Programmation fonctionnelle
Langage scheme

ENSEIRB

Octobre 2001, v1.2

Pour toute question/suggestion : mailto:sun@tuxfamily.org

Table des matières

Chapitre 1

Introduction générale

1.1 Historique des langages

Le langage Lisp est issu des besoins de l'intelligence artificielle dans les années 50. Il est utilisé pour :

- les démonstrations automatiques
- les jeux
- les représentations de la connaissance

Il permet la représentation de données symboliques. Dans les années 56-58, John Mac Carthy (du MIT) met au point le langage Lisp. Le COMMON LISP est une tentative de définition d'un Lisp universel à partir des Lisp précédents. Le scheme, où les fonctions sont définies plus proprement, voit le jour dans les années 70-80. Le e-Lisp est le langage d'extension du meilleur éditeur au monde : GNU Emacs. Il existe beaucoup d'autres Lisp : Hope (créé dans les années 80), ML (créé en 74). Tous les exemples donnés ici sont en guile , qui est un langage d'extension des logiciels du projet GNU. guile est livré en standard avec plusieurs distributions de Linux. C'est un langage de communication entre les différents langages.

1.2 Fondements mathématiques

Le Lisp est utilisé pour le lambda-calcul (Alonzo Church, 19^{e} siècle) et en Théorie des fonctions. Il y a deux concepts majeurs en Lisp :

- l'abstraction fonctionnelle
- l'application

En Lisp, il n'y a que des fonctions. On peut définir des opérations sur ces fonctions. Par exemple, la curryfication permet de ne considérer que des fonctions à 1 seul argument. Le langage Lisp est un ajout au lambda-calcul. On trouve en Lisp :

- un contexte global (pour éviter le passage de trop de paramètres)
- des types de base
- un choix de stratégie d'évaluation des expressions : passage des paramètres par valeur
- définition de formes spéciales : if (condition, alors, sinon), affectation

1.3 Programmation fonctionnelle

programmation	séquencement	exemple
fonctionnelle	composition	g(f())
impérative	en séquence	f() ; g()

Effets de bord :
- modification du contexte (affectation)
- Entrée/Sortie

1.3.1 Programmation impérative

Chaque instruction fait un effet de bord.

```
f();
g();
```

Cette séquence n'a un sens que si f et g font chacune un effet de bord.

1.3.2 Programmation fonctionnelle pure

Aucun effet de bord (ex : Hope). En scheme et en ML, il y a des affectations et des E/S.
Séquence :
- composition g(f())
- la séquence impérative n'a pas de sens.
- typage dynamique :
  ```
  (define a 0)
  (define a +)
  ```
  - inconvénients : fiabilité (le compilateur ne peut pas vérifier les typages des corps)
  - avantages : les syntaxes sont moins verbeuses. On peut définir des fonctions génériques.

L'interprétation et la compilation définissent une boucle d'interaction : REP, Read, Eval, Print.
Gestion de la mémoire : Lisp utilise la technique du ramasse-miette : On alloue automatiquement de la mémoire à la demande d'un nouvel objet. La mémoire est libérée automatiquement quand il n'y a plus de place (les objets présents en mémoire et non utilisés sont détruits.).

1.4 Bibliographie

- Structure et interprétation des langages
  Abelson et Sussman
  (Interédition)
- Programmer avec scheme
  Jacques Charazian
  (ITP)

Chapitre 2

Introduction au langage scheme

2.1 Eléments du langage

- les commentaires : `;`
- les nombres : 437 -3.14
- les booléens : `#f #t`
- les chaînes de caractères : `"Abcd"`
- les symboles (fun) : toute suite de caractères sauf les parenthèses et les espaces : `+rat *global* null? set!`
- les expression symboliques S-expr : nombre,booléen,chaîne,symbole
  `(S-expr1 S-expr2 .. S-expr_n)` : liste d'éléments séparés par des blancs

2.2 La boucle d'itération (ou top level)

REP : read (lecture S-expr), eval (évaluation), print (affichage+prompt).
Une session est une suite d'évaluations (ou top-level)
Convention : ? S-expr $->$ résultat (? : Prompt de l'interprète)

2.3 Ecriture des expressions

Les expressions sont préfixées et complètement parenthésées : (op arg1 arg2 ... arg_n)
ex : $\sqrt{\frac{\sin a^2+b^2}{y}}$ s'écrit :

```
(sqrt (/ (sin (+ (* a a) (* b b))) y))
```

Indentation :

```
(op arg1 arg2 ... arg_n)
```

ou

```
(op arg1
    arg2
    ...
    arg_n)
```

2.4 Définitions

2.4.1 Fonctions

(define `(nom arg1 arg2 ... arg_n) corps)`
`corps` est une suite d'expressions : `expr1 ... expr_n`
La fonction renvoie la valeur de la dernière expression.
ex :

```
(define (f x y) (+ x y))

?(f 1 3) -> 4
```

Notations : `(op arg1 arg_n)`
Evaluation :
- évaluation de `arg1 ... arg_n` dans un ordre non spécifié.
- application de op aux valeurs obtenues

Formes spéciales : fonctions particulières qui ne suivent pas ce schéma d'application

2.4.2 Variables

(define `symbole valeur)`
symbole est le nom de la variable.
ex :

```
? (define A "Abcde")
? (define A 0)
```

Action : ajout dans le dictionnaire de l'environnement global de la liaison : nom=A valeur=0
(A : nom du symbole, 0 : valeur entière)
define est une forme spéciale : seul son deuxième argument est évalué.

2.4.3 Définitions imbriquées

```
(define (somme_carre a b)
  (define (carre a)
     (* a a))
  (+ (carre a) (carre b)))
```

`carre` est une fonction locale : `? (carre 2)` $->$ `erreur`

2.4.4 Résultat indéfini

Le résultat d'un `define` dépend de l'interprète : Il peut être :
- le symbole qui est défini
- la valeur associée
- une valeur spéciale : `#unspecified` (spécifiant que la valeur est indéfinie)

En guile, c'est le troisième choix qui à été implémenté.
Cette valeur n'est pas affichée dans la boucle REP

2.4.5 Évaluation

- Atomes : le résultat de l'évaluation d'un atome est lui-même (Un atome est un nombre, un booléen ou une chaîne de caractère) :
  `? 1 -> 1`
- Symboles : `? A ->` Recherche de la valeur associée au symbole depuis l'environnement dans lequel se fait l'évaluation jusqu'à l'environnement global (Si rien n'est trouvé : erreur)
- S-expr `(symbole exp1 ... exp_n)` On commence par rechercher le symbole dans l'environnement. Dans le cas d'une forme spéciale, le mécanisme d'évaluation associé à cette forme est utilisé. Dans les autres cas, le mécanisme d'évaluation standard est appliqué.
- Forme spéciale **quote** ou **'** : ses arguments ne sont pas évalués
  ```
  ? (quote a) -> a
  ? (quote 1) -> 1
  ? (quote (+ a 3)) -> (+ a 3)
  ? (quote (1 2 3 4)) -> (1 2 3 4)
  ```
  `(quote arg)` s'écrit plus simplement 'arg

2.5 Autres opérateurs

2.5.1 Expressions arithmétiques

*** + - / 1+ 1- abs exp log sin modulo remainder sqrt**

- `1+` est l'incrémentation (de 1), `1-` est la décrémentation.
- `remainder` est le reste d'une division : `(remainder 5 3) -> 2`.
- `log` est le logarithme népérien.
- `sqrt` est la racine carrée d'un nombre.
- les fonctions `- + * /` peuvent avoir un ou plusieurs arguments.
  ```
  (* 2 2 2) -> 8
  (+ 2 2 2) -> 6
  (- 6 3 2) -> 1
  (/ 2)     -> 0.5
  (/ 8 2 2) -> 2
  ```

2.5.2 Opérateurs booléens relationnels

- sur les nombres : = <= >= < > **eqv ?** (mieux que `eq ?` pour les nombres longs)
- sur les booléens : **and or not**
- sur les symboles : **eq ?** : `(eq ? 'a 'b) -> #f`
- sur les listes : **equal ?** : `(equal ? '(1 2 3) '(1 2 3)) -> #t`
- prédicat de type : **number ? real ? integer ? string ? procedure ?**
  (Un prédicat est une procédure à valeur booléenne.)
  Teste si un objet est du type demandé (`procedure ?` teste si l'objet est une fonction).
- prédicat numérique : **zero ? odd ? even ?**
  `zero ?` teste si un nombre est égal à 0.
  `odd ?` teste si un nombre est pair (`even ?` : impair).
- autre prédicat : **null ?** : teste si une liste est vide.

2.5.3 Opérateurs conditionnels

Ce sont des formes spéciales.

- **(if** `cond alors sinon)`
  La condition `cond` est évaluée. Si elle est vraie (différente de `#f`), `alors` est évalué et on renvoie sa valeur de retour. Dans le cas contraire, `sinon` est évalué et on renvoie sa valeur de retour.
  ex : (if (zero ? x) 0 1)
- **(cond** `clause1 ... clause_n)`
  Les clauses sont évaluées séquentiellement jusqu'à un succès.
  Évaluation d'une clause : `(condition corps)`
  La condition est évaluée. Si le résultat est faux, c'est un échec, on passe à la clause suivante. Sinon, `corps` est évalué et on renvoie sa valeur de retour.
  ex :
  ```
  (define (type? x)
     (cond ((number? x) ‘‘x est un nombre’’)
           ((string? x) ‘‘x est une chaîne’’)
           (else ‘‘x est une paire pointée’’)))

  ? (type? 0) -> ‘‘x est un nombre’’
  ```
  La clause **else** est équivalente à `#t`.
- **(case** `expr clause1 ... clause_n)`
  Evaluation de l'expression, puis évaluation des clauses.
  Une clause est : `(liste corps)`
  Si le résultat de `expr` appartient à la liste, alors on évalue le corps et on sort avec le résultat de la dernière expression du corps. Le test fait en début de clause est `(memv expr liste)`
  ```
  (define (traduit-pronom p)
    (case p
      ((je) 'I)
      ((nous) 'We)
      (else "pronom inconnu")))
  ```
- **(begin** `e1...en)` permet de faire des blocs d'instructions. (ex : `(if cond (begin ...))`)

2.6 Opérateurs avec déclarations locales

- **let** : association en parallèle
- **let*** : association en séquence (permet a=1, b=a+1)
- **letrec** : récursif

Syntaxe : `(let liste_association corps)`
liste_association : `(assoc1 assoc2 ... assoc_n)`
assoc : `(nom expression)`
ex :

```
(let ((a 2)
      (b 6))
  (+ (* a a) (* b b)))
```

2.7 Opérateur lambda

Syntaxe : `(`**`lambda`**` (arg1 ... arg_n) corps)`
ex : `(lamdba (x y) (+ x y))`
la forme lambda permet une abstraction fonctionnelle.
Évaluation : les arguments ne sont pas évalués, le corps non plus : le résultat est une fonction. Cette valeur fonctionnelle peut être utilisée comme n'importe quel autre type de valeur : affectation, passage en argument lors d'une application, retournée en résultat de fonction... En plus, on peut l'appliquer.
Exemple d'application : `((lambda (x y) (+ x y)) 1 2) -> 3`
Exemple de définition :

```
(define plus
  (lambda (x y) (+ x y)))
```

est équivalent à

```
(define (plus x y)
  (+ x y))
```

Autre exemple :

```
(define (calcul x y)
  (let ((plus (lambda (x y) (+ x y))))
    (* x (plus x y))))
```

est équivalent à

```
(define (calcul x y)
  (define (plus x y) (+ x y))
  (* x (plus x y)))
```

2.8 Règles de visibilité

En Lisp, on peut imbriquer des environnements. On se pose ici les questions de la portée d'une déclaration, de la durée de vie d'un symbole et des règles de visibilité d'un symbole.

2.8.1 Portée

Quand on déclare un symbole, quelle est la zone du programme qui peut utiliser ce symbole ?
Niveau global : le programme en entier.
Niveau local : l'ensemble local plus les environnements imbriqués.

2.8.2 Durée de vie

Global : durée de l'exécution.
Local : durée de vie de l'environnement dans lequel il est déclaré. Potentiellement, un environnement local peut exister jusqu'à la fin de l'exécution (en particulier si on garde un pointeur dessus).

2.8.3 Visibilité

Deux possibilités :

- Gestion d'environnement statique
  Les noms de symboles déclarés localement masquent les mêmes nom éventuellement déclarés dans les environnements englobants.
  ex :
  ```
  (define a 6)

  (define (f a)
   (* a (puissance 2)))

  (define (puissance n)
    (if (zero? n)
      1
      (* a (puissance (1- n)))))
  ```
  D'où (ex : **scheme**) :
  `? (f 3) −> 3*6š`
- Gestion d'environnement dynamique
  Lors de l'application d'une fonction f à ses arguments effectifs, le corps de f est évalué dans le contexte de l'application de la fonction (au lieu dans le cas statique du contexte de définition de f).
  ex (**e-lisp**) :
  `? (f 3) −> 3*3š`

2.9 Récursivité

ex : factorielle

```
(define (fact n)
  (if (zero? n)
    1
    (* n (fact (1- n)))))
```

? `(fact 3) −> (* 3 (fact 2)) (* 2 (fact 1) (* 1 (fact 0))`
et `(fact 0) −> 1`
Processus récursif : nécessite une pile pour s'exécuter. On peut supprimer cette pile : **Récursivité terminale**
Le retour de l'appel récursif donne directement la valeur de la fonction qui l'appelle. Une fonction est récursive terminale si tous ses appels sont terminaux.
i.e. : Pour une fonction f, la dernière expression du corps est du type (f ...). Du coup, on n'empile rien, on n'a pas besoin de pile. C'est plus rapide. Il faut toujours essayer de faire des fonctions récursives terminales quand on fait beaucoup d'itérations.
ex :

```
(define (fact n r)
  (if (zero? n)
    r
    (fact (1- n) (* r n))))
```

Chapitre 3

Les listes

3.1 Expression symbolique

`(define semaine (lundi ... dimanche))` $->$ `erreur "lundi is not a function"`
On utilise la fonction quote. C'est pourquoi on parle d'expression symbolique.
(define semaine '(lundi ... dimanche))

3.2 Opérations de base

- **car** : renvoie le premier élément de la liste
  `(car '(1 2 3)) -> 1`
- **cdr** : renvoie le reste de la liste
  `(cdr '(1 2 3)) -> (2 3)`
- **cons** : rajoute un élément au début de la liste.
  `(cons 1 '(2 3)) -> (1 2 3)`
  `(cons '(1) '(2 3)) -> ((1) 2 3)`
  `(cons 1 '()) -> (1)`

3.3 La liste vide

```
(cdr '(6)) -> ()
```

La liste vide est **'()**

3.4 Composition des car et des cdr

Certaines fonctions permettent de simplifier l'écriture de `car` et de `cdr` imbriqués.
ex : accès au deuxième élément : `(car (cdr l))`
Il existe une fonction équivalente : `cadr`
Plusieurs fonctions du même genre existent (jusqu'à 4 imbrications de `car` et de `cdr`) : il suffit pour un `car` de mettre un "a" et pour un `cdr` de mettre un "d".

```
(caar l) <=> (car (car l))
(cadar l) <=> (car (cdr (car l)))
```

3.5 Fonctions sur les listes

- **list** : construit la liste de ses arguments
  `(list 1 2 3) -> (1 2 3)`
  `(list 1 2 '(1 2 3)) -> (1 2 (1 2 3))`
- **append** : concaténation de listes
  `(append '(1 2) '(3 4)) -> (1 2 3 4)`
- **length** : longueur d'une liste
- prédicat
  - **null ?** : teste si la liste est vide
  - **equal ?** : teste d'égalité pour les listes
    Différence avec `eq ?` : `eq ?` teste si les adresses en mémoire sont égales.
    ```
    (set! l1 '(1 2))
    (set! l2 '(1 2))
    (set! l3 l1)
    (eq? l1 l2) -> #f
    (eq? l1 l3) -> #t

    (eq? 1 1) -> #t
    (eq? 'a 'a) -> #t
    (eq? "a" "a") -> #f
    ```
  - **list ?** : teste s'il s'agit d'une liste
  - **member ?** : teste si l'élément appartient à la liste (test récursif avec equal ?)
  - **memq** : pareil que member ? avec eq ?
  - **memv** : pareil que member ? avec eqv ?

3.6 Programmation des listes plates

Une liste plate est une liste qui ne contient pas de sous-liste.

3.6.1 Longueur d'une liste

On a :

```
long[(a1 ... an)]=n
long[()]=0
long[(cons premier reste)]=1+long[reste]
```

D'où l'implémentation en scheme :

```
(define (long l)
  (if (null? l)
    0
    (+ 1 (long (cdr l)))))
```

Problème : `(long 2) -> error in function cdr : 2`
(il faudrait rajouter un test sur l'argument)

3.6.2 Concaténation de listes

```
concat[(1 2 3),(a b)]=(1 2 3 a b)
concat[(),l2]=l2
concat[(cons a l),l2]=(cons a (concat[l,l2]))
```

```
(define (concat l1 l2)
  (if (null? l1)
    l2
    (cons (car l1)
          (concat (cdr l1) l2))))
```

NB : `l2` n'est pas recopié, il y a partage des ressources

```
(define l3 (concat '(a) l))
(eq? l (cdr l3)) -> #t
```

3.6.3 La fonction iota

Définition : `? (iota 3) -> (3 2 1)`

```
(define (iota n)
  (if (zero? n)
    '()
    (cons n iota (1- n))))
```

Version récursive terminale :

```
(define (iota2 r l)
  (if (zero? n)
    l
    (iota2 (1- n) (cons n l))))

? (iota2 3 '()) -> (1 2 3)
```

L'introduction de `cons` dans l'appel récursif inverse l'ordre de la liste

3.6.4 Inversion de liste

Inversion naïve :iv

```
(1 2 3) -> (3 2) concat (1)
iv[()]=()
iv[(cons a r)]=append[iv[r],(a)]
```

```
(define (iv l)
  (if (null? l)
    '()
    (append (iv (cdr l))
            (list (car l)))))
```

Un `append` étant une série de `cons`, cette version est loin d'être optimale.

Inversion linéaire

```
(define (inv l1 l2)
  (if (null? l1)
    l2
    (inv (cdr l1) (cons (car l1) l2))))
```

appel : `(inv l '())`

3.6.5 Copie d'une liste

```
(define (copie l)
  (if (null? l)
    '()
    (cons (car l)
          (copie (cdr l)))))
```

3.7 Programmation des listes quelconques

Exemple : suppression d'un élément

– liste plate :
```
(define (supprimer element liste)
  (cond ((null? liste) '())
((eq? (car liste) element) (supprimer element (cdr liste)))
(else (cons (car liste) (supprimer element (cdr liste))))))
```
– liste quelconque :
```
(define (supprimer* element liste)
  (cond ((null? liste) '())
((list? (car liste)) (cons (supprimer* element (car liste))
   (supprimer* element (cdr liste))))
((eq? (car liste) element) (supprimer* element (cdr liste)))
(else (cons (car liste) (supprimer* element (cdr liste))))))
```

3.8 Les paires pointées (ou doublets)

(cons `1 2) -> (1 . 2)`
C'est une paire pointée. On la représente comme suit :

```
 /\
1  2
```

Pour tester si un objet est une paire pointée, on dispose du prédicat **pair ?** (On définit parfois son contraire, le prédicat `atom ?`).

```
(pair? '(1 . 2)) -> #t
(pair? '()) -> #f
(pair? '(1 2 3 4)) -> #t
```

Représentation de `(1 2 3 4)` :

```
  /\
 1 /\
  2 /\
   3 /\
    4 ()

[][]-> [][]-> [][]-> [][]-> ()
 \      \      \      \
  1      2      3      4
```

Une liste est en fait une succession de paires pointées dont le dernier élément est `()`. Ceci permet de caractériser une liste.

Dans le cas général, les paires pointées permettent de représenter des structures plus complexes telles que les arbres binaires étiquetés sur les feuilles.

3.9 Autres structures

3.9.1 Les arbres n-aires

On s'intéresse ici aux arbres n-aires étiquetés. Pour en faire des arbres étiquetés sur les feuilles, on ajoute un fils à la première position qui porte l'étiquette de son père (Pour bien comprendre, le mieux est de regarder le dessin ci dessous.). On peut transposer ces arbres en arbres binaires étiquetés sur les feuilles.

```
   1__4              .__4                        /\___.-- pas de noeud ici
  2/\               /|\                         1 /   \
  /\ 3             1 | 3                         /\   /\
 5 6\      ==>      /|\            ==>          2 /\ 3 /
    7              2 5 \                         5 /  4
                       /\                         /\
                      6  7                       6  7
```

3.9.2 Les a-listes

Ce sont des listes d'associations. Elles sont très utiles pour les dictionnaires, les tables, les fonctions :
`((clé1 . val1) ... (clé_n . val_n))`
`(assq clé a-liste)` : renvoie la paire pointée admettant clé dans son car (le test est fait avec `eq?`. Pour faire le test avec `equal`, il faut utiliser la fonction **assoc**)

```
(assq 'a '((b . 2) (a . 1) (c . 3))) -> (a . 1)
```

3.10 Définition d'objet

On définit pour tout objet des constructeurs (construction de l'objet), des accesseurs, qui permettent d'accéder à l'objet et des prédicats, qui font des tests sur l'objet. Ceci permet de cacher la structure de l'objet (en scheme, tout les prédicats se terminent par **?**).

Un exemple : les rectangles

```
                                              .
      ^                    /\               / \
      |    L              x /\             /\ /\
      |  o----o            y /\           x y L H
      |  |    | H           L /\
    y-|  o----o               H  ()
      |
      o----------->
         |
         x
```

Données : x, y, L, H
Représentation : liste (x y L H), ou paire pointée ((x . y) (L . H)). Le mieux serait d'utiliser les paires pointées.

Constructeur : cons-rectangle x y L H
Accesseurs : valeur-x, valeur-y, ...
Prédicat : dans ? point rectangle

On choisit ici comme implémentation la liste (x y L H)

```
(define cons-rectangle list)

(define valeur-x car)
(define valeur-y cadr)
(define valeur-L caddr)
(define valeur-H cadddr)

(define (dans? x0 y0 rectangle)
  (let ((x (valeur-x rectangle))
        (y (valeur-y rectangle))
        (L (valeur-L rectangle))
        (H (valeur-H rectangle)))
  (and (>= x0 x) (<= x0 (+ x L)) (>= y0 y) (>= y0 (+ y H)))))
```

3.11 Quasi-citation (quasi-quotation)

Notation : `‘s` ou `(`**quasiquote** `s)`
La quasiquote est semblable à la quote à ceci près que certains arguments peuvent modifier son comportement : `,` `,@`

– la virgule force l'évaluation :
`(define x 1)`
`` `(x ,x) -> (x 1) ``
– ,@ est utilisé dans le cas où l'évaluation retourne une liste. Il force l'évaluation et concatène la liste résultat :
`(define l '(1 2))`
`` `(,l ,@l) -> ((1 2) 1 2) ``

Exemple : Il existe deux façon différentes pour définir des fonctions
1 : `(define nom (lambda (par1 ... par_n) corps))`
2 : `(define (nom par1 ... par_n) corps)`

```
(define (f1->2 forme)
  (let ((nom (caadr forme))
        (corps (cddr forme))
        (l-param (cdadr forme)))
    `(define ,nom (lambda ,l-para ,@corps))))

(f1->2 '(define (f x y) x y)) -> (define f (lambda (x y) x y))
```

Chapitre 4

Les fonctionnelles

En anglais : higher order functions : fonctions pouvant recevoir en paramètre des fonctions et renvoyer des fonctions.

4.1 Fonctions en paramètre

```
(define (f-zero f)
  (f 0))
? (f-zero 1+)  -> 1
? (f-zero 0)   -> erreur: 0: not a procedure
? (f-zero car) -> erreur: 0: not a pair
```

Rappel : le prédicat `procedure ?` permet de tester si un objet est une fonction.

4.2 Fonction en résultat

Rappel : (lamdba (p1 ... pn) corps)

4.2.1 Multiplication par m

```
(define (multiplier_par_m m)
  (lambda(x) (* m x)))
(define double (multiplier_par_m 2))
? (double 3) -> 6
```

4.2.2 Dérivée d'une fonction f

On prend $f'(x) = \frac{f(x+dx)-f(x)}{dx}$

```
(define (deriver f dx)
  (lambda (x)
    (/ (- (f (+ x dx))
          (f x))
       dx)))
```

4.2.3 Composition de fonctions

```
(define (o f g)
  (lambda(x) (f (g x))))
(define 5-ieme (o car cddddr)) -> (lambda(x) (car (cddddr x)))
```

4.2.4 Curryfication

Opération sur le nombre d'argument des fonctions :

$f :\quad D_1 \times D_2 \rightarrow D_2$

$\quad (x, y) \mapsto f(x, y)$

$curry(f) = f^n : D_1 \rightarrow (D_2 \rightarrow D_3), x \mapsto (y \mapsto f(x, y))$

Addition : $+ : \Re \times \Re \rightarrow \Re, (x, y) \mapsto x + y$

$x \mapsto (y \mapsto x + y)$

$x = 1 \mapsto (y \mapsto y + 1)$

```
(define (curry f)
  (lambda(x)
    (lambda(y) (f x y))))
(define curry+ (curry +)) -----> (lambda(x) (lambda(y) (+ x y)))) où f=+
(define 1+ (curry+ 1)) -----> (lambda(y) (f x y)) où x=1 et f=+
```

4.2.5 Fonctionnelle récursive

On construit une fonction non récursive à partir d'une fonction récursive. On itère une fonction n fois :

si n=0, $x \mapsto x$ (identité : (lambda (x) x))

sinon $fo(iterer(f(n-1)))$

On obtient en résultat une fonction non récursive.

```
(define (iterer f n)
  (if (zero? n)
    (lambda (x) x)
    (o f (iterer f (1- n)))))

(define 3+ (iterer 1+ 3)) ===> (o 1+ (o 1+ (o 1+ (lamdba (x) x))))
```

Attention :

```
(define (iterer2 f n)
  (lambda (x)
    (if (zero? n)
        x
(o f (iterer2 f (1- n))))))
```

La lambda expression construit une fonction à chaque appel de `iterer2`. Cette fonction appelle `iterer2` au moment de son application : la boucle récursive se fait lors de l'application du résultat et à chaque tour, il y a construction d'une nouvelle fonction qui est aussitôt perdue.

```
(define 3+ (iterer2 1+ 3)) -> (lambda (x) (if (zero? n) ...
```

4.2.6 Fonctionnelle construisant une fonction récursive

Calcul par approximations successives (ex : racine carrée : $\|{a_n}^2 - d\| < \epsilon$). On connaît a_0. On définit a_n : $a_n = suivant(a_{n-1}, d)$. On a une condition d'arrêt : $arret?(a_n, d, \epsilon)$

On obtient en résultat une fonction récursive à 3 arguments : a_0, d et ϵ. La fonctionnelle est une fonction non récursive à deux paramètres : suivant et arrêt ?.

```
(define (approximation suivant arret?)
  (define (resultat a d e)
    (if (arret? a d e)
      a
      (resultat (suivant a) d e)))
  resultat)
```

4.3 Fonctionnelles et listes

4.3.1 La fonction map

$map(f, (x_1...x_n)) \mapsto (f(x_1)...f(x_n))$: $map(fonction, liste) \mapsto liste$

```
(define (u-map f l)
  (if (null? l)
    '()
    (cons (f (car l))
          (u-map f (cdr l)))))

(u-map 1- '(1 2 3 4)) -> (0 1 2 3)
```

map est plus général : soit n l'arité (le nombre d'arguments) de son argument fonctionnel ; alors il faut lui donner n listes de k éléments qui constituent les arguments de la fonction.
$map(f, (x_{11}...x_{1k})...(x_{n1}...x_{nk})) \mapsto (f(x_{11}...x_{n1})...f(x_{1k}...x_{nk}))$

Remarque : `and` et `or` ne sont pas des fonctions (`(map and ...) -> error`). Une solution est :

```
(map (lambda (x y) (and x y)) '(#f) '(#t))
```

Exemple :

```
(map odd? '(1 2 3)) -> (#f #t #f)
(map append '((a b) c) '((d) (e f g))) -> ((a b d) (c e f g))
(map (lambda (x) '(cons 'Monsieur x)) '(Durand Dupond))
```

4.3.2 La fonction apply

Cas où l'on dispose d'une fonction et de la liste de ses arguments. `(f lst-arg) -> error` : la liste est considérée comme étant l'argument de f.
On utilise (**apply** `f lst-arg`)

```
(apply + '(1 2 3)) -> 6
(apply + '(1 (+ 2 3))) -> error : <==> (+ 1 '(+ 2 3))
```

Ex : produit scalaire : $l1 = (x_1...x_n)$ $l2 = (y_1...y_n)$

```
(define (prod-scal l1 l2) (apply + (map * l1 l2)))
```

4.3.3 Autres exemples utiles

map et apply sont des fonctions de base qu'on trouve dans tous les Lisp. Les fonctions suivantes sont implémentées seulement dans certains Lisp (mais pas en scheme).

- **append-map** : Dans le cas où la fonction à appliquer renvoie une liste et qu'on souhaite concaténer tout les résultats
  ```
  (define (append-map f l)
    (apply append (map f l)))
  ```
  Ex : redéfinition de la fonction aplatir
  ```
  (define (aplatir l)
    (if (list? l)
      (append-map aplatir l)
      (list l)))
  ```
- **map-select** : l'application de son argument fonctionnel est conditionnée par la vérification d'un prédicat.
  ```
  (define (map-select f l p?)
    (append-map (lambda (x)
                  (if (p? x)
                      (list (f x))
                      '()))
                l))
  ```
  Version récursive terminale :
  ```
  (define (map-select f l p?)
    (cond ((null? l) '())
          ((p? (car l)) (cons (f (car l))
                              (map-select f (cdr l) p?)))
          (else (map-select f  (cdr l) p?))))
  ```
  Exemple :
  ```
  (map-select / '(a 4 2)
              (lambda (x)
                (and (number? x) (not (zero? x))))) -> (0.25 0.5)
  ```

4.3.4 Extension de la syntaxe des lambda

- `(lambda (x1...xn) corps)`
  On peut avoir n=0 (pas d'argument)
- `(lambda (x1...xk . reste) corps)`
  `reste` peut être vide. Ceci permet de définir des fonctions avec un nombre d'arguments variable.

Syntaxe avec `define` : `(define (nom x1...xn . reste) corps)`
ex :

- u-list : met ses arguments en liste (comme `list`) :
  ```
  (define (u-list . l) l)
  ```

– moyenne : calcule la moyenne de plusieurs nombres :
`(define (moyenne . l) (/ (apply + l) (length l)))`

Chapitre 5

La programmation impérative en scheme

5.1 L'affectation

set ! : prononcer set bang
De façon générale, les noms des fonctions effectuant une modification de l'environnement se terminent par **!**.

```
(set! nom expression)
```

nom : nom de la variable à modifier. Ce nom doit être défini. expression est une expression quelconque qui sera évaluée. Résultat : nom spécifié (à ne pas utiliser)

ex : compter le nombre d'appel à cons

```
(define nb-appel 0)
(define old-cons cons)
(define (cons a b)
  (set! nb-appel (1+ nb-appel))
  (old-cons a b))
```

Modification des car et des cdr : (**set-car !** `nom val)`
(**set-cdr !** `nom val)`

ex :

```
(define p '(1 2))
(set-car! p 0)
? p -> (0 2)
```

ex : Concaténation de listes : **append !**

```
(define (append! l1 l2)
  (define (append-aux l1)
    (if (null? (cdr l1))
      (set-cdr! l1 l2)
      (append-aux (cdr l1))))
  (if (null? l1)
    (set! l1 l2)
    (append-aux l1)))
```

Attention : les listes ne sont pas recopiées en mémoire

```
(define l1 '(1 2))
(define l2 '(3 4))
(append! l1 l2)
(set-cdr! l2 '(17))
l1 -> (1 2 3 17)
```

5.2 Entrée/Sortie

Lecture : **read** : lit une expression et construit une représentation interne qu'elle renvoie en résultat.

```
? (define e (read))
(+ 1 a)
? e -> (+ 1 a)
```

(L'expression n'est pas évaluée)

Écriture : **display**
Pour afficher une ligne vide, il existe la fonction **newline** (équivalente à `(display "\n")`. Pour afficher une erreur et interrompre le programme, on utilise la fonction **error**.

5.3 Fermeture et état interne

5.3.1 Définition

Fermeture : représentation interne de la fonction.
[corps,environnement lexical] (environnement dans lequel la fonction a été créée)

5.3.2 Compte bancaire

(ajouts, retraits)
On conserve le solde dans l'environnement local.
Les opérations se font dans le corps de la fermeture.
La fonction de création de compte est `creer-compte`. Elle renvoie une fonction.

```
(define (creer-compte solde)
  (lambda (x)
    (set! solde (+ solde x))
    solde))
```

Exemples d'utilisation :

```
? (define jean (creer-compte 500))
? (jean 100) -> 600
? (jean -200) -> 400
```

```
? (define paul (creer-compte 100))
? (define pauline paul) ; compte joint
? (paul 100) -> 200
? (pauline 200) -> 400
```

5.3.3 Générateur

On va créer un générateur de la suite de Fibonacci (définie par $x_0 = 0$, $x_1 = 1$ et $x_n = x_{n-1} + x_{n-2}$). Le générateur de la suite, gen-fib, doit créer une fermeture contenant dans son environnement les deux dernières valeurs de la suite et dans son code le calcul de la nouvelle valeur et la mise à jour des anciennes valeurs.

```
(define (gen-fib)
  (let ((a 0) ; Création de l'environnement lexical de la fermeture avec let
        (b 1))
    (lambda ()
      (let ((c b))   ; Environnement local à la fermeture
        (set! b (+ a b))
        (set! a c)
        a))))

? (define f1 (gen-fib))
? (f1) -> 1
? (f1) -> 1
? (f1) -> 2
```

L'environnement du premier “let” est conservé. Celui du deuxième disparaîtra au prochain ramasse-miette.

Chapitre 6

Modularisation en scheme

6.1 Fermeture et modularisation

On veut par exemple représenter l'ensemble des rationnels :
Constructeur : `make-rat` accesseur : `numerator`, `denominator`
On définit trois fonctions globales :

```
(define (make-rat x y)
  (cons x y))
(define numerator cat)
(define denominator cdr)
```

Problème : en scheme, il n'y a pas d'outil symbolique pour définir des modules permettant :
- le partage de variables ou de fonction
- le masquage d'implémentation
- de garantir la cohérence des modules :
  ```
  ? (define r (make-rat 2 3))
  ? (set-car! r 'a)
  ? r -> (a . 3) ---> r n'est plus un rationnel ...
  ```

On peut utiliser la notion de fermeture pour définir :
- un ensemble de variables et de fonctions dans un même environnement qui est l'environnement local de la fermeture
- l'interface, implémentée par le corps de la fermeture

Exemple des rationnels :

```
(define (make-rat x y)
  (define (numerator) x)
  (define (denominator y))
  (define (interface function)
    (case function
      ((numerator) (numerator))
      ((denominator) (denominator))
      (else (error "Fonction inconnue"))))
  interface)
```

Utilisation :

```
? (define r (make-rat 2 3))
? (r 'numerator) -> 2
```

Remarque :
- cohérence : `r` ne peut être qu'appliquée, et dans ce cas, elle a le contrôle des opérations sur son environnement.
- l'environnement lexical permet le partage de variable et de fonction qui peuvent être cachées à l'intérieur.

r est un objet et 'numerator est un message. On peut définir une fonction send, qui envoie un message à un objet :

```
(define (send objet message . arguments)
  (apply objet message arguments))

(send r 'denominator)
```

6.2 Exemple de modularisation

Modularisation fonctionnelle, pile fonctionnelle mutable, modularisation avec fermeture.

6.2.1 Pile fonctionnelle pure

pure : pas d'effet de bord : les fonctions ne modifient pas les piles
fonctionnelle : ensemble de fonctions :
- creer-pile : $->$ pile
- empiler : pile Œ A $->$ pile
- depiler : pile $->$ pile
- sommet : pile $->$ A
- vide ? : pile $->$ booléen

```
(define (creer_pile)
  '())

(define (empiler pile element)
  (cons element pile))

(define (depiler pile)
  (if (vide? pile)
    (error "Pile Vide")
    (cdr pile)))

(define (sommet pile)
  (if (vide? pile)
    (error "Pile Vide")
    (car pile)))

(define vide? null?)
```

6.2.2 Pile fonctionnelle mutable

Il faut modifier l'implémentation de la pile de façon à ce que son adresse de début soit toujours la même. On choisit ici de rajouter un élément au début de la pile : `'pile`.

- pile ! :creer : − > pile !
- pile ! :empiler : pile ! Œ A − > pile !
- pile ! :depiler : pile ! − > pile !
- pile ! :sommet : pile ! − > A
- pile ! :vide ? : pile ! − > booléen

```
(define (pile!:creer)
  (cons 'pile '()))

(define (pile!:empiler pile element)
  (set-cdr! pile (cons element (cdr pile))))

(define (pile!:depiler pile)
  (if (pile!:vide? pile)
    (error "Pile Vide")
    (set-cdr! pile (cddr pile))))

(define (pile!:sommet pile)
  (if (pile!:vide? pile)
    (error "Pile Vide")
    (cadr pile)))

(define (pile!:vide? pile)
  (and (pair? pile)
       (null? (cdr pile))))
```

6.2.3 Pile avec fermeture

Effets de bords (modifications de l'environnement) sur la variable pile.

- notion d'état, éventuellement modifié par certaines fonctions
- pas de problème pour modifier la variable pile depuis l'une des fonctions car elle est interne.

```
(define (creer-pile)
  (define (empiler element)
    (set! pile (cons element pile)))
  (define (depiler)
    (if (vide? pile)
        (error "Pile Vide")
        (set! pile (cdr pile))))
  (define (sommet)
  (if (vide? pile)
    (error "Pile Vide")
    (car pile)))
```

```
(define (vide?)
  (null? pile))
(define (interface message . arg)
  (case message
    ((empiler) (apply empiler arg))
    ((depiler) (depiler))
    ((sommet) (sommet))
    ((vide?) (vide?))
    (else (error "Message inconnu"))))
interface)
```

Chapitre 7

Les macros

7.1 Définition

Les macros permettent de définir des formes spéciales. Les arguments d'une macro ne sont pas évalués (cela correspond à peu près au passage par adresse en C). Une macro construit une liste. Cette liste est ensuite évaluée.

Construction : **(define-macro** `(nom-macro p1...pn) corps)`

Appel : (nom-macro e1...en)

- liaison des paramètres formels p1...pn avec les arguments e1...en non évalués.
- évaluation du corps dans cet environnement.
- le résultat de l'évaluation constitue l'expansion de la macro qui est ensuite évaluée dans l'environnement d'appel.

On peut voir le résultat de l'évaluation à l'aide de la fonction **macroexpand**. Les instructions sont des listes. Ce qui est amusant dans le langage scheme, c'est qu'on peut reconstruire tout le langage à partir de seulement quelques instructions.

Exemple :

```
(define-macro (first l)
  `(car ,l))              ; <==>  (list 'car l)

(macroexpand '(first (1 2 3))) -> (car (1 2 3))
(first (1 2 3)) -> 1
(define a (1 2))
(macroexpand '(first a)) -> (car a)
```

NB : il faut toujours évaluer les variables. Si on ne l'avait pas fait, on aurait dans la dernière ligne `(car l)`, ce qui provoquerait une erreur puisque cette instruction serait évaluée dans le contexte d'appel ou la variable `l` n'existe pas (de toute façon, il n'y aurait pas double évaluation).

On peut tester si l'objet est une macro à l'aide du prédicat **macro ?**. Les formes spéciales sont des macros.

```
(macro? and) -> #t
(macro? if) -> #t
(macro? define) -> #t
(macro? define-macro) -> #t
```

7.2 La fonction eval

Dans le même genre, il existe une fonction qui permet d'évaluer un argument (en général, une liste d'instuctions) : `(eval argument)`

```
(eval 10) -> 10
(eval '(+ 1 2)) -> 3
```

7.3 Exemple

7.3.1 Nouvelle forme spéciale

On veut créer la forme `(when test corps)` où le corps est une suite d'instructions (contrairement à `if`). Ce corps est évalué seulement si le test est vrai.

```
(define-macro (when test . corps)
  `(if ,test
       (begin . ,corps)
       ()))          ; si c'est faux, on retourne la liste vide (c'est un choix)
```

7.3.2 Documentation de fonction

On veut créer une nouvelle forme, semblable à celle existante en e-lisp qui permet d'associer une documentation à une fonction, qu'on stocke par exemple dans une a-liste (ici `doc-liste`) :

```
(defun last (l)
  "Retourne le dernier element de la liste ou () si elle est vide"
  (if (null? l)
    '()
    (if (null? (cdr l))
      (car l)
      (last (cdr l)))))

(define-macro (defun f arg doc . corps)
  (set! doc-liste (cons (cons f doc) doc-liste))
  `(define (,f ,@arg) ,@corps))
```

7.3.3 Encore une pile

On reprend l'exemple de la pile. La pile est cette fois ci une variable globale. Les fonctions `pile ! :vide ?` et `pile ! :sommet` n'ont pas besoin d'être des macros

```
(define-macro (pile!:creer pile)
  `(define ,pile ()))

(define-macro (pile!:empiler pile element)
  `(set! ,pile (cons ,element ,pile)))

(define-macro (pile!:depiler pile)
  `(set! ,pile (cdr ,pile)))
```

Index