```
? (cdr (cons 'a (cdadr '(((b
(c (d))))) )))
```

= ()

```
? (car (a b c))
```

**** eval : fonction indéfinie**
: a

```
? (cadr (cdr '(a (b c)) ))
```

= ()

```
? (cons (cdr '(a b)) (car
'((a) b) ))
```

= ((b) a)

III.6 - Autres fonctions de manipulation de liste

III.6.1 La fonction *list*

La fonction n-aire *list* retourne une liste formée des arguments spécifiés.

```
? (list ‘a ‘b)
= (a b)
? (list ‘(a b) ‘c)
= ((a b) c)
```

III.6.2 La fonction *append*

La fonction *append* retourne la concaténation des listes fournies en argument.

```
? (append ‘(a b) ‘(c d) )
= (a b c d)
```

III.7 - Définition de fonctions

Programmer en Lisp, c’est enrichir l’environnement existant avec de nouvelles fonctions. On définit des fonctions qui sont appelées comme les primitives du langage.
Créons par exemple une fonction qui retourne le second élément d’une liste :

```
? (de second (l)
?     (cadr l) )
= second
```

utilisation :

```
? (second ‘(1 2 3))
= 2
```

Autre fonction, qui retourne la différence du produit et de la somme de deux nombres :

```
? (de pms (x y)
?     (- (* x y) (+ x y)) )
= pms
? (pms 3 4)
```

= 5

de **est une fonction particulière ; en effet, ce n'est pas tant la valeur qu'elle retourne (le nom de la fonction définie) qui importe, mais plutôt le fait que l'environnement soit durablement changé. La fonction *de* est une fonction à "effet de bord".**

III.8 - T.D. 2 : Premières définitions de fonctions

Maîtrise des fonctions Lisp : *quote, car, cdr* et *cons* ; utilisation des fonctions Lisp : *list, append, de.*

A - Delta

Soit à trouver les racines d'une équation du second degré de la forme suivante :

$$ax^2 + bx + c = 0$$

Ecrire une fonction *delta* à trois arguments qui retourne le delta de l'équation (b^2 - 4ac). Ecrire ensuite une seconde fonction : *racines* qui retourne une liste de deux valeurs (les solutions de l'équation)

NB : On supposera Δ toujours positif.

(sqrt 16) -> 4

(- 5) -> -5

```
(de delta (a b c)
    (- (* b b) (* 4 a c)) )

(de racines (a b c)
    (list (/    (- -b (sqrt (delta a b c)))
                (* 2 a) )
          (/    (+ -b (sqrt (delta a b c)))
                (* 2 a) )))
```

B - Construction de liste

Construire la liste (A B C) de plusieurs façons

```
(cons 'A (cons 'B (cons 'C ())))
(append '(A) '(B C))
(list 'A 'B 'C)
(append (cons 'A ()) (list 'B 'C))
```

C - Listes

Ecrire les fonctions détaillées ci-dessous :

RemplaceUn : retourne une liste formée du premier élément de son premier argument placé au début de son second argument. ex :

```
? (remplaceun '(a b c) '(1 2 3 4))
= (a 2 3 4)
```

```
(de remplaceun (l1 l2)
        (cons   (car l1)
                (cdr l2) ))
```

Premiers : retourne la liste formée par les premiers éléments des deux paramètres. ex :

```
? (premiers '(a b c) '(1 2 3 4))
= (a 1)
```

```
(de premiers (l1 l2)
        (cons   (car l1)
                (car l2) ))
```

Seconds : idem pour les seconds éléments. ex :

```
? (second '(a b) '(1 2 3 4))
= (b 2)
```

```
(de seconds (l1 l2)
        (premiers   (cdr l1)
                    (cdr l2) ))
```

Reste : retourne une liste formée par les deux arguments passés en paramètre, privés de leur premier élément. ex :

```
? (reste '(a b c d) '(1 2 3))
= (b c d 2 3)
```

```
(de reste (l1 l2)
        (append   (cdr l1)
                  (cdr l2) ))
```

Insere : insère "beau" en deuxième position de la liste fournie en paramètre. ex :

```
? (insere '(le chapeau noir))
= (le beau chapeau noir)
```

```
(de insere (l)
        (cons   (car l)
                (cons   'beau
                        (cdr l) )))
```

Indefini : remplace les premier et quatrième éléments de la liste spécifiée en paramètre par "un". ex :

```
? (indefini '(le chien et le chat))
= (un chien et un chat)
```

```
(de indefini (l)
    (cons  'un
           (cons   (cadr l)
                   (cons  (caddr l)
                          (cons   'un
                                  (cdddr (cdr l)) )))))
```

D - Donner les expressions retournées par l'évaluateur dans chacun des cas suivants:

```
? (cons 'a ())
= (a)

? (append '(a b) '((b c)) )
= (a b (b c))

? (cons 'a '(list b c))
= (a list b c)

? (cons 'a (append '(f g)
(list 'b 'c)))
= (a f g b c)

? (caddr (cons 'a (list 'b '(c
d))))
= (c d)

? (append 'a (quote (b c)))
= (b c)

? (de foo (x)
?     (cons (car x)
?           (list 'et
?                 (caddr x)
? )))
= foo
? (foo '(chat souris iguane))
= (chat et iguane)

? (append (list 'a 'b) (cons
```

```
'c '(d)))
```

```
= (a b c d)
```

```
? foo
```

```
= ** err : variable indéfinie:
foo
```

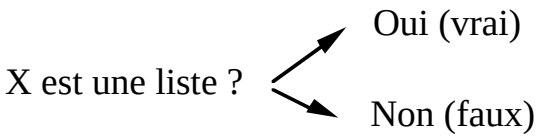
IV - Prédicats et conditionnelles

IV.1 - Prédicats

IV.1.1 - Introduction

Pour déterminer le type (liste, atome, nombre…) des valeurs passées en paramètre, ou pour comparer les valeurs entre elles, il existe des fonctions prédéfinies : les prédicats.

Chaque prédicat retourne une valeur booléenne indiquant si oui ou non la valeur de l'argument remplit la condition. Exemple :



Mais en Lisp, la valeur booléenne "faux" est assimilée à la valeur *()*, *nil*, la liste vide ; et la valeur booléenne "vrai" est assimilée à toute valeur différente de *()*.

NB : Certains prédicats devant retourner la valeur "vrai" retournent l'atome *t* (T pour True).*t* est un symbole dont la valeur est *true*.

(pour Prédicat)

IV.1.2 - Le prédicat *null*

si le test est vérifié, () sinon.

```
? (null ())
= t
? (null '(A B))
= ()
? (null 'A)
= ()
? (null t)
```

```
= ()? (null ())
= t
? (null (car '( () B) ))
= t
? (null (cdr '(A)))
 = t
? (null (cadr '(A B) ))
```

IV.1.3 - Les prédicats *atom* et *consp*

Prédicat *atom*

Le prédicat *atom* (ou *atomp*) teste si son argument est atomique (i.e. : un symbole, un nombre ou une chaîne de caractères), retourne *t* si le test est vérifié, *()* sinon.

```
? (atom ())
= t
? (atom 'ouf)
= t
```

```
? (atom 28)
= t
? (atom '(A B))
= ()
```

Prédicat consp

Le prédicat *consp* retourne son argument s'il s'agit d'une liste non vide, *()* sinon. En fait *consp* est le prédicat inverse de *atom*.

```
? (consp ())
= ()
? (consp 'A)
= ()
? (consp 4)
= ()
```

```
? (consp '(A B))
= (A B)
? (consp '( () ) )
= ( () )
```

IV.1.4 - Le prédicat *listp*

Le prédicat *listp* retourne vrai (donc *t*) si son argument est une liste, même vide, *()* sinon.

```
? (listp '(A B))
= t
? (listp ())
= t
? (listp '())
= t
```

```
? (listp 'A)
= ()
? (listp 4)
= ()
? (listp (cdr '(A)))
= t
```

N.B. : Différences en *listp* et *consp*

Les deux prédicats listp et consp sont très proches, en fait *()* est vu comme la liste vide par *listp*, alors qu'il est vu comme l'atome *nil* par *consp*.

Avantage de *consp* sur *listp* : *consp* retourne son argument lorsque cela est possible, et non l'atome *t* comme *listp*.

IV.1.5 - Les prédicats *symbolp, numberp* et *stringp*

Le prédicat *symbolp* retourne *t* si son argument est un symbole, *()* sinon.
Le prédicat *numberp* retourne son argument si c'est un nombre, *()* sinon.
Le prédicat *stringp* retourne son argument si c'est une chaîne de caractère, *()* sinon.

```
? (symbolp 4)
= ()
? (numberp 4)
= 4
? (stringp 4)
= ()
? (stringp "4")
= 4
```

```
? (symbolp 'foo)
= t
? (symbolp '(A B))
= ()
? (numberp 'A)
= ()
```

IV.1.6 - Le prédicat *equal*

La fonction *equal* retourne vrai si ses deux arguments sont identifiables (i.e. : sont soit un même objet, soit des copies l'un de l'autre)

```
? (equal () ())
= t
? (equal (+ 12 8) 20)
= t
? (equal '(a) 'a)
= ()
?(equal '(a b c) '(a b c))
= t
```

Il existe aussi une fonction nequal :

(nequal <a> <b>) <=> (not (equal <a> <b>))

```
? (nequal () ())
= ()

? (nequal t t)
= ()

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

IV.1.7 - Remarque sur "t"

A l'heure du second T

Lorsque cela est possible, un prédicat devant retourner "vrai" retournera son argument.

```
? (numberp 4)

= 4
```

Mais lorsque *nil* vérifie la condition du prédicat, cela n'est plus possible. Essayons de définir

une fonction *listp* qui retourne son argument au lieu de *t*

```
? (listp '(A B))
= (A B)
? (listp 'A)
= ()
? (listp ())
= ()…
```

Problème, *listp* souhaitait signifier à l'utilisateur que l'argument transmis était bien une liste, et donc le retourner. *Listp* retourne donc (). Mais lorsque l'utilisateur l'interprètera, il en déduira la valeur de vérité "faux", le contraire de ce que *listp* souhaitait.

En conséquence, tous les prédicats devant retourner "vrai" pour la valeur *nil*, ne pourront retourner leur argument. Ils devront retourner une valeur "vraie" immuable.

Certains, comme Vlisp retournent le nom du prédicat

```
? (atomp ())
= atomp
```

D'autres, créent un atome (une constante, un symbole…) t ayant pour valeur t. Ce qui entraîne de gros inconvénients :

- plus de variable *t* possible
- plus de fonction *t* possible
- plus de paramètre formel *t* possible

… en fait, le symbole *t* n'est plus disponible pour l'utilisateur.

• et surtout, non homogénéité des symboles Lisp : *t* est un symbole particulier. (A partir du moment ou un seul des symboles n'est pas "normal", la règle est transgressée, il pourrait bien en exister quelques autres, cela ne serait pas beaucoup plus grave !)

Remarque : Il existait déjà un symbole un peu particulier : nil

```
? (listp ())
= t
? (listp '())
= t
? (listp nil)
= t
? (listp 'nil)
= ()
```

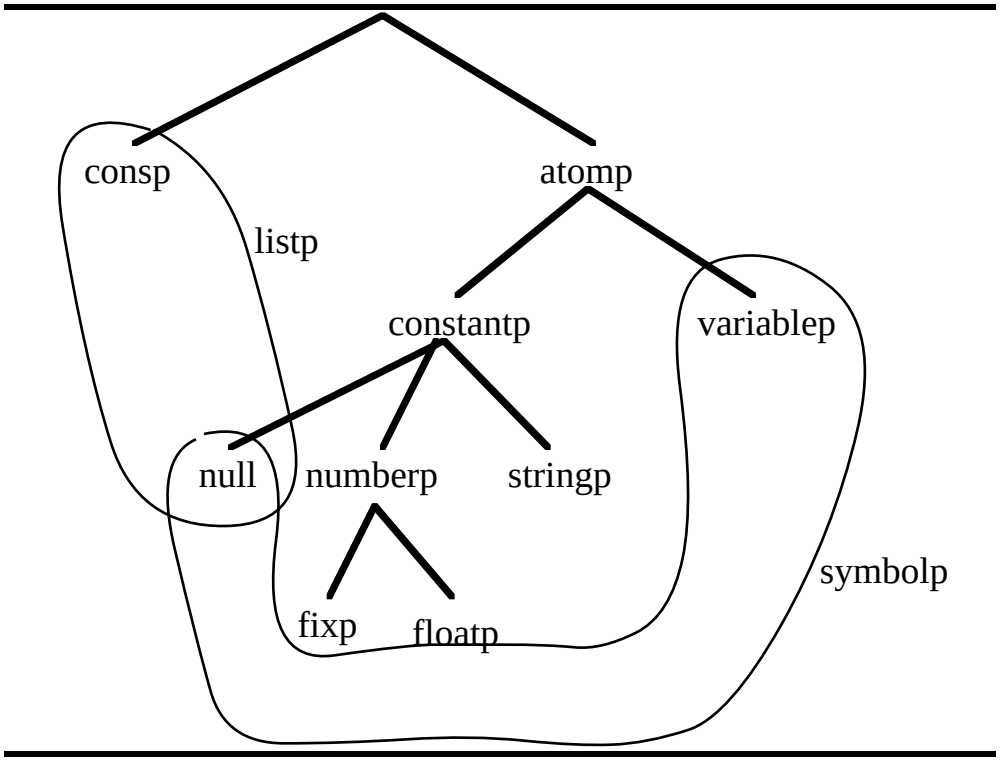
IV.1.8 - Prédicats sur les nombres

Il existe des prédicats de comparaison de nombres

```
? (< 4 3)
= ()
? (> 2 6)
= ()
? (< 2 9)
= 2
```

```
? (>= 4 1)
= 4
? (<> 3 3)
= ()
? (<= 3 3)
= 3
```

IV.1.9 - Arbre des prédicats Lisp



IV.1.10 - L'interprétation du résultat d'une fonction Lisp

Quelle est la différence entre une fonction Lisp "classique" et un prédicat ?

Aucune ! Tout est dans l'interprétation du résultat. Si le seul résultat souhaité est booléen, alors la fonction Lisp est utilisée en prédicat. En lisp, la frontière est entre prédicat et fonction est plus que floue puisque :

- **tous les prédicats ne retournent pas uniquement t ou ()**
- **est vrai tout ce qui n'est pas ().**

Exemples d'utilisation à contre sens :

```
(if (cdr l) …)
; plus agréable que (if (not (null (cdr l) )) …)
(cdr (consp l))
```

Un prédicat est donc une fonction pour laquelle tout ce qui intéresse est la valeur de vérité (vraie ou fausse) du résultat.

IV.2 - Conditionnelles

IV.2.1 - La fonction *if*

La fonction *if* correspond au *if then else* des langages de programmation classiques, à la seule différence que c'est un *if* ***fonctionnel***.

Beaucoup de langages tels C, Ada ou Basic sont impératifs (cf JUMP assembleur), il existe des instructions qui affectent la structure des données. Par contre, en lisp, la notion d'instruction n'est presque plus présente, le rôle d'une fonction est de retourner un résultat, non de modifier le contenu d'une variable.

En Lisp, *if* est une fonction.

Dans un langage impératif, la recherche de la majorité d'un individu s'écrirait de la manière suivante :

```
if age > 18
      then majorite := oui
      else majorité := non
```

alors qu'en Lisp :

```
(if   (> age 18)
      'oui
```

```
        'non)
```

sans devoir faire appel à une variable majorité

Syntaxe de la fonction *if* :

```
(if <condition> <action-si-oui> <action-si-non>)
```

La fonction *if* évalue son premier argument <condition>, si celui-ci est vrai, alors *if* retourne l'évaluation de son second argument : <action-si-oui> ; par contre, s'il est faux, *if* retourne l'évaluation de son troisième argument : <action-si-non>.

Ecrire la fonction *max* qui retourne le plus grand de deux entiers.

```
(de max (x y)
        (if (> x y)
            x
            y ))
```

Définir la fonction *null*

```
(de null (s)
        (equal s ()) ) ; eq en réalité !
```

La fonction *ifn* :

Il existe une fonction nommée *ifn*, qui se lit souvent "if null" car en Lisp la fonction *not* est identique à la fonction *null*, et *ifn* correspond bien au test de l'égalité à *()*.

Exemples :

```
(ifn (cdr liste) …)          IF NULL …
(ifn (> a b) …)              IF NOT …
```

IV.2.2 - La fonction *cond*

Historiquement *cond* est la conditionnelle la plus ancienne. Toutes les autres conditionnelles peuvent être redéfinies à partir de *cond*.

Syntaxe :

```
(cond   <e1>
        <e2>
        ...
```

```
            <en> )
```

Chaque argument <c1> .. <cn>, appelé “clause”, est de la forme :

```
(<tête> <queue>)
```

La fonction *cond* va évaluer la “queue” de la première clause dont la l’évaluation de la “tête” sera différente de *()*. *Cond* retournera alors la valeur de l’évaluation de cette queue de clause.

```
(cond
    (() 1)
    (t 5)
    (t 6) )
```

retourne 5.

Ecrire une fonction *type* qui retourne le type de son argument (atome, nombre ou liste)

```
(de type (x)
    (cond
        ((numberp x) ‘nombre)  ; dans cet ordre !
        ((atom x) ‘atom)
        (t ‘liste) )
```

En fait, la fonction cond permet de construire des structures de contrôle de la forme :

```
si      alors
        sinon si  alors
                  sinon si alors ...
```

```
(if   A                        (cond
      B       équivalent à           (A B)
      C)                             (t C) )
```

IV.2.3 - Les fonctions *and* et *or*

• *And*

And évalue successivement ses arguments jusqu’à ce que la valeur d’une évaluation soit égale à *()*. *and* retourne alors *()*. Sinon *and* retourne’évaluation de la dernière expression.

```
? (and)
```

```
= t

? (and 1 2 3)
= 3
? (and 1 () 3)
= ()
```

• *Or*

Or évalue successivement ses arguments jusqu'à ce que la valeur d'une évaluation soit différente de *()*, *or* retourne alors cette valeur. Si toutes les évaluations sont nulles, *or* retourne *()*.

```
? (or)
= ()
? (or 1 2)
= 1
? (or () () 2 3)
= 2
```

• Remarque

***And* retourne son premier argument faux, son dernier sinon ; *or* retourne son premier argument vrai, *()* s'il n'y en a pas.**

• Pourquoi les fonctions and et or sont des conditionnelles, ou comment faire le parallèle entre and, or et if.

```
(if a a b)   =   (or a b)

(if a b)     =   (and a b)
```

```
() )
```

• Exemple

Ecrire la fonction *nncar* (*not null car*) qui retourne le *car* de la liste passée en argument, seulement s'il est non nul (i.e : différent de *()*), et qui retourne *erreur* sinon.

```
(nncar '( () a b))      ->      erreur
(nncar '(1 2 3))        ->      1
```

```
(de nncar (l)
      (or (car l)
          'erreur))
```

IV.2.4 - La fonction *when*

```
(when   <test>                  si    <test>
        <oui 1>                       alors   <oui 1>
        <oui 2>         =                     <oui 2>
        ...                           ...
        <oui n>)                      <oui n>
                                      sinon ()
```

V - Récursivité

V.1 - Présentation

Il est parfois nécessaire d'itérer un calcul. Exemple (classique) :

$$n! = n \cdot \underbrace{(n-1) \cdot (n-2) \cdot \ \ldots \ \cdot 1}_{(n-1)!}$$

donc

$n! = n \cdot (n-1)!$

(pour $n > 0$, et avec $0! = 1$)

La fonction factorielle fait appel, dans sa définition, à elle même, elle est donc **récursive**. En voici la définition Lisp.

```
? (de fact (n)
?     (if (= n 0)
?         1
?         (* n (fact (1- n))) ))
= fact
? (fact 5)
=120
```

Comment se passe l'évaluation de `(fact 3)` ?

Premier appel à *fact*, n = 3

`(* 3 (fact 2))`

second appel, n = 2

`(* 3 (* 2 (fact 1)))`

troisième appel, n = 1

`(* 3 (* 2 (* 1 (fact  0))))`

quatrième appel, n = 0

le test est vrai, `(fact 0)` retourne 1,

l'expression en attente de vient donc :

`(* 3 (* 2 (* 1 1)))`, qui est évaluable,

`= 6`

V.2 - Définition de la fonction *membre*

Ecrire la fonction *membre* qui teste si un element est contenu dans une liste.

```
(membre 'a '(f a b))        ->  t
(membre 'a '(v e n))        ->  ()
```

• Expression du schéma recursif en français

Si l'élément recherché est égal au premier élément de la liste alors OK, sinon on cherche s'il est membre du reste de la liste (i. e. la liste privée de son premier argument)

• La fonction membre

```
(de membre (e l)
        (when l
            (if (equal e (car l))
                t
                (membre e (cdr l)) )))
```

V.3 - Comment bien écrire une fonction récursive

En général, écrire une fonction récursive consiste à

1 - Ecrire l'appel récursif

2 - Donner la condition d'arrêt

3 - Donner la valeur à retourner dans le cas d'arrêt

4 - Choisir la conditionnelle (if, when, or, and…)

V.4 - T.D. 3 : Définition de fonctions récursives

Utilisation des fonctions connues au travers de la récursivité.

A - Suite du TD 2 (Delta)

Soient à trouver les racines d'une équation du second degré de la forme suivante :

$$ax^2 + bx + c = 0$$

Ecrire une fonction *delta* à trois arguments qui retourne la valeur de delta pour l'équation $(b^2 - 4ac)$. Ecrire ensuite une seconde fonction : *racines* qui retourne une liste de deux valeurs (les racines de l'équation), une seule valeur (la racine double) ou nil (pas de racines)

```
(de racines (a b c)
  (cond ((> (delta a b c) 0)(list (/ (- -b (sqrt (delta a b c)))
                                     (* 2 a) )
                                  (/ (+ -b (sqrt (delta a b c)))
                                     (* 2 a) ) ) )
        ((= (delta a b c) 0)(/ -b (* 2 a)) )
        (t                    'pas-de-racines)  ))
```

B - Fonctions de recherche dans les listes

1) Définir un prédicat *estliste* qui retourne t si son argument est une liste, même vide.

ex :

```
(estliste '(a b))          ->  t
(estliste '())             ->  t
(estliste 4)               ->  ()
```

2) Ecrire la fonction *membre* sur des listes plates. (Membre teste si son premier argument est un élément de son second)

ex :

```
(membre 'a '(a b c))       ->  t ou (a b c) : "vrai"
(membre 'd '(a b c))       ->  ()
```

3) Ecrire *undescdr*, la fonction qui indique si son premier argument est une des copies d'un des cdrs de son second.

ex :

```
(undescdr '(3 4) '(1 2 3 4))->  t
(undescdr '(5) '(2 5 6))       ->   ()
```

4) Ecrire *niemecdr* qui retourne le nième *cdr* d'une liste.

ex :

```
(niemecdr 3 '(1 2 3 4 5 6)) ->  (4 5 6)
(niemecdr 5 '(1 2))         ->  ()
```

5) Ecrire une fonction *nieme* qui retourne le nième élément d'une liste, le *car* de cette liste étant l'élément de rang 0.

ex :

```
(nieme 2 '(a b c d))        ->  c
```

6) Définir *dernier*, une fonction qui retourne une liste composée du dernier élément de la liste paramètre.

ex :

```
(dernier '(a b c))          ->  (c)
(dernier '())               ->  ()
```

7) Définir la fonction *longueur* qui retourne la longueur d'une liste passée en paramètre.

ex :

```
(longueur '(a b c d))       ->  4
(longueur ())               ->  0
```

B

Utiliser (intelligemment) Lisp

Table des matières

Bibliographie

Ce cours a été réalisé à l'aide des documents suivants :

Jacques Ferber, Vol au dessus d'un nid de parenthèses
Emmanuel Saint-James
INRIA-ILOG, Le_Lisp Manuel de référence
Common-Lisp
Christian Quenniec
Christian Quenniec
Wertz

Index

Dialecte

Le dialecte d'un langage (un peu comme le dialecte d'une langue) représente une variante au langage de base, un ensemble de choix visant à implémenter physiquement un langage défini par sa philosophie. Le_Lisp, Turbo Pascal et Visual C++ son des dialectes respectivement de Lisp, Pascal et C++.

Machine virtuelle

Une machine X réalise un ensemble de fonctionnalités de base, ensuite, des fonctionnalités plus complexes sont décrites à partir des fonctionnalités primitives. Lorsque l'on souhaite faire émuler cette machine X (qui n'existe pas physiquement) à un ordinateur du marché, il suffit d'écrire comment est réalisée chacune des primitives sur la machine hôte. Cette machine hôte accueille donc la machine X qui n'existe que virtuellement.

Interprète ou interpréteur

Arbre Binaire

Nil

Nil vient du latin *nihil* qui signifie "rien" en Lisp la liste vide : () est appelée nil, mais nil correspond aussi à la valeur de vérité FAUX.

Liste plate

Une liste est plate si elle ne contient que des atomes. Le parcours récursif de liste plate est plus facile, car l'on sait que tous les cars successifs de la liste seront atomiques ; il ne sera donc pas nécessaire de lancer un appel récursif parallèle sur le car et sur le cdr de la liste.