Le LISP

Le langage LISP dont le nom est la contraction des termes « **Lis**t **P**rocessing » est un langage fonctionnel (également impératif mais on va l'ignorer) inventé en 1958 par John McCarthy (MIT).

Les Listes

Comme son nom l'indique, LISP traite des LISTES. Une liste est quelque chose qui commence par une parenthèse ouvrante "**(**" et qui se termine par une parenthèse fermante "**)**".

Syntaxe des Listes

Exemples de listes :

(A B C) // Liste avec 3 éléments
(Il Fait Beau) // Liste avec 3 éléments encore
(A (B C) D) // Liste avec 3 éléments dont le second est une liste de 2 éléments

Syntaxe d'une Liste en LISP :

Une *liste* commence par une parenthèse, contient optionnellement un nombre indéterminé d'éléments de type *item* ou est vide et se termine par une parenthèse
liste ::= ({item1 item2 … itemN})

Un item est soit une *liste* soit un *atome*
item ::= *liste* | *atome*

Une atome est un nombre ou bien un nom
atome ::= *nombre* | *nom*

Un *nombre* est composé de chiffres, signe et séparateur décimal)
nombre ::= n….n

Un nom est une suite de caractères alphanumériques commençant par une lettre et qui ne vomporte pas de *séparateur*
nom ::= a…a

Un *séparateur* est un caractère qui a un rôle particulier ou simplement de délimiter

Séparateur ::= . | espace | '(' | ')' | '[' | ']' | tab | ';' | Cr

Exemples d'objets LISP

Des atomes :
DO
RE
MI
CECI-EST-UN-ATOME-TRES-LONG
CUBE
CUBE1
1A

128	*un nombre*
-32600	*un nombre négatif*
HAHA	*un atome nom (atome alphanumérique)*
()	*une liste à 0 élément*
(HAHA)	*une liste à 1 élément*
(UNE LISTE)	*une liste à 2 éléments*
((UNE) LISTE)	*c'est une liste à 2 éléments, le premier élément est une liste à 1 élément, le deuxième élément est un atome ;*

Exemples de choses qui ne sont pas des objets LISP :

)	*rien ne peut commencer par une parenthèse fermante*
(TIENS (TIENS (TIENS	*les parenthèses fermantes sont manquantes*
(...)	*le caractère "point" n'est pas permis*

Quelle est la taille de ces listes ?

(JOHN GIVES MARY A BOOK)
((SUJET)(VERBE)(OBJET))
(((X + Y) + Z) --> (X + (Y + Z)))
(QUELQU-UN AIME MARY)
(DU PASSE FAISONS TABLE RASE)
(1 2 3 A B C C B A 3 2 1)
( )

La liste de taille 0 est nommée **NIL**. Pour LISP l'atome **NIL** et la liste **()** ont la même valeur.

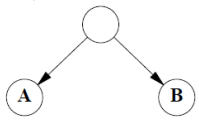
Les programmes LISP manipulent la plupart du temps des listes et leur code est écrit sous la forme d'une liste.

Une autre Représentation des listes :

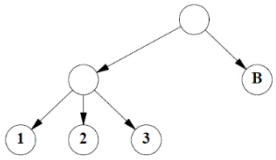
Cette représentation sous forme d'arbre permet de reconnaître visuellement à la fois les éléments et leur profondeur.

- *Une liste (ou sous-liste) est représentée comme un cercle vide.*
- *Les éléments d'une liste sont les cercles directement connectés au cercle de la liste, et la profondeur d'un élément se compte simplement par le nombre des flèches qu'il faut suivre (en partant du sommet) pour arriver à l'élément.*

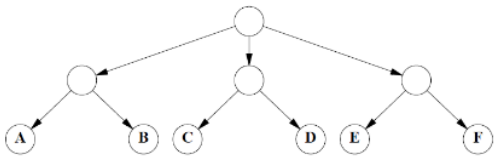
(A B)



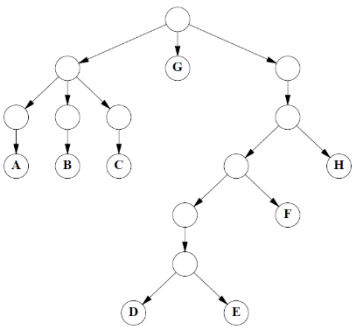
((1 2 3) B)



((A B) (C D) (E F))



(((A)(B)(C)) G (((((D E)) F) H)))



Les Fonctions

Les listes et les atomes sont les objets sur lesquels les programmes LISP travaillent. Il existe des listes spéciales qui servent à indiquer à l'évaluateur LISP les actions à réaliser. Ce sont le *formes* qui servent à appeler des fonctions.

Forme ::= (nom_de_la_fonction {argument1 … argumentN})

Un appel de fonction retourne une valeur : le résultat de la dernière évaluation d'expression

Quelques fonctions de base du LISP :

La fonction QUOTE :

Elle sert à retourner un objet LISP tel qu'il est sans l'évaluer :

Exemples d'utilisations de la fonction QUOTE :

(QUOTE Bonjour)	→	**Bonjour**	
(QUOTE (A B C))	→	**(A B C)**	
(+ 1 2)	→	**3**	*Forme addition*
(QUOTE (+ 1 2))	→	**(+ 1 2)**	*Avec QUOTE l'addition n'est pas évaluée (+ 1 2) est une liste*
'(+ 1 2)	→	**(+ 1 2)**	**'** *est équivalent à la forme QUOTE*

Les fonctions sur les listes : CAR, CDR, CADR, CADDR

Ces fonctions permettent d'accéder aux éléments d'une liste

La fonction CAR

CAR **:** retourne le premier élément de la liste passée en argument

Exemples d'utilisations de la fonction CAR :

(CAR '(A B C))	→	**A**
(CAR '(CAR '(B C)))	→	**CAR**
(CAR NIL)	→	**NIL**
(CAR ())	→	**NIL**

La fonction CDR

CDR **:** retourne la liste passée en argument privée de son premier élément (le CAR)

Exemples d'utilisations de la fonction CDR :

(CDR '(A B C))	→	**(B C)**
(CDR '(CAR '(B C)))	→	**(B C)**
(CDR NIL)	→	**NIL**
(CDR ())	→	**NIL**

Les fonctions CADR et CADDR :

CADR : est l'équivalent de CAR(CDR(arg)) donc CADR retourne ….

CADDR : est l'équivalent de CAR(CDR(CDR(arg))) donc CADDR retourne ….

Les fonctions pour la construction de listes

La fonction CONS permet de créer (CONStruire) une liste.

CONS (arg, argListe) : retourne la liste argListe avec la valeur de arg en tant que premier élément

Exemples d'utilisations de la fonction CONS :

(CONS 'A '(B C)) → **(A B C)**
(CONS '(A B) '(C D)) → **((A B) C D)**

Exercice :

Que retourne l'évaluation des expressions suivantes ?

(CONS 'A '(B C))
(CONS '(A B) '(C D))
(CONS (CAR '(A B C))(CDR '(A B C)))
(CAR (CONS 'A '(B C)))
(CDR (CONS 'A '(B C)))
(CAR '((A (B C)) D (E F)))
(CDR '((A (B C)) D (E F)))
(CADR (CAR '((A (B C)) D (E F)))
(CADDR '((A (B C)) D (E F))
(CONS 'NOBODY (CONS 'IS '(PERFECT)))
(CONS (CAR '((CAR A) (CDR A))) (CAR '(((CONS A B)))))

La définition de fonctions :

Pour compléter la liste des fonctions fournies par LISP il est possible de créer et d'utiliser nos propres fonctions. Créer une fonction consiste à la définir.

Lisp met à disposition une fonction particulière nommée **DEFUN** permettant de définir des fonctions.

(DEFUN nom_ fonction({nomArg1 … nomArgN}) (corps_de_la_fonction))

Exemple de définition d'une fonction pour faire la somme de deux nombres :

(DEFUN somme(a b) (+ a b))

Lorsque la fonction DEFUN est évaluée, l'évaluateur enregistre la fonction. La fonction **DEFUN** *retourne un atome nom ayant pour valeur le nom de la fonction définie. La fonction est maintenant utilisable (Appelable).*

L'évaluation de l'expression « **(somme 3 2)** » consiste en l'appel de la fonction somme définie en affectant la valeur 3 à **a** et la valeur 2 à **b**. Le retour de l'évaluation devrait être ….

TP LISP

Installation et test d'un interpréteur de LISP :

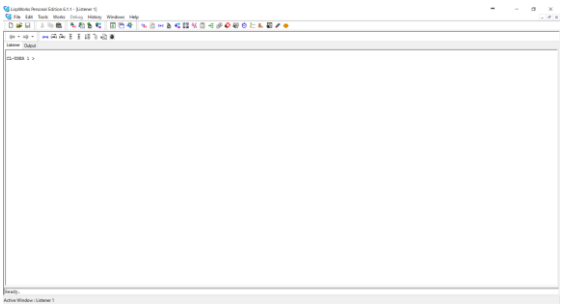
Téléchargement de LispWork :

Télécharger l'installeur « LispWorksPersonal61.exe » à partir du lien Dropbox suivant :
https://www.dropbox.com/s/a8pzuuws5fuli6d/LispWorksPersonal61.exe?dl=0

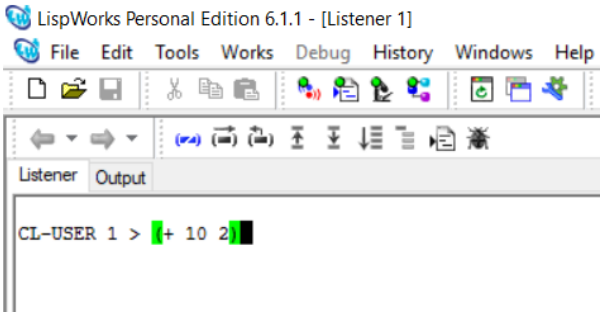
L'Installation se fait simplement en lançant « LispWorksPersonal61.exe »

Test de l'interpréteur LispWork :

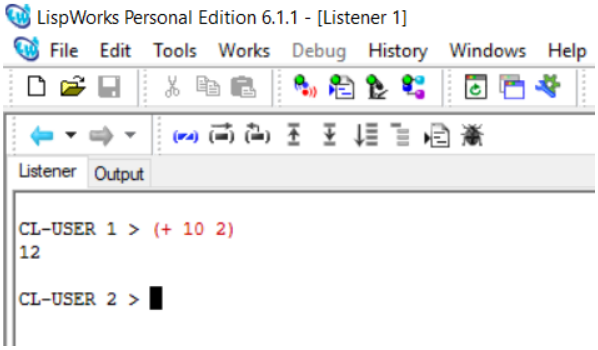
Lancer LispWork



La console s'affiche, l'évaluateur attend une expression à évaluer.



Saisir une expression à évoluer puis appuyer sur la touche Entrée



L'évaluateur affiche le résultat de l'évaluation puis attend une nouvelle expression à évaluer.

Tutorial LISP :

Déclaration et Affectation de variables :

```
> (setq a 5)              ;store a number as the value of a symbol
5
> a                       ;take the value of a symbol
5
> (let ((a 6)) a)         ;bind the value of a symbol temporarily to 6
6
> a                       ;the value returns to 5 once the let is finished
5
> (+ a 6)                 ;use the value of a symbol as an argument to a function
11
> b                       ;try to take the value of a symbol which has no value
Error: Attempt to take the value of the unbound symbol B
```

On va s'interdire au maximum ces instructions pour utiliser LISP comme un langage fonctionnel pur.

Conditions :

IF :

```
There are two special symbols, t and nil. The value of t is defined
always to be t, and the value of nil is defined always to be nil. LISP
uses t and nil to represent true and false. An example of this use is
in the if statement, described more fully later:

> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

COND :

```
> (cond
    ((evenp a) a)          ;if a is even return a
    ((> a 7) (/ a 2))      ;else if a is bigger than 7 return a/2
    ((< a 5) (- a 1))      ;else if a is smaller than 5 return a-1
    (t 17))                ;else return 17
2
```

CASE :

Identique au switch du C

```
> (case x
    (a 5)
    ((d e) 7)
    ((b f) 3)
    (otherwise 9))
```

Itérations :

```
> (setq a 4)
4
> (loop
   (setq a (+ a 1))
   (when (> a 7) (return a)))
8
> (loop
   (setq a (- a 1))
   (when (< a 3) (return)))
NIL

The next simplest is dolist: dolist binds a variable to the elements of
a list in order and stops when it hits the end of the list.

> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

```
The most complicated iteration primitive is called do. A do statement
looks like this:

> (do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y)
    (print 'working))
```

Exercices :

1 Représenter chacune de ces listes sous-forme d'arbre binaire. Puis écrire l'expression qui évaluée retournera l'atome C

L1 = (a b c d)
L2 = ((a b) (c d))
L3 = (((a) (b) (c) (d)))
L4 = ((((a) b) c) d)

2 Définir et tester les fonctions suivantes :

- **fac** qui calcule la factorielle pour un entier n supérieur ou égal à 0.
  >(fac 5)
  120

- **lnpe** qui renvoie la liste des n premiers entiers dans l'ordre décroissant 0 non compris.
  > (lnpe 5)
  (5 4 3 2 1)

- **fib** qui renvoie le nième terme de la suite de Fibonnacci :
  u(0) = 1, u(1) = 1, u(n) = u(n-1) + u(n-2)
  >(fib 5)
  8

- **longueur** qui renvoie la longueur d'une liste
  *vous pourrez utiliser le prédicat **(null L)** retourne **t** si L est null et **f** ou **NIL** sinon.*
  > (longueur '(a b c d e))
  5

- **dernier** qui renvoie le dernier élément d'une liste
  > (dernier '(a b c d e))
  E

- **conc** qui concatène deux listes
  > (conc '(a b) '(c d e))
  (A B C D E)

- **present** qui détecte la présence d'un élément dans une liste
  > (present 'a '(c d e a b))
  T

- **elimin** qui élimine toutes les occurrences d'un atome dans une liste
  > (elimin 'a '(c a d e a b))
  (C D E B)

- **renverse** qui renvoie la liste dans l'ordre inverse
  (on peut utiliser notre fonction **conc** ou la primitive **NCONC**)
  >(renverse '(a b c d e))
  (E D C B A)

- **rang_pair** qui renvoie tous les éléments de rang pairs d'une liste
  > (rang_pair '(a b c d e f))
  (B D F)

- **remplace** qui remplace toutes les occurrences de x par y dans l
  > (remplace 'a 'f '(c a d e a b))
  (C F D E F B)

- **renversebis** qui qui fait la même chose que renverse mais sans utiliser conc (en utilisant un accumulateur)

- **present2** qui détecte la présence à un niveau quelconque
  >(present2 'a '(1 (2 a) 3))
  T

- **elimin2** à tous les niveaux
  >(elimin2 'a '(1 (2 a) 3))
  (1 (2) 3)

- **remplace2** à tous les niveaux
  >(remplace2 'a 'b '(1 (2 a) 3))
  (1 (2 B) 3)

- **renverse2** qui renverse également les sous-listes
  >(renverse2 '(1 (2 a) 3))
  (3 (A 2) 1)

Les fonctions en tant qu'arguments :

Utiliser une fonction passée en argument :

Une fonction peut être passée en argument tout comme les atomes et les listes. La forme à utiliser est : **#'***nom_de_la_fonction*

Pour l'invoquer il faut la passer en premier argument de la primitive **funcall**, les arguments suivants passés à funcall sont les arguments pour utilisés pour l'appel de la fonction.

```
(defun  incrementeur(n) (+ n 1))
(defun  decrementeur(n) (- n 1))
(defun increm_decrem(n f) (funcall f n))
(increm_decrem 5  '#incrementeur)
> 6
(increm_decrem 5  '#decrementeur)
> 4
```

Certaines fonctions comme **defun** peuvent retourner des fonctions

```
CL-USER 1 > (funcall (defun mafunc(n) (+ n 1)) 2)
> 3
```

IL est possible de créer des fonctions à la volée (pendant l'évaluation). Ce sont les fonctions **Lambda**. Elles sont anonymes (pas de nom) et ne sont pas référencées pour être rappelées.

```
CL-USER 3 > (funcall (lambda (a b c) (+ a (* b c))) 1 2 3)
> 7
```

Les fonctions appliquées sur les listes :

Mapcar :

La fonction **mapcar** permet d'appliquer une fonction à tous les éléments d'une liste.
Le résultat de cette évaluation est une liste contenant le résultat de toutes ces évaluations.

Fonction unaire (1 argument) : (mapcar f '(a1 a2 ... an)) → (list (f a1) (f a2) ... (f an))
Fonction n-naire (n arguments) : (mapcar f '(a1... an) '(b1... bn) ...) → (list (f a1 b1 …) (f a2 b2 …) ...)

```
CL-USER 16 : 7 > (mapcar #'+ '(4 5 2) '(4 5 2))
(8 10 4)
```

Si les listes n'ont pas le même nombre d'arguments, dans le cas d'opérations n-aires, c'est la plus courte qui donne la taille du résultat.

Apply :

La fonction **apply** permet d'appliquer une fonction en utilisant comme arguments les éléments d'une liste.

```
CL-USER 18 : 8 > (apply #'+ '(3 4 5))
12

apply vs funccall : le mode de passage des arguments est différent
(apply #'+ '(5 6 7) )
(funcall #'+ 5 6 7)
```

Exercices :

1. Écrire une fonction qui évalue une expression donnée sous forme infixée complètement parenthésée. Toute expression est toujours de la forme (Opérande1 Opérateur Opérande2). Un Opérande est toujours un nombre ou une expression. Les opérateurs possibles sont uniquement '+', '-', '*'

```
> (eval_infix '((2 - 3) + ((5 * 6) - 1)))
28
```

2. Écrire de plusieurs manières différentes une fonction qui retourne le nombre le plus grand d'une liste. Si la liste est vide la fonction doit retourner NIL.

```
> (plus_grand '(7 3 28 22 11 3 7 9))
28
```

```
(DEFUN plus_grand_accu (l accu) (COND ((null l) NIL) ((null (CDR l)) (CAR l)) (t  (plus_grand_accu (CDR l) (IF (> (CAR l) accu) (CAR l) accu) ))))
```