TP OO
MACHINE LISP

I. Introduction

LISP est un langage qui a été créé dans les années 1960 par Mc Carthy. Initialement, il a été élaboré pour faire du traitement symbolique de listes (arbres). La description qui en est faite ici correspond à l'implémentation dite LeLisp proposé par J.Chailloux. Les caractéristiques principales du langage Lisp sont :

- ***C'est un langage fonctionnel** : le seul mécanisme d'exécution est l'évaluation fonctionnelle. Son fonctionnement est donc simple à décrire.*

- ***C'est un langage permettant de traiter des objets symboliques** : Il est uniquement structurel. La sémantique n'est pas contenue dans les constituants du langage.*

- ***Le langage Lisp est utilisable au travers d'un interprète** : C'est donc un système très interactif.*

- ***Programmes et données ont le même support syntaxique** : Cela permet de traiter des programmes par programme, On dit que le langage est homoiconique (littéralement : qui se décrit avec sa propre représentation).*

- ***C'est un système extensible** : Du fait des caractéristiques précédentes, augmenter les fonctionnalités du système se fait en ajoutant des fonctions.*

II. Bibliographie

John Mc Carthy http://en.wikipedia.org/wiki/John_McCarthy_%28computer_scientist%29

PH. Winston, BKP. Horn Lisp Addison Wesley, 1978

H. Wertz Lisp : Une introduction à la programmation - Masson 1985

L'historique de Lisp : http://www.softwarepreservation.org/projects/LISP/

III. Liens avec d'autres langages

Le langage Lisp est un langage fonctionnel qui s'apparente à beaucoup d'autres langages. C'est un langage plus simple que Caml car il n'est pas typé et il n'intègre pas l'évaluation partielle c.-à-d. qu'il ne permet pas d'avoir l'appel incomplet qui retourne une fonction où les variables connues ont été liées. C'est un langage qui se différencie de Caml ou de Scheme, … par le fait qu'il est « à portée dynamique » alors que les autres sont dits « à portée lexicographique » c.-à-d. qu'une variable n'est utilisable (accessible) que dans l'espace (block) syntaxique où elle est déclarée, en Lisp toute variable définie dans une fonction est accessible par les codes des fonctions appelées depuis celle-ci. La portée est donc l'arbre des appels et donc totalement dépendant de l'exécution.

IV. Organisation de la présentation

Dans la suite de ce document, on vous présente tout d'abord la structure de représentation de toute entité (objet, terme) Lisp, ceci constitue le chapitre « Objets LISP et fonctions de base associées ». Il est suivi par une présentation de l'interprétation de ces termes comme des évaluations fonctionnelles, ceci constitue le chapitre « Les programmes LISP ». Il est préférable de bien avoir compris le premier chapitre pour pouvoir aborder le second.

1 Objets LISP et fonctions de base associées

Dans ce paragraphe on étudie les objets de bases du langage Lisp et les fonctions intrinsèques qui y sont associées. On se place délibérément dans un système sans variable, où les symboles représentent des constantes. On introduira par la suite la notion de variable lorsqu'on abordera la notion de fonction.

1.1 Les objets LISP

On appelle globalement un objet LISP une S-EXPR. Une S-EXPR est soit un constituant atomique (ATOME) soit un objet composé (LISTE). Les LISTEs seront représentées en mémoire par des structures de données que l'on appelle des S-GRAPHEs fait de Scons.

1.1.1 Les ATOMEs

Un atome est un symbole dénoté par une chaîne de caractères. Seuls les caractères ', (,) et blanc ne sont pas autorisés dans les symboles. Exemples : toto, toto.p, 1homme, titi-ti et 333 sont des symboles valides.

1.1.2 Les LISTEs

Une liste est constituée à partir de listes ou d'atomes séparés par un ou plusieurs blancs. La liste ne comportant aucun élément est la liste vide, elle est notée de façon caractéristique. La forme BNF de la structure syntaxique des objets LISP est donc la suivante :

S-EXPR ::= *ATOME*
()
(*S-EXPR*$^{+}$**)**
ATOME ::= *SYMBOLE*

La liste vide est dénotée par ***()****. Le symbole* ***nil*** *(cf notion de variable) est associé à la liste vide et pourra être utilisé en lieu et place de* ***()****. SYMBOLE représente tout élément terminal qui est un symbole.* ***L'objet () a un statut particulier car il peut être considéré comme atome ou comme liste.*** *Cette interprétation est faite par les fonctions de base du système. Exemples d'objets Lisp :*

(toto toto.p)
((toto toto.p) toto)
(() ()) *correspond à la liste comportant deux listes vides.*

1.1.3 Les fonctions de base du langage Lisp

Il y a 5 primitives de base dans le langage Lisp {Car, Cdr, Cons, Atom, Eq}, les 3 premières permettent de manipuler les objets Lisp, ce sont le constructeur et les sélecteurs, et les 2 autres sont des prédicats.

1.1.3.1 Les primitives de manipulation des objets

Cons est la fonction qui permet de construire des listes, Car et Cdr sont les sélecteurs correspondants. Cons est une fonction totale alors que Car et Cdr sont des fonctions partielles.

1.1.3.1.1 La fonction CAR

CAR : S-EXPR -> S-EXPR

Elle délivre le premier élément de la liste passée en argument, exemples :

CAR	*appliqué à*	*(a b c)*	*délivre*	*le symbole a*
		((a b) c)		*la liste (a b)*
		()		*la liste () par convention*

a	*produit une erreur*
(() a b c)	*()*

1.1.3.1.2 La fonction CDR

CDR : S-EXPR -> S-EXPR

Elle délivre la valeur de la liste, passée en paramètre, privée de son premier élément. On verra un cas particulier dans le paragraphe La , exemples :

CDR	*appliqué à*	*(a b c)*	*délivre*	*la liste (b c)*
		(a (b c))		*la liste ((b c))*
		()		*la liste () par convention*
		a		*produit une erreur*
		(a ())		*la liste (())*
		((a b) c)		*(c)*

1.1.3.1.3 La fonction CONS

CONS : S-EXPR × S-EXPR -> S-EXPR

Cons est le constructeur de liste. Suivant la nature (ATOME ou LISTE) de son deuxième argument, la fonction Cons construit des listes différentes. On peut caractériser les fonctions Car, Cdr et Cons par l'axiomatisation suivante :

Car(Cons(S1,S2)) ≡ S1

Cdr(Cons(S1,S2)) ≡ S2

- CONS : S-EXPR × LISTE -> S-EXPR

- CONS : S-EXPR × ATOME -> S-EXPR

Lorsque le second argument est une liste, on remarque que Cdr(Cons(S1,S2)) renvoie une liste ce qui est conforme à la définition que l'on a fait de Cdr. Lorsque le second argument est un atome Cdr(Cons(S1,S2)) renvoie un atome. Dans ce cas la fonction Cons a construit ce que l'on appelle une paire pointée. Nous verrons lorsque nous étudierons la représentation en mémoire à quoi correspond une paire pointée, exemples :

CONS	*appliqué à*	*a*	*et*	*(b c)*	*délivre*	*(a b c)*
		a		*()*		*(a)*
		()		*(b c)*		*(() b c)*
		()		*()*		*(())*
		(a b)		*(c d)*		*((a b) c d)*
		(a b)		*()*		*((a b))*
		a		*b*		*(a . b)*
		(a b)		*c*		*((a b) . c)*
		()		*c*		*(() . c)*

Les paires pointées sont dénotées par un triplet CAR POINT et CDR (a . b). Le point est un élément syntaxique et doit donc être entouré de séparateurs. On remarque que la liste (a . (b c)) est équivalente à (a b c), le point est la représentation de l'opérateur Cons. Ceci prendra toute son importance lorsque l'on étudiera le problème de la liaison entre paramètres formels et paramètres effectifs (filtrage).

1.1.3.2 Les Prédicats de base du langage Lisp

Lisp est un langage non typé ce qui lui procure des avantages, mais peut être à la base de beaucoup d'erreurs. La notion de type n'apparaît donc qu'au travers de l'interprétation que l'on fait des objets Lisp manipulés. Aucune vérification n'est faite par LISP,

hormis le nombre d'arguments. Ce sont donc les fonctions qui doivent vérifier la conformité de leurs arguments. Si elles ne le font pas, on ne peut pas présager de l'exécution qui en découlera.

*Cependant LISP a besoin de manipuler les valeurs booléennes (vrai, faux), elles seront représentées par des valeurs de S-EXPR. Par convention **FAUX** est représenté par **la liste vide** et **VRAI** est représenté par toute autre valeur de S-EXPR différente de **()**. Le symbole **t** sera une notation conventionnelle pour VRAI (voir avec la notion de variable).*

1.1.3.2.1 Le prédicat ATOM

ATOM : S-EXPR -> S-EXPR

*Le prédicat Atom renvoie **t** si le paramètre est un atome et **()** dans le cas contraire. Ce prédicat considère **()** comme un atome, exemples :*

ATOM	*appliqué à*	*a*	*délivre*	*t*
		(a b c)		*()*
		()		*t*

1.1.3.2.2 Le prédicat EQ

EQ : S-EXPR × S-EXPR -> S-EXPR

*Le prédicat Eq renvoie **t** si les deux arguments sont **identiques**, **()** dans le cas contraire. Suivant la nature des arguments Eq teste une condition particulière.*

- EQ : ATOME × ATOME -> S-EXPR

Lorsqu'il s'agit de deux atomes Eq vérifie l'identité de ceux-ci, ils correspondent au même symbole.

- EQ : LISTE × ATOME -> S-EXPR ou ATOME × LISTE -> S-EXPR

*Dans ce cas Eq renvoie systématiquement **()** : impossibilité d'égalité entre un atome et une liste.*

- EQ : LISTE × LISTE -> S-EXPR

*Dans ce cas Eq renvoie **t** si les deux arguments désignent le même S-GRAPHE (identité) et **()** dans le cas contraire. Ce cas d'application de Eq est lié à la notion de variable Lisp que nous verrons par la suite.exemples :*

EQ	*appliqué à*	*a*	*et*	*a*	*délivre*	*t*
		a		*b*		*()*
		()		*()*		*t*
		(a b)		*(a b)*		*()*

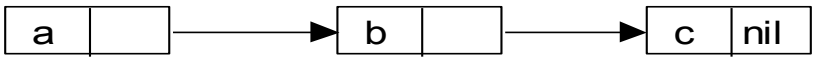
Dans le dernier exemple il s'agit de deux valeurs de listes qui sont équivalentes, mais pas identiques. C'est pourquoi la fonction Eq renvoie nil (les s-graphes sont différents : voir le paragraphe suivant).

1.2 Représentation des objets en mémoire

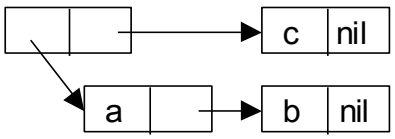
Il y a essentiellement deux structures de données dans un système LISP, une pour stocker les LISTEs et l'autre pour stocker les ATOMEs. On appelle la représentation d'une LISTE un S-GRAPHE. Chaque élément de liste est représenté par un doublet (S-CONS) dont le premier composant correspond au CAR et le second au CDR. L'autre structure de données importante dans un système LISP est la mémorisation des symboles.Un symbole est caractérisé par une chaîne de caractères. Des structures annexes peuvent apparaître pour des raisons d'optimisation ou d'extension du modèle.

structure d'un doublet de S-GRAPHE : 

la S-EXPR (a b c) est représentée par le S-GRAPHE :



la S-EXPR ((a b) c) est représentée par le S-GRAPHE :

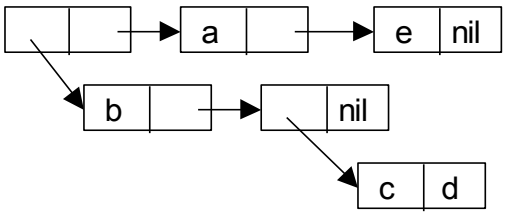


Une paire pointée est une structure particulière de S-GRAPHE. Elle a l'avantage de minimiser la place mémoire occupée. Elle correspond aussi à la tabulation d'une fonction, et plus généralement à toute association.

la S-EXPR (a . b) est représentée par le S-GRAPHE : 

Le S-GRAPHE ci-contre correspond aux S-EXPR ci-dessous :

((b (c . d)) a e)
((b (c . d)) a . (e))
((b (c . d)) a . (e . nil))
((b (c . d)) . (a . (e . nil)))
((b (c . d)) . (a . (e . nil)))
((b (c . d)) . (a . (e . nil)))
((b . ((c . d))) . (a . (e . nil)))
((b . ((c . d) . nil)) . (a . (e . nil)))

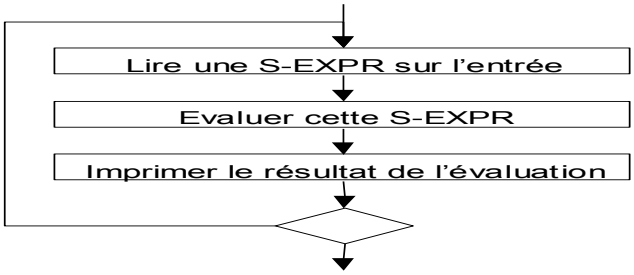


2 Les programmes LISP

Les programmes LISP s'articulent autour de la structure que nous venons de décrire et des primitives associées, ainsi que de quelques nouvelles fonctions liées au problème de l'évaluation fonctionnelle et du contrôle {Quote, Cond, Lambda et Flambda, Eval et Apply, De et Df}. Seule Cond est une vraie primitive, toutes les autres peuvent être décrites en Lisp lui-même. Cependant pour diverses raisons certaines sont souvent proposées sous la forme de primitives.

2.1 Principe de fonctionnement d'un interprète LISP

Le système LISP est un interpréteur qui reçoit ses ordres de l'utilisateur. La seule opération qu'il réalise est l'évaluation de la S-EXPR qu'on lui fournit. Le principe de fonctionnement de LISP peut donc être schématisé ainsi :



*On remarque que les notions de programmes et de données ne se distinguent que par le contexte où se situe la S-EXPR correspondante (**une S-EXPR n'est un programme que si on cherche à l'évaluer**).*

*Lorsqu'on appelle le système LISP, on démarre une session qui se terminera par un ordre d'arrêt de la session : fonction **quit**.*

2.2 Structure fonctionnelle (FORME fonctionnelle)

Comme nous venons de le dire toute S-EXPR est une forme fonctionnelle qui peut être évaluée. Nous sommes amenés à considérer le problème de l'évaluation d'un atome ou d'une liste (A rapprocher du λ–calcul).

2.2.1 Structure fonctionnelle atomique

A la notion de fonction, on attache la notion de paramètres. Un paramètre est un objet formel qui est lié à une valeur lors de l'appel de la fonction qui le définit. Cette notion est à rapprocher de celle de variable. En Lisp une variable est nommée à l'aide d'un symbole. Ceci pose un problème car un symbole a 2 significations, l'une correspondant à la première interprétation que l'on en a faite (constante symbolique) et la seconde correspondant à la notion de nom de variable. Afin de différencier les interprétations possibles on utilise les règles suivantes :

Lorsqu'il s'agit d'un symbole la valeur délivrée est celle qui a été associée à la "variable" ayant comme nom ce symbole.

- ***une variable est une forme fonctionnelle dont la valeur est celle de la variable***

Pour dénoter une constante symbolique, on fait précéder la suite de caractères définissant ce symbole du caractère quote '. On verra par la suite que l'on peut procéder autrement pour exprimer cette notion de constante symbolique. L'avantage de cette solution ultérieure est qu'elle fournit un sens fonctionnel au symbole quote, et permet de mieux généraliser ce mécanisme de constante.

- ***une constante est une forme fonctionnelle dont la valeur est le nom de la constante.***

Exemples : supposons que les variables a, b et c existent avec les associations suivantes :

***a** est associée à la valeur (a b c)*
***b** est associée à la valeur salut*
***c** est associée à la valeur a*

l'évaluation de	*a*	*donne comme résultat*	*(a b c)*
	'a		*a*
	b		*salut*
	c		*a*

2.2.2 Structure fonctionnelle complexe

*Une FORME, en LISP, est un appel de fonction. Dans le cas d'une structure complexe, elle contient donc la fonction à évaluer et les arguments sur lesquels porte l'évaluation (**voir lambda-calcul**). La structure syntaxique d'une forme est basée sur celle des objets Lisp puisque programmes et données sont dénotés avec la même structure de base. Nous donnons ici la grammaire "abstraite" d'une FORME. Une FORME est une S-EXPR que l'on cherche à évaluer*

FORME ::=	*ATOME*	*{variable}*
	'S-EXPR	*{constante}*
	(FONCT FORME)*	*{appel fonctionnel}*
FONCT ::=	*SYMBOLE*	*{nom de fonction}*
	(lambda PARAM FORME)*	*{fonction anonyme}*
PARAM ::=	$S\text{-}EXPR_{SYMBOLE}$	*{liste ou arbre de variables, les constantes sont interdites}*

Le fait que programmes et données aient même structure syntaxique pose un problème identique à celui des variables et constantes symboliques. Pour résoudre ce conflit nous allons utiliser la même technique consistant à faire précéder la constante par un quote '. Dans le cas d'une fonction anonyme (lambda-terme) PARAM est l'ensemble des noms des paramètres formels de la fonction, FORME est le corps de la fonction. Exemples de formes complexes :

```
'(a b c)
'a
(cons 'a (cons 'b '(c nil)))
((lambda (x) (car x)) '(a b c))
(cond ((eq 'a 'a) 'a) (t    (car a))
```

2.3 Mécanismes d'évaluation fonctionnelle

*Il existe plusieurs types d'évaluations fonctionnelles. Nous en décrivons tout d'abord deux principales. Le principe de la première forme d'évaluation correspond aux fonctions que l'on qualifie de **EXPR**, la seconde aux fonctions qualifiées de **FEXPR**.*

Lorsque LISP doit faire une évaluation, soit c'est un atome et c'est la valeur de la variable correspondante qui est renvoyée, soit c'est une liste et LISP considère qu'il a à faire à un appel fonctionnel, il traite celui-ci en fonction de la nature de la fonction qui doit être évaluée.

Un appel de fonction a la structure suivante :

(Foncteur Arg1 Arg2 …) où *Foncteur caractérise la fonction et Arg1,Arg2,… ses arguments*

2.3.1 Mécanisme d'évaluation des fonctions de type EXPRs

I) Lisp évalue Foncteur et obtient FCT qui est une valeur de fonction. Si Foncteur est un symbole, c'est une variable qui doit être associé à la définition d'une fonction (base ou utilisateur). Si c'est une lambda, il s'agit d'une définition anonyme fonction.

II) Lisp évalue la liste des arguments Arg1 Arg2 … et obtient une liste d'arguments évalués que l'on nomme Arg où le premier élément est la valeur de l'évaluation de Arg1 etc…

III) Lisp applique la fonction FCT à la liste Arg. Dans le cas d'une fonction de base l'application fonctionnelle correspond à la sémantique donnée dans le chapitre précédent. Dans le cas d'une fonction utilisateur (voir définition de fonctions) ou anonyme (lambda), l'application fonctionnelle consiste à lier les paramètres formels et les paramètres effectifs, puis à évaluer le corps de la fonction dans ce nouveau contexte (voir Notion de contexte).

***Remarque** :*

Ce mécanisme est valable pour les fonctions Car, Cdr, Cons, Atom et Eq. On peut rapprocher ce mécansime de celui d'appel de procédure dans les langages procéduraux avec passage de paramètres par valeur.

2.3.2 Mécanisme d'évaluation des fonctions de type FEXPRs

*Si on ne dispose que du mécanisme précédent, la fonction **cond** ne peut être réalisée. En effet le propre de la fonction cond est l'évaluation conditionnelle. Il ne faut pas que ses arguments soient évalués avant son exécution. Le mécanisme des FEXPRs va nous permettre de modéliser le comportement de telles fonctions. Il va nous permettre aussi d'intégrer le quote comme un mécanisme fonctionnel à part entière.*

Le principe de l'évaluation fonctionnelle des FEXPRs diffère de celui des EXPRs au niveau de la phase II. Dans le cas d'une fonction de type FEXPR les arguments de celle-ci ne sont pas évalués. La fonction est appliquée à la liste des arguments eux-mêmes. Ce type de fonction à l'intérêt de permettre de manipuler l'argument lui-même. On définit une FEXPR à l'aide de la fonction Flambda.

***Remarque** :*

1. *La fonction Quote rend, tel que, son argument.*
2. *La fonction Cond recherche la première alternative qui a une condition vrai. La valeur de Cond est alors celle de la FORME associée à l'alternative sélectionnée. Si aucune forme n'est sélectionnée la fonction Cond renvoie nil.*
3. *Le mécanisme peut s'apparenter à celui d'appel de procédure avec passage de paramètres par nécessité (lazy).*

exemples : On donne une forme et le résultat de son évaluation par l'interprèteur LISP.

3	*3*
(quote a)	*a*
'a	*a {racourci syntaxique de la forme précédente}*
a	*variable indéfinie*
‘3	*3*
(car (quote (a b c)))	*a*
(cdr '(a b c))	*(b c)*
(car (car '(a b c)))	*erreur : (car '(a b c)) rend **a**, l'application car à **a** émet l'erreur **a** n'est pas une liste*
(car '(car '(a b c)))	*car*
(car ''a)	*quote*
(cons 'a 'b)	*(a . b)*
(cons 'a '(a b c))	*(a a b c)*
(car (a b c))	*erreur la fonction a est indéfinie*
(cdr '(a b))	*(b)*
(cdr '(a . b))	*b*
(cons (cons 'a 'b) 'c)	*((a . b) . c)*

exemple d'évaluation fonctionnelle de la FORME : (cons (cons (cons 'a 'b) 'c) 'd)
La flèche -> indique la FORME à évaluer, la flèche <- indique le résultat de l'évaluation correspondante, le = correspond au résultat final.

```
->(cons (cons (cons 'a 'b) 'c) 'd)
        -> (cons (cons 'a 'b) 'c)
                -> (cons 'a 'b)
                        -> 'a
                        <- a
                        -> 'b
                        <- b
                <- (a . b)
                -> 'c
                <- c
        <- ((a . b) . c)
        -> 'd
        <- d
<- (((a . b) . c) . d)
= (((a . b) . c) . d)
```

La fonction "cond" est un "if" à choix multiple. Il a comme paramètre une liste de paires contenant chacune la condition et l'action associée. Sa structure générale est comme ceci : (Cond (<c1> <a1>)(<c2> <a2>)….(<cn> <an>)) où <ci> est une forme interprétée comme la condition et <ai> est un programme à évaluer si la condition est vraie. Si aucune des conditions n'est vraie la fonction cond renvoie nil. Exemples:

```
(cond   ((atom a) (cons a b))
        ((atom b) (cons a (cons b ())))
        ((eq a b) b))
```

cet exemple appliqué avec a et b valant	*'x*	*et*	*'y*	*renvoie*	*(x . y)*
	'(1 2)		*'y*		*((1 2) y)*
	'(1 2)		*'(3 4)*		*()*

2.4 Notion de contexte et liaison des paramètres

2.4.1 Notion de contexte

Comme dans les langages procéduraux, la notion de contexte existe et définit l'ensemble des associations symbole-valeur (variable) accessibles à un instant donné. Les variables LISP sont les paramètres formels des fonctions, elles sont liées aux valeurs des paramètres effectifs à l'appel de la fonction.

*La **durée de vie** d'une "variable" en Lisp est équivalente à la période d'activation de la fonction qui l'a définie. La période d'activation d'une fonction débute à l'appel de cette fonction et se termine lorsque l'on rend la valeur associée.*

*La **portée** d'une "variable" Lisp d'une fonction F1 est dynamique, ce qui veut dire que cette variable est accessible par n'importe qu'elle fonction F2 dont l'activation provient de l'appel de F1. Si à un instant donné deux fonctions F1 et F2 actives (F1 activée avant F2) possèdent une variable de même nom, seule celle de F2 est accessible. L'autre ne redeviendra accessible que lorsque F2 sera terminée. Exemples :*

((lambda (x y) (car (cons y (cdr x)))) '(a b) '(c d))

dans le contexte on ajoute les liaisons

x <-> (a b)
y <-> (c d)

on évalue alors dans ce contexte la forme suivante :

(car (cons y (cdr x))

Lorsque l'évaluation est terminée, les liaisons sont défaites et x et y ne correspondent plus à ces variables.

2.4.2 Les liaisons

Les liaisons dans LISP sont donc dynamiques. A la rencontre d'une lambda on ajoute dans le contexte les liaisons paramètres formels paramètres effectifs qu'elle définit, lorsque l'évaluation de la lambda se termine les liaisons sont défaites et le contexte initial est restitué. L'application fonctionnelle d'une Lambda se décompose en trois phases :

1. *liaison des arguments aux paramètres (les anciennes liaisons ne doivent pas être perdues).*
2. *évaluation du corps de la fonction.*
3. *destruction des liaisons effectuées en 1 et restitutions des liaisons précédentes.*

La liaison s'effectue entre la liste des arguments et la S-EXPR qui définit les paramètres de la fonction. La liaison dépend de la structure de la S-EXPR définissant les paramètres. Nous présentons ici quelques cas et les liaisons qui en découlent ainsi que leurs intérêts. Il s'agit d'une semi-unifcation (filtrage) entre un arbre de variables et un arbre de S-EXPRs.

1. *liste de variables : la liaison s'effectue alors en associant à chaque variable de la liste des paramètres la valeur de même rang dans la liste des arguments évalués.*

   ((lambda (a b c) (…)) <P1> <P2> <P3>)

2. *une variable : la liaison s'effectue entre cette variable et la liste des arguments évalués. Cette solution permet à une fonction d'avoir un nombre quelconque d'arguments. C'est elle qui gère cette liste comme elle veut. Par exemple on veut écrire une lambda qui construit la liste de ses arguments.*

   ((lambda a a) 'a 'b 'c) renvoie la liste (a b c)

   ((lambda a (car a)) 'x 'y 'z) renvoie x

3. *le cas général d'un arbre de variables : la liaison s'effectue en mettant en correspondance les deux structures arborescentes. Par exemple on a une structure d'arbre représentant une personne avec son nom, son prénom et sa date de naissance qui est composée de trois nombres, on peut définir une lambda comme suit :*

   ((lambda (nom prenom (jour mois . année)) …) 'MOI 'JE (cons 1 (cons 1 1))) ou

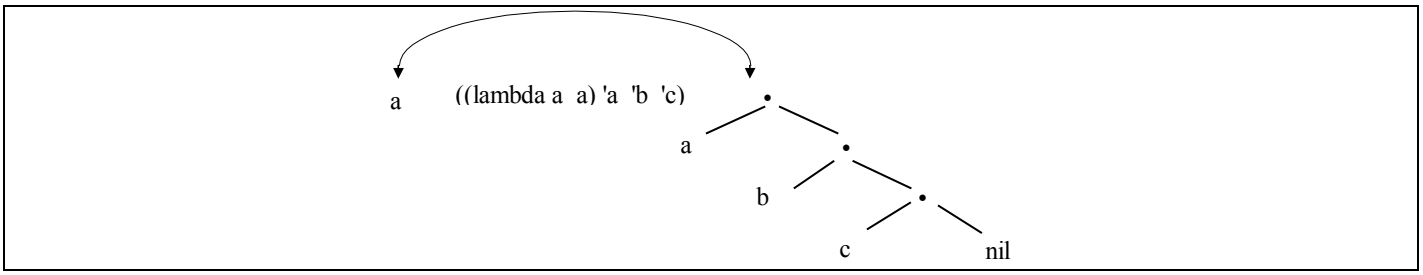
   ((lambda (nom prenom . (date))) (…) 'LUI 'IL '(0 0 . 0))

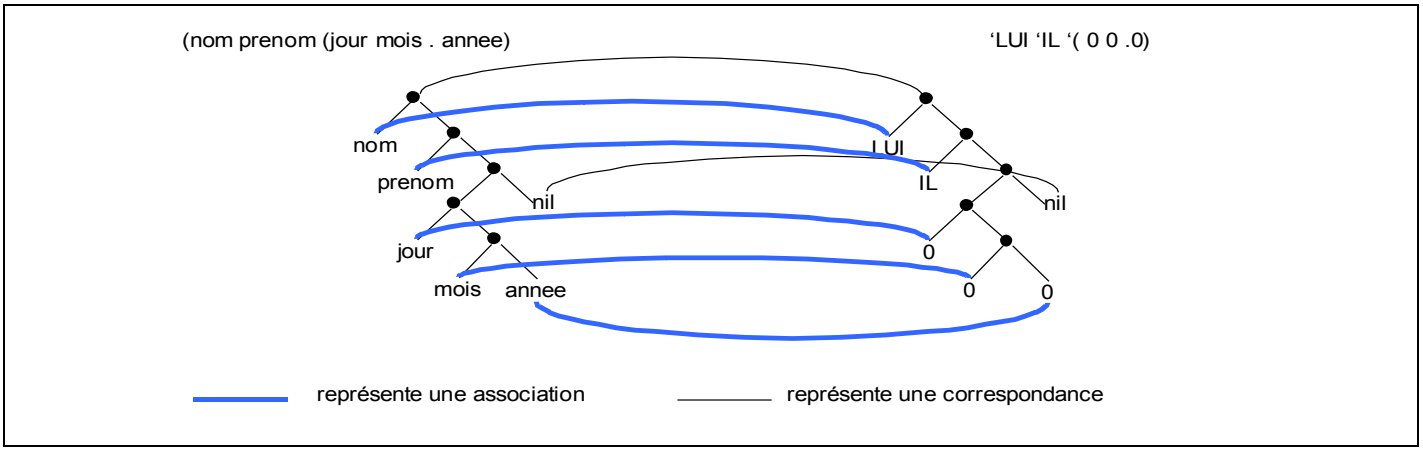
Cette possibilité est très utile pour noter une structuration et éviter la manipulation d'une expression de Car et de Cdr peu explicite. Elle est une façon simple de "typer" (au sens structurel) les arguments d'une fonction en explicitant la structure de ceux-ci.

En effet dans le cas de la définition ((lambda P (…)) 'MOI 'JE '(1 1 . 1))) le champ mois est le (car (cdr (car (cdr (cdr P)))) où P représente la structure personne.

La liaison est illégale si la correspondance ne peut pas se faire complètement. Ceci se produit lorsque l'on a une liste au niveau paramètre et un atome au niveau argument ou dans le cas inverse, et conduit à l'erreur de liaison.

Pour schématiser le principe de liaison, on peut utiliser la représentation arborescente comme ci-dessous :





Exemple : pour décrire le processus d'évaluation d'une FORME dans un CONTEXTE on utilise la notation suivante : <FORME ‖ CONTEXTE> où contexte est représenté par la liste de couples symbole-valeur. On considère, dans l'exemple, que l'appel initial se fait dans un contexte vide d'association.

```
-> <((lambda (x y) (cons ((lambda (y z) (cons x y)) x (car y)) y)) '(a b) '(c d)) ‖ ()>
        -> < (cons ((lambda (y z) (cons x y)) x (car y)) y) ‖ ((x . (a b)) (y . (c d)))>
                -> < ((lambda (y z) (cons x y)) x (car y)) ‖ ((x . (a b)) (y . (c d)))>
                        -> < (cons x y)    ‖ ((y . (a b)) (z . c)) (x . (a b)) (y . (c d)))>
                                -><x ‖ ((y . (a b)) (z . c)) (x . (a b)) (y . (c d)))>
                                <- (a b)
                                -><y ‖ ((y . (a b)) (z . c)) (x . (a b)) (y . (c d)))>
                                <- (a b)
                        <- ((a b) a b)
                <- ((a b) a b)
                -> <y ‖ ((x . (a b)) (y . (c d)))>
                <- (c d)
        <- (((a b) a b) c d)
<- (((a b) a b) c d)
=(((a b) a b) c d)
```

2.5 Les fonctions de l'évaluation (EVAL, APPLY)

Eval est une fonction qui ayant une FORME comme argument, rend la S-EXPR correspondant à l'évaluation de cette FORME. La fonction Apply correspond à la phase III de l'évaluation fonctionnelle, elle réalise l'application d'une fonction à une liste

d'arguments. Elle a comme argument la fonction et la liste d'arguments, elle rend comme valeur celle de l'application fonctionnelle réalisée.

Quand on introduit une FORME au terminal, le système l'interprète comme : évaluer la FORME. Exemple :

((lambda (x) (eval x)) '(cdr '(a b c)))	*retourne la liste (b c)*
((lambda (a) (eval a)) '(cdr a))	*retourne la liste (a).*
((lambda (a) (eval a)) '(eval a))	*exemple à méditer*
(eval '(cdr '(a b c)))	*retourne la liste (b c)*

Toutes les formes ci-dessous sont équivalentes

(cdr '(a b c))	*==*	*expression LISP*
((lambda (a) a) (cdr '(a b c)))	*==*	*la lambda se réduit à l'identité*
((lambda (a) (eval a)) '(cdr '(a b c)))	*==*	*Quote et eval s'annulent et donnent la forme ci-dessous.*
((lambda (a) (cdr a)) '(a b c))	*==*	*c'est la formulation dans le l-calcul*
(apply cdr '((a b c)))		*retourne la liste (b c)*
(eval 'X)		*où X est une des formes ci-dessus*

2.6 Définition de fonctions

*Le mécanisme de **Lambda** et de **Flambda** est le mécanisme de base pour la construction de fonctions de type EXPR et FEXPR. Son inconvénient est qu'il ne permet pas de mémoriser ces définitions. Pour mémoriser une définition de fonction on introduit dans Lisp deux nouvelles fonctions (**de** et **df**). Elles permettent d'associer, dans le contexte global, un symbole à une S-EXPR correspondant à la valeur d'une fonction de type EXPR ou FEXPR. Dès qu'une fonction est définie elle est utilisable.*

2.6.1 La fonction De

(de SYMBOLE PARAM FORME)

SYMBOLE est le nom de la fonction, PARAM décrit l'ensemble des paramètres formels de la fonction comme dans la description d'une Lambda ou d'une Flambda. La fonction De permet de définir de nouvelles fonctions dont l'évaluation fonctionnelle correspond à celle d'une Lambda. Exemples :

```
(de SINGLETON (L)
    (cond  ((atom L) t)
           (((eq nil nil (cdr L)) t)
           (t ())))
```

2.6.2 La fonction Df

La syntaxe est semblable à celle de la fonction De. La fonction Df permet d'introduire des fonctions dont l'évaluation fonctionnelle correspond à celle d'une Flambda.

(df SYMBOLE PARAM FORME)

exemples :

```
(df quote (l)      l)
        {La fonction Quote renvoie son    propre argument puisque celui-ci n'est pas évalué}
(df lambda l (cons 'lambda l))
(df flambda l (cons 'flambda l))
        {les fonctions lambda et flambda correspondent à l'identité, car elles sont leur propre valeur}
```

Fiche D'exercices

Exercice n°1.1

Ecrire les séquences de CAR et CDR qui permettent d'obtenir les symboles PECHE, ORANGE à partir des expressions suivantes :

```
(POMME ORANGE PECHE RAISIN)
((POMME ORANGE) (PECHE RAISIN))
((((POMME))) ((ORANGE)) (PECHE) RAISIN)
((((POMME) ORANGE) PECHE) RAISIN)
(((((RAISIN . (((POMME . PECHE)))) . (((ORANGE)))))))
```

Exercice n°1.2

Donner le S-GRAPHE de la S-EXPR suivante :

```
(L (A N ()) ((GA () (GE . LE) (L I . ())) S . P))
```

Exercice n°1.3

Ecrire la fonction SONC : LISTE x S-EXPR qui construit la liste faite de la liste 1er argument à laquelle on a ajouté en queue le second argument.

Exemples :	*(SONC '(R I C) 'M)*	*=> (R I C M)*
	(SONC '(R I C M) ())	*=> (R I C M ())*
	(SONC () 'RICM)	*=> (RICM)*
	(SONC () ())	*=> (())*

Ecrive la fonction REVERSE : LISTE qui rend la liste en sens inverse des éléments qui y sont contenus.

Exemple : (REVERSE '((1 (2) (3)) 4 (5 6) 7 8 (9))) => ((9) 8 7 (5 6) 4 (1 (2) (3)))

Exercice n°1.4

Ecrire la fonction AND : SEXPR x S-EXPR qui restitue la conjonction des 2 arguments

Exemples :	*(AND '(R I C) 'M)*	*=> T*
	(AND '(R I C M) ())	*=> ()*
	(AND () 'RICM)	*=> ()*
	(AND () ())	*=> (())*

Ecrire la fonction NOT : SEXPR qui restitue la négation de l'argument

Exemples :	*(NOT '(R I C))*	*=> ()*
	(NOTR ())	*=> T*

Exercice n°1.5

Ecrire la fonction CONCAT : S-EXPR x S-EXPR qui construit la liste concaténation des 2 arguments.

Exemple :	*(CONCAT '(R I) '(C M))*	*=> (R I C M)*
	(CONCAT 'R '(I C M))	*=> (R I C M)*
	(CONCAT '(R I C) 'M)	*=> (R I C M)*
	(CONCAT '(R I C M) ())	*=> (R I C M)*
	(CONCAT () '(R I C M))	*=> (R I C M)*
	(CONCAT '(R . I) '(C . M))	*=> (R I C . M)*
	(CONCAT '(R . I) '(C M))	*=> (R I C M)*
	(CONCAT '(R I) '(C . M))	*=> (R I C . M)*
	(CONCAT 'R I 'CM)	*=> (RI CM)*
	(CONCAT () ())	*=> ()*

Exercice n°1.6

Ecrire le prédicat EQUAL S-EPXRxS-EXPR qui indique si les deux arguments sont équivalents.

Exemple :	*(EQUAL 'a 'a)*	*=> T*
	(EQUAL '(a b c) '(a b c))	*=> T*
	(EQUAL 'a 'b)	*=> ()*
	(EQUAL '(a b) '(b a))	*=> ()*
	(EQUAL '() '())	*=> ()*

Exercice n°1.7

Donnez le résultat des évaluations ci-dessous, sachant que la fonction "append" admet 2 listes en paramètre et restitue la liste résultant de leur concaténation.

```
((lambda (x y)
    ((lambda (x y z) (cons x (append y z))) 'cond   '(((eq x y) (* 2 x)))   '((t (+ x y))))) 18 24)
((lambda (x y)
    (eval ((lambda (x y z) (cons x (append y z))) 'cond   '(((eq x y) (* 2 x)))   '((t (+ x y)))))) 18 24)
(apply cdr '((a b c)))
(apply (lambda (x) (cdr x)) '((a b c)))
(eval ((lambda (x) x) '(car '(a b c))))
(apply eval '((cdr '(a b c))))
```

Exercice n°1.8

Soient les définitions suivantes de fonctions :

```
(de F1(A) (CONCAT B A))

(de F2(B) (F1 '(I N G | | I S N |'| T | | I T)))

(de F3(C) (F1 '(I N G | | I S N |'| T | | I T)))
```

Quel est le résultat de (F2 '(A M A Z)) ?
Quel est le résultat de (F3 '(A M A Z)) ?

Exercice n°1.9