Programmation fonctionnelle en Scheme

Introduction

En 1958, Mc Carthy invente *Lisp* (*list processing*), le premier langage fonctionnel. Depuis de nombreux interpréteurs *lisp* ont été développé ; parmi eux *Emacs Lisp*, C*ommon Lisp* et *Scheme* (projet *GNU*). Le principe de fonctionnement d'un interpréteur est la boucle d'interaction *REP* (*read eval print*) : lecture d'une expression, évaluation d'une expression, et affichage du résultat.

Les langages fonctionnels se basent sur la théorie des fonctions, et en particulier sur les travaux de *Church (1941)* relatif au *λ-calcul*. Les concepts fondamentaux sont l'abstraction fonctionnelle, la notion d'application, et la stratégie d'évaluation des fonctions.

En programmation fonctionnelle, l'objet de base manipulé est *l'expression*. Le programme en lui-même est une expression, et elle va retourner un résultat. Par ailleurs, la composition des fonctions induit l'ordre des calculs. En *scheme* par exemple, on peut très bien programmer sans E/S et sans affectation ; toutefois ce langage n'est pas comme un langage fonctionnel pur qui interdirait complètement les effets de bord.

Guile (programmation hybride) sous *bash* permet d'interpréter les programmes s*cheme*. On notera que l'extension traditionnel des fichiers *scheme* est *scm*.

```
$ guile
?- (+ 2 (* 3 4))
14
?- (load "fichier.scm")
```

On peut avantageusement utilisé le *mode scheme* sous e*macs*, en éditant le source dans cet environnement.

- passage en mode scheme : *M–x scheme-mode* ou automatiquement avec *C–x C–f file.scm*
- lancement de l'interprète *guile* : *M–x run-scheme*
- évaluation d'une expression (*guile*) : *C–x C–e*.
- indentation : *C–i* ou *tab*
- indentation avec retour ligne : *C–j*

Sous *emacs*, on dispose du buffer **scratsh** qui est un interprète *lisp*. Les résultats sont dirigés dans le mini-buffer. L'évaluation d'une ligne est commandée par *C–x C–e*.

Table des matières

Expressions

Les expressions sont préfixées et complètement parenthésées : ***(op*** a_1 a_2 … a_n ***)***. L'*opérateur* est placé devant ses *arguments*. Chaque application est encadrée de parenthèses.

Expressions arithmétiques

On dispose des opérateurs traditionnels d'arité quelconque : **+ , - , * , /.**

```
?- (+ 1 2 3 4 5)
15
?- (/ 2)
2
?- (/ 8 2 2)
2
```

Lorsqu'il n'y a pas d'argument, le plus souvent, l'élément neutre est retourné :

```
?- (+)
0
?- (*)
1
```

Des fonctions issues de bibliothèques viennent compléter cette liste sommaire : ***abs*** , ***max*** , ***log*** , ***sin*** , ***remainder , modulo , quotient , sqrt , 1+ , 1- .***

Commentaires : *log* est le logarithme népérien, *remainder* est le reste de la division entière, *1+* est l'incrémentation de 1…

Atomes et Constantes du langage

- **Atomes** : l'évaluation d'un atome est lui-même.
  - nombres entiers (non limités en taille) : *-12554*
  - nombres réels : *1.5*
  - chaînes de caractères : "abcd"

```
?- 5
5
```

- **Constantes du langage**
  - booléens : ***#t*** et ***#f***
    En *lisp*, ***#f***, ***'( )*** et l'atome ***nil*** ont un résultat qui peut être considéré comme faux.
  - résultat indéfini (non affiché en *Scheme*) : *#unspecified*

Expressions symboliques ou *S-Expressions*

Les expressions symboliques sont des *symboles*, des *citations* ou des *listes*.

- Dans un environnement donné, **le symbole** est le nom de la variable, par lequel on fait référence à la valeur de la variable. Le symbole est une suite de caractères quelconques sauf . , *blanc* , *(,)* , *"* , et qui ne doit pas non plus désigner un atome.

```
?- pierre
erreur : symbole pierre non défini !
?- (define pierre 10)
?- pierre
10
```

- **la citation**

```
?- 'pierre
```

pierre

C'est l'abréviation de la forme spéciale ***quote***, qui n'évalue pas son argument.

*?- (**quote** pierre)*
pierre

- **la quasi-citation**

La ***quasiquote*** (ou accent grave : `)fonctionne comme la *quote* à ceci près que certains arguments peuvent modifier son comportement : notamment ***unquote*** (ou virgule ,) qui force l'évaluation et ***unquote-splicing*** (virgule arobasque : ,@) qui force l'évaluation en supprimant le premier jeux de parenthèse du résultat.

?- (define b 1)
?- `(a b) ou (quasiquote (a b))
(a b)
?- `(a ,b) ou (quasiquote (a (unquote b)))
(a 1)

?- (define L '(1 2))
?- `(0 ,L)
(0 (1 2))
?- `(0 ,@ L) ou (quasiquote (0 (unquote-splicing L)))
(0 1 2)

La quasi-citation prend toute son importance dans l'écriture des macros, où il s'agira de substituer du texte sans l'évaluer immédiatement.

- **les *listes***

On donne quelques exemples de listes, dites plates : *(1 2 3 4 5) , (lundi mardi mercredi).* De façon générale, une expression est une liste : *(+ (* x 2) 10).*

?- (1 2 3)
erreur : 1 n'est pas une fonction !

On utilise les citations pour manipuler les listes. En effet, la citation empêche l'application.

?- '(1 2 3)
(1 2 3)

On peut également utiliser la fonction ***list*** prévue à cet effet :

?- (list 1 2 3)
(1 2 3)

- On dispose de fonctions manipulant les symboles et les chaînes de caractères : *string-append* pour la concaténation, *string→symbol , symbol→string* pour des conversions…

?- (define name 'Aurélien)

?- (string-append "Bonjour " (symbol→string name))
"Boujour Aurélien"

?- (string→symbol "Bonjour")
Bonjour

Expressions booléennes

- Opérateurs relationnels : **>** , **<** , **>=** , **<=**

- Logique : ***not*** , ***and*** , ***or***. Ce sont des formes spéciales. Développer…

```
?- (and (> 2 3)
        (> 0 1))
#f
```

- ***eq?*** et ***equal?***
  On distinguera la fonction *eq?*, qui compare les adresses et la fonction *equal?*, qui compare les valeurs (parcours récursif des listes…) :

```
?- (eq? '(1 2 3)  '(1 2 3))
#f

?- (equal? '(1 2 3)  '(1 2 3))
#t

?- (let ((L1 '(1 2 3)) (L2 L1))
     (eq? L1 L2))
#t
```

Prédicats

Les prédicats sont des fonctions à valeurs booléennes. On donne ci-dessous la liste des plus courants…

- *number?*
- *real?*
- *integer?*
- *string?*
- *procedure?*
- *zero?*
- *odd?*
- *even?*
- *null?*
- *pair?*

Expressions conditionnelles

Là encore, il s'agit *de formes spéciales* évaluant leurs arguments de manière conditionnelle.

- ***if***

  *Syntaxe : (**if** condition alors sinon)*

  On évalue tout d'abord la *condition*. Si elle est vraie, alors on évalue l'expression *alors*, sinon on évalue l'expression *sinon* ; puis on retourne le résultat de l'expression évaluée.

  Exemple :

```
?- (if (> 2 3) 0 6)
6
```

- ***cond***

  *Syntaxe : (**cond** <clause 1> <clause 2> … <clause n> <clause else>)*
  avec *<clause> = (condition <corps>)*

  Les clauses sont évaluées dans l'ordre où elles apparaissent, de la façon suivante :
  - évaluation de la *condition* ;
  - si elle est fausse, on passe à la clause suivante (en l'absence de clause le résultat général du *cond* est faux) ;
  - sinon évaluation en séquence du *corps*, composé d'une suite d'expression, le résultat étant celui de la dernière.
  - Si aucune clause n'est vraie, on peut facultativement ajouter une *clause else = (else <corps>)*, qui sera évalué en dernier recours.

Exemple :

```
?- (cond ((number? x) 1)
         ((string? x) 2)
         ((null? x) 3)
         (else  4))
```

- ***case***

*Syntaxe : (**case** expression*
(<liste 1> <corps 1>)
(<liste 2> <corps 2>)
…
(<liste n> <corps n>)
(else <corps>))

Cette fonction est l'équivalent d'un *cond* dont la condition de branchement serait *(member 'expression liste).*

```
?- (case a
    ((0 2 4 6 8) (display "chiffre pair"))
    ((1 3 5 7 9) (display "chiffre impair")))
```

Lambda expressions

La *λ-expression* est une expression dont le résultat est une fonction. Elle se manipule comme les autres expressions : affectation, passage en paramètres…

*Syntaxe : (**lambda** <liste de paramètres> <corps>)*

Le *corps* est une suite d'expression évaluée séquentiellement, dont la dernière sert effectivement pour la définition de la lambda expression.

```
?- (lambda (x y) (+ x y))
```

Application d'une λ-expression :

```
?- ( (lambda (x) (* x x)) 2)
4
```

On commence par évaluer la *λ-expression* (construction de la représentation interne de la fonction) ; puis on évalue les paramètres (l'ordre d'évaluation des paramètres est inconnu) ; finalement, on évalue le corps de la fonction dans un contexte où les paramètres formels de la fonction sont liés aux valeurs des paramètres de l'appel (paramètres passés par valeur).

Evaluation d'une expression

- lire *expression*
- évaluer *expression*
- écrire le résultat

Si *l'expression* est un *identificateur*, alors on recherche cet identificateur dans l'environnement local de son apparition, etc. en remontant jusqu'à l'environnement global, s'il n'est pas trouvé. S'il est trouvé, le résultat est la valeur qui lui est associé, une erreur se produit : *identificateur non défini*. Dans le cas de multiple définition pour un même identificateur, c'est la plus haute dans la pile qui est utilisée.

Environnement et lambda expressions

Tout identificateur qui n'est pas un mot-clef syntaxique peut être utilisé comme *variable*. Une variable *liée* désigne un emplacement mémoire, où peut être conservée une *valeur*. En ce qui nous concerne, *l'environnement* représente l'ensemble des liaisons nom-valeurs qui sont accessibles à un instant donné, par un fragment de programme.

La plus fondamentale de ces *constructions liantes* est la lambda expression, car toutes les autres constructions peuvent s'exprimer en termes de lambda expressions. Les autres constructions sont le *let*, le *let**, le *letrec* et le *do*.

Il existe en *Scheme* un *environnement initial ou global*, qui contient des objets prédéfinis (comme *car*, *, ou *append*). On distinguera l'environnement global et *les environnements locaux ou dynamiques*.

Visibilité et persistance

Une entité est référencée dans un programme. La ***visibilité ou portée*** fait référence à la portion de texte d'un programme dans laquelle une référence à l'entité est possible. La ***persistance*** détermine l'intervalle de temps durant lequel une référence peut être faite à cette entité.

- *Portée lexicale* : c'est la visibilité habituelle des langages à structure de blocs, comme *Scheme*. Les références aux entités ainsi établies ne peuvent s'effectuer que dans la portion de programme qui est textuellement enchâssée dans la construction ayant créé l'entité. Par exemple, les paramètres d'une fonction sont accessibles uniquement dans le corps de la fonction.

- *Portée indéfinie* : Une entité a une portée indéfinie si on peut y faire référence n'importe où dans le programme. C'est le cas en particulier des variables globales, telles les fonctions prédéfinies du sytème : +, *, *max, sin, cons,* etc…

- *Persistance indéfinie ou illimitée* : Une entité a une persistance indéfinie si son existence se poursuit aussi longtemps que subsiste une possibilité d'y faire référence. Toutes les entités de *Scheme* jouissent d'une persistance indéfinie. Aucune n'est jamais détruite, et il n'existe pas de possibilité explicite de le faire. Cependant lorsqu'un objet n'est plus référencé, le système peut récupérer le mémoire qui lui a été allouée. C'est ce qui arrive, en général, aux valeurs des paramètres après exécution d'une fonction.

Notons qu'un identificateur peut être *masqué* par un autre identificateur de même nom, situé dans un bloc plus interne :

```
?- (define (f x)
     (define (g x)
       (+ x 2))
     (g (* x 3)))
```

Define

Lorsqu'une variable a été définie par une forme *define*, l'effet d'une nouvelle forme *define*, dans le même environnement, n'est pas d'ajouter une nouvelle liaison *nom / valeur* masquant la précédente, mais simplement de réaffecter une nouvelle valeur au même nom. La seconde forme *define* est donc équivalente à un *set!*.

- *définition de variables avec define*

  *Syntaxe : (**define** symbole expression)*

  Le *symbole* est relié au résultat de l'évaluation de *l'expression*.

```
?- (define a 0)
?- a
0
```

Par convention, on signale les variables globales en les encadrant avec le symbole '*' :

```
?- (define *global* 10)
```

- *définition de lambda expressions avec define*

Syntaxe : (define symbole (lambda (<liste de paramètres> <corps>)))

Pour la définition d'une *lambda-expression*, on dispose d'une facilité d'écriture :

Syntaxe : (define (symbole param_1 … param_n) <corps>)

Exemple :

```
?- (define (carre x)
     (* x x))
?- (carre 2)
4
```

A la différence d'un *define* traditionnel, le *corps* se compose d'une ou plusieurs expressions. Ces expressions sont évaluées séquentiellement et la dernière est associée au symbole.

```
?- (define (constante) 1 2 3)          ; fonction sans paramètre
?- (constante)
?- 3
```

Cette séquence prend tout son intérêt si elle produit des effets de bords, souvent utile pour la création d'un environnement local.

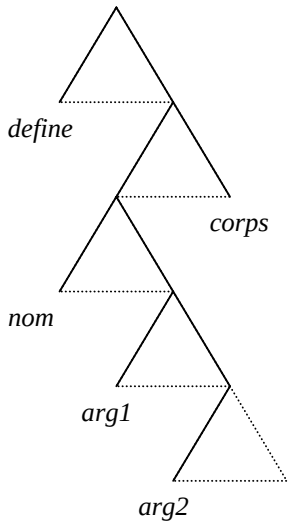
```
?- (define (norme_carre a b)
     (define a² (* a a))          ; variable locale
     (define b² (* b b))          ; variable locale
     (+ a²  b²))

?- (define (nome_carre a b)
     (define (carre x) (* x x))        ; fonction auxiliaire
     (+ (carre a) (carre b)))
```

Extension de la syntaxe des lambda

Pour mieux comprendre les extensions suivantes, on rappelle sur un exemple simple la représentation interne en liste des fonctions.

```
?- (define (nom arg1 arg2) corps)
```



- Le nombre d'arguments est connu :

  Syntaxes : *(lambda (x_1 … x_n) corps)*
  (define (nom x_1 … x_n) corps)

- Au moins 1 argument est connu, le reste des arguments sont placés dans la liste *reste* qui peut être vide :

  Syntaxes : *(lambda (x_1 . reste) corps)*
  (define (nom x_1 . reste) corps)

- Au moins *k* arguments connus , le reste dans la liste *reste* :

  Syntaxes : *(lambda (x_1 … x_k . reste) corps)*
  (define (nom x_1 … x_k . reste) corps)

- Plus généralement encore, les arguments sont tous placés dans la liste *liste* :

  Syntaxes : *(lambda (. liste) <corps>)*
  (define (nom . liste) corps)

On propose une méthode permettant de réinterpréter la liste des paramètres passés en argument à une fonction. *self* désigne la fonction elle-même.

```
?- (define (f a b c)
     (letrec   ((self (lambda (arg1 . reste_args) reste_args)))
       (self a b c)))

?- (f 1 2 3)
(2 3)
```

Expressions à environnement local : *let* et *let**

Syntaxe : ***(let*** *<liste d'affectations> <corps>)*

Le *let* permet de définir un environnement local, en affectant des valeurs à des variables. Le *corps* est une suite d'expressions évaluées séquentiellement dans cet environnement, et le résultat renvoyé par le *let* est celui de la dernière expression . En dernier lieu, l'environnement disparaît.

- *le let //* : Les définitions ne sont pas ordonnées, à la différence du *let séquentiel*.

```
?- (let ((x 1) (y 2))
     ( + x y))
3
```

Remarque : Ne pas oublier le double parenthèsage !

- *le let séquentiel*, autorise l'utilisation des affectations précédentes pour définir les suivantes *:* ***let****

```
?- (let* ((x 1) (y (* 2 x)))
     (+ x y))
3
```

- *le let mutuellement récursif :* ***letrec***

```
?- (letrec ((loop (lambda () (if test (begin expression (loop))))))
       (loop))
```

Environnement des λ-expression

La *λ-expression* compte des symboles de trois types :

- les symboles correspondant aux paramètres de la *λ-expression ;*

- les symboles définis localement à l'intérieur de la *λ-expression ;*

```
?- (lambda (x)
     (let ((a 0) (b 1))
          (+ x b a)))
```

- les symboles *libres* qui dépendent de l'environnement où apparaît la *λ-expression ;*

```
?- (define a 0)     ; variable globale
?- (define b 1)     ; variable globale
?- (lambda (x) (+ x b a))
```

Listes

Définition d'une liste

Tout simplement, en définissant une variable…

```
?- (define L '(1 2 3 4 5))
?- L
(1 2 3 4 5)
```

Lorsque tous les éléments de la liste sont des atomes de même nature, on parle de *liste plate*. De façon générale, il n'y a pas de restriction sur la nature des objets contenus par une liste ; on parle alors de liste quelconque.

```
?- (define L '(a (2) (3 4 5)))
?- L
(a (2) (3 4 5))
```

Manipulation des listes

On dispose de trois fonctions de base pour la manipulation récursive des listes : *cons* , *car* , *cdr*. On dispose en outre d'une fonction testant si la liste est vide.

- ***cons***

  Construction d'une liste dont le premier élément est le résultat de l'évaluation de son premier argument, et dont la suite est son deuxième argument.

  Syntaxe : (cons expression <liste>), le deuxième argument doit être une liste.

  ```
  ?- (cons lundi '(mardi mercredi))
  (lundi mardi mercredi)

  ?- (cons '( ) (1 2 3))
  (1 2 3)
  ```

- ***car***

  Renvoie le premier élément de son argument (le type du résultat varie selon le type des éléments de la liste).

  Syntaxe : (car <liste>)

  ```
  ?- (car '(1 2 3))
  1
  ```

- ***cdr***

  Renvoie la liste constituée de son argument privée de son premier élément.

  Syntaxe : (cdr <liste>)

  ```
  ?- (cdr '(1 2 3))
  (2 3)
  ?- (cdr '(1))
  ( )
  ```

- ***null?***

  Cette fonction retourne vrai si la liste est vide, sinon elle retourne faux.

  ```
  ?- (null? '( ))
  #t
  ```

Composition des deux accesseurs *car* et *cdr*

On peut composer efficacement les fonctions ***cdr*** et ***car*** selon le schéma $cl_1l_2l_3l_4r$ avec 4 lettres au maximum, telles que $l_i = \begin{cases} a \rightarrow car \\ d \rightarrow cdr \end{cases}$. Par exemple ***caar***, ***cdar***, …

Exemple :

```
?-  (cadr '(1 2 3)) = (car (cdr '(1 2 3)))
2
```

Fonctions diverses sur les listes

On propose ci-dessous quelques fonctions évoluées de manipulation des listes.

- ***list?***

  Ce prédicat retourne vrai si son argument est une liste.

  ```
  ?- (list? '(1 2 3))
  #t
  ```

- ***list***

  ```
  - ? (list 1 2 3)
  (1 2 3)
  ```

- ***append***

  ```
  - ? (append '(1 2 3) '(4 5 6))
  (1 2 3 4 5 6)
  ```

- ***reverse***

  ```
  ?- (reverse '(1 2 3))
  (3 2 1)
  ```

- ***list-ref*** *:* retourne le *ième* élément, à compter de zéro.

  ```
  ?- (list-ref '(a b c d) 2)
  c
  ```

- ***list-tail*** *:* retire les *i* premiers éléments de la liste

  ```
  ?- (list-tail '(a b c d e) 3)
  (d e)
  ```

- ***member***

  ```
  ?- (member 'd '(a b c))
  #f
  ?- (member 'b '(a b c))
  (b c)                ; ce qui signifie #t
  ```

Paires pointées et Structures complexes

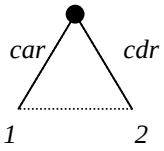
Les paires pointées

On donne un exemple de paire pointée :

```
?- (cons 1 2)
(1 . 2)

?- (pair? '(1 . 2))
#t
```

La différence avec une liste vient du fait que le second argument du *cons*[1] n'est plus nécessairement une liste mais peut être un atome ! La représentation mémoire se dessine alors comme suit.



Ainsi si l'on définit une paire pointée, on peut accéder au membre droit au moyen de *car* et au membre gauche au moyen de *cdr*.

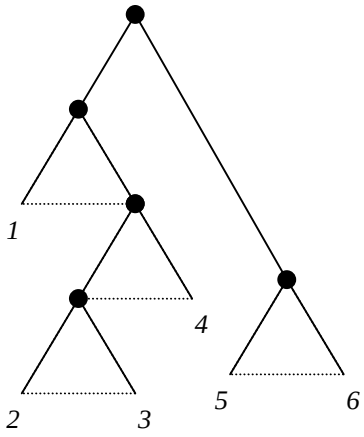
```
?- (define  P '(1 . 2))
?- (car P)
1
?- (cdr P)
2
```

Structures complexes

Les structures complexes sont construites à partir d'imbrications quelconques de *cons*.

```
?- (cons (cons 1 (cons (cons 2 3) 4)) (cons 5 6))
((1 . ((2 . 3) . 4)) . (5 . 6))
```

On donne la représentation mémoire de la structure complexe précédente.



Les listes

Les *listes* sont des structures complexes particulières dont chaque *cons* a un deuxième argument tel que ce soit une nouvelle paire pointée, ou bien la liste vide *'( )*. Le *cons* le plus à droite doit être obligatoirement la liste vide. On distinguera *les listes d'atomes, les listes de listes et les listes quelconques.*

```
?- cons 1 . '( ))
```

[1] On rappelle que le *cons* a deux arguments.

(1 . ( ))

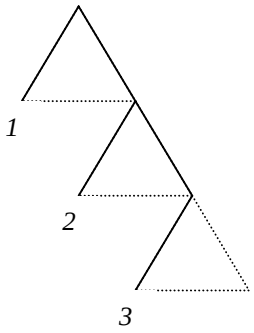
?- (cons 1 (cons 2 (cons 3 '( ))))
(1.(2.(3.( ))))

En réalité, on dispose d'un affichage simplifié pour ce type de structure… ce qui donne respectivement :

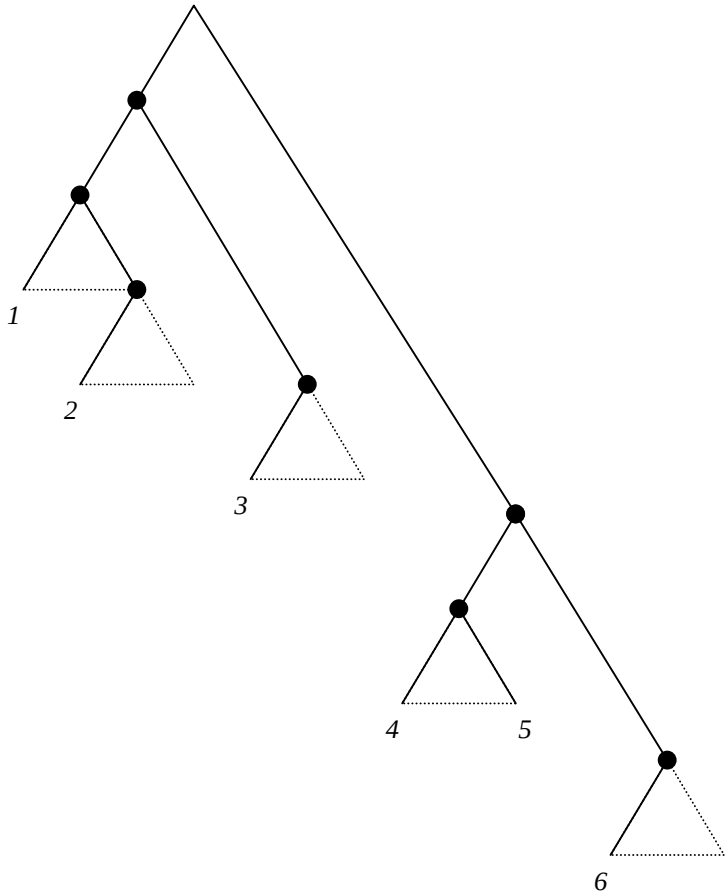
?- '(1. ( ))
(1)

?- '(1.(2.(3.( ))))
(1 2 3)

Considérons l'exemple simple de la liste *(1 2 3)*. La représentation de la liste est la suivante :



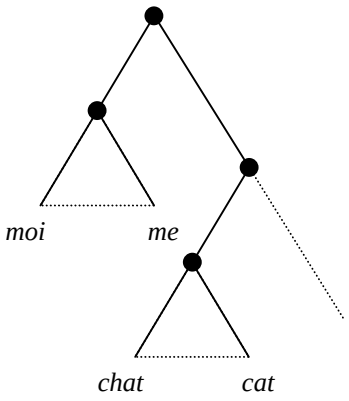
Voyons le cas générique d'une liste quelconque : *(((1 2) 3) (4 . 5) 6)*.



les A – listes

Ce sont des listes de paires pointées.

?- (define dico '((moi . me) (chat . cat) (chien . dog)))



On dispose de la fonction ***assq*** qui renvoie la paire pointée associée à son fils gauche (1er argument) dans une *A – liste* (2ème argument).

?- (assq 'chat dico)
(chat . cat)

Récursivité

Sans affectation, on utilise la récursivité pour programmer des boucles.

Exemple de définition récursive de la factorielle :

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

Au retour de l'appel récursif, il faut multiplier le résultat par *n* ; ce qui implique que *n* doit être stocké dans la pile.

Récursivité terminale

La récursivité terminale est un mécanisme permettant à l'interprète *scheme* d'implémenter certaines boucles récursives comme des boucles itératives, c'est-à-dire sans utiliser de pile d'appel.

Pour ce faire, il faut et il suffit que tous les appels récursifs de la fonction, que l'on cherche à définir, n'entrent dans aucun calcul. Une méthode consiste de créer une fonction récursive auxiliaire ayant en plus comme paramètre le résultat. Il reste, ensuite, à initialiser ce résultat pour obtenir un premier appel correct.

Exemple de définition récursive terminale de la factorielle :

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

Programmation récursive avec les listes plates

Considérons l'exemple simple de calcul de la somme des éléments d'une liste d'atome numérique.

```
?- (define (somme L)
    (if (null? L)          ← test d'arrêt
        0                  ← résultat lors de l'appel terminal
        (+ (car l)         ← définition récursive, où survient l'appel récursif
            (somme (cdr L)))))
```

Fonctionnelles

On traite dans cette partie de *l'utilisation dynamique des fonctions*, par opposition aux définitions statiques obtenues précédemment grâce à la forme *define* ou par des *lambda expressions*.

La *fonctionnelle* est une fonction ayant en argument une fonction ou bien un résultat qui est une fonction.

Fonction en argument

Un exemple simple de fonctionnelle :

```
?- (define (of_zero f) (f 0))
?- (of_zero (lambda (x) (* x x)))
0
```

Calcul général de la somme d'une série utilisant une fonctionnelle ; *terme* et *suivant* sont des fonctions :

```
?- (define (serie a b terme suivant)
(if (> a b)
    0
    (+ (terme a) (serie (suivant a) b terme suivant))))
```

On propose ci-après une autre écriture évitant l'empilement des trois derniers arguments :

```
?- (define (serie a b terme suivant)
     (define (serie_aux a)
          (if (> a b)
          0
          (+ (terme a) (serie_aux (suivant a)))))
     (serie_aux a))
```

Fonction en résultat

On donne l'exemple d'une fonctionnelle construisant la dérivée d'une fonction qu'elle reçoit en paramètre :

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

Composition des fonctions et Curryfication

- **Composition** :

```
?- (define (compose g  f)
     (lambda (x) (g (f x))))
```

- **Curryfication** : on se ramène à des fonctions à un seul paramètre.

```
?- (define (curry f)
     (lambda (x) (lambda (y) (f x y))))
```

Fonctionnelles et Listes

- ***apply***

  Syntaxe : (apply f liste_arg) avec *liste_arg = '(arg1 arg2 …)*

  Lors de l'appel avec *apply*, tous les éléments de la liste sont passés un à un, en argument, à la fonction *f*, ce qui revient à l'application suivante *(f arg1 arg2 …).*

```
?- (apply + '(1 2 3))        ; schéma de calcul (+ 1 2 3)
6

?- (apply cons '(a b))
(a . b)

?- (define (moyenne . L)
     ( / (apply + L) (lenght L)))
```

- ***map***

Syntaxe : (map f liste_1 … liste_n)

La fonctionnelle *map* prend comme premier argument une fonction *f* d'arité quelconque déterminée (ici *n*), qu'elle applique un à un aux éléments des listes passées en argument, pour former la liste résultat. A priori, toutes les listes doivent avoir la même dimension, qui est aussi la dimension de la liste en retour.

```
?- (map + '(1 2) '(3 4) '(5 6)) ; le schéma de calcul est ((+ 1 3 5) (+ 2 4 6))
(9 12)

?- (map 1+ '(1 2 3))        ; cas particulier de l'arité unaire
(2 3 4)
```

Le *map* permet d'effectuer des traitements parallèles.

```
?- (map car '((1 2 3) (4 5 6) (7 8 9)))    ; attention car est  un opérateur unaire sur liste.
(1 4 7)

?- (map cdr '((1 2 3) (4 5 6) (7 8 9)))
((2 3) (5 6) (8 9))
```

Voici un exemple plus consistant :

```
?- (define (moyenne_carrés . L)
     ( / (apply + (map (lambda (x) (* x x)) L))
        (length L)))
```

- ***for-each***

Syntaxe : (for-each f liste)

La syntaxe de *for-each* et ses arguments sont identiques à ceux de *map*, mais *for-each* ne fournit pas de résultats[2] : la fonction est appliquée uniquement pour ses effets de bords éventuels. à la différence de *map*, *for-each* nous assure que la fonction transmise en paramètre sera appliquée séquentiellement sur les éléments successifs des listes.

```
?- (for-each display '( "Bonjour " "maman…"))
Bonjour maman…
```

[2] caractéristique des fonctions à effets de bord, aucun résultat n'est affiché, sauf bien sûr quand l'effet de bord s'applique à l'écran lui-même (ex. : *display*)

Macros

Limitation des fonctions

Les fonctions définies par les *lambda expressions* sont telles que leurs arguments sont systématiquement évalués avant l'évaluation du corps, lors d'une application.

Par conséquent, il va être impossible de définir de nouvelles formes spéciales et donc d'étendre le langage.

Définition et Appel

*Syntaxe : (**define-macro** (nom arg …) corps)*

Lors de l'application d'une *macro*, l'interprèteur commence par construire les liaisons entre les paramètres formels de la macro et les arguments *non évalués* de l'appel. Puis, il procède à l'évaluation du corps de la macro dans cet environnement et il termine en évaluant dans le contexte d'appel le résultat de l'évaluation précédente.

```
?- (define-macro (ifn test alors sinon)
     (list 'if test sinon alors))
```

Pour connaître le résultat de la première évaluation, c'est-à-dire l'expansion syntaxique, on peut utiliser la fonction ***macroexpand*** *ou* ***macroexpand-n*** avec *n* un entier précisant le niveau d'expansion (ce qui s'avère utile dans la cas de macro-définitions récursives) .

```
?- (macroexpand '(ifn #t 0 1))
(if #t 1 0)

?- (ifn #t 0 1)
1
```

On peut aussi avantageusement utiliser *la quasi-citation* dans l'écriture du corps des macros :

```
?- (define-macro (ifn test alors sinon)
     `(if ,test ,sinon ,alors))
```

Utilisation des macros

Les macros se différencient des fonctions sur plusieurs points importants :

- Une fois expansé, l'appel à une macro ne prend pas en compte les modification utérieures de la macro.
- Les macros ne sont pas des fonctions, donc on ne peut pas les envoyer en paramètre à des fonctionnelles (ex. : *or, and*). Une solution consiste à encapsuler la macro dans une fonction.
- Une macro peut modifier ses paramètres. ? ? ?
- Une macro peut implémenter de nouvelles formes spéciales (c'est-à-dire une forme qui n'évalue pas systématiquement tous ses arguments).

De façon général, on préférera écrire des fonctions plutôt que des macros, à chaque fois que cela sera possible.

Capture de noms

L'utilisation de macros peut entraîner divers désagréments. L'un des moindres n'est pas le phénomène de *capture*. Celui-ci intervient lorsque l'expansion de la macro utilise des symboles qui sont redéfinis dans l'environnement dans lequel opère la macro.

```
?- (define-macro (echanger! a b)
     `(let ((tmp ,a))
```

```
        (set!  ,a  ,b)
        (set!  ,b  tmp)))

?- (define tmp 1)
?- (define x 2)
?- (echanger tmp x)          ; capture de tmp par la définition locale : (let ((tmp tmp)) …) ! ! !
```

Une solution en *Scheme* pour contourner ce problème consiste à utiliser la fonction ***gensym*** (fonction non standard) qui génère des noms de symboles à chaque appel.

```
?- (define-macro (echanger! a b)
    (let ((tmp (gensym)))
        `(let ((,tmp ,a))
            (set!  ,a  ,b)
            (set!  ,b  ,tmp))))
```

Modification de variables (effets de bord)

Affectation

Les fonctions suivantes travaillent par effets de bord, et par conséquent n'entrent pas dans le cadre d'une programmation purement fonctionnelle. Les fonctions produisant des effets de bord retourne, le plus souvent, un résultat indéfini, qui ne s'affiche pas.

- ***set!*** *(*lire *"set bang")*

  Syntaxe : (set! symbole expression)
  avec *symbole* un symbole possédant effectivement une adresse, et non pas une constante.

  Description : L'interpréteur commence par évaluer *expression*, puis recherche le symbole dans l'environnement et modifie la liaison en substituant la valeur associée par le résultat de l'évaluation.

  ```
  ?- (define a 1)
  ?- (set! a 0)        ; effet de bord ! ! !
  ?- a
  0
  ```

  Attention au partage d'objet, qui vient du fait que l'on manipule pas directement les valeurs, mais leurs liens (ou adresses).

  ```
  ?- (define a 1)
  ?- (define b 2)
  ?- (set! a b)            ; a reçoit lien vers la valeur de b.
  ?- a
  2
  ?- (set! b 0)            ; on modifie b.
  ?- a                     ; comme le lien est conservé, la modification de b se répercute sur a.
  ?- 2                     ; et NON ! ! ! pourquoi ? ? ? (même problème avec define au lieu de set!)
  ```

- ***set-car!***

  Syntaxe : (set-car! paire_pointée expression)

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

- ***set-cdr!***

  Syntaxe : (set-cdr! paire_pointée expression)

  ```
  ?- (define L '(1 2))
  ?- (set-cdr! L 0)
  ?- L
  (1 . 0)
  ```

  Attention au partage d'objet, qui vient du fait que l'on ne manipule pas directement les valeurs, mais leurs liens (ou adresses).

  ```
  ?- (define L '(1))
  ?- (define K '(2 3))
  ?- (set-cdr! L K)            ; (cdr L) reçoit le lien vers la valeur de K.
  ?- L
  (1 2 3)
  ```

```
?- (set-car! K 0)          ; comme le lien est conservé, la modification de K se répercute sur L.
?- L
?- (1 0 3)
```

L'application de *define* pour redéfinir *K* ne se répercute pas sur *L*, car l'adresse de *K* devrait changer. Ainsi les *set-car!* suivants n'auront aucune répercussion sur *L*.

Fermeture et Modélisation d'états internes

La *fermeture* est une fonction munie d'un environnement local susceptible d'évoluer avec le temps.

On donne l'exemple de la gestion d'un compte bancaire. La fonction dont conserver le solde du compte en interne.

```
?- (define (make-compte montant)
     (lambda (m)
          (set! montant (+ montant m))
          montant))
```

La variable *montant* fait partie de l'environnement (ici, l'environnement d'appel) associé à *la lambda expression*. La fermeture est précisément l'ensemble <*environnement de la fonction, corps de la fonction*>.

```
?- (define paul (make-compte 1000))        ; paul = fermeture
?- (paul 100)
1100
?- (paul -500)
600
```

Pour définir l'environnement nécessaire, on peut simplement utiliser un *let*.

Fonctions diverses

display, read, error

```
?- (display "hello world !")
hello world !

?- (error "division par zéro interdite !")
division par zéro interdite !

?- (read)
```

trace, untrace

Tracer les appels d'une fonction.

```
?- (trace fonction)

?- (untrace fonction)
```

begin

La forme *begin* évalue en séquence ses paramètres ; le résultat est celui de la dernière

```
?- (begin 1 2 3)
3
```

eval

En *lisp*, *eval* peut être utilisé par le programmeur.

```
?- (eval '(+ 2 3))
5
```

Mais en *scheme*, le choix standard est de cacher cette fonction au programmeur. Certains interprètes la fournissent tout de même, mais elle n'est pas standard.
Cette fonction a en réalité deux arguments : l'expression à évaluer et l'environnement dans lequel évaluer cette expression. Lorsque *eval* est employé avec un seul argument, l'environnement est pris global par défaut.

Index