Programmation récursive

Quatrième saison

Version 1.2

Sommaire

1. Spécification de deug-eval . . . 2
1.1. Introduction . . . 2
1.2. Spécification de la fonction `deug-eval` . . . 2
1.2.1. Cas des programmes « complets » . . . 2
1.2.2. Auto-évaluation . . . 3
1.3. Grammaire de `DEUG-Scheme` . . . 4
1.4. Structure générale du listing . . . 4
2. Utilitaires généraux . . . 5
3. Barrière syntaxique . . . 5
4. Implantation de l'évaluateur . . . 7
4.1. Implantation de la fonction `deug-eval` . . . 7
4.2. Implantation de la fonction `evaluation` . . . 7
4.2.1. Implantation de la fonction `alternative-eval` . . . 7
4.2.2. Implantation de la fonction `conditionnelle-eval` . . . 8
4.2.3. Implantation de la fonction `sequence-eval` . . . 8
4.2.4. Implantation de la fonction `application-eval` . . . 10
4.2.5. Implantation de la fonction `bloc-eval` . . . 10
4.2.6. Implantation de la fonction `corps-eval` . . . 10
5. Barrière d'interprétation . . . 11
5.1. Introduction . . . 12
5.2. Valeurs non fonctionnelles . . . 12
5.3. Valeurs fonctionnelles . . . 12
5.3.1. Spécification de la barrière d'abstraction . . . 12
5.3.2. Implantation de la barrière d'abstraction . . . 12
5.3.3. Barrière d'abstraction des primitives . . . 13
5.3.4. Barrière d'abstraction des fonctions définies par le programmeur . . . 15
6. Barrière d'abstraction des environnements . . . 16
6.1. État des lieux . . . 16
6.2. Notion de bloc d'activation . . . 19
6.3. Spécification de la barrière des environnements . . . 19
6.4. Implantation (via barrière d'abstraction de bas niveau) . . . 20
6.4.1. Définition de `variable-val` . . . 21
6.4.2. Définition de `env-enrichissement` . . . 21
6.4.3. Définition de `env-extension` . . . 24
6.4.4. Définition de `env-add-liaisons` . . . 24
6.5. Implantation barrière d'abstraction de bas niveau . . . 25
6.5.1. Rappel de la spécification . . . 25
6.5.2. Structure de données . . . 25
6.5.3. Définition des fonctions . . . 25
6.6. Implantation barrière d'abstraction des blocs d'activation . . . 25
6.6.1. Rappel de la spécification . . . 25
6.6.2. Structure de données . . . 25
6.6.3. Définitions des fonctions de la barrière d'abstraction . . . 26
6.7. Environnement initial . . . 27
6.7.1. Description des primitives . . . 27
6.7.2. Définition de la fonction `env-initial` . . . 27
7. Annexe : source de deug-eval . . . 28

1. Spécification de deug-eval

1.1. Introduction

L'évaluation convertit un texte en une valeur. Ainsi, le cœur d'un interprète Scheme, celui de *DrScheme* par exemple, est :

```
(display (eval (read)))
```

La fonction `display` permet d'imprimer une valeur quelconque :

;;; *display : Valeur -> Rien*
;;; *(display val) imprime la valeur val*

La fonction `eval` convertit une S-expression Scheme — représentant un programme — en sa valeur :

;;; *S-Expression/Programme/ -> Valeur*
;;; *ERREUR lorsque "prog" ne représente pas un programme Scheme valide*
;;; *(eval prog) rend la valeur du programme Scheme représenté par la*
;;; *S-expression "prog".*

La fonction `read` permet de lire une S-expression quelconque (symbole, nombre, liste, vecteur...). Elle convertit un flux de caractères (provenant d'un clavier, d'un fi chier) en une S-expression Scheme :

;;; *read : -> S-Expression*
;;; *(read) lit une S-expression tapée au clavier*

Insistons sur le rôle de cette fonction : il ne faut pas confondre le programme qui est une idée dans la tête du programmeur et ses diverses représentations. Pour un éditeur de texte, c'est une suite de caractères, pour le système d'exploitation un fi chier, pour Scheme la valeur produite par `read` c'est-à-dire une S-expression, pour un débogueur des octets liés par des pointeurs. En conclusion, un programme est la description d'un calcul qui peut revêtir différentes représentations suivant les outils dont on dispose et, pour nous écrivain d'un interprète, c'est une S-expression qui nous est fournie par la fonction `read` .

1.2. Spécification de la fonction `deug-eval`

Dans cette partie, nous voudrions (ré)écire la fonction `eval` de l'interprète sous la forme d'une fonction `deug-eval` qui a comme donnée une S-expression — représentant un programme — et qui retourne la valeur de cette S-expression. Ainsi :

```
(deug-eval '(+ 2 3)) → 5
```

et la fonction `deug-eval` a comme spécifi cation :

;;; *deug-eval: Deug-Programme -> Valeur*
;;; *(deug-eval p) rend la valeur du programme (de Deug-Scheme) «p».*

Le langage Scheme que nous analyserons ne sera pas un Scheme complet, ceci pour quatre raisons :

– notre objectif est de voir (tous) les mécanismes fondamentaux et cela ne servirait à rien de traiter un certain nombre de constructions Scheme car elles se traitent comme d'autres constructions que nous avons traitées ;
– nous avons vu en cours et en TD que des constructions de Scheme, très pratiques, ne sont pas indispensables car on peut les remplacer par d'autres (il en est ainsi du `and` , du `or` , du `cond` ...). Lorsque l'on écrit un évaluateur, plus le langage est petit, moins on a de choses à faire.
– c'est ainsi que cela se passe dans la réalité : on commence par ne traiter qu'une partie du langage et, ensuite, on complète l'interpréteur ;
– et enfi n, pour une raison technique : un programme Scheme – tel qu'il est défi ni par la carte de référence – est une suite de défi nitions et d'expressions, une défi nition ou une expression étant représentée par une S-expression. Ainsi un programme Scheme est constitué par un certain nombre de S-expressions et la fonction qui évalue un programme Scheme devrait avoir un nombre quelconque d'arguments. Comme nous ne savons pas faire, nous décidons que nos programmes seront réduit à une S-expression.

1.2.1. Cas des programmes « complets »

La restriction précédente n'est pas une restriction fondamentale : si nous avons un programme *P* entier – *i.e.* qui contient des défi nitions et des expressions — à évaluer, il suffi t de l'envelopper dans une forme `(let () P)` avant de

le soumettre à la fonction deug-eval . Par exemple, si nous voulons faire évaluer le programme suivant (qui comporte une définition et une expression) :

```
(define (f n)
   (if (= n 0)
       1
       (* n (f (- n 1)))))   ; fin définition de f
(f 3)
```

nous écrirons :

```
(deug-eval
   '(let ()
       (define (f n)
          (if (= n 0)
              1
              (* n (f (- n 1)))))   ; fin définition de f
       (f 3)))
```

1.2.2. Auto-évaluation

De plus en plus fort ! deug-eval étant écrit en Scheme et évaluant un programme Scheme, on peut le faire évaluer par lui-même. Par exemple, le source de deug-eval étant dans la fenêtre de définition et étant compilé, la fenêtre d'interaction peut contenir :

```
(deug-eval '(let ()
     ...
     ; ; ; deug-eval : Deug-Programme -> Valeur
     ; ; ; (deug-eval p) rend la valeur du programme
     ; ; ; (de Deug-Scheme) "p".
     (define (deug-eval p)
     ...
     (deug-eval '(+ 2 3))))
```

le source de deug-eval

Le deug-eval de la première ligne est la fonction définie dans la fenêtre de définition (et est donc évalué par DrScheme) alors que le deug-eval de la dernière ligne est la fonction définie dans le source de deug-eval . Le second deug-eval est donc évalué par la définition interne de deug-eval qui est elle-même évaluée par le premier deug-eval . On parle d'**auto-évaluation**.

Remarque très importante : nous avons dit que nous ne traiterions pas toutes les constructions de Scheme. Mais, si l'on veut pouvoir faire de l'auto-évaluation, il faut que le langage utilisé pour écrire deug-eval soit inclus dans le langage que deug-eval évalue (en pratique, on s'est arrangé pour que ces deux langages soient égaux).

Remarque non moins importante : dans toute cette partie, nous utiliserons deux langages Scheme : celui de la carte de référence et le langage pour lequel nous écrivons un interprète. Systématiquement (ou tout du moins nous essaierons !), nous nommerons *Scheme* le langage de la carte de référence et *DEUG-Scheme* celui qu'évalue deug-eval . Lors de l'auto-évaluation, nous avons même trois langages Scheme :

- celui de DrScheme qui évalue le premier deug-eval ,
- celui – c'est un Deug-Scheme – qui est évalué par le premier deug-eval (dans l'exemple précédent, tout ce qui est dans la boîte encadrée appartient à ce langage),
- et enfin – c'est aussi un Deug-Scheme, mais pas le même que le précédent puisqu'il n'est pas évalué par la même fonction – celui qui est évalué par le second deug-eval (dans l'exemple précédent, `'(+ 2 3)` appartient à ce langage).

et encore, rien ne nous interdit de faire de l'auto-auto-évaluation : on aurait alors quatre niveaux de langages. Aussi nous parlerons de **Scheme sous-jacent** : le Scheme sous-jacent du Deug-Scheme évalué par le premier deug-eval est le Scheme de DrScheme et le Scheme sous-jacent du Deug-Scheme évalué par le second deug-eval est le Deug-Scheme évalué par le premier deug-eval .

Dernière remarque : l'auto-évaluation peut vous sembler un exercice de style. Il n'en est rien ! ne serait-ce parce qu'elle est excellente pour tester l'évaluateur. En effet, son écriture utilisant toutes les constructions du langage, il les

reste toutes, alors qu'il faudrait de nombreux tests « classiques » pour tout tester.

1.3. Grammaire de DEUG-Scheme

Un programme est écrit dans un langage de programmation qui fixe les règles grammaticales que tous les programmes écrits dans ce langage doivent respecter. La syntaxe de bas niveau de Scheme est rudimentaire (des blancs, des parenthèses et des mots sans oublier quelques règles d'abréviations concernant l'apostrophe). La grammaire de Scheme comporte des formes spéciales, des applications fonctionnelles, des variables et des citations. Ce faisant on plaque une interprétation sur les S-expressions qui sont lues.

La grammaire du sous-ensemble de Scheme utilisé dans ce cours est indiqué en tête du listing donné en annexe (page 28).

- Comme déjà indiqué, un programme *DEUG-Scheme* est simplement une expression *DEUG-Scheme* (il n'y en a qu'une et il n'y a pas de définition au top-level).
- Il n'y a pas de `and` , de `or` et de `let *`, ces constructions n'étant pas essentielles.
- En revanche, nous avons conservé le `cond` , qui n'est pas essentiel non plus, mais qui est tellement pratique pour écrire les sources de `deug-eval` (et nous l'avons donc mis dans le langage en vue de l'auto-évaluation).

1.4. Structure générale du listing

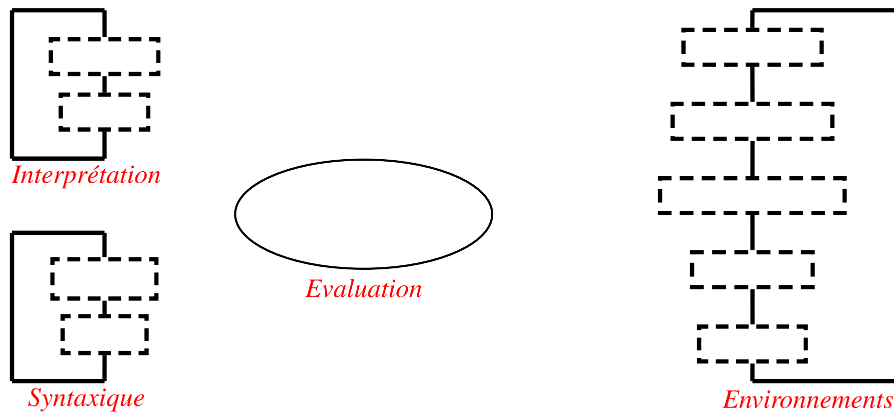
Tout d'abord, comme nous l'avons déjà dit, le listing contient (bien sûr sous forme de commentaires) la grammaire du langage utilisé :

```
5    ;;;; {{{ Grammaire du langage
32   ;;;; }}} Grammaire du langage
```

On trouve ensuite le source proprement dit. Dans un premier temps, nous définissons des fonctions de service utiles pour écrire le programme :

```
34   ;;;; {{{ Utilitaires généraux
100  ;;;; }}} Utilitaires généraux
```

Pour le programme proprement dit, comme pour les expressions booléennes que nous avons vues dans les cours précédents, nous écrirons une barrière syntaxique, une barrière d'interprétation et une barrière d'abstraction des environnements :



Il est facile de déterminer la barrière syntaxique (il suffit de regarder la grammaire), et nous le ferons tout de suite.

En revanche, pour la barrière d'évaluation et la barrière d'abstraction des environnements, il n'est pas aisé de trouver *a priori* les fonctions nécessaires, aussi nous reporterons cette étude plus tard.

Ainsi, nous donnerons d'abord la barrière syntaxique :

```
102  ;;;; {{{ Barrière-syntaxique
224  ;;;; }}} Barrière-syntaxique
```

qui nous permettra d'écrire l'évaluateur (la fonction deug-eval) :

```
226   ;;;; {{{ Evaluateur
350   ;;;; }}} Evaluateur
```

au dessus d'une barrière d'interprétation :

```
352   ;;;; {{{ Barrière-interpretation
475   ;;;; }}} Barrière-interpretation
```

et d'une barrière d'abstraction des environnements :

```
477   ;;;; {{{ Environnements-H (barrière de haut niveau)
677   ;;;; }}} Environnements-H (barrière de haut niveau)
```

Enfin, nous avons écrit, bien sûr sous forme de commentaires, le mode d'emploi de notre logiciel :

```
679   ;;;; {{{ Mode d'emploi
697   ;;;; }}} Mode d'emploi
```

2. Utilitaires généraux

Dans un premier temps, nous définissons des fonctions utilitaires classiques (`cadr` , `cddr` ... `length` , `member` ... qui ont presque toutes été vues dans le présent ouvrage, celles qui n'ont pas été vues étant très faciles. Deux fonctions, `deug-erreur` et `deug-map` , demandent tout de même une explication.

Fonction `deug-erreur`

Nous avons défini la fonction `deug-erreur` pour personaliser les messages d'erreurs de `deug-eval` (ils commencent tous par *deug-eval*) :

```
39 ;;;  Signaler une erreur et abandonner l'évaluation.
   (define (deug-erreur fn message donnee)
     (erreur 'deug-eval fn message donnee) )
```

Fonction `deug-map`

Nous avons défini une fonction `deug-map` (et non `map`) car, en Scheme, la fonction `map` opère sur un nombre quelconque de listes, la fonction appliquée étant une fonction n-aire (alors que nous l'avons toujours utilisée – et nous l'utiliserons encore ainsi dans le présent logiciel – avec une fonction unaire et une seule liste) :

```
75 ;;;  deug-map: (alpha -> beta) * LISTE[alpha] -> LISTE[beta]
   ;;;  (deug-map f L) rend la liste des valeurs de «f» appliquée aux termes
   ;;;  de la liste «L».
   (define (deug-map f L)
     (if (pair? L)
         (cons (f (car L)) (deug-map f (cdr L)))
         '() ) )
```

3. Barrière syntaxique

La barrière syntaxique est constituée d'une longue kyrielle de fonctions pour reconnaître (reconnaisseurs qui ont pour type de résultat `bool`) et extraire (accesseurs) de l'information des diverses formes syntaxiques possibles. Contrairement à ce que nous avons fait jusqu'alors, elles sont données règle de grammaire par règle de grammaire et non en écrivant d'abord les reconnaisseurs puis les accesseurs.

Règle `expression := ...` *(début)*

```
108  ; ; ; variable ? : Expression -> bool
124  ; ; ; citation ? : Expression -> bool
133  ; ; ; conditionnelle ? : Expression -> bool
137  ; ; ; conditionnelle-clauses : Conditionnelle -> LISTE[Clause]
141  ; ; ; alternative ? : Expression -> bool
145  ; ; ; alternative-condition : Alternative -> Expression
149  ; ; ; alternative-consequence : Alternative -> Expression
153  ; ; ; alternative-alternant : Alternative -> Expression
159  ; ; ; sequence ? : Expression -> bool
163  ; ; ; sequence-exps : Sequence -> LISTE[Expression]
```

Noter, encore une fois, comme ces spécifications suivent la grammaire. Pour l'implantation, rien à dire, hormis peut-être les implantations un peu longues de `variables?` (pour exclure les mots-clefs) et `citation?` (car nous avons mis ensemble les constantes et les données « quotées »).

Noter la définition de `alternative-alternant` : l'alternant est facultatif. Lorsque la condition est fausse et que l'alternant est absent, la fonction `alternative-alternant` retourne false, ce qui est permis par la norme de Scheme.

Règle `expression` := . . . *(suite et fin)*

```
167  ; ; ; bloc ? : Expression -> bool
171  ; ; ; bloc-liaisons : Bloc -> LISTE[Liaison]
175  ; ; ; bloc-corps : Bloc -> Corps
179  ; ; ; application ? : Expression -> bool
183  ; ; ; application-fonction : Application -> Expression
187  ; ; ; application-arguments : Application -> LISTE[Expression]
```

On suit toujours la règle de grammaire...

Règle `clause` := . . .

```
191  ; ; ; clause-condition : Clause -> Expression
195  ; ; ; clause-expressions : Clause -> LISTE[Expression]
```

Rien à dire...

Règle `liaison` := . . .

```
199  ; ; ; liaison-variable : Liaison -> Variable
203  ; ; ; liaison-exp : Liaison -> Expression
```

Rien à dire...

Règle `corps` := . . .

```
207  ; ; ; definition ? : Corps -> bool
     ; ; ; (definition ? corps) rend #t ssi le premier élément du corps
     ; ; ; "corps" est une définition
```

Ici, nous avons donné la spécification complète car c'est une forme de règle de grammaire que nous n'avons encore pas vue : un corps est une suite (éventuellement vide) de définitions suivie d'une suite (non vide) d'expressions. Deux applications de cette règle diffèrent donc – et c'est pourquoi nous donnons un reconnaisseur – selon que le premier élément est, ou n'est pas, une définition :

Règle `definition` := . . .

```
213  ; ; ; definition-nom-fonction : Definition -> Variable
217  ; ; ; definition-variables : Definition -> LISTE[Variable]
221  ; ; ; definition-corps : Definition -> Corps
```

Facile...

4. Implantation de l'évaluateur

4.1. Implantation de la fonction `deug-eval`

Comme dans le cas des expressions booléennes, on évalue une expression dans un environnement où des variables (nommant des fonctions ou des données) sont liées à leurs valeurs.

Lorsque vous avez écrit des programmes Scheme en TP, vous avez utilisé des fonctions comme `+`, `-`, `car` ... Vous étiez donc dans un certain environnement, l'environnement initial.

Ainsi la définition de `deug-eval` est :

```
229 ;;; deug-eval: Deug-Programme -> Valeur
    ;;; (deug-eval p) rend la valeur du programme (de Deug-Scheme) «p».
    (define (deug-eval p)
      (evaluation p (env-initial) ) )
```

avec :

```
234 ;;; evaluation: Expression * Environnement -> Valeur
    ;;; (evaluation exp env) rend la valeur de l'expression «exp» dans
    ;;; l'environnement «env».
624 ;;; env-initial: -> Environnement
    ;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
    ;;; contient toutes les primitives.
```

4.2. Implantation de la fonction `evaluation`

La définition de la fonction `evaluation` n'est qu'un aiguillage qui analyse la nature de l'expression à évaluer et appelle une fonction ad-hoc, avec comme arguments le contenu de la forme syntaxique (et non l'expression à analyser) :

```
237 (define (evaluation exp env)
      ;; (discrimine l'expression et invoque l'évaluateur spécialisé)
      (cond
       ((variable? exp)          (variable-val exp env))
       ((citation? exp)          (citation-val exp))
       ((alternative? exp)       (alternative-eval
                                  (alternative-condition exp)
                                  (alternative-consequence exp)
                                  (alternative-alternant exp) env))
       ((conditionnelle? exp)    (conditionnelle-eval
                                  (conditionnelle-clauses exp) env))
       ((sequence? exp)          (sequence-eval (sequence-exps exp) env))
       ((bloc? exp)              (bloc-eval (bloc-liaisons exp)
                                            (bloc-corps exp) env))
       ((application? exp)       (application-eval
                                  (application-fonction exp)
                                  (application-arguments exp) env))
       (else (deug-erreur 'evaluation "pas un programme" exp))) )
```

Ces différentes fonctions sont des évaluateurs spécialisés, dont nous étudions les définitions dans les paragraphes qui suivent, sauf

- la fonction `variable-val` qui devra être une des fonctions de la barrière des environnements,
- la fonction `citation-val` qui devra être une des fonctions de la barrière d'interprétation.

4.2.1. Implantation de la fonction `alternative-eval`

La définition de cette fonction ne devrait pas vous poser de problème :

```
256 ;;; alternative-eval: Expression3 * Environnement -> Valeur
    ;;; (alternative-eval condition consequence alternant env) rend la valeur de
    ;;; l'expression «(if condition consequence alternant)» dans l'environnement «env».
    (define (alternative-eval condition consequence alternant env)
      (if (evaluation condition env)
          (evaluation consequence env)
          (evaluation alternant env)) )
```

4.2.2. Implantation de la fonction `conditionnelle-eval`

Nous avons dit que la conditionnelle n'était pas essentielle (toute conditionnelle peut être réécrite avec des alternatives), mais que nous la mettions tout de même dans DEUG-Scheme car elle est très pratique (nous nous en sommes déjà servi dans la définition de `evaluation`).

Dans DEUG-Scheme, c'est la seule forme non essentielle. Les formes `and` et `or` ont été directement réécrites dans la définition de la fonction `deug-eval` plutôt que par programme (dans la première version de `deug-eval`, il y n'en y avait que quelques unes, dans les reconnaisseurs syntaxiques surtout).

En fait, dans les vrais évaluateurs, existe une phase dite de macro-expansion qui réduit le nombre de constructions qu'emploie un programme à celles qui sont vraiment essentielles et c'est ce que nous allons faire pour la conditionnelle :

```
264 ;;; LISTE[Clause] * Environnement -> Valeur
    ;;; (conditionnelle-eval clauses env) rend la valeur, dans l'environnement «env»,
    ;;; de l'expression «(cond c1 c2 ... cn)», «c1», «c2»... «cn» étant les éléments
    ;;; de la liste «clauses».
    (define (conditionnelle-eval clauses env)
      (evaluation (conditionnelle-expansion clauses) env) )
```

la fonction `conditionnelle-expansion` ayant comme spécification :

```
271 ;;; conditionnelle-expansion: LISTE[Clause] -> Expression
    ;;; (conditionnelle-expansion clauses) rend l'expression, écrite avec des
    ;;; alternatives, équivalente à l'expression «(cond c1 c2 ... cn)»,
    ;;; «c1», «c2»... «cn» étant les éléments de la liste «clauses».
```

Nous n'étudierons pas l'implantation de cette fonction car vous l'avez vue en TD.

4.2.3. Implantation de la fonction `sequence-eval`

```
289 ;;; sequence-eval: LISTE[Expression] * Environnement -> Valeur
    ;;; (sequence-eval exps env) rend la valeur, dans l'environnement «env», de
    ;;; l'expression «(begin e1 ... en)», «e1»... «en» étant les éléments de la liste
    ;;;  «exps».
    ;;; (Il faut évaluer tour à tour les expressions et rendre la valeur de la
    ;;;  dernière d'entre elles.)
```

Tout d'abord notons qu'une séquence peut être vide (i.e. l'expression `(begin)` est correcte). La norme ne précise pas ce que l'on doit rendre dans ce cas ; nous avons choisi de rendre `#f`.

Rappelons que pour évaluer `(begin e1 e2 ... en)`, il faut évaluer tour à tour les expressions `e1`, `e2` ... `en` et rendre la valeur de cette dernière. Ainsi la dernière expression doit être traitée de façon différente. Pour ce faire, pour l'efficacité, nous avons déjà vu sur d'autres exemples qu'il fallait définir une fonction interne, de même spécification que la fonction principale, mais dont la donnée est une liste non vide (et nous en profitons pour globaliser la varaible `env`) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        ... à faire
        ... à faire
        ... à faire
        ... à faire                                    )
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

Pour implanter la fonction sequence-eval+ , deux cas selon qu'il n'y a qu'une expression (c'est alors la dernière) ou plusieurs :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
          ... à faire
          ... à faire
          ... à faire                                  )
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

s'il n'y en a qu'une, il suffi t de l'évaluer (et de rendre sa valeur) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
          ... à faire
          ... à faire
            (evaluation (car exps) env)))
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

s'il y en a plusieurs, on évalue la première sans se soucier de sa valeur puis on évalue la séquence des expressions qui restent (le tout à l'aide d'une séquence) :

```
295 (define (sequence-eval exps env)
      ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
      ;; même fonction, sachant que la liste "exps" n'est pas vide et en globalisant la variable "env".
      (define (sequence-eval+ exps)
        (if (pair? (cdr exps))
            (begin (evaluation (car exps) env)
                   (sequence-eval+ (cdr exps)) )
            (evaluation (car exps) env)))
      ;; expression de (sequence-eval exps env) :
      (if (pair? exps)
          (sequence-eval+ exps)
          #f ) )
```

4.2.4. Implantation de la fonction application-eval

```
309 ;;; application-eval: Expression * LISTE[Expression] * Environnement -> Valeur
    ;;; (application-eval exp-fn arguments env) rend la valeur de l'invocation de
    ;;; l'expression «exp-fn» aux arguments «arguments» dans l'environnement «env».
```

Notons que `exp-fn` est une expression dont la valeur doit être fonctionnelle et que `arguments` est une liste d'expressions. Ainsi, on doit :

- évaluer la valeur de la fonction,
- évaluer les valeurs des arguments ; ceux-ci sont représentés par une liste, on pense donc à un map (rappelons que nous avons nommé `deug-map` cette fonction), mais ce n'est pas aussi simple car la fonction `evaluation` a deux arguments, l'expression mais aussi l'environnement : comme nous l'avons déjà fait dans ce cas, nous définirons une fonction interne,
- appliquer (on parlera ici d'invocation) la valeur de la fonction aux valeurs des arguments.

De plus, nous voudrions vérifier que la valeur de l'expression `exp-fn` est bien une valeur fonctionnelle.

D'où la définition :

```
312 (define (application-eval exp-fn arguments env)
      ;; eval-env : Expression -> Valeur
      ;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
      (define (eval-env exp)
        (evaluation exp env))
      ;; expression de (application-eval exp-fn arguments env) :
      (let ((f (evaluation exp-fn env)))
        (if (invocable? f)
            (invocation f (deug-map eval-env arguments))
            (deug-erreur 'application-eval
                         "pas une fonction" f ) ) ) )
```

Les fonctions `invocable?` et `invocation` doivent faire partie de la barrière d'interprétation :

```
;;; invocable?: Valeur -> bool
;;; invocation: Invocable * LISTE[Valeur] -> Valeur
```

- `(invocable? val)` rendant vrai si, et seulement si, `val` est une fonction (primitive ou définie par le programmeur),
- `(invocation f vals)` rendant la valeur de l'application de `f` aux éléments de `vals`.

4.2.5. Implantation de la fonction bloc-eval

```
324 ;;; bloc-eval: LISTE[Liaison] * Corps * Environnement -> Valeur
    ;;; (bloc-eval liaisons corps env) rend la valeur, dans l'environnement «env»,
    ;;; de l'expression «(let liaisons corps)».
```

Rappelons la sémantique du `let` : nous devons évaluer le corps dans un environnement obtenu en ajoutant les liaisons à l'environnement courant. D'où la définition :

```
327 (define (bloc-eval liaisons corps env)
      (corps-eval corps (env-add-liaisons liaisons env)) )
```

la fonction `env-add-liaisons` devant être une fonction de la barrière des environnements :

```
;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
```

`(env-add-liaisons liaisons env)` rendant l'environnement obtenu en ajoutant, à l'environnement `env`, les liaisons `liaisons`.

4.2.6. Implantation de la fonction corps-eval

```
330 ;;; corps-eval: Corps * Environnement -> Valeur
    ;;; (corps-eval corps env) rend la valeur de «corps» dans l'environnement «env»
```

Tout d'abord des rappels :

- un corps est une suite de définitions et d'expressions,
  - les définitions sont toutes écrites avant les expressions,

– il peut ne pas y avoir de définitions,
– il y a obligatoirement une expression,
– en fait les expressions constituent une séquence (implicite) ;
– pour évaluer un corps, on évalue la séquence du corps dans l'environnement obtenu en enrichissant l'environnement courant avec les définitions du corps.

D'où la définition :

```
332 (define (corps-eval corps env)
      (let ((def-exp (corps-separation-defs-exps corps)))
        (let ((defs (car def-exp))
              (exp (cadr def-exp)))
          (evaluation exp (env-enrichissement env defs)) ) ) )
```

la fonction `corps-separation-defs-exps` permettant de séparer les définitions et les expressions présentes dans le corps. Plus précisément :

```
338 ;;; corps-separation-defs-exps: Corps -> (LISTE[Definition] * LISTE[Expression])
    ;;; (corps-separation-defs-exps corps) rend une liste dont le premier élément est
    ;;; la liste des definitions du corps «corps» et les autres éléments sont les
    ;;; expressions de ce corps.
```

et la fonction `env-enrichissement` devant être une fonction de la barrière des environnements :

Dans barrière des environnements :

```
;;; env-enrichissement: Environnement * LISTE[Definition] -> Environnement
```

`(env-enrichissement env defs)` rendant l'environnement `env` étendu avec un bloc d'activation pour les définitions fonctionnelles `defs` .

Implantation de la fonction `corps-separation-defs-exps`

```
338 ;;; corps-separation-defs-exps: Corps -> (LISTE[Definition] * LISTE[Expression])
    ;;; (corps-separation-defs-exps corps) rend une liste dont le premier élément est
    ;;; la liste des definitions du corps «corps» et les autres éléments sont les
    ;;; expressions de ce corps.
```

Noter bien le type du résultat de cette fonction : c'est une liste (hétérogène), le premier élément étant une liste de définitions et les autres éléments étant des expressions (les expressions du corps).

L'idée de la définition est simple :
– si le premier élément du corps est une définition, nous devons la rajouter dans la liste des définitions du corps privé de cette définition (d'où un appel récursif),
– si le premier élément n'est pas une définition, la liste des définitions est vide, liste vide que nous devons mettre devant la liste des expressions :

```
342 (define (corps-separation-defs-exps corps)
      (if (definition? (car corps))
          (let ((def-exp-cdr
                  (corps-separation-defs-exps (cdr corps))))
            (cons (cons (car corps)
                        (car def-exp-cdr))
                  (cdr def-exp-cdr)))
          (cons '() corps) ) )
```

Et nous avons fini d'écrire l'évaluateur, il ne nous reste plus qu'à implanter la barrière d'interprétation et la barrière des environnements.

5. Barrière d'interprétation

5.1. Introduction

Nous avons dit que nous devions manipuler deux sortes de valeurs, les valeurs non fonctionnemlles (les entiers, les booléens... les listes...) et les valeurs fonctionnelles et, dans l'évaluateur, nous avons utilisé des fonctions différentes pour ces deux types de valeurs. Nous retrouvons donc ces deux sortes de valeurs dans la barrière d'interprétation :

```
352  ;;;;   {{{ Barrière-interpretation
357  ;;;;       {{{ Valeurs-non-fonctionnelles
366  ;;;;       }}} Valeurs-non-fonctionnelles
368  ;;;;       {{{ Valeurs-fonctionnelles
474  ;;;;       }}} Valeurs-fonctionnelles
475  ;;;;   }}} Barrière-interpretation
```

5.2. Valeurs non fonctionnelles

Une seule fonction a été utile dans l'écriture de l'évaluateur :

```
360 ;;; citation-val: Citation -> Valeur/non fonctionelle/
    ;;; (citation-val cit) rend la valeur de la citation «cit».
```

Nous décidons d'implanter les valeurs non fonctionnelles par leur valeur dans le Scheme sous-jacent. La défi nition est alors très simple :

```
362 (define (citation-val cit)
      (if (pair? cit)
          (cadr cit)
          cit ) )
```

5.3. Valeurs fonctionnelles

5.3.1. Spécification de la barrière d'abstraction

Tout d'abord, récapitulons les fonctions (Scheme) que nous avons utilisées dans l'évaluateur pour manipuler les fonctions (de Deug-Scheme) :

```
;;; invocable?: Valeur -> bool
;;; (invocable? val) rend vrai ssi «val» est une fonction (primitive ou définie
;;; par le programmeur)
;;; invocation: Invocable * LISTE[Valeur] -> Valeur
;;; (invocation f vals) rend la valeur de l'application de «f» aux éléments de
;;; «vals».
```

Noter que nous devrons aussi créer des valeurs fonctionnelles (pour enrichir l'environnement et pour défi nir l'environnement initial).

5.3.2. Implantation de la barrière d'abstraction

En fait, les fonctions sont de deux natures très différentes, celles-ci pouvant être :

1. les fonctions défi nies (dans Deug-Scheme) par le programmeur,
2. les fonctions primitives, c'est-à-dire
   – celles qui sont fournies par le langage (comme +, car..),
   – que nous utilisons (ou non, mais toutes celles qui sont utilisées doivent être prédéfi nies si l'on veut pouvoir autoévaluer l'interprèteur) dans l'écriture de l'interprète,
   – qui, dans un interprète du commerce, sont codées en langage machine.

   Bien sûr, nous ne les coderons pas dans le langage machine, nous les coderons tout simplement en utilisant une fonction du Scheme sous-jacent, c'est-à-dire le Scheme dans lequel on écrit l'évaluateur (par exemple, DrScheme ou, lors de l'auto-évaluation, Deug-Scheme).

Aussi nous décidons d'implanter la barrière d'abstraction des valeurs fonctionnelles en créant, et utilisant, une barrière d'abstraction des primitives et une barrière d'abstraction des fonctions défi nies par le programmeur. Ainsi la structure de la partie du listing qui met en œuvre la barrière d'interprétation pour les valeurs fonctionnelles est :

```
368  ;;;;        {{{  Valeurs-fonctionnelles
     définition de invocable?
     définition de invocation
390  ;;;;            {{{  Primitives
437  ;;;;            }}}  Primitives
439  ;;;;            {{{  Fonctions-definies
473  ;;;;            }}}  Fonctions-definies
474  ;;;;        }}}  Valeurs-fonctionnelles
```

Implantation de `invocable?`

```
374  ;;;  invocable?: Valeur -> bool
     ;;;  (invocable? val) rend vrai ssi «val» est une fonction (primitive ou définie
     ;;;  par le programmeur)
     (define (invocable? val)
       (if (primitive? val)
           #t
           (fonction? val) ) )
```

Remarque : dans la première version de `deug-eval`, nous avions écrit :

```
(define (invocable? val)
  (or (primitive? val) (fonction? val) ) )
```

Nous avons transformé la définition pour ne pas avoir besoin de la forme `or`.

Implantation de `invocation`

```
382  ;;;  invocation: Invocable * LISTE[Valeur] -> Valeur
     ;;;  (invocation f vals) rend la valeur de l'application de «f» aux éléments de
     ;;;  «vals».
     (define (invocation f vals)
       (if (primitive? f)
           (primitive-invocation f vals)
           (fonction-invocation f vals) ) )
```

5.3.3. Barrière d'abstraction des primitives

Structure de données

Nous décidons d'implanter une primitive par un 4-uplet :

- le premier élément est le symbole *primitive* (pour reconnaître les primitives parmi les listes),
- le second élément est la fonction du Scheme sous-jacent qui implante la primitive,
- le troisième élément est un comparateur (= ou >=),
- le dernier élément est un entier naturel, ces deux derniers éléments permettant de spécifier l'arité de la primitive. Par exemple,
  - pour une primitive d'arité 2, le troisième élément est = et le quatrième élément est 2,
  - pour une primitive qui peut avoir 1, 2, 3... arguments, le troisième élément est >= et le quatrième élément est 1.

Par exemple, la primitive correspondant à `'+` est `(list '*primitive* + >= 1)`.

Implantation de `primitive?`

Pour savoir si une valeur est une primitive, il suffit de vérifier que c'est une liste ayant au moins un élément dont le premier élément est le symbole `*primitive*` :

```
400  ;;;  primitive?: Valeur -> bool
     ;;;  (primitive? val) rend vrai ssi «val» est une fonction primitive.
     (define (primitive? val)
       (if (pair? val)
           (equal? (car val) '*primitive*)
           #f) )
```

Implantation de `primitive-creation`

```
407 ;;; primitive-creation: N-UPLET[(Valeur... -> Valeur) (num * num -> bool) num]
    ;;;                    -> Primitive
    ;;; (primitive-creation f-c-n) rend la primitive implantée par la fonction (du
    ;;; Scheme sous-jacent) «f», le premier élément de «f-c-n», et dont l'arité est
    ;;; spécifiée par le comparateur «c», deuxième élément de «f-c-n» et l'entier «n»,
    ;;;  troisième élément de «f-c-n».
    (define (primitive-creation f-c-n)
      (cons '*primitive f-c-n) )
```

Spécification de `primitive-invocation`

La définition de `primitive-invocation` est un peu plus compliquée. Voyons tout d'abord sa spécification :

```
416 ;;; primitive-invocation: Primitive * LISTE[Valeur] -> Valeur
    ;;; (primitive-invocation p vals) rend la valeur de l'application de la
    ;;; primitive «p» aux éléments de «vals».
```

Idée pour l'implantation de `primitive-invocation`

Raisonnons sur un exemple en considérant l'évaluation (par DrScheme) de `(deug-eval '(+ 2 3))` :

`(deug-eval '(+ 2 3))` ⇒ ...

en nommant `env-ini` la valeur de l'environnement initial, en « faisant tourner à la main » notre évaluateur, au bout de quelques pas on obtient (comme exercice, vous pouvez vérifier cette assertion) :

```
... ⇒     (primitive-invocation (variable-val '+ env-ini)
                                '(2 3) env-ini)
```

et comme nous avons dit que la primitive correspondant à `'+` était implantée par la liste `(list '*primitive * + >= 1)` :

```
... ⇒     (primitive-invocation (list '*primitive * + >= 1)
                                '(2 3) env-ini)
```

et nous voulons obtenir

```
... ⇒     (+ 2 3)
```

Comment, à partir de `(list '*primitive * + >= 1)` (qui est la valeur de l'argument `primitive`) et `'(2 3)` (qui est la valeur de l'argument `vals`) obtenir `(+ 2 3)` ?

- pour retrouver la fonction `+`, c'est facile puisque c'est le second élément de la liste `(list '*primitive * + >= 1)`,
- pour obtenir 2 et 3 remarquons que nous ne pouvons pas écrire, comme corps de `primitive-invocation` , `((cadr primitive) vals)` puisque, sur l'exemple, cela donnerait `(+ '(2 3))` qui n'est pas le résultat recherché. Si on veut le faire, à la main, sur notre exemple, c'est facile : 2 (resp. 3) est le premier (resp. deuxième) élément de la liste `'(2 3)` et le calcul de `(primitive-invocation (list '*primitive * + >= 1) '(2 3) env-ini)` doit être :

```
... ⇒     ((cadr (list '*primitive * + >= 1)) (car '(2 3))
                                              (cadr '(2 3)))

... ⇒     (+ 2 3)
```

Mais, si cette extraction est facile au coup par coup et à la main, elle est plus délicate lorsque l'on veut écrire un programme qui l'effectue en général. En effet l'extraction dépend du nombre d'arguments de la primitive, autrement dit de la longueur de la liste `vals` .

Implantation de `primitive-invocation`

De plus, nous voudrions vérifier (une fois n'est pas coutume) que la primitive est invoquée avec un nombre d'argument correct.

```
419 (define (primitive-invocation primitive vals)
      (let ((n       (length vals))
            (f       (cadr primitive))
            (compare (caddr primitive))
            (arite   (cadddr primitive)))
        (if (compare n arite)
            (cond
             ((= n 0) (f))
             ((= n 1) (f (car vals)))
             ((= n 2) (f (car vals) (cadr vals)))
             ((= n 3) (f (car vals) (cadr vals) (caddr vals)))
             ((= n 4) (f (car vals) (cadr vals)
                         (caddr vals) (cadddr vals) ))
             (else
              (deug-erreur 'primitive-invocation
                           "limite implantation (arités quelconques < 5)"
                           vals)))
            (deug-erreur 'primitive-invocation "arité incorrecte" vals) ) ) )
```

- Tout d'abord, nous nommons le nombre d'arguments,
- ainsi que les informations issues de la représentation de la primitive : la fonction du Scheme sous-jacent, l'opération de comparaison et la valeur de l'arité ;
- après avoir vérifié que l'arité était correcte (en affichant éventuellement un message d'erreur), il ne reste plus qu'à appliquer la fonction du Scheme sous-jacent aux arguments et, pour ce faire, nous faisons un `cond` sur le nombre d'arguments.

Noter que, de part l'implantation, nous limitons les arités quelconques (à 4) ce qui est fait, dans certains interprètes Scheme (mais, bien sûr avec une valeur plus grande).

5.3.4. Barrière d'abstraction des fonctions définies par le programmeur

Pour déterminer la structure de données que nous utilisons pour les fonctions définies par le programmeur, considérons un « exemple » d'une définition de fonction :

```
(define (f v1 v2)
  corps)
```

et d'une d'application de cette fonction (`e1` et `e2` étant des expressions) :

```
(f e1 e2)
```

Lorsque nous avons étudié la sémantique des applications de fonctions, nous avons dit que l'on évaluait (par substitution) les arguments et que, ensuite, on remplaçait l'appel de fonction par le corps de la fonction en substituant aux variables les valeurs des arguments correspondant, la valeur des autres variables – fonctionnelles ou non fonctionnelles – étant pris dans l'environnement où est définie la fonction.

Dans l'implantation que nous avons donné de `deug-eval`, l'évaluateur spécialisé `application-eval` évalue les différents arguments. Dans notre exemple, supposons que `e1` est évalué en `a1` et `e2` est évalué en `a2`.

```
(f e1 e2)  ⇒  (f a1 a2)
```

Pour évaluer l'application `(f e1 e2)`, il ne reste donc plus qu'à évaluer le corps – de la définition de la fonction – en substituant aux variables – de la définition de la fonction – (`e1` et `e2`) les valeurs des arguments correspondant (`a1` et `a2`), dans l'environnement où est définie la fonction. Autrement dit, en terme d'environnement, on doit évaluer le corps de la définition dans l'environnement obtenu en ajoutant à l'environnement où est définie la fonction les liaisons `v1 -> a1` et `v2 -> a2` :

Structure de données

En résumé, pour pouvoir ensuite évaluer des applications de la fonction, nous devons connaître la liste des variables et le corps de sa définition ainsi que l'environnement où est définie la fonction. Nous décidons alors d'implanter les

fonctions définies par le programmeur par un 4-uplet :

– le premier élément est le symbole `*fonction*` (pour reconnaître les fonctions définies par le programmeur parmi les listes),
– le second élément est la liste des variables de la définition de la fonction,
– le troisième élément est le corps de la définition de la fonction,
– le quatrième élément est l'environnement où est définie la fonction.

Implantation de `fonction?`

Pour savoir si une valeur est une fonction définie par le programmeur, il suffit de vérifier que c'est une liste ayant au moins un élément dont le premier élément est le symbole `*fonction*` :

```
448 ;;; fonction?: Valeur -> bool
    ;;; (fonction? val) rend vrai ssi «val» est une fonction créée par le programmeur.
    (define (fonction? val)
      (if (pair? val)
          (equal? (car val) '*fonction*)
          #f ) )
```

Implantation de `fonction-creation`

L'implantation de `fonction-creation` va de soi :

```
464 ;;; fonction-creation: Definition * Environnement -> Fonction
    ;;; (fonction-creation definition env) rend la fonction définie par
    ;;; «definition» dans l'environnement «env».
    (define (fonction-creation definition env)
      (list '*fonction*
            (definition-variables definition)
            (definition-corps definition)
            env ) )
```

Implantation de `fonction-invocation`

Comme nous l'avons dit, pour évaluer une application de la fonction, il suffit d'évaluer le corps de la définition – qui est le troisième élément du 4-uplet qui mémorise la fonction – en étendant l'environnement – quatrième élément du 4-uplet – en liant les variables – second élément du 4-uplet – avec les arguments. D'où la définition :

```
455 ;;; fonction-invocation: Fonction * LISTE[Valeur] -> Valeur
    ;;; (fonction-invocation f vals) rend la valeur de l'application de
    ;;; la fonction définie par le programmeur «f» aux éléments de «vals».
    (define (fonction-invocation f vals)
      (let ((variables (cadr f))
            (corps (caddr f))
            (env (cadddr f)) )
        (corps-eval corps (env-extension env variables vals)) ) )
```

la fonction `env-extension` devant être une fonction de base des environnements avec comme spécification :

Dans barrière des environnements :

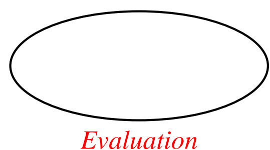
```
;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».
```

Et l'implantation de la barrière d'évaluation est terminée.

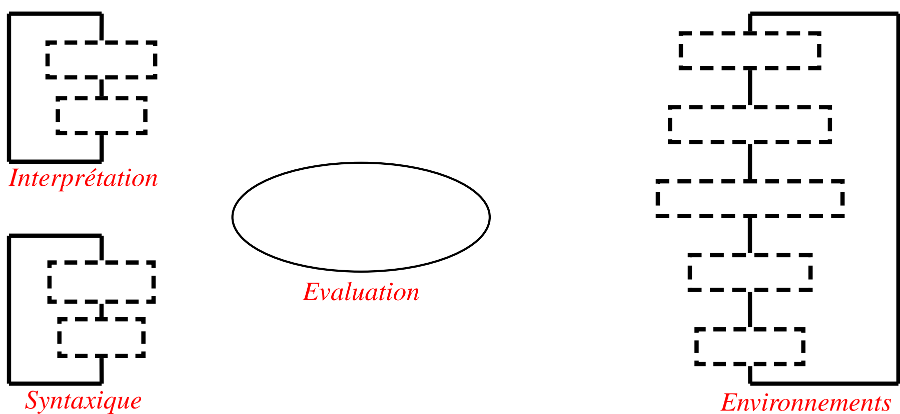
6. Barrière d'abstraction des environnements

6.1. État des lieux

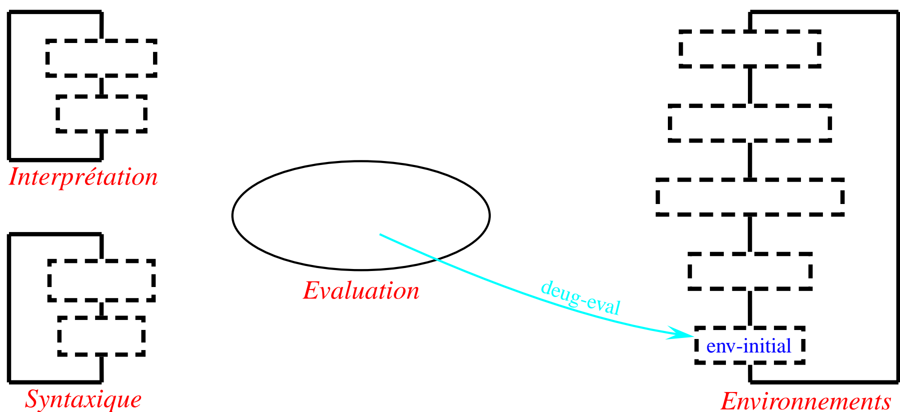
Avant d'étudier la barrière d'abstraction des environnements, nous voudrions schématiser la structure de l'évaluateur (en oubliant les fonctions de service). Nous voulions donc écrire un évaluateur :



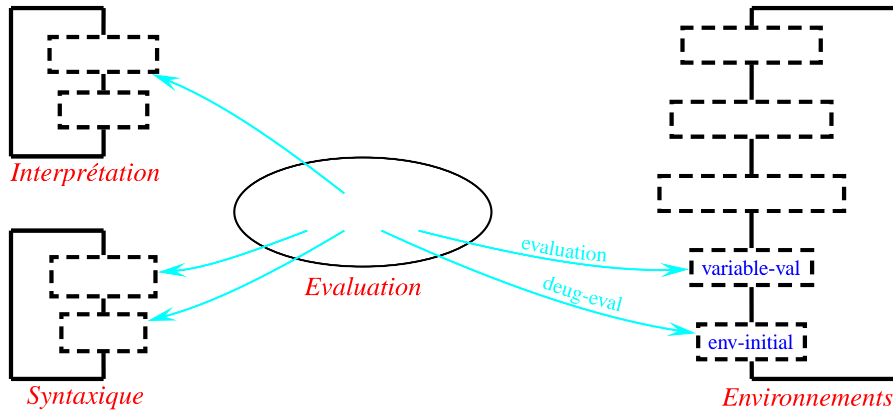
en sachant, dès le départ, qu’il faudrait une barrière syntaxique, que nous avons définie en premier, une barrière d’interprétation que nous venons de définir, et une barrière des environnements :



Après avoir défini la barrière syntaxique, nous avons défini la fonction `deug-eval` en utilisant la fonction `evaluation` et la fonction – de la barrière des environnements – `env-initial` :

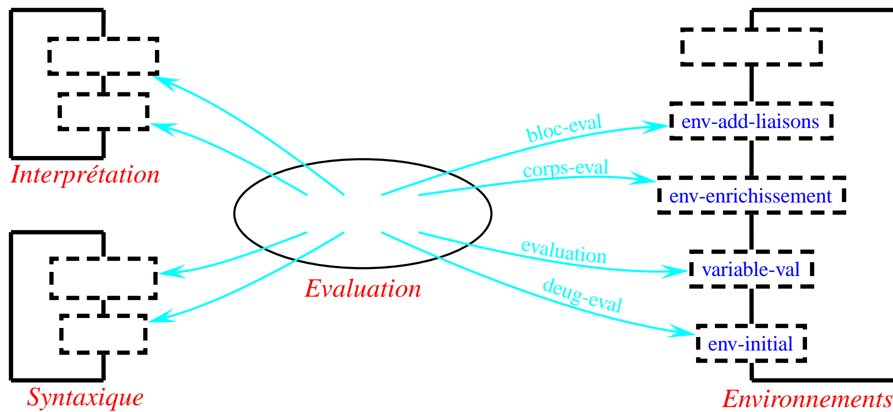


La définition de la fonction `evaluation` utilise les fonctions de la barrière d’abstraction, des évaluateurs spécialisés, la fonction – de la barrière d’interprétation – `citation-val` et la fonction – de la barrière des environnements – `variable-val` :

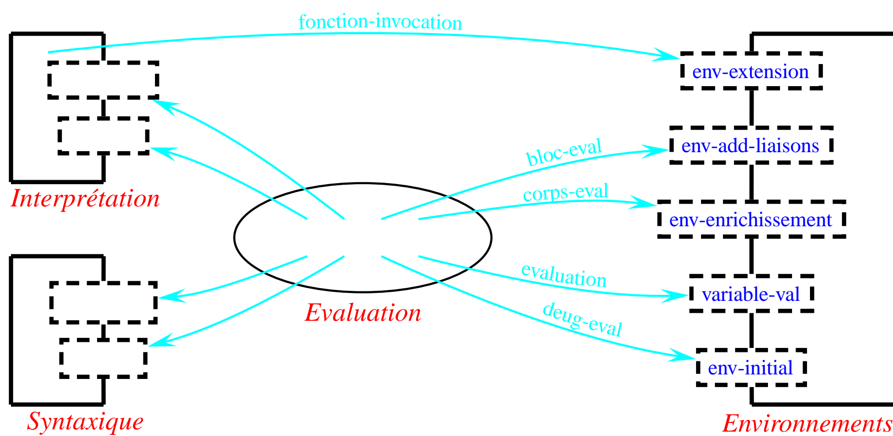


En écrivant les différents évaluateurs spécialisés, nous avons utilisé des fonctions de la barrière syntaxique et de la barrière d'interprétation ainsi que les fonctions suivantes de la barrière d'abstraction des environnements :

- la fonction `env-add-liaisons` lors de l'écriture de la fonction `bloc-eval`,
- la fonction `env-enrichissement` lors de l'écriture de la fonction `corps-eval` :



Enfi n, lorsque nous avons implanté la barrière d'interprétation, en écrivant la défi nition de la fonction `fonction-invocation`, nous avons utilisé la fonction `env-extension` de la barrière d'abstraction des environnements :



6.2. Notion de bloc d'activation

Considérons l'« exemple » suivant :

```
(let ((v1 exp1)
      (v2 exp2))
  (define (f1 x1 x2 x3)
    corps-f1)
  (define (f2 y1)
    corps-f2)
  exp-bloc)
```

a) Liaisons : ce bloc commence par des liaisons et nous devons étendre l'environnement courant en ajoutant deux associations variable – valeur.

b) Définitions fonctionnelles : nous avons ensuite deux définitions fonctionnelles et nous devons enrichir l'environnement avec deux associations variable – valeur fonctionnelle.

c) Application de fonction : enfin, dans `exp-bloc`, lors de l'application de `f1` (resp. `f2`), nous devons évaluer `corps-f1` (resp. `corps-f2`) dans l'environnement obtenu en étendant l'environnement mémorisé en liant les trois (resp. une) variables aux valeurs des trois (resp. un) arguments.

Ainsi, dans toutes ces étapes, nous devons ajouter à l'environnement courant un ensemble de couples d'associations variable – valeur : nous nommerons **bloc d'activation** un tel ensemble (noter que cette terminologie, classique en informatique, est due à l'implantation des fonctions).

6.3. Spécification de la barrière des environnements

Rappelons tout d'abord les spécifications des fonctions sur les environnements que nous avons utilisées, en commençant par deux fonctions qui vont de soi :

```
624 ;;; env-initial: -> Environnement
    ;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
    ;;; contient toutes les primitives.

480 ;;; variable-val: Variable * Environnement -> Valeur
    ;;; (variable-val var env) rend la valeur de la variable «var» dans
    ;;; l'environnement «env».
```

Précision

Considérons la définition suivante que nous avons déjà vue en cours (la fonction rend la liste obtenue en ajoutant l'élément donné à la fin de la liste donnée), définition où nous avions globalisé une variable :

```
(define (&d L x)
  (define (&d-x L)
    (if (pair? L)
        (cons (car L)
              (&d-x (cdr L)))
        (list x))) ; fin &d-x
  (&d-x L)) ; fin &d
```

Dans cette définition, de quels blocs d'activation sont extraits les valeurs des différentes occurrences des variables ?

– la valeur de `L` de la dernière ligne doit être trouvée dans le bloc d'activation créé par la fonction sous-jacente liée à `&d` (la fonction externe),
– la valeur de `L` des lignes 3, 4 et 5 doit être trouvée dans le bloc d'activation, ou plus exactement dans le dernier bloc d'activation (récursivité), créé par la fonction sous-jacente liée à `&d-x` (la fonction interne),
– la valeur de `x` de l'avant dernière ligne devrait, de même, être trouvée dans le bloc d'activation créé par la fonction sous-jacente liée à `&d-x` (la fonction interne), mais, comme cette variable n'est pas présente dans ce bloc d'activation (ce n'est pas une des variables de la fonction), on doit aller la rechercher dans un bloc créé précédemment, en l'occurence dans le bloc d'activation créé par la fonction sous-jacente liée à `&d` (la fonction externe).

Continuons notre revue des fonctions de la barrière d'abstraction des environnements en étudiant les trois dernières fonctions (elles posent plus de problèmes que les deux premières), en commençant par le cas relativement simple de l'ajout d'associations variable – valeurs. Nous avons utilisé deux fonctions :

```
492 ;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
    ;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
    ;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».
```

```
503 ;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
    ;;; (env-add-liaisons liaisons env) rend l'environnement obtenu en ajoutant,
    ;;; à l'environnement «env», les liaisons «liaisons».
```

Rappel : nous avons eu besoin de la fonction `env-extension` pour ajouter des liaisons variable — valeur lors de l'écriture de la définition de la fonction `fonction-creation` et nous avons eu besoin de la fonction `env-add-liaisons`) pour ajouter des liaisons variable — valeur lors de l'écriture de la définition de la fonction `bloc-eval` .

Remarquer que ces deux fonctions ont la même finalité mais qu'elles sont différentes de part leur signature : la fonction `env-extension` a comme données (autres que l'environnement) deux listes, la liste des variables et la liste des valeurs, alors que la fonction `env-add-liaisons` a comme donnée (autre que l'environnement) une seule liste, liste dont chaque élément est une association variable — valeur.
Enfin, nous avons eu besoin de la fonction `env-enrichissement` pour ajouter des définitions fonctionnelles lors de l'écriture de la définition de la fonction `corps-eval` – qui est appelée entre autres par `bloc-eval` :

```
516 ;;; env-enrichissement: Environnement * LISTE[Definition] -> Environnement
    ;;; (env-enrichissement env defs) rend l'environnement «env» étendu avec un
    ;;; bloc d'activation pour les définitions fonctionnelles «defs».
```

Précision : considérons l'« exemple » suivant d'un bloc où le corps possède deux définitions :

```
(let ()
  (define (f1 x)
    corps-f1)
  (define (f2 y)
    corps-f2)
  exp-bloc)
```

Pour évaluer le corps, on doit évaluer *exp-bloc* dans l'environnement obtenu en enrichissant l'environnement courant avec les deux fonctions `f1` et `f2` . Ces deux fonctions sont représentées par un 4-uplet formé du symbole `*fonction*`, de la liste des variables (i.e. `'(x)` – resp. `'(y)`), du corps de la fonction (i.e *corps-f1* – resp. *corps-f2*) et de l'environnement, `env` dans lequel on doit évaluer les applications de ces deux fonctions.

Mais quel est l'environnement `env` ? Question posée autrement : de quoi pouvons-nous nous servir dans le corps de `f1` ? Bien sûr, de l'environnement où est évalué le corps (*i.e.* celui où est évalué le bloc, plus les liaisons définies derrière le `let`), mais pas seulement : `f1` elle-même doit appartenir à l'environnement (cas de la récursivité), et aussi `f2` (et plus généralement, tous les noms des fonctions définies dans le corps).

Mais cela veut dire que lorsque nous évaluons la définition de `f1` , nous devons le faire en utilisant un environnement où il y a `f2` alors que nous ne savons même pas que `f2` existe ! Clairement, la possibilité d'utiliser quelque chose qui n'existe pas encore ne facilite pas l'écriture de l'évaluateur (de nombreux langages de programmation interdisent cela). On pourrait se dire que, pour simplifier cette écriture il suffit que, nous aussi, on l'interdise. Malheureusement, cette possibilité est absolument nécessaire si on veut pouvoir écrire des récursivités croisées (dans l'exemple si `f1` apparaît dans *corps-f2* et `f2` apparaît dans *corps-f1*). Or, par exemple, c'est exactement ce que nous avons – en plus complexe car il y en a de partout – dans le présent logiciel ; par exemple, `evaluation` appelle `alternative-eval` qui appelle `evaluation` .

6.4. Implantation (via barrière d'abstraction de bas niveau)

Afin que l'on puisse aller chercher une valeur (que ce soit de variable, que ce soit de fonction) dans un ajout antérieur, tout en donnant la priorité au derniers ajouts, on implante les environnements sous forme d'une suite de blocs d'activation, celui auquel on accède en premier étant celui qui est ajouté en dernier (en informatique, on parle de pile).

6.4.1. Définition de variable-val

L'idée de la définition de la fonction `variable-val` est alors très simple : on regarde si la variable est présente dans le premier bloc d'activation, auquel cas on va y chercher sa valeur et, si ce n'est pas le cas, on appelle récursivement la fonction sur le reste de l'environnement (*i.e.* l'environnement obtenu en supprimant le premier bloc d'activation). Et, bien sûr, pour que l'on puisse effectuer ce calcul, il faut qu'il y ait un bloc d'activation, autrement dit que l'environnement ne soit pas vide. D'où la définition :

```
480 ;;; variable-val: Variable * Environnement -> Valeur
    ;;; (variable-val var env) rend la valeur de la variable «var» dans
    ;;; l'environnement «env».
    (define (variable-val var env)
      (if (env-non-vide? env)
          (let ((bloc (env-1er-bloc env)))
            (let ((variables (blocActivation-variables bloc)))
              (if (member var variables)
                  (blocActivation-val bloc var)
                  (variable-val var (env-reste env)))))
          (deug-erreur 'variable-val "variable inconnue" var) ) )
```

sous réserve que l'on ait, dans la barrière d'abstraction de bas niveau, les fonctions suivantes :

```
540 ;;; env-non-vide?: Environnement -> bool
    ;;; (env-non-vide? env) rend #t ssi l'environnement «env» n'est pas vide

551 ;;; env-1er-bloc: Environnement -> BlocActivation
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-1er-bloc env) rend le premier (i.e. celui qui a été ajouté en
    ;;; dernier) bloc d'activation de l'environnement «env».

558 ;;; env-reste: Environnement -> Environnement
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-reste env) rend l'environnement obtenu en supprimant le premier
    ;;; bloc d'activation de l'environnement «env».
```

et, dans la barrière d'abstraction des blocs d'activation,les fonctions suivantes :

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»

576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
```

Pour les fonctions qui ajoutent des blocs d'activation, commençons par celle qui enrichit l'environnement avec des définitions fonctionnelles car, étant la plus complexe, c'est elle qui détermine les fonctions de base utilisées.

6.4.2. Définition de env-enrichissement

Rappels

Pour une définition de fonction, on doit lier le nom de la fonction à une fonction du Scheme sous-jacent, fonction qui est défini par :

```
464 ;;; fonction-creation: Definition * Environnement -> Fonction
    ;;; (fonction-creation definition env) rend la fonction définie par
    ;;; «definition» dans l'environnement «env».
    (define (fonction-creation definition env)
      (list '*fonction *
            (definition-variables definition)
            (definition-corps definition)
            env ) )
```

le problème étant que l'environnement env de la fonction précédente doit contenir toutes les fonctions définies dans le corps pour lequel on enrichi l'environnement avec la présente définition, y compris elle-même, y compris toute fonction dont la définition suit la définition que l'on est en train d'analyser). Gros problème : pour créer les différentes fonctions on doit utiliser un environnement qui ne sera défini que lorsque toutes les fonctions seront crées ! On ne s'en sort pas ! Eh bien si (mais je pense que vous vous en doutiez) !

Notons tout d'abord que la fonction que l'on est en train de définir ne sera réelement exécutée qu'après qu'on ait défini toutes les fonctions du corps, dans la partie <expressions> de ce corps, autrement dit après que l'on ait lu toutes les informations pour définir complètement l'environnement.

Remarque : on peut dire que c'est à cause de cela qu'en Scheme les parties définitions de fonctions et expressions sont séparées dans un corps. En passant, notons que ce n'est pas le cas au toplevel qui, encore une fois, a un statut particulier et rappelons que nous n'avons pas de tel toplevel en Deug-Scheme.

Idée

Comment allons nous faire ? Considérons le « corps » :

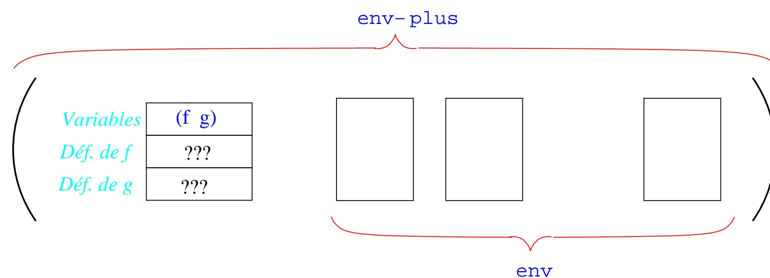
```
(define (f ...) ...) ; def-f
(define (g ...) ...) ; def-g
```

Dans un premier temps, nous créons un bloc d'activation qui contient :

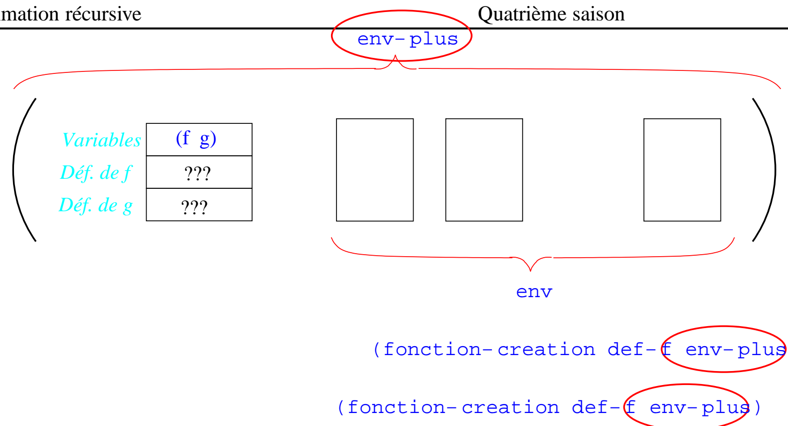
- la liste des variables de fonctions définies dans le bloc,
- autant de « cases » qu'il y a de variables définies dans le bloc, mais ces « cases » sont vides et il faudra ensuite les remplir avec la valeur des différentes fonctions (que nous calculerons en utilisant les définitions des fonctions).

Variables	(f g)
Déf. de f	???
Déf. de g	???

Nous pouvons alors ajouter cet environnement en tête de l'environnement courant :

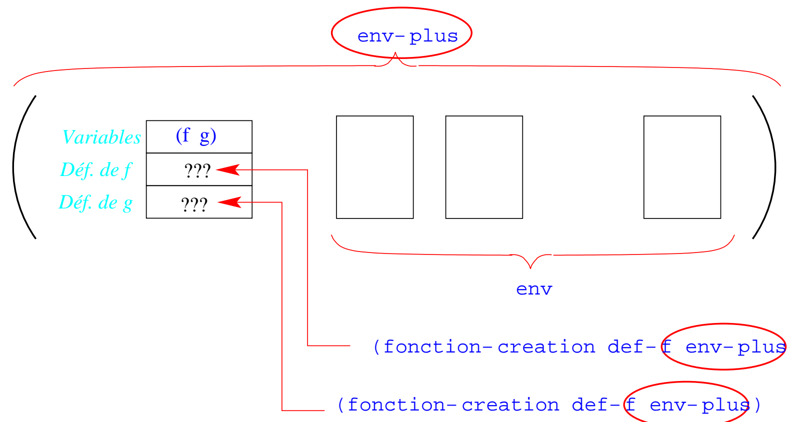


et nous pouvons alors définir les fonctions, du Scheme sous-jacent, qui implantent les deux fonctions :



Noter bien que l'environnement utilisé dans les applications de `fonction-creation` est l'environnement `env-plus` créé précédemment, environnement où il y a les différentes fonctions défi nies dans le corps.

Et il ne reste plus qu'à remplir les cases non remplies du bloc d'activation :



Noter que cette opération est une fonction très particulière puisqu'elle ne retourne pas de résultat (c'est comme les fonctions `display` et `newline`) et qu'elle modifi e l'existant (au second semestre, on parlera de procédure). En Scheme, on parle tout de même de fonction – ou plus exactement Scheme parle toujours de procédure – mais le type du résultat est `Rien` et, par convention, le nom d'une telle fonction se termine par un point d'exclamation.

Pour mettre en œuvre cette idée, nous avons besoin des fonctions suivantes :

Fonction de la barrière d'abstraction des environnements

```
545 ;;; env-add: Environnement * BlocActivation -> Environnement
    ;;; (env-add bloc env) rend l'environnement obtenu en ajoutant devant
    ;;; l'environnement «env» le bloc d'activation «bloc».
```

Fonctions de la barrière d'abstraction des blocs d'activation

```
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.

594 ;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
    ;;; (blocActivation-mettre-valeurs! bloc vals)  affecte les valeurs «vals» (données
    ;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
```

Définition de `env-enrichissement`

La défi nition va alors de soi, en utilisant la fonction `map` , ou plutôt `deug-map` , et une fonction interne pour pouvoir appliquer cette denière :

```
516 ;;; env-enrichissement: Environnement * LISTE[Définition] -> Environnement
    ;;; (env-enrichissement env defs) rend l'environnement «env» étendu avec un
    ;;; bloc d'activation pour les définitions fonctionnelles «defs».
    (define (env-enrichissement env defs)
      (let ((noms (deug-map definition-nom-fonction defs)))
        (let ((bloc (blocActivation-creation noms)))
          (let ((env-plus (env-add bloc env)))
            (define (fonction-creation-env-plus definition)
              (fonction-creation definition env-plus))
            (begin
              (blocActivation-mettre-valeurs!
               bloc
               (deug-map fonction-creation-env-plus defs))
              env-plus ) ) ) ) )
```

6.4.3. Définition de `env-extension`

La défi nition de cette fonction est plus simple que la précédente. On pourrait implanter cette fonction « d'un coup » et non en deux étapes comme ci-dessus mais cela ajouterait des fonctions de base nécessaires et comme cela se fait bien comme ça... Ainsi on fabrique l'environnement résultat en

- créant un bloc d'activation de bonne taille et où la case de la liste des variables est remplie,
- remplissant les autres cases avec les valeurs :

```
492 ;;; env-extension: Environnement * LISTE[Variable] * LISTE[Valeur] -> Environnement
    ;;; (env-extension env vars vals) rend l'environnement «env» étendu avec
    ;;; un bloc d'activation liant les variables «vars» aux valeurs «vals».
    (define (env-extension env vars vals)
      (if (= (length vars) (length vals))
          (let ((bloc (blocActivation-creation vars)))
            (begin
              (blocActivation-mettre-valeurs! bloc vals)
              (env-add bloc env) ) )
          (deug-erreur 'env-extension "arité incorrecte" (list vars vals)) ) )
```

6.4.4. Définition de `env-add-liaisons`

Nous avons déjà dit que cette fonction a la même sémantique que la précédente sauf qu'elle n'a pas la même signature (les liaisons sont données comme une liste d'associations alors que, dans la fonction prcédentes, les liaisons sont données par deux listes). Il suffi t donc de fabriquer ces deux listes et d'appliquer la fonction précédente :

- la détermination de la liste des variables se fait facilement en utilisant la fonction `deug-map` appliquée sur la fonction `liaison-variable` qui, étant donnée une liaison, retourne la variable liée,
- la détermination de la liste des valeurs s'effectue en deux applications de la fonction `deug-map` :
  - la première « mappe » la liste des liaisons avec la fonction `liaison-exp` (qui, étant donnée une liaison, retourne l'expression de la valeur de la variable liée),
  - la seconde « mappe » la liste ainsi obtenue avec une fonction qui évalue ces expressions dans l'environnement donné (on a donc besoin d'une fonction interne) :

```
503 ;;; env-add-liaisons: LISTE[Liaison] * Environnement -> Environnement
    ;;; (env-add-liaisons liaisons env) rend l'environnement obtenu en ajoutant,
    ;;; à l'environnement «env», les liaisons «liaisons».
    (define (env-add-liaisons liaisons env)
      ;; eval-env : Expression -> Valeur
      ;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
      (define (eval-env exp)
        (evaluation exp env))
      ;; expression de (env-add-liaisons liaisons env) :
      (env-extension env
                     (deug-map liaison-variable liaisons)
                     (deug-map eval-env (deug-map liaison-exp liaisons)) ) )
```

6.5. Implantation barrière d'abstraction de bas niveau

6.5.1. Rappel de la spécification

```
536 ;;; env-vide: -> Environnement
    ;;; (env-vide) rend l'environnement vide
540 ;;; env-non-vide?: Environnement -> bool
    ;;; (env-non-vide? env) rend #t ssi l'environnement «env» n'est pas vide
545 ;;; env-add: Environnement * BlocActivation -> Environnement
    ;;; (env-add bloc env) rend l'environnement obtenu en ajoutant devant
    ;;; l'environnement «env» le bloc d'activation «bloc».
551 ;;; env-1er-bloc: Environnement -> BlocActivation
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-1er-bloc env) rend le premier (i.e. celui qui a été ajouté en
    ;;; dernier) bloc d'activation de l'environnement «env».
558 ;;; env-reste: Environnement -> Environnement
    ;;; ERREUR lorsque l'environnement donné est vide
    ;;; (env-reste env) rend l'environnement obtenu en supprimant le premier
    ;;; bloc d'activation de l'environnement «env».
```

6.5.2. Structure de données

Environnement = LISTE[BlocActivation]

6.5.3. Définition des fonctions

Trop facile, laissée en exercice.

6.6. Implantation barrière d'abstraction des blocs d'activation

6.6.1. Rappel de la spécification

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»
576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.
594 ;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
    ;;; (blocActivation-mettre-valeurs! bloc vals)  affecte les valeurs «vals» (données
    ;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
```

6.6.2. Structure de données

Comment créer des « cases » vides que l'on peut remplir ensuite ? Pour ce faire, il existe en Scheme la notion de vecteur. D'où :

BlocActivation = VECTEUR[LISTE[Variable] Valeur...]

Un vecteur est une suite de cases numérotées (à partir de 0) et on peut :

1 - créer un vecteur, ayant un nombre de cases donné, les cases n'étant pas remplies, avec la fonction `make-vector` :

```
;;; make-vector : nat -> VECTEUR[alpha]
;;; (make-vector taille) rend un vecteur de dimension "taille"
```

2 - affecter une valeur à une case désignée par son numéro d'ordre – rappelons que la première case a comme numéro 0 :

*;;; vector-set! : VECTEUR[alpha] * nat * alpha -> Rien*
;;; (vector-set! v i val) affecte au ("i"+1)'ème élément de "v" la
;;; valeur "val". Par exemple, "(vector-set! v 0 val)" affecte au
;;; premier élément de "v" la valeur "val".

Notons que cette fonction Scheme ne retourne pas de résultat et qu'elle modifie un de ses arguments. Rappelons que le type du résultat est alors *Rien* et que, par convention, le nom se termine par un point d'exclamation.

2 - connaître la valeur contenue dans une case :

*;;; vector-ref : VECTEUR[alpha] * nat -> alpha*
;;; (vector-ref v i) retourne le ("i"+1)'ème élément du vecteur "v".
;;; Par exemple, "(vector-ref v 0)" retourne le premier élément
;;; de "v", "(vector-ref v 1)" retourne le deuxième élément...

6.6.3. Définitions des fonctions de la barrière d'abstraction

Définition de `blocActivation-variables`

Trés simple puisque c'est le contenu de la première case du vecteur :

```
570 ;;; blocActivation-variables: BlocActivation -> LISTE[Variable]
    ;;; (blocActivation-variables bloc) rend la liste des variables définies
    ;;; dans le bloc d'activation «bloc»
    (define (blocActivation-variables bloc)
      (vector-ref bloc 0) )
```

Définition de `blocActivation-val`

L'idée est très simple : on cherche le rang, *i*, de la variable dans la liste des variables et il ne reste plus qu'à rendre le contenu de la $i^{ème}$ case du vecteur :

```
576 ;;; blocActivation-val: BlocActivation * Variable -> Valeur
    ;;; HYPOTHESE: «var» est une variable définie dans «bloc»
    ;;; (blocActivation-val bloc var) rend la valeur de la variable «var»
    ;;; dans le bloc d'activation «bloc».
    (define (blocActivation-val bloc var)
      (let ((i (rang var (blocActivation-variables bloc))))
        (vector-ref bloc i) ) )
```

Définition de `blocActivation-creation`

Très simple (ne pas oublier de remplir la case 0) :

```
584 ;;; blocActivation-creation: LISTE[Variable] -> BlocActivation
    ;;; (blocActivation-creation vars) rend un bloc d'activation contenant
    ;;; la liste des variables «vars», avec la place qu'il faut pour les valeurs
    ;;; de ces variables, cette place n'étant pas remplie.
    (define (blocActivation-creation vars)
      (let ((bloc (make-vector (+ 1 (length vars)))))
        (begin
          (vector-set! bloc 0 vars)
          bloc ) ) )
```

Définition de `blocActivation-mettre-valeurs!`

Pour assigner les valeurs, la difficulté vient du fait que l'on ne sait pas combien il y en a : on doit donc définir une fonction (interne) récursive qui remplit les cases d'un vecteur, à partir d'un indice donné, avec les éléments successifs d'une liste donnée et appliquer cette fonction avec comme indice initial 1 (premier indice où l'on trouve une valeur) et la liste de valeurs.

```
594 ;;;  blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
    ;;;  (blocActivation-mettre-valeurs! bloc vals)  affecte les valeurs «vals» (données
    ;;;  sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
    (define (blocActivation-mettre-valeurs! bloc vals)
      ;;  remplir!: nat * LISTE[Valeur] -> Rien
      ;;  (remplir! i vals) remplit les cases du vecteur «bloc», à partir de
      ;;  l'indice «i», avec les valeurs de la liste «vals» (et dans le même ordre).
      (define (remplir! i vals)
        (if (pair? vals)
            (begin
              (vector-set! bloc i (car vals))
              (remplir! (+ i 1) (cdr vals)) ) ) )
      (remplir! 1 vals) )
```

6.7. Environnement initial

Pour créer l'environnement initial il suffi t d'étendre l'environnement vide avec des liaisons nom-de-la-primitive — valeur-de-la-primitive. Pour ce faire, nous pouvons penser utiliser la fonction `env-extension` ou la fonction `env-add-liaisons`. Nous décidons d'utiliser `env-extension` (comme exercice, vous pouvez essayer de le faire avec `env-add-liaisons`; attention, une liaison est une association variable — expression et `env-add-liaisons` évalue les expressions).

C'est rrès facile (vous pouvez essayer)... sauf qu'il faut écrire (décrire) les primitives avec deux listes : la première qui contient les différents noms et la seconde qui contient les différentes valeurs, la première valeur correspondant au premier nom, la seconde valeur correspondant au second nom... la 27ème valeur correspondant au 27ème nom et gare à celui qui intervertit deux noms ou deux valeurs !

6.7.1. Description des primitives

Aussi, pour éviter des erreurs, nous décrivons chaque primitive par quatre éléments, le nom d'une primitive, la fonction correpondante du Scheme sous-jacent et l'arité représentée par un comparateur et un entier et nous fabriquerons une desciption – implantée par un n-uplet – de la primitive en utilisant la fonction `description-primitive` :

```
633 ;;;  description-primitive: Variable * (Valeur ... -> Valeur)
    ;;;      * (num * num -> bool) * num  -> DescriptionPrimitive
    ;;;  (description-primitive var f comparator arite) rend la description de la
    ;;;  primitive désignée par «var», implantée dans le Scheme sous-jacent par «f» et
    ;;;  dont l'arité est définie par «comparator» «arite».
    (define (description-primitive var f comparator arite)
      (list var f comparator arite))
```

Les primitives sont alors données comme une liste de description sous une forme très lisible (voir lignes 641 à 674):

```
641 ;;;  descriptions-primitives: -> LISTE[DescriptionPrimitive]
    ;;;  (descriptions-primitives) rend la liste des descriptions de toutes les
    ;;;  primitives
    (define (descriptions-primitives)
      (cons (description-primitive 'car     car     =  1)
     ...
      (cons (description-primitive 'erreur  erreur  >= 2)
674         '()))))))))))))))))))))))))))))))) )
```

Noter que nous avons utilisé `cons` (et non `list` qui nous aurait facilité la tâche) car, si nous avions utilisé `list`, pour l'auto-évaluation, il aurait fallu que les primitives ayant un nombre quelconque d'arguments puissent être appelées avec au moins une trentaine d'arguments (voir ce que cela implique pour l'imlantation de la fonction `primitive-invocation` ...).

6.7.2. Définition de la fonction `env-initial`

Comme nous l'avons déja dit, il faut créer les deux listes (celle des variables et celle des valeurs) à partir de cette liste. Bien sûr, nous le faisons en utilisant `deug-map` (et même deux fois pour calculer les valeurs).

```
624 ;;; env-initial: -> Environnement
;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
;;; contient toutes les primitives.
(define (env-initial)
  (env-extension (env-vide)
                 (deug-map car (descriptions-primitives))
                 (deug-map primitive-creation
                           (deug-map cdr (descriptions-primitives)) ) ) )
```

7. Annexe : source de deug-eval

```
1 ;;;; $Id: eval4fr.scm,v 1.12 2003/01/16 16:03:52 titou Exp $
2 ;;;; Copyright (C) 2000 by <Titou.Durand@ufr-info-p6.jussieu.fr>
3 ;;;;                   and <Christian.Queinnec@lip6.fr>
4
5 ;;;; {{{ Grammaire du langage
6 ;;;; Le langage interprété est défini par la grammaire suivante:
7 ;;;;   deug-programme := expression
8 ;;;;       expression := variable
9 ;;;;                   | constante | (QUOTE donnée)        ; citation
10 ;;;;                  | (COND clause *)                    ; conditionnelle
11 ;;;;                  | (IF condition conséquence [alternant]); alternative
12 ;;;;                  | (BEGIN expression *)               ; séquence
13 ;;;;                  | (LET (liaison *) corps)            ; bloc
14 ;;;;                  | (fonction argument *)              ; application
15 ;;;;        condition := expression
16 ;;;;      conséquence := expression
17 ;;;;        alternant := expression
18 ;;;;           clause := (condition expression *)
19 ;;;;                  | (ELSE expression *)
20 ;;;;         fonction := expression
21 ;;;;         argument := expression
22 ;;;;        constante := nombre | chaîne | booléen | caractère
23 ;;;;           donnée := constante
24 ;;;;                  | symbole
25 ;;;;                  | (donnée *)
26 ;;;;          liaison := (variable expression)
27 ;;;;            corps := definition * expression expression *
28 ;;;;       définition := (DEFINE (nom-fonction variable *) corps)
29 ;;;;     nom-fonction := variable
30 ;;;;         variable := tous les symboles de Scheme autres que les mots-clés
31 ;;;;          symbole := tous les symboles de Scheme
32 ;;;; }}} Grammaire du langage
33
34 ;;;; {{{ Utilitaires généraux
35 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;        ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
36 ;;;; Nécessaires pour l'auto-amorçage (on pourrait également les placer
37 ;;;; dans l'environnement initial).
38
39 ;;; Signaler une erreur et abandonner l'évaluation.
40 (define (deug-erreur fn message donnee)
41   (erreur 'deug-eval fn message donnee) )
42
43 ;;; cadr: LISTE[alpha]/au moins deux termes/ -> alpha
44 ;;; (cadr L) rend le second terme de la liste «L».
45 (define (cadr L)
46   (car (cdr L)) )
47
```

```
48 ;;;  cddr:  LISTE[alpha]/au  moins  deux  termes/  ->  LISTE[alpha]
49 ;;;  (cddr  L)  rend  la  liste  «L»  privée  de  ses  deux  premiers  termes.
50 (define  (cddr  L)
51   (cdr  (cdr  L))  )
52
53 ;;;  caddr:  LISTE[alpha]/au  moins  trois  termes/  ->  alpha
54 ;;;  (caddr  L)  rend  le  troisième  terme  de  la  liste  «L».
55 (define  (caddr  L)
56   (car  (cdr  (cdr  L)))  )
57
58 ;;;  cdddr:  LISTE[alpha]/au  moins  trois  termes/  ->  LISTE[alpha]
59 ;;;  (cdddr  L)  rend  la  liste  «L»  privée  de  ses  trois  premiers  termes.
60 (define  (cdddr  L)
61   (cdr  (cdr  (cdr  L)))  )
62
63 ;;;  cadddr:  LISTE[alpha]/au  moins  quatre  termes/  ->  alpha
64 ;;;  (cadddr  L)  rend  le  quatrième  terme  de  la  liste  «L».
65 (define  (cadddr  L)
66   (car  (cdr  (cdr  (cdr  L))))  )
67
68 ;;;  length:  LISTE[alpha]  ->  nat
69 ;;;  (length  L)  rend  la  longueur  de  la  liste  «L».
70 (define  (length  L)
71   (if  (pair?  L)
72        (+ 1 (length  (cdr  L)))
73        0  )  )
74
75 ;;;  deug-map:  (alpha  ->  beta)  *  LISTE[alpha]  ->  LISTE[beta]
76 ;;;  (deug-map  f  L)  rend  la  liste  des  valeurs  de  «f»  appliquée  aux  termes
77 ;;;  de  la  liste  «L».
78 (define  (deug-map  f  L)
79   (if  (pair?  L)
80        (cons  (f  (car  L))  (deug-map  f  (cdr  L)))
81        '()  )  )
82
83 ;;;  member:  alpha  *  LISTE[alpha]  ->  LISTE[alpha]  +  #f
84 ;;;  (member  e  L)  rend  le  suffixe  de  «L»  débutant  par  la  première
85 ;;;  occurrence  de  «e»  ou  #f  si  «e»  n'apparaît  pas  dans  «L».
86 (define  (member  e  L)
87   (if  (pair?  L)
88        (if  (equal?  e  (car  L))
89             L
90             (member  e  (cdr  L))  )
91        #f  )  )
92
93 ;;;  rang:  alpha  *  LISTE[alpha]  ->  nat
94 ;;;  (rang  e  L)  rend  le  rang  de  l'élément  donné  dans  la  liste  «L»
95 ;;;  (où  on  sait  que  l'élément  apparaît).  Le  premier  élément  a  pour  rang  un.
96 (define  (rang  e  L)
97   (if  (equal?  e  (car  L))
98        1
99        (+ 1 (rang  e  (cdr  L)))  )  )
100 ;;;;  }}}  Utilitaires  généraux
101
102 ;;;;  {{{  Barrière-syntaxique
103 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
104 ;;;;  Ces  fonctions  permettent  de  manipuler  les  différentes  expressions
105 ;;;;  syntaxiques  dont  Scheme  est  formé.  Pour  chacune  de  ces  différentes
106 ;;;;  formes  syntaxiques,  on  trouve  le  reconnaisseur  et  les  accesseurs.
107
```

```
108 ;;;   variable?:    Expression    ->  bool
109 (define   (variable?    exp)
110   (if  (symbol?   exp)
111       (cond  ((equal?   exp  'cond)     #f)
112              ((equal?   exp  'else)     #f)
113              ((equal?   exp  'if)       #f)
114              ((equal?   exp  'quote)    #f)
115              ((equal?   exp  'begin)    #f)
116              ((equal?   exp  'let)      #f)
117              ((equal?   exp  'let *)    #f)
118              ((equal?   exp  'define)   #f)
119              ((equal?   exp  'or)       #f)
120              ((equal?   exp  'and)      #f)
121              (else                      #t)  )
122       #f)  )
123
124 ;;;   citation?:    Expression    ->  bool
125 (define   (citation?    exp)
126   (cond  ((number?    exp)   #t)
127          ((string?    exp)   #t)
128          ((char?   exp)      #t)
129          ((boolean?    exp)  #t)
130          ((pair?   exp)      (equal?   (car   exp)  'quote))
131          (else               #f)  )  )
132
133 ;;;   conditionnelle?:      Expression    ->  bool
134 (define   (conditionnelle?      exp)
135   (if  (pair?   exp)  (equal?   (car   exp)  'cond)   #f)  )
136
137 ;;;   conditionnelle-clauses:        Conditionnelle     ->  LISTE[Clause]
138 (define   (conditionnelle-clauses        conditionnelle)
139   (cdr  conditionnelle)      )
140
141 ;;;   alternative?:     Expression    ->  bool
142 (define   (alternative?     exp)
143   (if  (pair?   exp)  (equal?   (car   exp)  'if)  #f)  )
144
145 ;;;   alternative-condition:       Alternative    ->  Expression
146 (define   (alternative-condition       alt)
147   (cadr   alt)  )
148
149 ;;;   alternative-consequence:       Alternative    ->  Expression
150 (define   (alternative-consequence       alt)
151   (caddr   alt)  )
152
153 ;;;   alternative-alternant:       Alternative    ->  Expression
154 (define   (alternative-alternant       alt)
155   (if  (pair?   (cdddr   alt))
156       (cadddr   alt)
157       #f  )  )
158
159 ;;;   sequence?:    Expression    ->  bool
160 (define   (sequence?    exp)
161   (if  (pair?   exp)  (equal?   (car   exp)  'begin)   #f)  )
162
163 ;;;   sequence-exps:     Sequence    ->  LISTE[Expression]
164 (define   (sequence-exps     seq)
165   (cdr   seq)  )
166
167 ;;;   bloc?:   Expression    ->  bool
```

```
168 (define (bloc? exp)
169   (if (pair? exp) (equal? (car exp) 'let) #f) )
170
171 ;;; bloc-liaisons: Bloc -> LISTE[Liaison]
172 (define (bloc-liaisons bloc)
173   (cadr bloc) )
174
175 ;;; bloc-corps: Bloc -> Corps
176 (define (bloc-corps bloc)
177   (cddr bloc) )
178
179 ;;; application?: Expression -> bool
180 (define (application? exp)
181   (pair? exp) )
182
183 ;;; application-fonction: Application -> Expression
184 (define (application-fonction appl)
185   (car appl) )
186
187 ;;; application-arguments: Application -> LISTE[Expression]
188 (define (application-arguments appl)
189   (cdr appl) )
190
191 ;;; clause-condition: Clause -> Expression
192 (define (clause-condition clause)
193   (car clause) )
194
195 ;;; clause-expressions: Clause -> LISTE[Expression]
196 (define (clause-expressions clause)
197   (cdr clause) )
198
199 ;;; liaison-variable: Liaison -> Variable
200 (define (liaison-variable liaison)
201   (car liaison) )
202
203 ;;; liaison-exp: Liaison -> Expression
204 (define (liaison-exp liaison)
205   (cadr liaison) )
206
207 ;;; definition?: Corps -> bool
208 ;;; (definition? corps) rend #t ssi le premier élément du corps
209 ;;; «corps» est une définition
210 (define (definition? corps)
211   (if (pair? corps) (equal? (car corps) 'define) #f) )
212
213 ;;; definition-nom-fonction: Definition -> Variable
214 (define (definition-nom-fonction def)
215   (car (cadr def)) )
216
217 ;;; definition-variables: Definition -> LISTE[Variable]
218 (define (definition-variables def)
219   (cdr (cadr def)) )
220
221 ;;; definition-corps: Definition -> Corps
222 (define (definition-corps def)
223   (cddr def) )
224 ;;;; }}} Barrière-syntaxique
225
226 ;;;; {{{ Evaluateur
227 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
```

```
228
229 ;;; deug-eval: Deug-Programme -> Valeur
230 ;;; (deug-eval p) rend la valeur du programme (de Deug-Scheme) «p».
231 (define (deug-eval p)
232   (evaluation p (env-initial) ) )
233
234 ;;; evaluation: Expression * Environnement -> Valeur
235 ;;; (evaluation exp env) rend la valeur de l'expression «exp» dans
236 ;;; l'environnement «env».
237 (define (evaluation exp env)
238   ;; (discrimine l'expression et invoque l'évaluateur spécialisé)
239   (cond
240    ((variable? exp)          (variable-val exp env))
241    ((citation? exp)          (citation-val exp))
242    ((alternative? exp)       (alternative-eval
243                               (alternative-condition exp)
244                               (alternative-consequence exp)
245                               (alternative-alternant exp) env))
246    ((conditionnelle? exp)    (conditionnelle-eval
247                               (conditionnelle-clauses exp) env))
248    ((sequence? exp)          (sequence-eval (sequence-exps exp) env))
249    ((bloc? exp)              (bloc-eval (bloc-liaisons exp)
250                                         (bloc-corps exp) env))
251    ((application? exp)       (application-eval
252                               (application-fonction exp)
253                               (application-arguments exp) env))
254    (else (deug-erreur 'evaluation "pas un programme" exp))) )
255
256 ;;; alternative-eval: Expression 3̂ * Environnement -> Valeur
257 ;;; (alternative-eval condition consequence alternant env) rend la valeur de
258 ;;; l'expression «(if condition consequence alternant)» dans l'environnement «env».
259 (define (alternative-eval condition consequence alternant env)
260   (if (evaluation condition env)
261       (evaluation consequence env)
262       (evaluation alternant env)) )
263
264 ;;; LISTE[Clause] * Environnement -> Valeur
265 ;;; (conditionnelle-eval clauses env) rend la valeur, dans l'environnement «env»,
266 ;;; de l'expression «(cond c1 c2 ... cn)», «c1», «c2»... «cn» étant les éléments
267 ;;; de la liste «clauses».
268 (define (conditionnelle-eval clauses env)
269   (evaluation (conditionnelle-expansion clauses) env) )
270
271 ;;; conditionnelle-expansion: LISTE[Clause] -> Expression
272 ;;; (conditionnelle-expansion clauses) rend l'expression, écrite avec des
273 ;;; alternatives, équivalente à l'expression «(cond c1 c2 ... cn)»,
274 ;;; «c1», «c2»... «cn» étant les éléments de la liste «clauses».
275 (define (conditionnelle-expansion clauses)
276   (if (pair? clauses)
277       (let ((premiere-clause (car clauses)))
278         (if (equal? (clause-condition premiere-clause) 'else)
279             (cons 'begin (clause-expressions premiere-clause))
280             (cons 'if
281                   (cons (clause-condition premiere-clause)
282                         (cons (cons 'begin (clause-expressions premiere-clause))
283                               (let ((seq (conditionnelle-expansion (cdr clauses))))
284                                 (if (pair? seq)
285                                     (list seq)
286                                     seq ) ) ) ) ) ) )
287       '() ) )
```

```
288
289 ;;;  sequence-eval:  LISTE[Expression] * Environnement -> Valeur
290 ;;;  (sequence-eval exps env) rend la valeur, dans l'environnement «env», de
291 ;;;  l'expression «(begin e1 ... en)», «e1»... «en» étant les éléments de la liste
292 ;;;  «exps».
293 ;;;  (Il faut évaluer tour à tour les expressions et rendre la valeur de la
294 ;;;  dernière d'entre elles.)
295 (define (sequence-eval exps env)
296   ;; sequence-eval+ : LISTE[Expression]/non vide/ -> Valeur
297   ;; même fonction, sachant que la liste «exps» n'est pas vide et en globalisant
298   ;; la variable «env».
299   (define (sequence-eval+ exps)
300     (if (pair? (cdr exps))
301         (begin (evaluation (car exps) env)
302                (sequence-eval+ (cdr exps)) )
303         (evaluation (car exps) env)))
304   ;; expression de (sequence-eval exps env) :
305   (if (pair? exps)
306       (sequence-eval+ exps)
307       #f ) )
308
309 ;;;  application-eval:  Expression * LISTE[Expression] * Environnement -> Valeur
310 ;;;  (application-eval exp-fn arguments env) rend la valeur de l'invocation de
311 ;;;  l'expression «exp-fn» aux arguments «arguments» dans l'environnement «env».
312 (define (application-eval exp-fn arguments env)
313   ;; eval-env : Expression -> Valeur
314   ;; (eval-env exp) rend la valeur de «exp» dans l'environnement «env»
315   (define (eval-env exp)
316     (evaluation exp env))
317   ;; expression de (application-eval exp-fn arguments env) :
318   (let ((f (evaluation exp-fn env)))
319     (if (invocable? f)
320         (invocation f (deug-map eval-env arguments))
321         (deug-erreur 'application-eval
322                      "pas une fonction" f ) ) ) )
323
324 ;;;  bloc-eval:  LISTE[Liaison] * Corps * Environnement -> Valeur
325 ;;;  (bloc-eval liaisons corps env) rend la valeur, dans l'environnement «env»,
326 ;;;  de l'expression «(let liaisons corps)».
327 (define (bloc-eval liaisons corps env)
328   (corps-eval corps (env-add-liaisons liaisons env)) )
329
330 ;;;  corps-eval:  Corps * Environnement -> Valeur
331 ;;;  (corps-eval corps env) rend la valeur de «corps» dans l'environnement «env»
332 (define (corps-eval corps env)
333   (let ((def-exp (corps-separation-defs-exps corps)))
334     (let ((defs (car def-exp))
335           (exp (cadr def-exp)))
336       (evaluation exp (env-enrichissement env defs)) ) ) )
337
338 ;;;  corps-separation-defs-exps:  Corps -> (LISTE[Definition] * LISTE[Expression])
339 ;;;  (corps-separation-defs-exps corps) rend une liste dont le premier élément est
340 ;;;  la liste des definitions du corps «corps» et les autres éléments sont les
341 ;;;  expressions de ce corps.
342 (define (corps-separation-defs-exps corps)
343   (if (definition? (car corps))
344       (let ((def-exp-cdr
345               (corps-separation-defs-exps (cdr corps))))
346         (cons (cons (car corps)
347                     (car def-exp-cdr))
```

```
348          (cdr  def-exp-cdr)))
349          (cons  '()  corps)  ) )
350  ;;;;    }}}  Evaluateur
351
352  ;;;;    {{{  Barrière-interpretation
353  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
354  ;;;;  Un  programme   Scheme  décrit  deux  sortes  d'objets:   les  valeurs  non  fonctionnelles
355  ;;;;  (les  entiers,   les  booléens...   les  listes...)   et  les  valeurs  fonctionnelles
356
357  ;;;;        {{{  Valeurs-non-fonctionnelles
358  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
359
360  ;;;   citation-val:    Citation   ->  Valeur/non   fonctionelle/
361  ;;;   (citation-val    cit)  rend  la  valeur  de  la  citation   «cit».
362  (define   (citation-val    cit)
363    (if  (pair?  cit)
364         (cadr  cit)
365         cit  ) )
366  ;;;;        }}}  Valeurs-non-fonctionnelles
367
368  ;;;;        {{{  Valeurs-fonctionnelles
369  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
370  ;;;;   Il  y  a  deux  types  de  fonctions,   les  fonctions  prédéfinies   (reconnues   par
371  ;;;;   primitive?)   et  les  fonctions   du  programme   en  cours  d'évaluation   (créées   par
372  ;;;;   fonction-creation).
373
374  ;;;   invocable?:   Valeur   ->  bool
375  ;;;   (invocable?   val)  rend  vrai  ssi  «val»  est  une  fonction   (primitive   ou  définie
376  ;;;   par  le  programmeur)
377  (define   (invocable?   val)
378    (if  (primitive?   val)
379         #t
380         (fonction?   val)  ) )
381
382  ;;;   invocation:   Invocable   *  LISTE[Valeur]    ->  Valeur
383  ;;;   (invocation   f  vals)  rend  la  valeur  de  l'application   de  «f»  aux  éléments   de
384  ;;;   «vals».
385  (define   (invocation   f  vals)
386    (if  (primitive?   f)
387         (primitive-invocation      f  vals)
388         (fonction-invocation      f  vals)  ) )
389
390  ;;;;            {{{  Primitives
391  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
392  ;;;;   Une  primitive   est  implantée   par  un  4-uplet:
393  ;;;;   - le  premier   élément   est  le  symbole  *primitive  *  (pour   les  reconnaître),
394  ;;;;   - le  second   élément   est  la  fonction   du  Scheme   sous-jacent   qui  implante
395  ;;;;     la  primitive,
396  ;;;;   - le  troisième   élément   est  un  comparateur    (=  ou  >=),
397  ;;;;   - le  dernier   élément   est  un  entier   naturel,   ces  deux  derniers   éléments
398  ;;;;     permettant   de  spécifier   l'arité   de  la  primitive.
399
400  ;;;   primitive?:   Valeur   ->  bool
401  ;;;   (primitive?   val)  rend   vrai  ssi  «val»  est  une  fonction   primitive.
402  (define   (primitive?   val)
403    (if  (pair?  val)
404         (equal?   (car  val)  '*primitive  *)
405         #f)  )
406
407  ;;;   primitive-creation:      N-UPLET[(Valeur...      ->  Valeur)   (num  *  num  ->  bool)   num]
```

```
408 ;;;                    -> Primitive
409 ;;; (primitive-creation f-c-n) rend la primitive implantée par la fonction (du
410 ;;; Scheme sous-jacent) «f», le premier élément de «f-c-n», et dont l'arité est
411 ;;; spécifiée par le comparateur «c», deuxième élément de «f-c-n» et l'entier «n»,
412 ;;;  troisième élément de «f-c-n».
413 (define (primitive-creation f-c-n)
414   (cons '*primitive* f-c-n) )
415
416 ;;; primitive-invocation: Primitive * LISTE[Valeur] -> Valeur
417 ;;; (primitive-invocation p vals) rend la valeur de l'application de la
418 ;;; primitive «p» aux éléments de «vals».
419 (define (primitive-invocation primitive vals)
420   (let ((n       (length vals))
421         (f       (cadr primitive))
422         (compare (caddr primitive))
423         (arite   (cadddr primitive)))
424     (if (compare n arite)
425         (cond
426          ((= n 0) (f))
427          ((= n 1) (f (car vals)))
428          ((= n 2) (f (car vals) (cadr vals)))
429          ((= n 3) (f (car vals) (cadr vals) (caddr vals)))
430          ((= n 4) (f (car vals) (cadr vals)
431                      (caddr vals) (cadddr vals) ))
432          (else
433           (deug-erreur 'primitive-invocation
434                        "limite implantation (arités quelconques < 5)"
435                        vals)))
436         (deug-erreur 'primitive-invocation "arité incorrecte" vals) ) ) )
437 ;;;;           }}} Primitives
438
439 ;;;;           {{{ Fonctions-definies par le programmeur
440 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;        ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
441 ;;;;  Une fonction définie par le programmeur est implantée par un 4-uplet:
442 ;;;;  - le premier élément est le symbole *fonction* (pour les reconnaître),
443 ;;;;  - le second élément est la liste des variables de la définition de la
444 ;;;;    fonction,
445 ;;;;  - le troisième élément est le corps de la définition de la fonction,
446 ;;;;  - le quatrième élément est l'environnement où est définie la fonction.
447
448 ;;; fonction?: Valeur -> bool
449 ;;; (fonction? val) rend vrai ssi «val» est une fonction créée par le programmeur.
450 (define (fonction? val)
451   (if (pair? val)
452       (equal? (car val) '*fonction*)
453       #f ) )
454
455 ;;; fonction-invocation: Fonction * LISTE[Valeur] -> Valeur
456 ;;; (fonction-invocation f vals) rend la valeur de l'application de
457 ;;; la fonction définie par le programmeur «f» aux éléments de «vals».
458 (define (fonction-invocation f vals)
459   (let ((variables (cadr f))
460         (corps     (caddr f))
461         (env       (cadddr f)) )
462     (corps-eval corps (env-extension env variables vals)) ) )
463
464 ;;; fonction-creation: Definition * Environnement -> Fonction
465 ;;; (fonction-creation definition env) rend la fonction définie par
466 ;;; «definition» dans l'environnement «env».
467 (define (fonction-creation definition env)
```

```
468   (list   '*fonction   *
469          (definition-variables       definition)
470          (definition-corps      definition)
471          env  ) )
472
473 ;;;;             }}}  Fonctions-definies      par  le  programmeur
474 ;;;;         }}}  Valeurs-fonctionnelles
475 ;;;;     }}}  Barrière-interpretation
476
477 ;;;;     {{{  Environnements-H      (barrière    de  haut   niveau)
478 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;;;;;  ;;
479
480 ;;;   variable-val:     Variable    * Environnement     -> Valeur
481 ;;;   (variable-val     var  env)  rend  la  valeur   de  la  variable    «var»   dans
482 ;;;   l'environnement      «env».
483 (define   (variable-val     var  env)
484   (if  (env-non-vide?     env)
485       (let  ((bloc   (env-1er-bloc     env)))
486         (let  ((variables    (blocActivation-variables        bloc)))
487           (if  (member   var  variables)
488               (blocActivation-val       bloc  var)
489               (variable-val     var  (env-reste    env)))))
490       (deug-erreur     'variable-val     "variable    inconnue"    var)  )  )
491
492 ;;;   env-extension:      Environnement     * LISTE[Variable]     * LISTE[Valeur]     ->  Environnement
493 ;;;   (env-extension      env  vars  vals)  rend  l'environnement      «env»  étendu   avec
494 ;;;   un  bloc  d'activation     liant   les  variables    «vars»   aux  valeurs   «vals».
495 (define   (env-extension      env  vars  vals)
496   (if  (=  (length   vars)  (length   vals))
497       (let  ((bloc   (blocActivation-creation        vars)))
498         (begin
499           (blocActivation-mettre-valeurs!          bloc  vals)
500           (env-add   bloc  env)  )  )
501       (deug-erreur     'env-extension      "arité   incorrecte"    (list   vars   vals))   )  )
502
503 ;;;   env-add-liaisons:      LISTE[Liaison]     * Environnement     -> Environnement
504 ;;;   (env-add-liaisons      liaisons    env)  rend  l'environnement      obtenu   en  ajoutant,
505 ;;;   à  l'environnement      «env»,   les  liaisons    «liaisons».
506 (define   (env-add-liaisons      liaisons    env)
507   ;;  eval-env   : Expression    -> Valeur
508   ;;  (eval-env   exp)  rend  la  valeur   de  «exp»   dans   l'environnement      «env»
509   (define   (eval-env    exp)
510     (evaluation    exp  env))
511   ;;  expression    de  (env-add-liaisons      liaisons    env)  :
512   (env-extension      env
513                     (deug-map    liaison-variable      liaisons)
514                     (deug-map    eval-env    (deug-map    liaison-exp    liaisons))    )  )
515
516 ;;;   env-enrichissement:       Environnement     * LISTE[Definition]      ->  Environnement
517 ;;;   (env-enrichissement       env  defs)  rend  l'environnement      «env»  étendu   avec  un
518 ;;;   bloc  d'activation     pour  les  définitions     fonctionnelles     «defs».
519 (define   (env-enrichissement       env  defs)
520   (let  ((noms   (deug-map    definition-nom-fonction        defs)))
521     (let  ((bloc   (blocActivation-creation        noms)))
522       (let  ((env-plus    (env-add   bloc  env)))
523         (define   (fonction-creation-env-plus         definition)
524           (fonction-creation      definition    env-plus))
525         (begin
526           (blocActivation-mettre-valeurs!
527            bloc
```

```
528              (deug-map   fonction-creation-env-plus    defs))
529             env-plus   ) ) ) ) )
530
531 ;;;;         {{{  Environnements-B     (barrière   de  bas  niveau)
532 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
533 ;;;;   Les  environnements     sont  représentés    par  la  structure   de  données
534 ;;;;   LISTE[BlocActivation]
535
536 ;;;   env-vide:    ->  Environnement
537 ;;;   (env-vide)    rend  l'environnement     vide
538 (define   (env-vide)    '())
539
540 ;;;   env-non-vide?:     Environnement     ->  bool
541 ;;;   (env-non-vide?     env)  rend  #t  ssi  l'environnement     «env»  n'est  pas  vide
542 (define   (env-non-vide?     env)
543   (pair?   env)  )
544
545 ;;;   env-add:   Environnement    * BlocActivation     ->  Environnement
546 ;;;   (env-add   bloc  env)  rend  l'environnement     obtenu   en  ajoutant   devant
547 ;;;   l'environnement     «env»   le  bloc  d'activation    «bloc».
548 (define   (env-add   bloc  env)
549   (cons  bloc  env)  )
550
551 ;;;   env-1er-bloc:     Environnement     ->  BlocActivation
552 ;;;   ERREUR  lorsque   l'environnement     donné   est  vide
553 ;;;   (env-1er-bloc     env)  rend  le  premier   (i.e.  celui   qui  a  été  ajouté   en
554 ;;;   dernier)  bloc  d'activation    de  l'environnement     «env».
555 (define   (env-1er-bloc     env)
556     (car  env)  )
557
558 ;;;   env-reste:    Environnement     ->  Environnement
559 ;;;   ERREUR  lorsque   l'environnement     donné   est  vide
560 ;;;   (env-reste    env)  rend  l'environnement     obtenu   en  supprimant    le  premier
561 ;;;   bloc  d'activation    de  l'environnement     «env».
562 (define   (env-reste    env)
563   (cdr  env)  )
564
565 ;;;;              {{{  Blocs  d'activation
566 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;          ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
567 ;;;;   Les  blocs  d'activation    sont  représentés    par  la  structure   de  données:
568 ;;;;   VECTEUR[LISTE[Variable]       Valeur...]
569
570 ;;;   blocActivation-variables:        BlocActivation     ->  LISTE[Variable]
571 ;;;   (blocActivation-variables        bloc)  rend  la  liste  des  variables   définies
572 ;;;   dans  le  bloc  d'activation    «bloc»
573 (define   (blocActivation-variables        bloc)
574   (vector-ref    bloc  0)  )
575
576 ;;;   blocActivation-val:      BlocActivation     * Variable   ->  Valeur
577 ;;;   HYPOTHESE:    «var»  est  une  variable   définie   dans  «bloc»
578 ;;;   (blocActivation-val      bloc  var)  rend  la  valeur   de  la  variable   «var»
579 ;;;   dans  le  bloc  d'activation    «bloc».
580 (define   (blocActivation-val      bloc  var)
581   (let  ((i  (rang  var  (blocActivation-variables        bloc))))
582      (vector-ref    bloc  i)  )  )
583
584 ;;;   blocActivation-creation:       LISTE[Variable]     ->  BlocActivation
585 ;;;   (blocActivation-creation       vars)  rend  un  bloc  d'activation    contenant
586 ;;;   la  liste  des  variables   «vars»,  avec  la  place  qu'il  faut  pour  les  valeurs
587 ;;;   de  ces  variables,    cette  place  n'étant   pas  remplie.
```

```
588 (define (blocActivation-creation vars)
589   (let ((bloc (make-vector (+ 1 (length vars)))))
590     (begin
591       (vector-set! bloc 0 vars)
592       bloc ) ) )
593
594 ;;; blocActivation-mettre-valeurs!: BlocActivation * LISTE[Valeur] -> Rien
595 ;;; (blocActivation-mettre-valeurs! bloc vals) affecte les valeurs «vals» (données
596 ;;; sous forme de liste) dans le bloc d'activation «bloc» (qui est un vecteur)
597 (define (blocActivation-mettre-valeurs! bloc vals)
598   ;; remplir!: nat * LISTE[Valeur] -> Rien
599   ;; (remplir! i vals) remplit les cases du vecteur «bloc», à partir de
600   ;; l'indice «i», avec les valeurs de la liste «vals» (et dans le même ordre).
601   (define (remplir! i vals)
602     (if (pair? vals)
603         (begin
604           (vector-set! bloc i (car vals))
605           (remplir! (+ i 1) (cdr vals)) ) ) )
606   (remplir! 1 vals) )
607 ;;;;            }}} Blocs d'activation
608 ;;;;       }}} Environnements-B (barrière de bas niveau)
609
610 ;;;;       {{{ Environnement-initial
611 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;         ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;;;;; ;;
612 ;;;;  L'environnement initial est composé des primitives.
613 ;;;;  Pour faciliter la description, dans ce logiciel, des différentes
614 ;;;;  primitives, nous les écrirons sous la forme d'une liste de
615 ;;;;  descriptions de primitive.
616 ;;;;  Un élément du type DescriptionPrimitive est une description (presque
617 ;;;;  textuelle) d'une primitive. C'est une liste dont
618 ;;;;  - le premier élément est la variable représentant, dans Deug-Scheme, la
619 ;;;;    primitive considérée,
620 ;;;;  - les trois autres éléments sont les trois éléments qui décrivent
621 ;;;;    l'implantation de la primitive (la fonction du Scheme sous-jacent,
622 ;;;;    le comparateur et l'arité).
623
624 ;;; env-initial: -> Environnement
625 ;;; (env-initial) rend l'environnement initial, i.e. l'environnement qui
626 ;;; contient toutes les primitives.
627 (define (env-initial)
628   (env-extension (env-vide)
629                  (deug-map car (descriptions-primitives))
630                  (deug-map primitive-creation
631                            (deug-map cdr (descriptions-primitives)) ) ) )
632
633 ;;; description-primitive: Variable * (Valeur ... -> Valeur)
634 ;;;       * (num * num -> bool) * num -> DescriptionPrimitive
635 ;;; (description-primitive var f comparator arite) rend la description de la
636 ;;; primitive désignée par «var», implantée dans le Scheme sous-jacent par «f» et
637 ;;; dont l'arité est définie par «comparator» «arite».
638 (define (description-primitive var f comparator arite)
639   (list var f comparator arite))
640
641 ;;; descriptions-primitives: -> LISTE[DescriptionPrimitive]
642 ;;; (descriptions-primitives) rend la liste des descriptions de toutes les
643 ;;; primitives
644 (define (descriptions-primitives)
645   (list
646    (description-primitive 'car car = 1)
647    (description-primitive 'cdr cdr = 1)
```

```
648    (description-primitive       'cons            cons            =   2)
649    (description-primitive       'list            list            >=  0)
650    (description-primitive       'vector-length   vector-length   =   1)
651    (description-primitive       'vector-ref      vector-ref      =   2)
652    (description-primitive       'vector-set!     vector-set!     =   3)
653    (description-primitive       'make-vector     make-vector     =   1)  ;   ou  2
654    (description-primitive       'pair?           pair?           =   1)
655    (description-primitive       'symbol?         symbol?         =   1)
656    (description-primitive       'number?         number?         =   1)
657    (description-primitive       'string?         string?         =   1)
658    (description-primitive       'boolean?        boolean?        =   1)
659    (description-primitive       'vector?         vector?         =   1)
660    (description-primitive       'char?           char?           =   1)
661    (description-primitive       'equal?          equal?          =   2)
662    (description-primitive       '+               +               >=  0)
663    (description-primitive       '*               *               >=  0)
664    (description-primitive       '-               -               =   2)
665    (description-primitive       '=               =               =   2)
666    (description-primitive       '<               <               =   2)
667    (description-primitive       '>               >               =   2)
668    (description-primitive       '<=              <=              =   2)
669    (description-primitive       '>=              >=              =   2)
670    (description-primitive       'remainder       remainder       =   2)
671    (description-primitive       'display         display         =   1)  ;   ou  2
672    (description-primitive       'newline         newline         =   0)  ;   ou  1
673    (description-primitive       'read            read            =   0)
674    (description-primitive       'erreur          deug-erreur     =   3)  ))
675
676 ;;;;        }}}  Environnement-initial
677 ;;;;   }}}  Environnements-H     (barrière   de  haut  niveau)
678
679 ;;;;   {{{  Mode  d'emploi
680 ;;;;  NOTA:  sous  DrScheme,   on  doit  faire  tourner   ce  code  dans  un  environnement    où  est
681 ;;;;  définie   la  fonction   «erreur»,   pour  faire  tourner   ce  code  sous  d'autres   systèmes
682 ;;;;  Scheme,  il  faut  définir   une  fonction   «erreur»   d'arité   supérieure   ou  égale  à  2.
683
684 ;  Mise  en  oeuvre  sous  DrScheme:
685 ;  ouvrir  fichier  eval4fr.scm   puis  évaluer   (deug-eval   '(+  2  3))
686 ;  puis  évaluer
687 ;  (deug-eval   '(let  ((a  5))  (define   (f  n)  (if  (=  n  0)  1  (*  n  (f  (-  n  1)))))
688 ;                                (f  a)))
689 ;  Pour  auto-interpréter     l'évaluateur,    il  suffit  d'écrire   (en  supposant
690 ;  que  <DEUGEVAL>   est  le  programme   définissant   deug-eval:
691 ;  (deug-eval   '(let  ()  <DEUGEVAL>   (deug-eval   '(+  2  3))))
692 ;  puis
693 ;  (deug-eval   '(let  ()
694 ;     <DEUGEVAL>
695 ;     (deug-eval   '(let  ((a  5))  (define   (f  n)  (if  (=  n  0)  1  (*  n  (f  (-  n  1)))))
696 ;                                      (f  a)))))
697 ;;;;   }}}  Mode  d'emploi
698 ;;;;   end  of  eval4fr.scm
```