 'val (valeur m env))
                 largev))
      (t  (subst-list-sauf-quote  largev
                          (cadr m)
                          (caddr m))) ))
 )
)
```

c) Construction Itérative : Φ-DOVE

```
(defun E-step-prol (f)
  (setq mem-prol (cdr f))
  (cons 'prol! (cons '--> mem-prol))
 )

(defun E-step-prol! (f)
  (cond ((null (cadr f)) (list 'val nil))
        ((and (listp (cadr f))
              (output? (cadr f))) (cadadr f))
        ((erreur-val? (caddr f)) (caddr f))
        (t (cons 'prol! (esap (cdr f) mem-prol)))
       )
 )

(defun erreur-val? (x)
 (and (val? x) (erreur? (cadr x)))
 )

(defun arg-prol (f)
  (cdr f)
)

(defun E-step! (x)
  (E-step x en)
 )

(defun Esap (l lg)   ;E-step-AUX-prol
  (cond
   ((null l) (list 'val nil))
   ((equal (car l) nil) (list 'val nil))
   ((equal (car l) '-->)
```

```
  (cond ((atom (cadr l))
         (cons (cadr l)
               (cons '-->
                     (cddr l))))
        ((equal (caadr l) 'val) (cons '--> (cddr l)))
        ((erreur? (caadr l)) (caadr l))
        ((equal (caadr l) 'output) (list (list 'output (cadr (cadr l)))))
        ((equal (caadr l) 'def) (cons '-->
                                      (cons (E-step! (cadr l)) (cddr l))
                                      ))
;       ((primitive? (caadr l))
;        (E! (cadr l))
;             (cons (cadr l)
;                   (cons '-->
;                     (cddr l))))
        ((equal (caadr l) 'go)
         (append (delete '--> (avant (cadadr l) lg))
                 (list (cadadr l) '-->)
                 (delete '--> (après (cadadr l) lg))))
        ((equal (caadr l) 'cond)
         (cond ((or (equal (resultat1 (cadr l)) '(val T))
                    (equal (resultat1 (cadr l)) 'T)
                    (and (atom (resultat1 (cadr l)))
                         (not (null (resultat1 (cadr l))))))
                (cons '--> (action1-S (cadr l))))
               (t (cons '-->
                        (cons (E-step-cond-prol (cadr l)) (cddr l))))
               ))
        (t (cons '--> (cons (E-step! (cadr l)) (cddr l)))) ; 9 juil. 90
        ))
   (t (Esap (cdr l) lg))
  )
 )
```

d) Conditionnelle itérative Φ-DOVE

```
(defun E-step-cond-prol (f )
  (cond
    ; ((or (or (numberp (resultat1 f))
    ;          (and (not (null (resultat1 f))) (atom (resultat1 f)))) ; à simplifier
    ;    (equal (resultat1 f) '(val T))
    ;    (equal (resultat1 f) 'T))  (action1-S f))
    ((and (val? (resultat1 f)) (equal (caadr (resultat1 f)) 'erreur))
       ; (or  (equal (car (resultat1 f)) 'erreur) inutil car erreur appel "val"
      (resultat1 f))
       ; plus tard 2ème rôle de lambda (en lisp-i)
    ((or (equal (resultat1 f) '(val nil))
         (equal (resultat1 f) nil)) (cons 'cond (cddr f)))
    (t (cons 'cond
             (cons (cons (E-step (resultat1 f) en) (action1-S f))
                   (cddr f))))
  )
 )
```

e) Utilitaire Φ-DOVE

```
(defun supprime-val (ll)
  (cond ((null ll) nil)
        ((and (listp (car ll)) (equal (caar ll) 'val))
         (cons (cadar ll) (supprime-val (cdr ll))))
        (t (cons (car ll) (supprime-val (cdr ll))))
       )
  )

(defun action1 (f)
  (cadadr f)
  )
```

```
(defun resultat1 (f)
  (caadr f)
)

(defun val? (x)
  (and (listp x) (equal (car x) 'val))
  )

(defun output? (x)
  (and (listp x) (equal (car x) 'output))
  )

(defun erreur? (x)
 (and (listp x) (equal (car x) 'erreur))
 )

(defun lambda? (l)
  (and (listp l) (equal (car l) 'lambda))
)

(defun lambda-val? (l)
  (and (listp l) (equal (car l) 'val) (lambda? (cadr l)))
)

(defun argu-val (l)
  (argu (cadr l))
)

(defun body-val (l)
  (body (cadr l))
)

(defun action1-S (f)
  (cdadr f)
  )
```

4.1.6 Gestion des erreurs

```
(defun member-erreur (l)
  (cond ((null l) nil)
        ((atom (car l)) (member-erreur (cdr l)))
        ((equal (caar l) 'erreur) (car l))
        (t (member-erreur (cdr l)))
       )
  )
```

4.1.7 Boucles de haut niveau (INIs & ITs)

print-eval-read, interfaces utilisateurs, dovetelleurs particuliers.

; les INIs

```
(defun ini ()
  (prog (a)
   b (setq a (read))
     (cond ((equal a '(fini))  (go c))
           (t (pprint (list 'ini (E a en))) ))
     (terpri)
     (go b)
   c (print (list 'termine 'ini))
     ))

(defun ini-st ()
  (prog (aa bb)
    a (setq aa (read))
      (cond ((equal aa '(fini)) (go c)))
```

```
     (setq bb (E-step aa en))
    b (setq bb (E-step bb en))
     (pprint (sgv bb))
     (cond
      ((val? bb) (pprint (list 'resultat 'ini-st (cadr bb))) (terpri) (go a)))
     (go b)
    c (print (list 'c-est-termine 'ini-st))
   )
  )

(defun ini-st-val ()
  (prog (aa bb)
    a (setq aa (read))
      (cond ((equal aa '(fini)) (go c)))
      (setq bb (E-step aa en))
    b (setq bb (E-step bb en))
      (pprint bb)
      (cond
       ((val? bb) (pprint (list 'resultat 'ini-st-val (cadr bb))) (terpri) (go a)))
      (go b)
    c (print (list 'c-est-termine 'ini-st-val))
   )
  )

(defun ini-st-n ()
  (prog (aa bb n)
    a (setq aa (read))
      (setq n 0)
      (cond ((equal aa '(fini)) (go c)))
      (setq bb (E-step aa en))
    b (setq bb (E-step bb en))
      (setq n (+ 1 n))
      (pprint (sgv bb))
      (cond
       ((val? bb)  (pprint (list 'resultat 'ini-st-n (cadr bb))) (terpri) (print n)
                   (terpri) (go a)))
      (go b)
    c (print (list 'c-est-termine 'ini-st-n))
   )
  )


(defun ini-st-n-muette ()
  (prog (aa bb n)
    a (setq aa (read))
      (setq n 0)
      (cond ((equal aa '(fini)) (go c)))
      (setq bb (E-step aa en))
    b (setq bb (E-step bb en))
      (setq n (+ 1 n))
      (cond
       ((val? bb)
        (pprint (list 'resultat 'ini-st-n-muette (cadr bb))) (terpri) (print n)
        (terpri) (go a)))
      (go b)
    c (print (list 'c-est-termine 'ini-st-n-muette))
   )
  )

(defun ini-st-rec ()
  (prog () a  (print (list 'ini-st-rec (it-st-rec (read))))  (terpri)  (go a))
  )
; attention avec fini on descend de 2 niveaux évidemment! (et au niveau 0 ACL
; pour l'arrêter on ABORT.



;les IS & les IT-ST
```

```
(defun is (x)
  (it-st x)
)

(defun isv (x)
  (it-st-val x)
 )

(defun it-st (x)
 (prog (a c)
       (setq a x)
     b (setq c (read-char))
       (cond ((val? a) (print a) (print '********) (go c))
             ((equal c #\Space )(pprint (sgv a))(setq a (E-step a en))(go b))
             ((equal c #\q) (go c))
              (t (print '(barre d'espacement!...(ou "q" pour quitté)))(go b)))
      (go b)
     c (print 'termine-it-st))
)


(defun it-st-val (x)
 (prog (a c)
       (setq a x)
     b (setq c (read-char))
       (cond ((equal c #\Space )(setq a (E-step a en))(pprint a)(go b))
             ((equal c #\q) (go c))
              (t (print '(barre d'espacement!...(ou "q" pour quitté)))(go b)))
      (go b)
     c (print 'termine-it-st))
)

(defun sgv (l)
  (cond ((null l) nil)
        ((numberp l) l)
        ((atom l) l)
        ((atom (car l)) (cons (car l) (sgv (cdr l))))
        ((equal (caar l) 'val) (cons (cadar l) (sgv (cdr l))))
        (t (cons (sgv (car l)) (sgv (cdr l))))
       )
 )


(defun it-st-rec-aux (x n env)
 (cond ((val? x) (list 'nombre-de-steps n 'resultat (cadr x)))
       (t (it-st-rec-aux (E-step x env) (+ n 1) env))
      )
 )

(defun isr (x)
  (it-st-rec-aux x 0 en)
 )


(defun it-st-rec (x)
  (it-st-rec-aux x 0 en)
)
```

;les dovetelles

```
(defun dovetelle (x y)
 (prog (a b n n1)
   (setq a x)
   (setq b y)
   (setq n 0)
   a1 (cond ((val? a) (print (list 'resultat (cadr a))) (terpri) (go a3))
            ((val? b) (print (list 'resultat (cadr b))) (terpri) (go a4)))
```

```
    (setq a (E-step a en))
    (setq n (+ n 1))
    (print x) (pprint (sgv a))
    (setq b (E-step b en))
    (terpri)  (print '----------) (terpri)
    (print y) (pprint (sgv b))
    (terpri)  (print '----------) (terpri)
   (go a1)
 a3 (print (list x 'a 'gagne 'avec n 'étapes)) (setq n1 n)(terpri) (terpri)
    (print (list 'on 'regarde 'ce 'que 'donne y '? ))
    (cond ((equal (read) 'yes) (go a33))(t (go fin)))
 a4 (print (list y 'a 'gagne 'avec n 'étapes)) (setq n1 n)(terpri) (terpri)
    (print (list 'on 'regarde 'ce 'que 'donne x '? ))
    (cond ((equal (read) 'yes) (go a44))(t (go fin)))
a33 (setq b (E-step b en))
    (setq n (+ n 1))
    (pprint (sgv b))
    (cond ((val? b) (print (list 'resultat (cadr b)))
                (print (list 'nombre 'd 'etapes 'pour y '= n))
               (terpri) (go fin)))
   (go a33)
a44 (setq a (E-step a en))
    (setq n (+ n 1))
    (pprint (sgv a))
    (cond ((val? a) (print (list 'resultat (cadr a)))
                (print (list 'nombre 'd 'étapes 'pour x '= n))
               (terpri) (go fin)))
   (go a44)
 fin (terpri)(terpri)
    (print (list 'termine 'rapport '= (/ n n1))) (terpri)
    (print (list 'et 'la 'difference '= (- n n1)))
))

;ex (dovetelle '(fact 2) '(fact3 2) )

(defun dovetelle-muette (x y)
 (prog (a b n n1)
  (setq a x)
  (setq b y)
  (setq n 0.0)
  a1 (cond ((val? a) (print (list 'resultat (cadr a))) (terpri) (go a3))
           ((val? b) (print (list 'resultat (cadr b))) (terpri) (go a4)))
    (setq a (E-step a en))
    (setq n (+ n 1))
   ; (print x) (pprint (sgv a))
    (setq b (E-step b en))
   ; (terpri)  (print '----------) (terpri)
   ; (print y) (pprint (sgv b))
   ; (terpri)  (print '----------) (terpri)
    (go a1)
 a3 (print (list x 'a 'gagne 'avec n 'etapes)) (setq n1 n)(terpri) (terpri)
    (print (list 'on 'regarde 'ce 'que 'donne y '? ))
    (cond ((equal (read) 'yes) (go a33))(t (go fin)))
 a4 (print (list y 'a 'gagne 'avec n 'etapes)) (setq n1 n)(terpri) (terpri)
    (print (list 'on 'regarde 'ce 'que 'donne x '? ))
    (cond ((equal (read) 'yes) (go a44))(t (go fin)))
a33 (setq b (E-step b en))
    (setq n (+ n 1))
   ; (pprint (sgv b))
    (cond ((val? b) (print (list 'resultat (cadr b)))
                (print (list 'nombre 'd 'etapes 'pour y '= n))
               (terpri) (go fin)))
   (go a33)
a44 (setq a (E-step a en))
    (setq n (+ n 1))
   ; (pprint (sgv a))
    (cond ((val? a) (print (list 'resultat (cadr a)))
```

```
                    (print (list 'nombre 'd 'etapes 'pour x '= n))
                   (terpri) (go fin)))
     (go a44)
  fin (terpri)(terpri)
     (print (list 'termine 'rapport '= (/ n n1))) (terpri)
     (print (list 'et 'la 'difference '= (- n n1)))
))
```

; (defun filtre-continuation (c l) autant faire l'hybride.
; rend réellement efficace si on utilise "FPRIM"

; NEW IS

```
(defun ini-break-val ()
 (prog (a c)
    d (setq a (read))
      (cond ((equal a '(fini)) (go c)) )
         ;  (t (go b)))
    b (setq c (read-char))
      (cond ((equal c #\Space )
            (setq a (E-step a en))
            (cond ((val? a) (pprint a) (terpri)
                  (print 'termine)
                  (terpri) (go d))
                  (t (pprint a) (terpri) (terpri) (go b))))
           ((equal c #\q) (go c))
           ((equal c #\R)
           (prog (bb)
             (setq bb a)
            e (setq bb (E-step bb en))
            ; (pprint bb)
            (cond
             ((val? bb)
                (pprint (list 'resultat 'ini-st-val (cadr bb)))
                              (terpri) (go d))
             (t (go e))) ))
            (t (print '(barre d'espacement!...ou "q" pour quitter))
              (print '(ou R pour resumer... et ensuite return))
              (terpri) (terpri) (terpri)
             (go b)))
     (go b)
    c (print 'termine-ini-break-val))
)

(defun ini-break ()
 (prog (a c)
    d (setq a (read))
      (cond ((equal a '(fini)) (go c)) )
         ;  (t (go b)))
    b (setq c (read-char))
      (cond ((equal c #\Space )
            (setq a (E-step a en))
            (cond ((val? a) (pprint (sgv a)) (terpri)
                  (print 'termine)
                  (terpri) (go d))
                  (t (pprint (sgv a)) (terpri) (terpri)(go b))))
           ((equal c #\q) (go c))
           ((equal c #\R)
           (prog (bb)
             (setq bb a)
            e (setq bb (E-step bb en))
             ; (pprint (sgv bb))
            (cond
             ((val? bb)
                (pprint (list 'resultat 'ini-st-val (cadr bb)))
                              (terpri) (go d))
             (t (go e))) ))
            (t (print '(barre d'espacement!...ou "q" pour quitter))
```

```
            (print '(ou R pour resumer... et ensuite return))
            (terpri) (terpri) (terpri)
           (go b)))
    (go b)
   c (print 'termine-ini-break))
)
```

COMPLEXITÉ DE BLUM, ET FONCTION TRACE

```
(defun blum (e)
  (blum-aux e 0)
 )

(defun blum-aux (e n)
  (cond ((val? e) n)
        (t (blum-aux (print (e-step e en)) (+ n 1)))  ; option
     )
 )
; ce Blum sera inapproprié dans (cond ((> (blum expr) 10) (do ...))
; si expr ne s'arrête pas.


(defun blum-exec (e m)     ; permet de calculer les βi
  (blum-exec-aux e 0 m m e)
 )

(defun blum-exec-aux (e n m l mp)
  (cond ((val? e) n)
        ((= 0 m) (list 'beta mp '> l))
        (t (blum-exec-aux (e-step e en) (+ n 1) (- m 1) l mp))
     )
 )


; ==========================================

(defun trac (e)              ; (def 'trac '(lambda (e) (trac-aux e nil)))
  (trac-aux e nil)           ; + (fprim trac-aux)
 )

(defun trac-aux (e l)
  (cond ((val? e) l)
        (t (trac-aux (print (e-step e en)) (lcons (e-step e en) l)))
     )
 )

; même remarque

(defun trac-exec (e m)
  (trac-exec-aux e nil m m e)
 )

(defun trac-exec-aux (e trace m n mp)
  (cond ((val? e) trace)
        ((= 0 m) (append trace (list 'beta mp '> n)))
        (t (trac-exec-aux
            ; (print
                 (e-step e en)
            ; )
                 (lcons (e-step e en) trace)
                (- m 1)
                n
                mp))       )    )
```

; ex: (trac-exec '(fact 2) 10) aussi bien en common lisp, qu'avec F-lisp.

; ex (trac-exec '(infini) 50) marche avec (def 'infini '(lambda () (infini)))

(en Φ-Lisp)

SOLUTION 1

```
-------------------------------------------
(def 'trac '(lambda (e) (trac-aux e nil)))

(def 'trac-aux '(lambda (e l)
  (cond ((val? e) l)
        (t (trac-aux (print (e-step e (env))) (lcons (e-step e (env)) l)))
       ))
 )
-----------------------------------------------
(def 'trac-moi-F '(lambda (x) (trac (list x) )))
(def 'trac-moi (k trac-moi-F))

ainsi, + avec (fprim trac) (fprim lcons)
==========================================
```

SOLUTION 2

```
-------------------------------------------
(def 'trac-exec '(lambda (e m)
  (trac-exec-aux e nil m m e))
 )

(def 'trac-exec-aux '(lambda (e trace m n mp)
  (cond ((val? e) trace)
        ((= 0 m) (append trace (list (list 'beta '> n))))
        (t (trac-exec-aux  (print (e-step e (env)) )
                           (lcons (e-step e (env)) trace)
                           (- m 1) n mp))
     )
 )
 )
-------------------------------------------
(def 'trace-moi-F '(lambda (x n) (trac-exec  (list x n) n)))
(def 'trace-moi (k trace-moi-F))    avec (Blum '(trace-moi i) i)

ainsi, + avec (fprim trac-exec)
exemple :1 ère étape de (trace-moi 10)
(TRAC-EXEC (LIST (S '(LAMBDA (X N) (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                 (LIST '(LAMBDA (X N)
                          (TRAC-EXEC (LIST (S X (LIST X)) N) N))))
            10)
         10)

1 ère étape de (trac '(trace-moi 10))
(TRAC-EXEC (LIST (S '(LAMBDA (X N) (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                 (LIST '(LAMBDA (X N)
                          (TRAC-EXEC (LIST (S X (LIST X)) N) N))))
            10)
         10)


19 ième étape de (trac '(trace-moi 10)) :

(TRAC-EXEC (LIST (S '(LAMBDA (X N) (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                 (LIST '(LAMBDA (X N)
                          (TRAC-EXEC (LIST (S X (LIST X)) N) N))))
            10)
         10)
période = 19.

==========================================
```

SOLUTION 3

```
-----------------------------------------
(def 'descrip-trac '(lambda (x)
          (list 'lambda nil (list 'trac (list 'quote (list x))))))
(def 'D-T-Moi (k descrip-trac))
-----------------------------------------
=======================================
(def 'descrip-trac '(lambda (x)
(list 'lambda nil (list 'trac
                  (list 'quote (list x))))))

(def 'D-T-Moi (k descrip-trac))

(equal ((D-T-Moi)) (trac '(D-T-Moi)))

; (k descrip-trac) est un programme DTM qui appliqué au vide donne une
; description complète de sa trace, sous la forme d'un programme qui
; appliqué au vide donne cette trace (la trace de DTM).

; à l'inverse, le programme suivant, appliqué à un nombre n, donne
; directement le segment de longueur n de sa trace initiale.
=========================================


exemples :

(trace-moi i) fonctionne avec i < 7, et ne s'arrête pas sur i > 6.
 (trac-exec '(trace-moi 7) 7) s'arrête pourtant, mais
(trac-exec '(trace-moi 10) 10) ne s'arrête pas !!!, avec ini-break,
on s'aperçoit que cela est du au fait que trac-exec est primitive

ces deux suivants marchent
(def 'trace-moi-F-sans-variable '(lambda (x) (trac-exec (list x) 5)))
(def 'tr-moi (k trace-moi-F-sans-variable))

ces deux suivants ne s'arrêtent pas
(def 'trace-moi-F-sans-variable '(lambda (x) (trac-exec (list x) 10)))
(def 'tr-moi (k trace-moi-F-sans-variable))

=> avec trac-exec défini dans F-lisp



(def 'mystère-1 '(TRAC-EXEC (VAL
        ((LAMBDA (N)
           (TRAC-EXEC (LIST (S '(LAMBDA (X N)
                                  (TRAC-EXEC (LIST (S X (LIST X)) N) N))
                               (LIST '(LAMBDA
                                    (X N)
                                     (TRAC-EXEC
                                       (LIST (S X (LIST X)) N)
                                      N))))
                           N)
                      N))
         1))
        1)
 )

de 1 à 7, mystère s'arrête, après il diverge (même E-step diverge sur lui,
 c'est normal)

remarque : (Blum '(trace i)) donne 8 pour i = 1 à 6.
(Blum '(trace i)) diverge pour i > 6. mais blum-exec et trac-exec aussi car il
s'applique à mystère 8 (ou 7).

conclusion si on lui demande de se tracer lui-même avec un nombre de step
plus grand que le nombre de step qu'il utilise pour s'arrêter, il ne
sait plus s'arrêter.(!!).
```

```
si on utilise trac-exec définit en F-lisp, blum n'est plus constant, et
croît, si bien que (trace-moi n) donne des valeurs sur des segments plus grand
de sa trace. (blum '(trace-moi 1)) donne 34, (blum '(trace-moi 21)) donne 334.

|#

; ================================================
```

; utilitaire pour mesurer la période (et l'imprimer convenablement)

```
(defun new-member (a ll)
  (cond ((null ll) nil)
        ((equal a (cadar ll)) ll)
        (t (new-member a (cdr ll)))
       )
  )

(defun new-pprint (ll)
  (cond ((null ll) nil)
        (t (print (caar ll))
                (pprint (cadar ll)) (terpri)
                (new-pprint (cdr ll)) (terpri))
       )
  )

(defun periode-trace (l)
  (periode-trace-aux (list (list 1 (car l))) (cdr l) 1)
  )

(defun periode-trace-aux (l1 l2 n)
  (cond ((null l2) l1)
        ((new-member (car l2) l1)
         (append l1 (new-member (car l2) l1)  ))
      (t (periode-trace-aux (lcons (list (+ n 1) (car l2)) l1) (cdr l2) (+ n 1)))
       )
  )

(defun pptr (tr)     ; pprint de deux (!) périodes de la trace
  (new-pprint (periode-trace tr)))

;=======================================================
```

4.1.8 Environnement(s)

Pas le temps : on se contentera d'une gestion un peu barbare.

4.1.9 Applications : S & K

```
(defun usk ()
 (setq en (append  '(

(i (lambda (x) x))
(longueur length)
(liste list)

(f1 (lambda (x) (cons 'truc x)))

(u (lambda (x y)
  (e (cons x (mapquote y))) ) )

(mapquote (lambda (l)
  (cond ((null l) nil)
        (t (cons (list (quote quote) (p1 l))
                 (mapquote (p2 l)) )) )))

(enlève-premiers (lambda (n l)
  (cond ((=? n 0) l)
        (t (enlève-premiers(- n 1) (p2 l))) )) )
```

```
(premiers (lambda (n l)
  (cond ((=? n 0) nil)
        (t (cons (p1 l)
                 (premiers (- n 1) (p2 l)) )) )) )

(arguments(lambda (l) (p1 (p2 l)) ))

(corps (lambda (l)
  (p1 (p2 (p2 l))) ))

(substitue-arguments (lambda (ln la ll)   ; subst-list-sauf-quote
  (cond ((equal ln nil) ll)
        (t (subst-sauf-quote (car ln)
                             (car la)
                               (subst-list-sauf-quote (cdr ln)
                                                      (cdr la)
                                                      ll))))))


(subst-sauf-quote (lambda (n a ll)
  (cond ((equal ll a) n)
        ((null ll) nil)
        ((equal (car ll) (quote quote)) ll)
        ((equal (car ll) a) (cons n
                                  (subst-sauf-quote n a (cdr ll))))
        ((atom (car ll)) (cons (car ll)
                               (subst-sauf-quote n a (cdr ll))))
        (t (cons (subst-sauf-quote n a (car ll))
                 (subst-sauf-quote n a (cdr ll)))) )))

(s (lambda (programme donnée)
  (liste (quote lambda)
         (enlève-premiers (longueur donnée)
                          (arguments programme))
         (subst-list-sauf-quote (mapquote donnée)
                                (premiers (longueur donnée)
                                                    (arguments programme))
                                (corps programme))  )))

(diag (lambda (f)
  (list (quote lambda)
        (arguments f)
        (subst-sauf-quote
         (list (quote s)
               (car (arguments f))
               (list (quote list)
                     (car (arguments f))))
         (car (arguments f))
         (corps f)
        ) )))


(k (lambda (f)
  (s (diag f) (list (diag f))) ))


) en )
)
)
```

Exercice : Exprimer en Φ-DOVE, avec k, la différence entre "je sens que mes cheveux se dressent sur la tête" et "je sens mes cheveux se dresser sur la tête" cf Sellars 1965.

4.1.10 Fission & fusion (projet)

Fission dans le même environnement ;
Fission avec fission de l'environnement (cf dovetellage) ;
Fission hybride ;
Application : la duplication (avec délais nul, positif, ou négatif) ;
Fusion ad hoc ;
Fusion quantique.

4.1.11 Anonymisation

1) Prédicat de parité anonyme

```
(def 'F-pair-impair
    '(lambda (x y z)
       (cond ((equal y 1)
            (cond ((equal z 0) t)
                 (t ((x 2) (- z 1)) )))
           ((equal y 2)
            (cond ((equal z 0) nil)
                 (T ((x 1) (- z 1)) )))
           (t 'erreur))
      ))


  (def 'G-pair-impair
    '(lambda (x y) (s F-pair-impair (list x y))))


  (def 'E-pair-impair
    (k G-pair-impair))



  (def 'pair? '(lambda (x)
       ((e-pair-impair 1) x)))


))

#|
? (ini)

 (def 'F-pair-impair
    '(lambda (x y z)
       (cond ((equal y 1)
            (cond ((equal z 0) t)
                 (t ((x 2) (- z 1)) )))
           ((equal y 2)
            (cond ((equal z 0) nil)
                 (T ((x 1) (- z 1)) )))
           (t 'erreur))
      ))


(INI (VOILA (F-PAIR-IMPAIR (LAMBDA (X Y Z) (COND ((EQUAL Y 1) (COND ((EQUAL
Z 0)
 T) (T ((X 2) (- Z 1))))) ((EQUAL Y 2) (COND ((EQUAL Z 0) NIL) (T ((X 1) (- Z 1)
)))) (T 'ERREUR))))))

(def 'G-pair-impair
    '(lambda (x y) (s F-pair-impair (list x y))))
(INI (VOILA (G-PAIR-IMPAIR (LAMBDA (X Y) (S F-PAIR-IMPAIR (LIST X Y))))))

(def 'E-pair-impair
    (k G-pair-impair))
```

```
(INI (VOILA (E-PAIR-IMPAIR (LAMBDA (Y) (S F-PAIR-IMPAIR (LIST (S '(LAMBDA (X
Y)
(S F-PAIR-IMPAIR (LIST (S X (LIST X)) Y))) (LIST '(LAMBDA (X Y) (S F-PAIR-IMPAIR
 (LIST (S X (LIST X)) Y))))) Y))))))

(def 'pair? '(lambda (x)
       ((e-pair-impair 1) x)))
(INI (VOILA (PAIR? (LAMBDA (X) ((E-PAIR-IMPAIR 1) X)))))

(pair? 5)
(INI NIL)

(pair? 6)
(INI T)

(pp (e-pair-impair 1))

(LAMBDA (Z)
  (COND ((EQUAL '1 1)
        (COND ((EQUAL Z 0) T)
              (T
                ((('LAMBDA (Y)
                    (S F-PAIR-IMPAIR
                      (LIST (S '(LAMBDA (X Y)
                                  (S F-PAIR-IMPAIR (LIST (S X (LIST X)) Y)))
                            (LIST '(LAMBDA
                                     (X Y)
                                     (S
                                       F-PAIR-IMPAIR
                                        (LIST (S X (LIST X)) Y)))))
                            Y)))
                  2)
                  (- Z 1)))))
        ((EQUAL '1 2)
         (COND ((EQUAL Z 0) NIL)
               (T
                 ((('LAMBDA (Y)
                     (S F-PAIR-IMPAIR
                       (LIST (S '(LAMBDA (X Y)
                                   (S F-PAIR-IMPAIR (LIST (S X (LIST X)) Y)))
                             (LIST '(LAMBDA
                                      (X Y)
                                      (S
                                        F-PAIR-IMPAIR
                                         (LIST (S X (LIST X)) Y)))))
                             Y)))
                   1)
                   (- Z 1)))))
        (T 'ERREUR)))
```

2) Interprète anonyme

Ici je rend anonyme un petit interprète, c-à-d une boucle Eval/Apply. La difficulté supplémentaire est due au fait que, à la différence d'une boucle comme pair/impair, Eval et Apply n'ont pas le même nombre d'arguments :

eval (forme,Env) apply (machine, liste, Env)

```
(setq mini '(

; petit interpreteur destiné à être rendu anonyme

(def 'evalue '(lambda (exp env)
                (cond ((atome? exp) (valeur exp env))
                      (t (applique (evalue (p1 exp) env)
                                   (liste (evalue (p1 (p2 exp)) env)
                                    (evalue (p1 (p2 (p2 exp))) env))
                                   env)) ))  )
```

```
(def 'applique '(lambda (nom largev env)
                  (cond ((=? nom '+) (plus (p1 largev)
                                           (p1 (p2 largev))))
                        (t (applique (evalue nom env)
                                     largev
                                     env)) )) )

; petit environnement :

(def 'en '((plus +)(un 1)(deux 2)))


; anonymisation :

(def 'F-e-a '(lambda (x y d)
  (cond

( (equal y 1) ;on veut evalue

;(pp (subst '(x 1) 'evalue
;      (subst '(p1 d) 'exp
;        (subst '(p1 (p2 d)) 'env
;          (subst '(x 2) 'applique
;            (p1 (p2 (p2 evalue)))
;          )
;        )
;      )              ))

;ce qui donne:

(COND ((ATOME? (P1 D)) (VALEUR (P1 D) (P1 (P2 D))))
    (T
     ((X 2)
       (liste ((X 1) (liste (P1 (P1 D)) (P1 (P2 D))))
              (LISTE ((X 1) (liste (P1 (P2 (P1 D))) (P1 (P2 D))))
                     ((X 1) (liste (P1 (P2 (P2 (P1 D)))) (P1 (P2 D)) ))
                     (P1 (P2 D)) ))

          )))
             ) ;fin du equal

( (equal y 2) ; on veut applique

;(pp (subst '(x 1) 'evalue
;     (subst '(x 2) 'applique
;      (subst '(p1 d) 'nom  ;différent de exp !
;       (subst '(p1 (p2 d)) 'largev
;        (subst '(p1 (p2 (p2 d))) 'env
;         (p1 (p2 (p2 applique)))
;        )
;       )
;      )
;     )
;    )
;   )
; ce qui donne

 (COND ((=? (P1 D) '+) (PLUS (P1 (P1 (P2 D))) (P1 (P2 (P1 (P2 D))))))
     (T
      ((X 2) (liste ((X 1) (liste (P1 D) (P1 (P2 (P2 D)))))
                    (P1 (P2 D))
                    (P1 (P2 (P2 D)))) ) ))
                          ) ;fin du equal

(t 'ERREUR)) ;fin du premier cond
)) ;fin du lambda et du def
```

```
                                (def 'G-e-a '(lambda (x y) (s F-e-a (list x y))))

                                (def 'e-e-a  '(k G-e-a))


                                ; essai:

                                ;((e-e-a 1) (list 'plus en))
                                ;+
                                ;((e-e-a 1) (list '(plus un un) en))
                                ;2
                                ;((e-e-a 2) (list 'plus '(1 2) en))
                                ;3
                                ;(evalue '(plus (plus un un) (plus un deux)) en)
                                ;5
                                ;
                                ;((e-e-a 1) (list '(plus (plus un un) (plus un deux)) en))
                                ;5
                                ;
; le rendre moins redondant, simulation de LET :
                                ;
                                ; facile grâce à:
                                ;
                                ;((lambda (r) (r 2 4)) 'plus)
                                ;6
                                ;((lambda (r) (r (r 3 5) (r 2 8))) 'plus)
                                ;18

                                ;(def '++ '(lambda (x y) ((lambda (r) (r x y)) 'plus)))
                                ;OK
                                ;OK
                                ;(++ 4 5)
                                ;9


                                (def 'ev-petit
                                   '(lambda (exp env)
                                      ((lambda (r)
                                        (cond ((atome? exp) (valeur exp env))
                                          (t (applique (r (p1 exp) env)
                                                       (liste (r (p1 (p2 exp)) env)
                                                            (r(p1 (p2 (p2 exp))) env))
                                                         env)
                                    )  )) 'evalue) ))

                                ;(ev-petit '(plus un un) en)
                                ;2
                                ;(ev-petit '(plus (plus un deux) (plus (plus un un) deux)) en)
                                ;7

; à présent il suffit de réappliquer les substitutions adéquates sur
; ev-petit et ap-petit. Ecrire un programme qui fait ça, y compris l'uni-argumentarisation:
; donc : (evalue x y) -> (r (list x y))...
;
; une meilleur idée est d'opérer la lambda abstraction sur F-e-a, directement sur x:
; (...(x 2)...(x 1)...) -> ((lambda (r) (...(r 2)...(r 1)...) x).

                                (def 'F-e-a-p '(lambda (x y d)     ((lambda (r)
                                  (cond

                                ( (equal y 1) ;on veut évaluer
```

```
(COND ((ATOME? (P1 D)) (VALEUR (P1 D) (P1 (P2 D))))
    (T
     ((r 2)
       (liste ((r 1) (liste (P1 (P1 D)) (P1 (P2 D))))
            (LISTE ((r 1) (liste (P1 (P2 (P1 D))) (P1 (P2 D))))
                 ((r 1) (liste (P1 (P2 (P2 (P1 D)))) (P1 (P2 D)) ))
                (P1 (P2 D)) ))

          )))
             ) ;fin du equal

( (equal y 2) ; on veut appliquer


 (COND ((=? (P1 D) '+) (PLUS (P1 (P1 (P2 D))) (P1 (P2 (P1 (P2 D))))))
     (T
      ((r 2) (liste ((r 1) (liste (P1 D) (P1 (P2 (P2 D)))))
                (P1 (P2 D))
                (P1 (P2 (P2 D)))) ) ))
                      ) ;fin du equal

(t 'ERREUR)) ;fin du premier cond
) ;fin du lambda (r)
x
)
) ;fi du premier lambda
) ;fin def.

(def 'G-e-a-p '(lambda (x y) (s F-e-a-p (list x y))))

(def 'e-e-a-p  '(k G-e-a-p))

 ("p" pour petit).

essai:
((e-e-a-p 1) (list '(plus un un) en))
2
((e-e-a-p 1) (list '(plus (plus un un) (plus un deux)) en))
5
((e-e-a-p 2) (list 'plus
                '(1 2)
                en))
3
(pp (e-e-a 1))
(LAMBDA (D)
  (COND ((EQUAL '1 1)
        (COND ((ATOME? (P1 D)) (VALEUR (P1 D) (P1 (P2 D))))
            (T
              (('(LAMBDA (Y)
                  (S F-E-A
                     (LIST (S '(LAMBDA (X Y)
                                 (S F-E-A (LIST (S X (LIST X)) Y)))
                           (LIST '(LAMBDA
                                    (X Y)
                                      (S F-E-A (LIST (S X (LIST X)) Y)))))
                        Y)))
                2)
                (LISTE
                 (('(LAMBDA (Y)
                     (S F-E-A
                        (LIST (S '(LAMBDA (X Y)
                                    (S F-E-A (LIST (S X (LIST X)) Y)))
                              (LIST '(LAMBDA
                                       (X Y)
                                         (S F-E-A (LIST (S X (LIST X)) Y)))))
                           Y)))
                  1)
                  (LISTE (P1 (P1 D)) (P1 (P2 D))))
                 (LISTE
```

```
                ((('(LAMBDA (Y)
                   (S F-E-A
                      (LIST (S '(LAMBDA (X Y)
                                  (S F-E-A (LIST (S X (LIST X)) Y)))
                               (LIST '(LAMBDA
                                       (X Y)
                                       (S
                                        F-E-A
                                          (LIST (S X (LIST X)) Y)))))
                            Y)))
                 1)
                  (LISTE (P1 (P2 (P1 D))) (P1 (P2 D))))
                ((('(LAMBDA (Y)
                   (S F-E-A
                      (LIST (S '(LAMBDA (X Y)
                                  (S F-E-A (LIST (S X (LIST X)) Y)))
                               (LIST '(LAMBDA
                                       (X Y)
                                       (S
                                        F-E-A
                                          (LIST (S X (LIST X)) Y)))))
                            Y)))
                 1)
                  (LISTE (P1 (P2 (P2 (P1 D)))) (P1 (P2 D))))
                 (P1 (P2 D))))))))
          ((EQUAL '1 2)
           (COND ((=? (P1 D) '+) (PLUS (P1 (P1 (P2 D))) (P1 (P2 (P1 (P2 D))))))
                 (T
                  ((('(LAMBDA (Y)
                     (S F-E-A
                        (LIST (S '(LAMBDA (X Y)
                                    (S F-E-A (LIST (S X (LIST X)) Y)))
                                 (LIST '(LAMBDA
                                         (X Y)
                                           (S F-E-A (LIST (S X (LIST X)) Y)))))
                              Y)))
                   2)
                   (LISTE
                    ((('(LAMBDA (Y)
                       (S F-E-A
                          (LIST (S '(LAMBDA (X Y)
                                      (S F-E-A (LIST (S X (LIST X)) Y)))
                                   (LIST '(LAMBDA
                                           (X Y)
                                             (S F-E-A (LIST (S X (LIST X)) Y)))))
                                Y)))
                     1)
                      (LISTE (P1 D) (P1 (P2 (P2 D)))))
                     (P1 (P2 D)) (P1 (P2 (P2 D))))))))
          (T 'ERREUR)))
NIL
(pp (e-e-a-p 1))
(LAMBDA (D)
  ((LAMBDA (R)
     (COND ((EQUAL '1 1)
            (COND ((ATOME? (P1 D)) (VALEUR (P1 D) (P1 (P2 D))))
                  (T
                   ((R 2)
                     (LISTE ((R 1) (LISTE (P1 (P1 D)) (P1 (P2 D))))
                      (LISTE ((R 1) (LISTE (P1 (P2 (P1 D))) (P1 (P2 D))))
                       ((R 1) (LISTE (P1 (P2 (P2 (P1 D)))) (P1 (P2 D))))
                       (P1 (P2 D))))))))
           ((EQUAL '1 2)
            (COND ((=? (P1 D) '+)
                   (PLUS (P1 (P1 (P2 D))) (P1 (P2 (P1 (P2 D))))))
                  (T
                   ((R 2)
                     (LISTE ((R 1) (LISTE (P1 D) (P1 (P2 (P2 D))))) (P1 (P2 D))
                      (P1 (P2 (P2 D))))))))
```

```
        (T 'ERREUR)))
   '(LAMBDA (Y)
     (S F-E-A-P
       (LIST (S '(LAMBDA (X Y) (S F-E-A-P (LIST (S X (LIST X)) Y)))
             (LIST '(LAMBDA (X Y) (S F-E-A-P (LIST (S X (LIST X)) Y)))))
           Y)))))
NIL

attention fin de setq mini:
                                        ))
```

4.1.12 Diverses routines

```
(defun enlevedebut (n l)
  (cond ((equal n 0) l)
        (t (enlevedebut (- n 1) (cdr l)))
))

(defun npremier (n l)
  (cond ((equal n 0) nil)
        (t (cons (car l)
                 (npremier (- n 1) (cdr l))
        ))
))

(defun argu (l)
  (cadr l)
)

(defun body (l)
  (caddr l)
)
```

4.1.13 Bibliographie locale

ABELSON H., **SUSSMAN** G. J., **SUSSMAN** J., **1985**, *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge / McGraw-Hill Book Company, New York.

ALLEN J., **1978**, *Anatomy of Lisp*, McGraw-Hill Book Company, New York.

CLEAVE J.P., **1961**, *Creative Functions*, Zeitschr. f. math. Logik und Grundlagen d. Math. Bd. 7, pp. 205-212.

DAVIS M., **1956**, *A note on universal Turing machines*, Automata Studies, Annals of mathematics studies, no 34, pp. 167-175, Princeton, N.Y.

DAVIS M., **1957**, *The definition of universal Turing machines*, Proceedings of the American Mathematical Society, Vol 8, pp. 1125-1126.

FRIEDMAN D., **1974**, *The Little Lisper*, Science Research Associates, Palo Alto, California, Trade edition, with FELLEISEN M., The MIT Press, Cambridge, USA, 1987.

ROGERS H., **1958**, *Gödel Numbering of the Partial Recursive Functions*, Journal of Symbolic Logic, 23, pp. 331-341.

SELLARS W. S., **1965**, *The Identity Approach to the Mind-Body Problem*, Rev. Metaphysics 18, pp. 430-451.