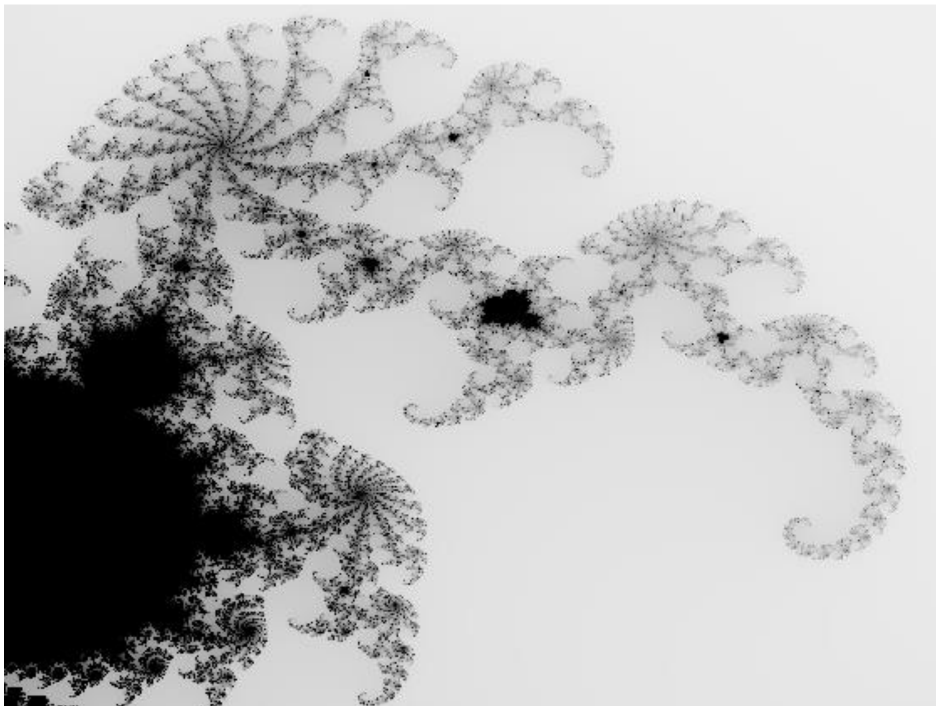
# Introduction
# à
# AutoLISP®

Programmation d'AutoCAD®



© Gilles Chanteau

# *Sommaire*

**1 Généralités** **3**
1.1 Types d'expressions, types de données 3
1.2 Syntaxe et évaluation (*quote type*) 3
**2 Saisie directe** **6**
2.1 Opérateurs arithmétiques (*+ - * / sqrt rem expt exp log Boole logand logior*) 6
2.2 Variables (*setq eval set*) 7
**3 Programmes** **9**
3.1 Chargement de fichiers (*load appload*) 9
3.2 Chargement automatique (*autoload S::STARTUP*) 10
**4 Éditeur Visual LISP®** **11**
4.1 Présentation 11
4.2 Les fenêtres 11
4.3 Mise en forme 12
4.4 Sélection par double clic 13
4.5 Aperçu des fonctions de débogage 13
**5 Commandes AutoCAD®** (*command pause vl-cmdf*) **14**
**6 Définitions de fonctions** (*defun defun-q*) **15**
6.1 (defun c:symbole …) vs (defun symbole …) 15
**7 Entrées utilisateur** (*getint getreal getpoint getcorner getangle getorient getdist getstring getkword*) **16**
7.1 Initialisation (*initget*) 16
7.2 Boites de dialogue (*getfiled acad_coclordlg acad_truecoclordlg*) 17
**8 Listes et points** **18**
8.1 Construction des listes (*quote list cons*) 18
8.2 Accès aux éléments (*car cdr last nth vl-position member length*) 18
8.3 Manipulation des listes (*append reverse subst vl-remove*) 19
8.4 Liste d'association et paire pointée (*assoc*) 19
**9 Variables système** (*getvar setvar*) **20**
9.1 Variables d'environnement (*getenv setenv*) 20
**10 Fonctions géométriques** (*distance angle polar inters sin cos atan trans*) **21**
**11 Affichage de texte** (*prompt princ prin1 print alert textscr graphscr*) **22**
**12 Décisions conditionnelles** (*if cond*) **23**
12.1 Comparaison (*= /= < > eq equal zerop minusp wcmatch*) 23
12.2 Oprérateurs logiques (*and or not null*) 24
12.3 Type de donnée (*atom listp vl-consp numberp*) 25
**13 Procédures itératives et récursives** **26**
13.1 Boucle et incrémentation (*repeat while 1+ 1-)* 26
13.2 Traitement de liste (*foreach mapcar lambda apply vl-every vl-some vl-member-if(-not) vl-remove-if(-not) vl-sort*) 26
13.3 Fonctions récursives 28
**14 Chaînes de caractères et Fichier ASCII** **30**
14.1 Chaînes de caractères (*strcat strlen strcase substr vl-string-* read*) 30
14.2 Conversions (*itoa atoi rtos atof distof angtos angtof ascii chr vl-string->list vl-list->string float fix cvunit*) 30
14.3 Fichiers ASCII (*findfile open close read-char read-line write-car write-line*) 32
**15 Gestion des erreurs** *(*error* vl-catch-all-…)* **33**
**16 Accès aux objets** **35**
16.1 Entité unique (*entsel nentsel nentselp entlast entnext entdel handent*) 35
16.2 Données DXF des objets (*entget entmake entmakex entmod entupd*) 36
16.3 Jeux de sélection (*ssget ssadd ssdel sslength ssmemb ssname ssnamex ssgetfirst sssetfirst*) 37
16.4 Filtres de sélection 38
16.5 Tables (*tblnext tblsearch tblobjname*) 39
16.6 Données étendues et dictionnaires (*regapp, xdroom, xdsize, dictnext dictsearch dictadd dictremove dictrename namedobjdict*) 40

© Gilles Chanteau

# *1 Généralités*

Inventé par John McCarthy en 1958 au Massachusetts Institute of Technology, le LISP, acronyme de List Processing est le deuxième plus vieux langage de programmation. Langage de haut niveau, il est généralement classé parmi les langages fonctionnels. Il a été très en vogue dans les années 1970/80 dans les domaines de la recherche et de l'intelligence artificielle.
AutoLISP® est un dialecte du LISP spécialement implémenté pour fonctionner avec AutoCAD®.
Dans la suite de ce document, il ne sera question que d'AutoLISP®.

## 1.1 Types d'expressions, types de données

Les expressions LISP se partagent entre deux types : **liste** et **atome**.
Tout ce qui n'est pas une liste est un atome (excepté **nil**).

### 1.1.1 Les atomes

Les atomes peuvent être des expressions dites autoévaluantes :
- les nombres **entiers**, sans séparateur décimal : 1, -27, 23504 … (compris entre -2147483648 et 2147483647)
- les nombres **réels**, avec séparateur décimal, 1.0, 0.05, 3.375 …
- les **chaînes**, ensembles de caractères contenus entre des guillemets : "a", "Le résultat est : 128.5".

ou des expressions évaluables :
- les **symboles**, qui sont des suites de caractères (exceptés parenthèses, guillemets, apostrophes, points et points virgules) non exclusivement constituées de chiffres.

Les symboles peuvent être affectés à des **fonctions** (prédéfinies ou définies par l'utilisateur), ou à des données, dans ce cas, on parle de **variables**, les données stockées dans les variables peuvent être des expressions LISP et aussi des pointeurs vers des objets spécifiques (jeux de sélection, noms d'entités, descripteurs de fichiers…).
Certains symboles sont protégés, outre les fonctions LISP prédéfinies, il s'agit de **T** et **pi**.

### 1.1.2 Les listes

La **liste** est l'expression fondamentale de tous les langages LISP.
Une liste commence avec une parenthèse ouvrante et se termine avec une parenthèse fermante, entre ces parenthèses, chaque élément est séparé par une espace. Les éléments constituant les listes sont des expressions LISP de tout type y compris des listes. Ces imbrications multiples expliquent la multiplication des parenthèses caractéristique à ce langage.
Un programme LISP est une liste d'expressions qui peuvent être elle mêmes des listes. Ce qui fait dire que LISP est défini récursivement.

### 1.1.3 Nil

**nil** est une expression atypique en ce sens qu'elle exprime plusieurs concepts.
Elle exprime la notion de vide : un symbole auquel aucune valeur n'est affectée est **nil**, une liste qui ne contient aucun élément est **nil** aussi.
**nil** sert aussi pour exprimer le résultat Booléen faux (false) opposé à **T** vrai (true).
Malgré que son type soit **nil**, elle est considérée à la fois comme un **atome** et comme une **liste**.

## 1.2 Syntaxe et évaluation (quote type)

Un programme LISP (langage dit « fonctionnel ») est essentiellement constitué d'**appels de fonction**.
Une **fonction**, en informatique comme en mathématique est un ensemble d'instructions qui retourne un résultat dépendant des **arguments** (paramètres) qui lui sont passés, les mêmes arguments passés à une fonction retournent toujours le même résultat.
Une expression LISP retourne toujours un résultat (fusse-t-il **nil**), il n'existe pas en LISP d'instruction qui ne retourne rien.
Le résultat de l'évaluation d'un **appel de fonction** est retourné à l'endroit même de cet appel.
AutoLISP® fournit de nombreuses fonctions prédéfinies.

© Gilles Chanteau

La programmation LISP consiste à définir de nouvelles fonctions en utilisant les fonctions préexistantes, qu'elles soient natives ou définies par l'utilisateur.

### 1.2.1 Évaluation des applications de fonction

AutoCAD® intègre un interpréteur qui permet d'évaluer les expressions LISP.
L'interpréteur est accessible directement à la ligne de commande.

Les expressions de type liste destinées à être évaluées sont appelées des **appels de fonction** (ou **applications de fonction**).
La Notation dite « préfixe » du LISP en détermine la structure : le premier élément est une **fonction** (opérateur), les suivants les **arguments** requis par cette fonction.

Suivant les fonctions les arguments peuvent être optionnels et leur nombre variable.

```
(fonction [argument …])
```

Par exemple 2 + 3 s'écrit en LISP : `(+ 2 3)`

```
Commande: (+ 2 3)
5
```

L'évaluation d'une expression LISP commence par le contrôle de l'appariement des parenthèses.

```
Commande: (+ 2 3))
; erreur: parenthèse fermante supplémentaire en entrée
```

S'il manque une (ou des) parenthèse(s) fermantes, il est demandé à l'utilisateur de corriger :

```
Commande: (+ 2 3
(_>
```

Ceci fait, l'évaluation est faite et son résultat est retourné :

```
Commande: (+ 2 3
(_> )
5
```

Ensuite l'interpréteur recherche la définition de la fonction (premier terme de la liste). Si cette définition n'est pas trouvée, un message d'erreur est retourné :

```
Commande: (toto)
; erreur: no function definition: TOTO
```

Si la fonction est définie, la validité de chaque argument (termes suivants) est évaluée, au premier argument incorrect trouvé, un message d'erreur est retourné :

```
Commande: (+ "2" "3")
; erreur: type d'argument incorrect: numberp: "2"
```

Les arguments pouvant être aussi des applications de fonction, celles-ci seront évaluées tour à tour et les résultats retournés à l'endroit même de chaque expression.
Par exemple, dans l'expression `(sqrt (+ (* 4 4) (* 3 3)))`,
`(+ (* 4 4) (* 3 3))` est l'argument de la fonction `sqrt`,
`(* 4 4)` et `(* 3 3)` les arguments de la fonction `+`,
4 et 4 ou 3 et 3 ceux de la fonction `*`.

L'évaluation procède de façon récursive en réduisant chaque appel de fonction à une expression autoévaluante à partir des expressions les plus imbriquées.

Le suivi de l'expression ci-dessus peut être décrit comme suit :

Saisie : **(* 4 4)**
Résultat : **16**
Saisie : **(* 3 3)**
Résultat : **9**
Saisie : **(+ 16 9)**
Résultat : **25**
Saisie : **(SQRT 25)**
Résultat : **5.0**

Soit : `(sqrt (+ (* 4 4) (* 3 3)))`
=> `(sqrt (+ 16 (* 3 3)))`
=> `(sqrt (+ 16 9))`
=> `(sqrt 25)`
=> `5.0`

### 1.2.2 Données sous forme de listes (quote)

AutoLISP® intègre de nombreuses fonctions qui requièrent comme argument une (ou des) liste(s) -voir chapitre 8. Ce type de liste est considéré comme une donnée unique, par exemple, un point se définit comme la liste de ses coordonnées (x y [z]).
Ces listes n'étant pas des applications de fonction, il faut empêcher leur évaluation.
On utilise pour cela la fonction **quote**.
`(quote expression)`

`(quote (1 2 3))` retourne : **(1 2 3)**
`(quote (+ 2 3))` retourne : **(+ 2 3)**

Cette fonction, très largement utilisée, s'abrège avec une apostrophe :

`'(1 2 3)` est équivalent à `(quote (1 2 3))`

### 1.2.3 Fonction type

La fonction **type** retourne le type de donnée de l'expression qui lui est passée comme argument.
Les différents types de données AutoLISP® sont :

INT : nombre entier
REAL : nombre réel
STR : chaîne de caractères
SYM : symbole
LIST : liste
ENAME : nom d'entité
PICKSET : jeu de sélection
FILE : descripteur de fichier
SUBR : fonction AutoLISP® interne ou fonction chargée depuis des fichiers FAS ou VLX
USUBR : fonction définie par l'utilisateur chargée depuis un fichier LSP
VL-CATCH-ALL-APPLY-ERROR : erreur retournée par une expression vl-catch-all-apply

`(type 10)` retourne : **INT**
`(type 25.4)` retourne : **REAL**
`(type "test")` retourne : **STR**
`(type '(1 2 3))` retourne : **LIST**
`(type quote)` retourne : **SUBR**
`(type nil)` retourne : **nil**

# 2 Saisie directe

Les expressions LISP pouvant être entrées directement à la ligne de commande, toute expression qui débute par une parenthèse est interprétée comme une expression LISP.

## 2.1 Opérateurs arithmétiques (+ - * / sqrt rem expt exp log)

Il est donc possible d'utiliser AutoLISP® comme une calculatrice, directement à la ligne de commande, avec les opérateurs numériques.
Ces expressions peuvent être utilisées pour répondre à une invite dans une commande.

NOTA : les fonctions **+**, **-**, *, **/**, **rem** et **expt** retournent un entier si tous leurs arguments sont entiers, un réel si au moins un des arguments est réel.

**+** : addition `(+ [nombre nombre] …)`

    (+ 3 4 2) retourne : 9
    (+ 5 9.0) retourne : 14.0

**-** : soustraction `(- [nombre nombre] …)`

    (- 9 4 2) retourne : 3
    (- 5.2 6) retourne : -0.8

* : multiplication `(* [nombre nombre] …)`

    (* 2.5 3) retourne : 7.5
    (* 2 6 -4) retourne : -48

**/** : division `(/ [nombre nombre] …)`

    (/ 20 3) retourne : 6
    (/ 20.0 3) retourne : 6.66667

**rem** : reste de la division `(rem [nombre nombre …])`

    (rem 20 3) retourne : 2
    (rem 55 7 3) => (rem (rem 55 7) 3) => (rem 6 3) retourne : 0

**expt** : exposant `(exp nombre puissance)`

    (expt 5.0 2) retourne : 25.0
    (expt 2 5) retourne : 32

**exp** : exponentielle `(exp nombre)`

    (exp 1) retourne : 2.71828
    (exp 2.5) retourne : 12.1825

**log** : logarithme `(log nombre)`

    (log 2.7182818) retourne : 1.0
    (/ (log 32) (log 2)) retourne : 5.0

### 2.1.1 Opérateurs logiques binaires (Boole logand logior)

**Boole** `(Boole operateur int1 [int2 ...])`
C'est l'opérateur générique de comparaison "bit à bit". Il requiert au moins deux arguments mais est généralement utilisé avec trois : un entier définissant l'opérateur et deux entiers à comparer.

L'argument `operateur` est un entier de 0 à 15 représentant une des seize fonctions Booléennes possibles avec deux variables.

Les arguments `int1` et `int2` sont comparés bit à bit (logiquement) sur la base de la "table de vérité" ci-dessous en fonction des combinaisons de bit d'opérateur.
Si le résultat de la comparaison est VRAI, la valeur décimale du bit est ajoutée au résultat retourné.

| Table de vérité | | |
|---|---|---|
| bit opérateur | int1 | int2 |
| 1 | 1 | 1 |
| 2 | 1 | 0 |
| 4 | 0 | 1 |
| 8 | 0 | 0 |

Certaines valeurs d'opérateur donnent les opérations Booléennes standard :
AND : 1 (les deux entrées sont 1)
OR : 7 (1+2+4 une des entrées ou les deux sont 1)
XOR : 6 (2+4 une seule des deux entrées est 1)
NOR : 8 (aucune des entrées n'est 1)

`(Boole 6 6 5)` retourne : **3** (2 est uniquement dans 6 et 1 uniquement dans 5)
`(Boole 4 6 5)` retourne : **1** (1 est uniquement dans 5)

Les fonctions **logand** et **logior** acceptent un nombre indéterminé de nombres entiers comme arguments et retournent respectivement le AND et le OR logique d'une comparaison bit à bit de ces entiers.
`(logand 2 7)` retourne : **2**
`(logior 3 5)` retourne : **7**

## 2.2 Variables (setq eval set)

Il est possible de stocker des données (résultat d'expression) dans des symboles ; la variable ainsi définie conserve sa valeur dans le dessin pendant toute la session tant qu'elle n'est pas redéfinie.

### 2.2.1 Fonction setq

`(setq symbole expression [sym2 expr2] …)`

La fonction **setq** sert à définir des variables en affectant à un symbole (premier argument), une expression (second argument).

Il est possible dans la même expression de faire passer plusieurs paires d'arguments. L'expression retourne le résultat de la dernière expression.
Pour faire afficher la valeur d'une variable à la ligne de commande, il suffit d'entrer le nom de la variable précédé d'un point d'exclamation (!).

```
Commande: (setq a 10 b (/ 1.0 3))
0.333333

Commande: !a
10

Commande: !b
0.333333
```

La valeur stockée dans la variable pourra être réutilisée pour répondre à l'invite d'une commande ou dans une autre expression.

```
Commande: cercle
Spécifiez le centre du cercle ou [3P/2P/Ttr (tangente tangente rayon)]: 10,20

Spécifiez le rayon du cercle ou [Diamètre]: !a
10


Commande: (* 2 pi a)
62.8319
```

### 2.2.2 Fonction eval

`(eval expr)`

La fonction **eval** retourne le résultat de l'évaluation de l'expression qui lui est passée comme argument.

```
Commande: (setq c 'a)
A

Commande: (setq d '(* a a))
(* A A)

Commande: (eval c)
10

Commande: (eval d)
100
```

Cette dernière évaluation pourrait se décomposer de la sorte :

```
(eval d)
=> (eval '(* A A))
=> (eval (quote (* 10 10)))
=> (* 10 10)
=> 100
```

### 2.2.3 Fonction set

`(set 'sym expr)` ou `(set expr expr)`

La fonction **set** est équivalente à la fonction **setq** à ceci près qu'elle évalue les deux arguments qui lui sont passés. On peut l'employer à la place de **setq** à condition de "quoter" le symbole.

```
Commande: (set 'b a)
10

Commande: (set c "test")
"test"

Commande: !c
A

Commande: !a
"test"

Commande: !b
10
```

La fonction **set** est surtout utilisée associée à la fonction **read** (voir chapitre 14.1).

```
Commande: (set (read "d") 2.5)
2.5

Commande: !d
2.5
```

© Gilles Chanteau

# 3 Programmes

Pour éviter de saisir les expressions une à une, on préfèrera les regrouper dans un programme.
Un programme AutoLISP® est un fichier ASCII (fichier texte) avec l'extension .lsp. Il peut être écrit dans éditeur de texte tel que le bloc-notes. Il est préférable de ne pas utiliser de traitement de textes trop sophistiqué tels que Word® qui ajoute des caractères de mise en page. Il existe de nombreux éditeurs plus ou moins spécialisés dans l'édition de programmes, AutoCAD® intègre l'éditeur Visual LISP® (voir chapitre 4).

## 3.1 Chargement de fichiers (load appload)

Comme dit plus haut un programme LISP peut être directement collé sur la ligne de commande, la validation entraîne sont chargement dans le dessin.

### 3.1.1 Fonction load

La fonction LISP **load** permet de charger un fichier LISP (extension .lsp) dans le dessin courant.
Si le fichier est dans un répertoire du chemin de recherche des fichiers de support (menu Outils > Options… > onglet Fichiers), il n'est pas nécessaire d'entrer le chemin complet du fichier, le nom sans l'extension suffit :

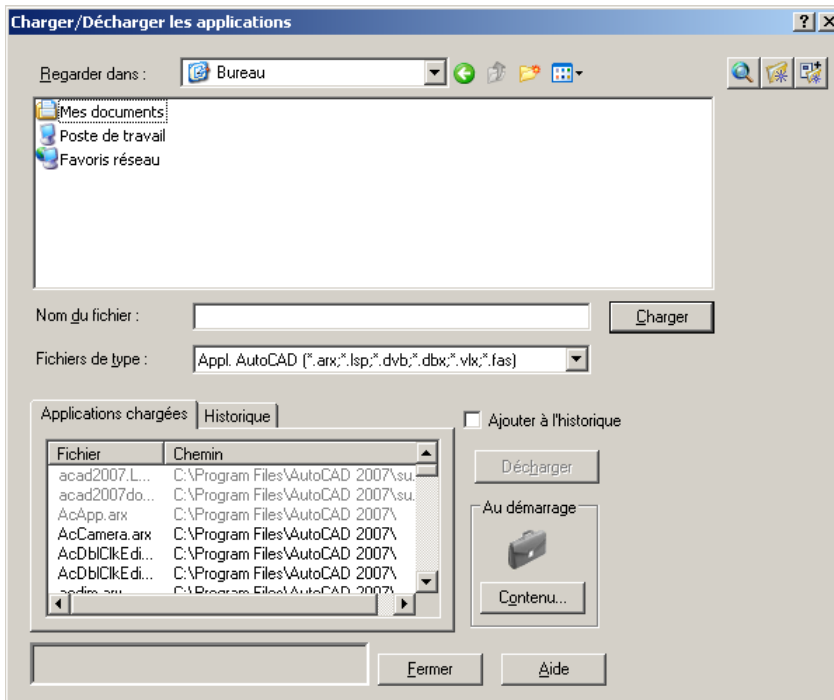
```
Commande: (load "toto")
C:TOTO
```

ou, avec le chemin complet :

```
Commande: (load "C:\\mes_lisps\\toto.lsp")
C:TOTO
```

### 3.1.2 Commande charger une application (APPLOAD)

La commande AutoCAD® pour charger un fichier LISP se nomme APPLOAD, accessible aussi par le menu *Outils* > *Charger une application…*
Si le fichier est dans un répertoire du chemin de recherche des fichiers de support (menu *Outils* > *Options…* > *onglet Fichiers*), il n'est pas nécessaire d'entrer le chemin complet du fichier, le nom sans l'extension suffit.
Pour que le fichier soit chargé automatiquement à chaque démarrage d'AutoCAD® ou ouverture d'un dessin, il suffit de l'ajouter à la liste dans la boite de dialogue Contenu… (valise).




### 3.1.3 Cliquer/déposer

Il est aussi possible de charger un fichier LISP dans le dessin courant en effectuant un « cliquer/déposer » depuis l'explorateur Windows ou le bureau dans la fenêtre d'AutoCAD®.

## 3.2 Chargement automatique (autoload S::STARTUP)

Trois fichiers définis par l'utilisateur : acad.lsp, acaddoc.lsp et le fichier MNL du même nom que le fichier de personnalisation courant (.CUI ou .MNS), sont chargés automatiquement par AutoCAD®.
Ces fichiers peuvent contenir des expressions LISP : routines et expressions de chargement de fichiers LISP. Ces expressions peuvent utiliser les fonctions **load** (voir ci-dessus) ou **autoload**.

Les fichiers acad.lsp et acaddoc.lsp ou MNL autre que acad.mnl n'existent pas par défaut, il appartient à l'utilisateur de les créer dans un répertoire du chemin de recherche.

Au démarrage d'une session, AutoCAD® cherche un fichier acad.lsp et l'exécute s'il le trouve.
Le fichier acaddoc.lsp, quant à lui, est exécuté à l'ouverture de chaque document.
Ce comportement est celui par défaut et peut être modifié avec la variable système ACADLSPASDOC.
Au chargement de chaque fichier de personnalisation (CUI ou MNS) AutoCAD® recherche un fichier MNL de même nom et l'exécute s'il le trouve. Le fichier MNL est exécuté après le fichier acaddoc.lsp. On place dans ce fichier les expression et routines LISP nécessaire à ce fichier de personnalisation.

Par ailleurs, AutoCAD® fournit deux fichiers : acad20XX.lsp et acad20XXdoc.lsp ayant le même comportement que les fichier décris ci-dessus. Ces fichiers contiennent des instructions LISP nécessaires au fonctionnement d'AutoCAD®. Il convient de ne pas modifier ces fichiers.

Pour de plus amples renseignements sur l'utilisation de ces fichiers, voir dans l'aide : *Guide de personnalisation > Introduction aux interfaces de programmation > AutoLISP® et Visual LISP® > Chargement et exécution automatique des routines AutoLISP®.*

### 3.2.1 Fonction autoload

La fonction **autoload** permet de charger un fichier LISP seulement quand un appel à une commande définie dans le fichier est exécuté.
Si les commandes cmd1 cmd2 sont définies dans toto.lsp, l'expression suivante dans un fichier de chargement automatique, entraînera le chargement de toto.lsp au premier appel de cmd1 ou cmd2.

```
(autoload "toto" '("cmd1" "cmd2"))
```

### 3.2.2 Fonction S::STARTUP

Les fichiers de démarrage acad.lsp, acaddoc.lsp et MNL sont chargés avant que le dessin ne soit complètement initialisé. Ils ne peuvent donc contenir d'expressions utilisant la fonction **command** (qui n'est opérationnelle qu'après initialisation.
Il est possible d'inclure dans un des fichiers de démarrage une fonction **S::STARTUP** qui ne sera exécutée qu'après l'initialisation du dessin.

Généralement la fonction **S::STARTUP** est définie avec la fonction **defun-q**. Ceci permet de l'amender facilement avec **append**.

```
(defun-q zext ()
  (command "_zoom" "_extent")
  (princ)
)

(setq S::STARTUP (append S::STARTUP zext))
```

Si aucune fonction **S::STARTUP** n'existe, elle est créée avec le contenu de **zext**.
S'il existait déjà une fonction **S::STARTUP**, le contenu de **zext** lui est ajouté.

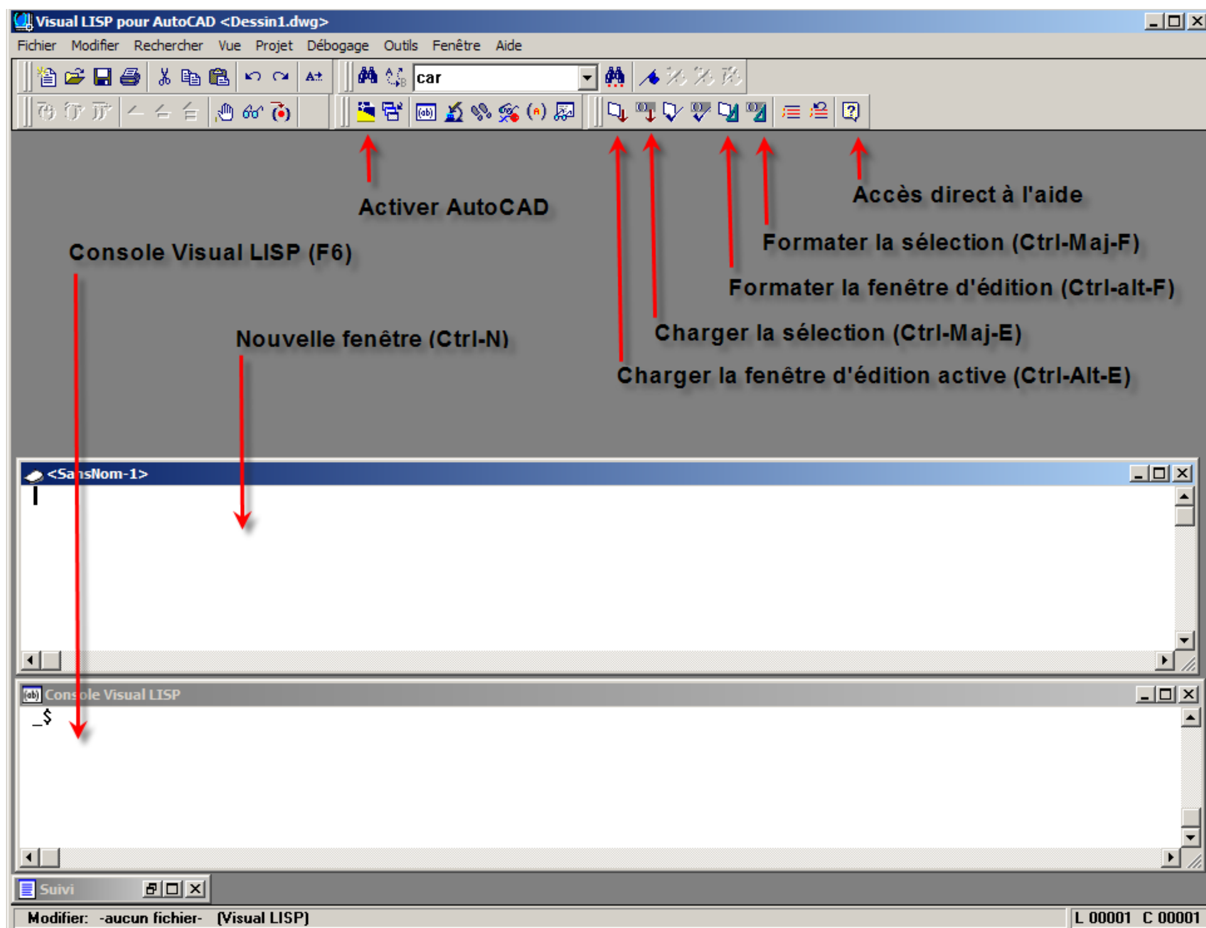
# *4 Éditeur Visual LISP®*

S'il est possible d'entrer des expressions LISP à la ligne de commande, et d'écrire le code de routines dans un éditeur de texte type bloc-notes, AutoCAD® fourni un éditeur de LISP, puissant outil de création et de modification de routines LISP.

## 4.1 Présentation

On accède à l'éditeur Visual LISP® depuis AutoCAD®, par le menu Outils > AutoLISP® > Editeur Visual LISP® ou par les commandes VLIDE ou VLISP.

On retrouve dans l'éditeur Visual LISP® les outils communs à tous les éditeurs de texte : Nouveau (Ctrl-N), Ouvrir (Ctrl-O), Enregistrer (Ctrl-S), Imprimer (Ctrl-P), Couper (Ctrl-X), Copier (Ctrl-C), Coller (Ctrl-V), Rechercher (Ctrl-F), Remplacer (Ctrl-H).
Il est aussi doté d'outils spécifiques pour faciliter la mise en forme, charger et évaluer des expressions ou des routines, ainsi que d'outils de débogage.




## 4.2 Les fenêtres

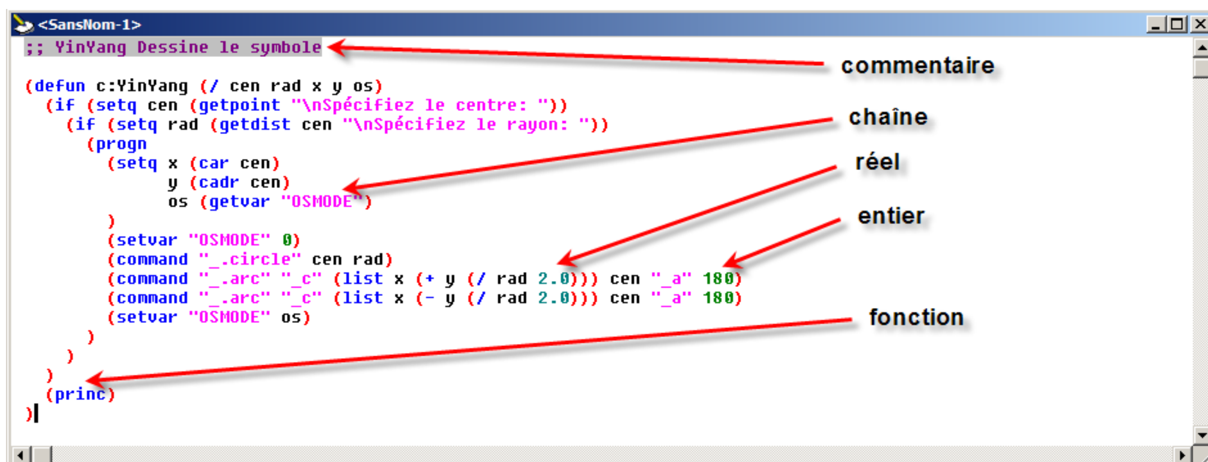
Comme dans n'importe quel éditeur, il est possible d'ouvrir des fichiers existants (Ctrl-O), ou des nouvelles fenêtres (Ctrl-N).
La console Visual LISP® (F6) est une fenêtre spéciale de l'éditeur dans laquelle on peut évaluer directement des expressions ou des symboles. C'est aussi dans la console que sont retournés les résultats des évaluations faites depuis les autres fenêtres avec Charger la sélection (Ctrl-Maj-E).

## 4.3 Mise en forme

Pour une meilleure lisibilité du code, les données apparaissent de différentes couleurs suivant leur type et le code est mis en forme.

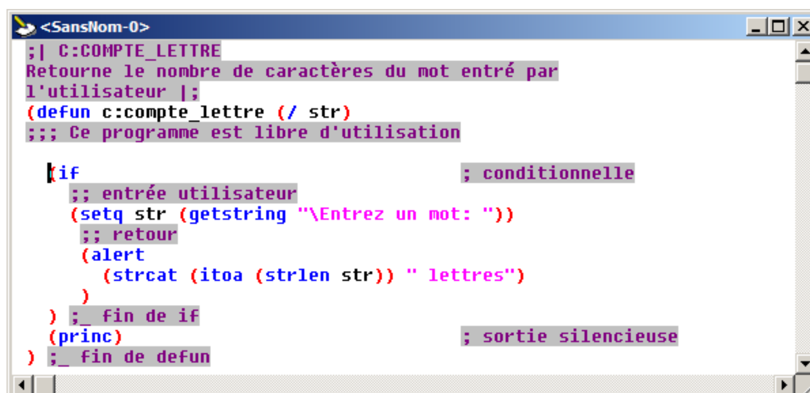
Les données apparaissent en :
- `rouge` pour les parenthèses
- `bleu` pour les fonctions LISP prédéfinies et les symboles protégés (pi, nil et T)
- `rose` pour les chaînes de caractère
- `vert` pour les nombres entiers
- `bleu-vert` pour les nombres réels
- `noir` pour tous les autres symboles
- `surlignés en gris` pour les commentaires

Le formatage du code (paramétrable) consiste principalement à donner un retrait à chaque ligne correspondant à l'imbrication de la ligne au sein des parenthèses (indentation). On peut formater toute la fenêtre (Ctrl-Alt-F) ou une sélection (Ctrl-Maj-F).




Sur une ligne tout ce qui est derrière un point-virgule n'est pas interprété, c'est un commentaire.
Les blocs de commentaire compris entre `;|` et `|;` peuvent avoir plusieurs lignes
Suivant le nombre de points-virgules devant la ligne de commentaire, le formatage du code dans l'éditeur Visual LISP® placera différemment la ligne de commentaire :

`;;;` le commentaire est placé complètement à gauche
`;;` le commentaire prend le retrait du code à sa position
`;` le commentaire se place en retrait à droite à 40 caractères (défaut)
`;_` commentaire de fin, se place à une espace de la dernière parenthèse




© Gilles Chanteau

## 4.4 Sélection par double clic

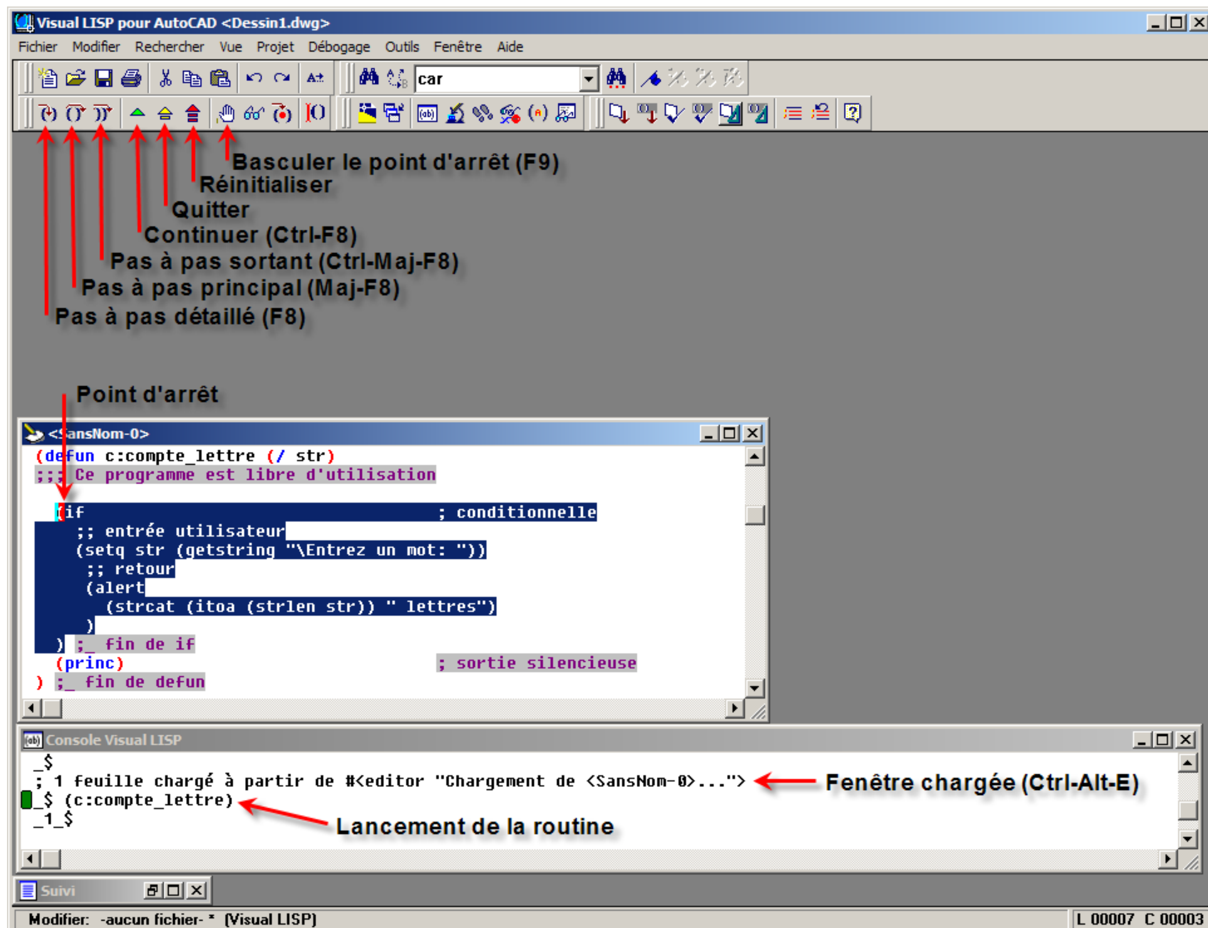
- un double clic au début ou à la fin d'un mot sélectionne le mot entier.
- un double clic avant un guillemet ouvrant ou après un guillemet fermant sélectionne toute la chaîne de caractère.
- un double clic après une parenthèse fermante ou avant une parenthèse ouvrante sélectionne toute l'expression depuis ou jusqu'à la parenthèse correspondante.

Un symbole (variable) ou une expression sélectionnée dans une fenêtre peut être évaluée directement (Ctrl-Maj-E), le résultat est retourné dans la console.

Si après avoir sélectionné le symbole d'une fonction prédéfinie (bleu), on clique sur l'icône de l'aide [?] (Ctrl-F1), l'aide s'ouvre directement à la page de cette fonction.

## 4.5 Aperçu des fonctions de débogage

L'éditeur Visual LISP® fournit de nombreuses fonctions facilitant le débogage.




Pour suivre le déroulement de l'interprétation des expressions pendant l'exécution d'une routine, il est possible de placer un, ou plusieurs, points d'arrêt dans le code.
Après chargement et lancement de la routine, l'interprétation s'interrompt au point d'arrêt et l'expression correspondante est sélectionnée On peut alors suivre l'interprétation des expressions à l'aide des outils de pas à pas ou continuer jusqu'au point d'arrêt suivant.
À tout moment, il est possible d'utiliser la console pour évaluer des variables ou des expressions.

Si le déroulement d'une routine génère une erreur, on peut déterminer l'expression en cause en activant Arrêt sur erreur (Menu Débogage), puis en utilisant l'outil : Source de la dernière interruption (Ctrl-F9). Une fois l'erreur localisée, on doit Réinitialiser (Ctrl-Q).

## 5 Commandes AutoCAD® (command pause vl-cmdf)

```
(command [arguments] ...)
```

La fonction **command** permet d'appeler depuis une expression LISP les commandes natives d'AutoCAD®. Les arguments pour la fonction **command** sont le nom de la commande suivi des options et entrées nécessaire à son exécution.
Pour les commandes ayant une version "ligne de commande" et une version "boite de dialogue", c'est la première qui est utilisée par défaut. Pour forcer l'ouverture de la boite de dialogue, il faut faire précéder l'appel à command par l'expression : `(initdia)`

**command** utilisé sans arguments `(command)` équivaut à un Échap.

Une chaîne vide (*blank*) `""` utilisée comme argument équivaut à un Enter.

**`pause`** utilisé comme argument interrompt l'exécution pour permettre une entrée par l'utilisateur.

Lorsque l'argument requis pour la fonction **command** est un nombre, il peut être spécifié indifféremment sous la forme d'un entier, d'un réel ou d'une chaîne de caractères.

```
(command "_circle" '(0 0 0) 10)
(command "_circle" '(0.0 0.0 0.0) 10.0)
(command "_circle" "0.0,0.0,0.0" "10.0")
```

Plusieurs commandes AutoCAD® peuvent être appelées successivement dans la même expression, exemple pour dessiner un point en 0,0 et un cercle de centre 0,0 et de rayon 10.0 :

```
(command "_point" '(0.0 0.0 0.0) "_circle" '(0.0 0.0 0.0) 10.0)
```

De même que l'appel à une commande peut être décomposé en plusieurs expressions, ceci sert, par exemple à faire passer une liste de points (pt_lst) comme argument :

```
(command "_pline")
(foreach p pt_lst (command p))
(command "")
```

Ou encore, pour laisser la main à l'utilisateur tant que la commande est active :

```
(command "_line")
(while (not (zerop (getvar "CMDACTIVE")))
  (command pause)
)
```

NOTA : Les fonctions foreach, while, not, zerop, getvar sont expliquées plus loin.

Il est important de noter que lors de l'exécution de la fonction **command** les accrochages aux objets permanents sont actifs. Il est donc prudent de les désactiver temporairement pour s'assurer que les entités seront créées précisément sur les points spécifiés. Une méthode consiste à utiliser l'option "auc" ou de préférence "_non" avant de spécifier le point pour forcer l'accrochage à "aucun".
Par ailleurs les paramètres requis par la fonction command sont les mêmes que ceux entrés à la ligne de commande : les points sont exprimés en coordonnées SCU courant, les angles dans le système d'unité courant etc.

```
(command "_line" "_non" '(0 0) "_non" '(20 10) "")
```

La fonction **vl-cmdf** s'utilise comme **command** mais évalue les arguments avant d'exécuter la commande. Si **command** retourne toujours **nil**, vl-**cmdf** retourne **T** si tous les arguments sont valides, **nil** sinon.

NOTA : Pour une meilleure compatibilité entre les différentes versions d'AutoCAD® est préférable d'utiliser, pour les noms de commande et leurs options, les noms en langage international. Ces noms sont précédés d'un tiret bas (*underscore*). La fonction AutoLISP® **getcname** traduit les noms de commande exprimés dans la langue de la version courante en langage international et vice-versa :

`(getcname "ligne")` retourne: `"_LINE"`
`(getcname "_move")` retourne : `"DEPLACER"`

# *6 Définitions de fonctions (defun defun-q)*

La fonction **defun** sert à définir une fonction.
Le premier argument est un symbole décrivant le nom de la fonction, le second une expression contenant les arguments requis par la  fonction et les variables locales (arguments et variables sont séparé par une barre oblique (/)).
Si la fonction ne requiert ni argument ni variable, l'expression doit quand même être présente : liste vide ().
Puis viennent les expressions dont l'évaluation retournera le résultat désiré.

```
(defun symbole ([arguments…] [/ variables...]) expression...)
```

## 6.1 (defun c:symbole …) vs (defun symbole …)

En faisant précéder  le nom de la fonction par **c:** la fonction est définie comme une commande AutoCAD® qu'il sera possible de lancer depuis la ligne de commande (ou d'un menu) en tapant directement le nom de la fonction. Dans l'autre cas la fonction ne peut être lancée que dans une expression AutoLISP®. C'est l'équivalent d'une nouvelle fonction LISP utilisable comme les fonctions prédéfinies.

```
(defun Fonction (arg1 arg2 / var1 var2) ...)
```

`arg1` et `arg2` sont les arguments (non optionnels) de `Fonction`, ils doivent être spécifiés conformément aux besoin de `Fonction`.

`(Fonction arg1)` -> ; erreur: nombre d'arguments insuffisants

`(Fonction arg1 arg2)` -> ; erreur: type d'argument incorrect: ... Si `arg1` ou `arg2` ne sont pas conformes au type d'arguments nécessaires à `Fonction`.

`var1` et `var2` sont des variables définies dans `Fonction` (avec **setq** par exemple).
Si elles sont déclarées, comme dans l'exemple, on les appelle des variables locales.
Les valeurs qui leur sont affectées dans `Fonction` ne le seront que pendant la durée de son exécution, elles reprendront ensuite la valeur qu'elles avaient avant l'appel à `Fonction` (nil ou autre).
Si au sein de `Fonction`, une valeur est attribuée à `var3` avec **setq**, et qu'elle n'a pas été déclarée, on l'appelle variable globale, elle conserve sa valeur dans le dessin pendant toute la session.
Il peut être intéressant de conserver une valeur dans une variable globale pour la récupérer lors d'une autre exécution de routine, dans ce cas, une convention consiste à écrire le nom de la variable entre astérisques, il est aussi prudent de choisir un nom qui a peu de chance d'être utilisé par une autre fonction : *Fonction1_var3*

Il existe une autre fonction AutoLISP® équivalente à **defun** : **defun-q**.
Cette fonction est conservée surtout pour assurer la compatibilité avec les versions antérieures.
Si la syntaxe est la même qu'avec la fonction **defun**, la différence est le type de donnée retournée.
**defun** crée des objet de type USUBR quand **defun-q** retourne une liste.
Ceci peut présenter un intérêt dans certains cas (fort peu nombreux, en fait). La structure de la fonction étant une liste, elle peut aisément être redéfinie "programmatiquement" avec les fonctions de manipulation des listes.

AutoLISP® offre aussi la possibilité de définir des fonctions anonymes. L'utilisation de **lambda** est expliquée au chapitre 13.2.

© Gilles Chanteau

# *7 Entrées utilisateur (getint getreal getstring getkword getangle getcorner getdist getorient getpoint)*

De nombreuses fonctions permettent de suspendre l'exécution du programme afin que l'utilisateur puisse entrer une donnée. Le nom de ces fonctions commence par get...
Toutes ces fonctions acceptent un argument (optionnel) : une chaîne qui correspond à l'invite qui sera affichée en ligne de commande.

- données numériques : **getint** et **getreal** retournent respectivement un entier et un réel

```
(getint [message]) (getreal [message])
```

Exemple : `(setq ech (getreal "\nEntrez le facteur d'échelle: "))`

- données alphabétiques : **getstring** et **getkword**

```
(getstring [esp] [message])
```

**getstring** accepte un autre argument (optionnel) : **esp**. Si celui-ci est spécifié et non **nil**, la chaîne entrée pourra contenir des espaces, sinon la touche espace sera considérée comme une validation de l'entée.
**getkword** est utilisée pour spécifier des "mots clés" (options), son utilisation est liée à celle de la fonction **initget** (voir ci-dessous)

- données géométriques : **getangle**, **getcorner**, **getdist**, **getorient**, **getpoint**

```
(getangle [pt] [message])
(getcorner [pt] [message])
(getdist [pt] [message])
(getorient [pt] [message])
(getpoint [pt] [message])
```

Toutes ces fonctions permettent indifféremment une entrée au clavier ou une saisie à l'aide du périphérique de pointage.
Elles acceptent un argument (optionnel) : **pt**, un point à partir duquel s'affiche une ligne élastique jusqu'au pointeur (avec **getcorner** la ligne élastique est remplacée par un rectangle).
**getpoint** et **getcorner** retournent un point.
**getangle**, **getdist** et **getorient** retournent un nombre réel. **getangle** et **getorient** acceptent les entrées au clavier dans l'unité angulaire courante mais retournent un angle exprimé en radians croissant en sens trigonométrique. **getorient** retourne l'angle "absolu" (0.0 à l'Est), **getangle** l'angle "relatif" qui tient compte d'un éventuel paramétrage de l'angle de base 0.0 (variable système ANGBASE).

## 7.1 Initialisation (initget)

Les fonctions **getXXX** contrôlent le type de donnée entré et affichent un message si le type n'est pas valide :

```
Commande: (setq dist (getdist "\nSpécifiez la distance: "))
Spécifiez la distance: 25cm
Nécessite une distance numérique ou deux points.
Spécifiez la distance: 25
25.0
```

La fonction **initget** permet d'initialiser les valeurs "autorisées" pour les fonctions **getXXX** (exceptée **getstring**) ainsi que pour les fonctions **entsel**, **nentsel** et **nenselp** (voir 16.1).

Un appel **initget** n'affecte que l'appel à la fonction **getXXX** ou ***entsel*** qui lui succède.
**initget** retourne toujours **nil**.

```
(initget [bits] [chaine])
```

Les arguments (optionnels) sont :
`bits` : la somme des codes binaires suivants (pour les principaux) :

1 : refuser uniquement Entrée
2 : refuser 0
4 : refuser les nombres négatifs
8 : Autoriser la spécification d'un point hors des limites (si LIMCECK = 1)

`chaine` : une chaîne contenant les mots clés autorisés séparés par des espaces.

Exemples :
```
(initget 7)
(setq dist (getdist "\nSpécifiez la distance: "))
```
Seuls un nombre strictement positif sera accepté.

Avec **getkword**
```
(initget 1 "Oui Non")
(setq del (getkword "\nEffacer l'objet source ? [Oui/Non]: "))
```
L'utilisateur ne peut entrer que **oui**, **non**, **o** ou **n** (les lettres en capitales dans l'argument **chaîne** sont des raccourcis)
NOTA : le fait de spécifier les mots clés sous la forme : [Oui/Non] dans l'invite de la fonction **getkword** les fait apparaître dans le menu contextuel.

## 7.2 Boites de dialogue (getfiled acad_colordlg acad_truecolordlg)

Pour certaines données, AutoLISP® fournit des fonctions qui utilisent les boites dialogue standard.

**getfiled** `(getfiled titre defaut ext drapeau)`
Retourne le nom du fichier sélectionné.
`titre` : titre de la boite de dialogue
`defaut` : nom du fichier par défaut affiché dans la boite ou ""
`ext` : extension(s) autorisée(s) pour le choix d'un fichier (s'il y a plusieurs extensions, les séparer par un point-virgule) ou une chaîne vide pour "tous"
`drapeau` : un entier qui est somme des codes binaires suivants :

1 Créer un nouveau fichier (ne pas utiliser pour sélectionner un fichier déjà existant)
4 Permet à l'utilisateur d'entrer une extension arbitraire (ou pas d'extension)
8 Si 1 n'est pas spécifié, ouvre directement sur le fichier par défaut si celui-ci est dans un répertoire du chemin de recherche
16 Si ce bit est spécifié ou si l'argument défaut se termine par un délimiteur de chemin (\\ ou /), La case "Nom du fichier" est vide et la recherche commence au chemin spécifié.
32 Si 1 est spécifié (création d'un nouveau fichier), n'affiche pas la boite de dialogue "Ce fichier existe déjà, voulez vous l'écraser ?"
64 Ne transfère pas les fichiers à distance si l'utilisateur spécifie une URL
128 N'accepte aucune URL

```
(getfiled "Créer un fichier" "C:\\" "txt" 1)
```

**acad_colordlg** et **acad_truecolordlg** pour les boites dialogue de choix de couleur respectivement de l'index et "couleurs vraies" (versions 2004 et postérieures).

**acad_colordlg** `(acad_colordlg index [drapeau])`
`index` : l'indice de la couleur proposée par défaut (de 0 à 256 inclus)
`drapeau` : si spécifié nil rend inaccessible les couleurs DuBloc et DuCalque

```
(acad_colordlg 1)
```

**acad_truecolordlg** fonctionne comme **acad_colordlg** en remplaçant l'index par une paire pointée (groupes DXF 62, 420 ou 430)

```
(acad_truecolordlg '(62 . 30) nil)
```

# *8 Listes et points*

Les listes étant des éléments fondamentaux dans le langage LISP, il existe de nombreuses fonctions pour les construire, accéder à leurs éléments, les manipuler.

## 8.1 Construction des listes (quote list cons)

Pour construire une liste dont on connaît, à priori, tous les éléments et l'attribuer à une variable, le plus simple est d'utiliser la fonction **quote** (ou son abréviation : ').
`(setq lst '(1 "chaîne" 12.57))` retourne : `(1 "chaîne" 12.57)`

Si la liste contient des expressions nécessitant une évaluation (appels de fonctions ou variables), la fonction **quote** empêchera ces évaluations, on utilise alors la fonction **list**.
`(setq a 12)`
`(setq lst '((+ 2 3) a))` retourne : `((+ 2 3) a)`
`(setq lst (list (+ 2 3) a))` retourne : `(5 12)`

Une autre fonction fondamentale du LISP est la fonction **cons** utilisée pour construire une liste. **cons** requiert 2 arguments : élément et une liste et retourne la liste avec le nouvel élément ajouté au début.
`(cons "a" '("b" "c"))` retourne `("a" "b" "c")`
**nil** étant considéré comme une liste vide,
`(cons 1 nil)` retourne : `(1)`

## 8.2 Accès aux éléments (car cdr last nth member vl-position)

Les premières fonctions LISP sont **car** et **cdr**. Elles sont présentes dans tous les langages LISP.
**car** retourne le premier élément d'une liste et **cdr** la liste privée du premier élément.
`(car '(6 1 8)` retourne : `6`
`(cdr '(6 1 8)` retourne : `(1 8)`
Ces fonctions peuvent se combiner :
`(car (cdr '(6 1 8))` s'écrit : `(cadr '(6 1 8))` et retourne : `1`, soit le premier élément de la liste privée du premier élément (soit le second élément de la liste).
`(car (cdr (cdr '(6 1 8))))` ou
`(car (cddr '(6 1 8)))` ou
`(cadr (cdr '(6 1 8)))` ou, plus simplement
`(caddr '(6 1 8))` retournent : `8`

Ces trois premières combinaisons sont très utilisées pour accéder aux coordonnées des points. En LISP un point est défini comme une liste de 2 ou 3 nombres (selon si c'est un point 2d ou 3d) : les coordonnées du point. **car** retourne la coordonnée X, **cadr** le Y et **caddr** le Z.
AutoLISP® permet ce type de combinaison jusqu'à 4 niveaux d'imbrication, ceci permet d'accéder aux éléments des listes imbriquées.
`(setq lst '((1 2 3) (4 5 6) (7 8 9)))`
`(caddar lst)` retourne : `3`
`(caadr lst)` retourne : `4`

La fonction **last** retourne le dernier élément d'une liste.

On peut aussi accéder aux éléments d'une liste avec la fonction **nth** qui retourne l'élément de la liste à l'indice spécifié. La fonction inverse : **vl-position** retourne l'indice de l'élément. L'indice du premier élément est toujours 0.
`(nth 3 '("a" "b" "c" "d" "e" "f"))` retourne : `"d"`
`(vl-position "e" '("a" "b" "c" "d" "e" "f"))` retourne : `4`

La fonction **member** requiert 2 arguments : une expression et une liste. Elle parcourt la liste en cherchant une occurrence de l'expression et retourne la liste à partir de cette première occurrence.
`(member 3 '(1 9 3 5 3 8))` retourne : `(3 5 3 8)`
`(member 7 '(1 9 3 5 3 8))` retourne : `nil`

## 8.3 Manipulation des listes (append reverse subst vl-remove)

**append** fusionne en une liste unique les listes qui lui sont passées comme argument
`(append '(1 2) '(3 4 5))` retourne : `(1 2 3 4 5)`

On inverse une liste avec la fonction **reverse**.
`(reverse '("a" "b" "c"))` retourne : `("c" "b" "a")`

La fonction **subst** sert à substituer toutes les occurrences d'un élément par un nouvel élément. Elle requiert 3 arguments : le nouvel élément, l'élément à substituer et la liste à traiter
`(subst "a" 2 '(1 2 3 2 1))` retourne : `(1 "a" 3 "a" 1)`

Pour supprimer toutes les occurrences d'un élément dans une liste on utilise la fonction **vl-remove**
`(vl-remove 2 '(1 2 3 2 1))` retourne : `(1 3 1)`

Exemples de routines simples

```
(setq lst '(0 1 2 3 4 5 6 7 8 9))
```

Une liste sans le dernier élément :

```
(defun butlast (lst)
  (reverse (cdr (reverse lst)))
)
```

`(butlast lst)` retourne : `(0 1 2 3 4 5 6 7 8)`

Faire passer le premier élément à la fin de la liste

```
(defun permute (lst)
  (append (cdr lst) (list (car lst)))
)
```

`(permute lst)` retourne : `(1 2 3 4 5 6 7 8 9 0)`

## 8.4 Listes d'association et paires pointées (assoc)

Une paire pointée est une liste particulière, elle ne contient que 2 éléments et ceux-ci sont séparés par un point : (1 . "test"), (40 . 25.4)…
On construit une paire pointée avec **quote** (') ou avec **cons**. Dans ce cas, le second argument est un atome et non une liste comme montré plus haut.
`(cons 70 3)` retourne : `(70 . 3)`

Les paires pointées servent à la constitution de listes d'association (ou liste associative). Il s'agit de liste dont les éléments sont des listes et/ou des paires pointées. Le premier élément de chaque sous liste servant de clé pour le localiser sans se soucier de sa position dans le liste.
La fonction **assoc** permet de retrouver un élément d'une liste d'association grâce à sa clé (si dans la liste plusieurs éléments ont la même clé, seul le premier est retourné).
`(assoc 40 '((1 . "test") (40 . 25.4) (70 . 3)))` retourne : `(40 . 25.4)`

Les paires pointées permettent un fonctionnement cohérent tant pour créer un élément dans liste d'association avec **cons** que pour récupérer une valeur d'élément avec **cdr** et **assoc** que cette valeur soit un atome ou une liste.

`(setq a_lst (cons 1 "test") (cons 10 '(20.0 30.0 0.0)))`
retourne : `((1 . "test") (10 20.0 30.0 0.0))`

`(cdr (assoc 1 a_lst))` retourne : `"test"`
`(cdr (assoc 10 a_lst))` retourne : `(20.0 30.0 0.0)`

# *9 Variables système (getvar setvar)*

Les fonctions **getvar** et **setvar** permettent, respectivement, de récupérer ou de modifier la valeur d'une variable système. Toutes deux requièrent comme argument le nom de la variable système (chaîne ou symbole quoté) **setvar** requiert en plus la nouvelle valeur à affecter.

Souvent on modifie la valeur d'une variable système pour les besoins de la bonne exécution d'un programme, Dans ce cas, le programme devra restaurer la valeur initiale de la (ou des) variable(s) système modifiée(s).
Pour ce faire, on affecte la valeur initiale de la variable système à une variable, puis on change la valeur et, à la fin du programme, on restaure la valeur initiale.

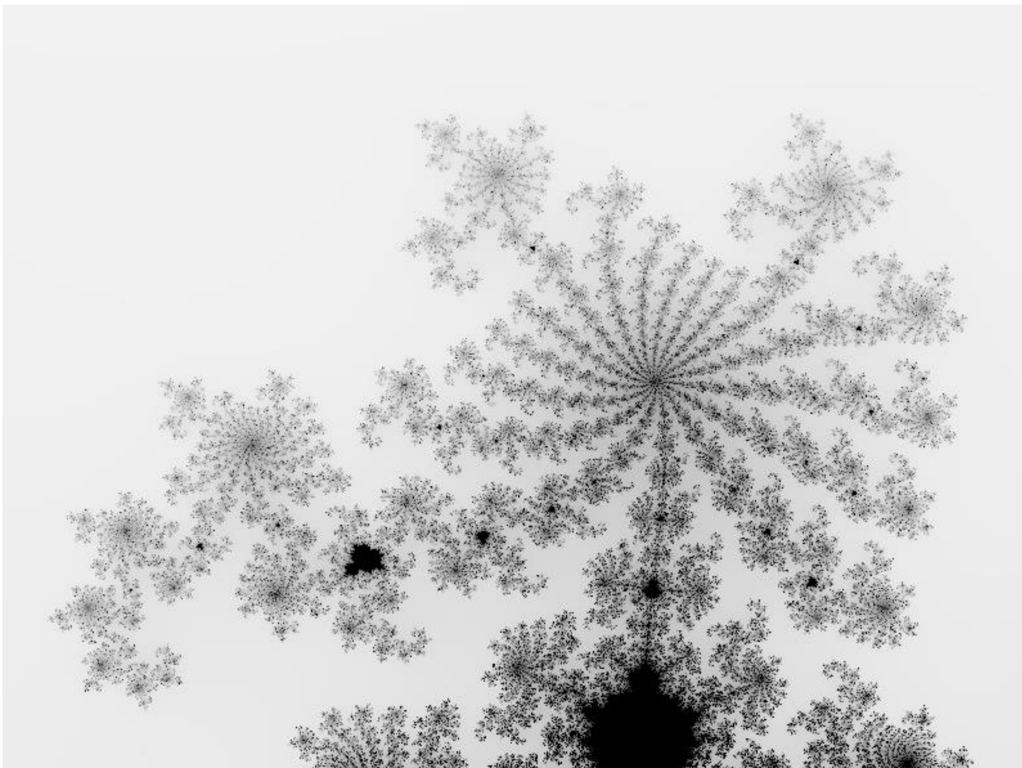
```
(setq osm (getvar "OSMODE"))
(setvar "OSMODE" 0)
(command "_line" pt1 pt2 "")
(setvar "OSMODE" osm)
```

## 9.1 Variables d'environnement (getenv setenv)

Les variables d'environnement ressemblent aux variables système. Elles permettent de stocker des informations en dehors de l'application. Les valeurs des variables d'environnement sont écrites sous forme de chaînes dans la base de registre. Comme les variables système, beaucoup servent à conserver des paramètres personnalisables.
L'utilisation des fonctions **getenv** et **setenv** est semblable à celle **getvar** et **setvar** à ceci près que les valeurs sont toujours de type STRING (chaîne) et qu'il est possible, avec **setenv** de créer ses propres variables d'environnement.

On retrouve les variables créées avec setenv dans la clé du registre suivante* :
HKEY_CURRENT_USER\Software\Autodesk\AutoCAD\R17.0\ACAD-5001:40C\FixedProfile\General

* la partie : R17\ACAD-5001:40C\ change selon les versions



## *10 Fonctions géométriques (distance angle polar inters sin cos atan trans)*

Il est logique qu'un langage dédié à un logiciel de DAO fournisse des fonctions géométriques.

**distance** `(distance pt1 pt2)`
Retourne la distance entre les deux points qui lui sont passés comme arguments.

**angle** `(angle pt1 pt2)`
Retourne l'angle entre la droite définie par les deux points qui lui sont passés comme arguments et l'axe X du SCU courant. L'angle est retourné en radians et est mesuré dans le sens trigonométrique. Si les points n'appartiennent pas au plan XY du SCU courant, ils sont projetés sur ce plan.

**polar** `(polar pt ang dist)`
Définit un point à partir de données polaires. Les arguments requis sont : le point d'origine, l'angle (exprimé en radians) et la distance.
`(polar '(10 10 10) (/ pi 4) 10)` retourne : **`(17.0711 17.0711 10.0)`**

**inters** `(inters pt1 pt2 pt3 pt4 [surseg])`
Retourne l'intersection des segments (ou des droites) pt1 pt2 et pt3 pt4. Si l'argument `surseg` est omis ou non **nil** , l'intersection doit se trouver sur les deux segments. Si aucune intersection n'est trouvée **inters** retourne **nil**.
`(setq a '(0 0) b '(1 1) c '(5 0) d '(5 8))`
`(inters a b c d)` ou `(inters a b c d T)` retournent : **`nil`**
`(inters a b c d nil)` retourne **`:(5.0 5.0)`**

**cos**, **sin** implémentent les fonctions trigonométriques cosinus et sinus. Elles requièrent un nombre comme argument exprimant un angle en radians.
**atan** `(atan nombre1 [nombre2])`
Retourne l'arc tangent (en radians) de `nombre1` s'il est l'unique argument ou de `nombre1` divisé par `nombre2` si `nombre2` est aussi exprimé.

**trans** `(trans pt depuis vers [depl])`
Traduit les coordonnées d'un point (ou d'un vecteur) d'un système de coordonnées vers un autre.
Les arguments `depuis` et `vers` peuvent être : un nombre entier (code), un nom d'entité (ENAME) ou un vecteur 3d (direction d'extrusion).

Si l'argument est un entier, il peut avoir les valeurs suivantes :
0 : Système de Coordonnées Général (SCG ou *WCS*)
1 : Système de Coordonnées Utilisateur courant (SCU ou *UCS*)
2 : Système de Coordonnées d'Affichage (SCA ou *DCS*) de la fenêtre courante de l'espace objet s'il est employé avec 0 ou 1. Employé avec 3, il exprime le SCA de l'espace objet de la fenêtre courante de l'espace papier.
3 : SCA de l'espace papier (utilisé uniquement avec 2).

Si un nom d'entité est utilisé comme argument, il doit être de type ENAME (tel que retourné par les fonctions **entlast**, **entnext**, **entsel**, **nentsel** ou **ssname**). Il exprime alors le Système de Coordonnées Objet (SCO ou *OCS*) de l'entité. Ce format ne sert que pour certaines entités crées ou insérées dans un Système de Coordonnées dont le plan XY n'est pas parallèle à celui du SCG.
Un vecteur 3d (direction d'extrusion) définit aussi un SCO.

L'argument `depl`, s'il est spécifié et non **nil** spécifie que `pt` doit être considéré comme un vecteur (déplacement).

Dans un SCU ayant subi une rotation de 90° sur l'axe Z (sens trigonométrique) :
`(trans '(1.0 2.0 3.0) 0 1)` retourne : **`(2.0 -1.0 3.0)`**
`(trans '(1.0 2.0 3.0) 1 0)` retourne : **`(-2.0 1.0 3.0)`**

## 11 Affichage de texte (prompt prin1 princ print alert textscr graphscr)

De nombreuses fonctions permettent d'afficher des messages.

**prompt** `(prompt msg)`
Affiche le message `msg` (chaîne) sur la ligne de commande et retourne toujours **nil**.

**prin1**, **princ** et **print** requièrent les mêmes arguments : une expression et, optionnellement, un descripteur de ficher. Les trois fonctions affichent sur la ligne de commande le résultat de l'évaluation de l'expression et retournent ce même résultat. Si le second argument est présent et représente un descripteur de fichier ouvert en écriture, ce résultat est écrit dans le fichier (voir chapitre 14).
Les différences entre ces trois fonctions concernent l'affichage dans les cas où l'argument est une chaîne :
- **princ** affiche la chaîne sans guillemets
- **prin1** affiche la chaîne avec guillemets
- **print** affiche la chaîne avec guillemets, précédée d'un saut de ligne et succédée d'une espace.

De plus ces fonctions n'ont pas le même comportement si la chaîne contient des caractères de contrôle* :
**prin1** et **print** affichent les caractères de contrôle, **princ** et **prompt** affichent leur évaluation.

```
Commande: (setq str "Ceci est un \"test\"")
"Ceci est un \"test\""

Commande: (prompt str)
Ceci est un "test"nil

Commande: (princ str)
Ceci est un "test""Ceci est un \"test\""

Commande: (prin1 str)
"Ceci est un \"test\"""Ceci est un \"test\""

Commande: (print str)

"Ceci est un \"test\"" "Ceci est un \"test\""
```

On remarque aussi que le texte affiché est immédiatement suivi du résultat de l'évaluation de l'expression. Comme dit plus haut, une expression LISP retourne toujours un résultat et ce résultat s'affiche sur la ligne de commande. Un programme LISP étant une expression (parfois très longue), il retournera sur la ligne de commande le résultat de la dernière expression. Pour éviter ceci et sortir "silencieusement" du programme, on utilise comme dernière expression un appel à **princ** ou **prin1** sans argument : (princ) ou (prin1).

La fonction **alert** permet d'afficher un message dans une petite boite de dialogue.

Les fonctions **textscr** et **graphscr** permettent, respectivement d'ouvrir et de fermer la fenêtre de texte d'AutoCAD

* Les caractères de contrôle sont soit des caractères non imprimables (saut de ligne, retour chariot, Echap…) soit des caractères qui pourraient rendre l'interprétation du code caduque : des guillemets à l'intérieur d'une chaîne. Pour pouvoir écrire ces caractère dans une chaîne on les fait précéder du "caractère d'échappement" : la barre oblique inversée ou *anti-slash* (\). Ce caractère devra donc, lui aussi, être précédé d'un caractère d'échappement.
Caractères de contrôle :
\\ : anti-slash
\" : guillemet
\n : saut de ligne
\r : retour chariot
\e : Echap
\nnn : caractère dont le code octal est nnn

## *12 Décisions conditionnelles (if cond progn)*

On ne fait pas de programmation "intelligente" sans fonctions conditionnelles permettant d'établir des structures décisionnelles.

La fonction conditionnelle de base en LISP est **if**.
`(if expr_test alors [sinon])`

En LISP, il n'est pas demandé à l'expression de test (`expr_test)` de retourner une valeur purement Booléenne : vrai ou faux (**T** ou **nil**) mais est considéré comme "vrai" tout ce qui est non **nil**, autrement dit, si l'expression test retourne une valeur quelconque autre que **nil** le test est considéré comme "vrai" et l'expression `alors` est évaluée. Si `expr_test` retourne **nil** et que l'expression `sinon` est présente, c'est elle qui sera évaluée.
Comme **if** n'accepte qu'une seule expression `alors` (et une seule `sinon)` et qu'il peut être nécessaire d'évaluer plusieurs expressions on les regroupe avec la fonction **progn**. Cette fonction évalue séquentiellement les expressions qu'elle contient et retourne le résultat de la dernière.

```
(initget 1 "Oui Non")
(setq kw (getkword "\nVoulez vous tout effacer ? [Oui/Non]: "))
(if (= "Oui" kw)
  (progn
    (command "_erase" "_all" "")
    (princ "\nTout a été effacé")
  )
  (princ "\nRien n'a été effacé")
)
```

AutoLISP® fournit une autre fonction de décision conditionnelle : **cond**
`(cond [(test resultat ...) ...])`
**cond** accepte un nombre indéterminé de listes comme arguments. Le premier élément de chaque liste est évalué tour à tour jusqu'à ce qu'un de ces éléments retourne un résultat non nil. Le reste des expressions contenues dans cette liste est alors évalué.
Il est courant d'utiliser T comme dernière expression test (défaut). Si tous les tests précédents ont échoués, les expressions de cette dernière liste seront évaluées.

```
(setq num (getreal "\nEntrez un nombre: "))
(cond
  ((= 0 (rem num 2)) (alert "Nombre entier pair"))
  ((= 1 (rem num 2)) (alert "Nombre entier impair"))
  (T (alert "Nombre réel"))
)
```

Les expressions de test peuvent utiliser nombre de fonctions AutoLISP® qui retournent un résultat de type Booléen (T ou nil). Ces fonctions peuvent être des prédicats de comparaison ou de type de données ou des opérateurs logiques.

## 12.1 Comparaisons (= /= < > eq equal zerop minusp wcmatch)

AutoLISP® fournit trois prédicat d'égalité : **=**, **eq** et **equal**.

**=** accepte un nombre indéfini d'arguments mais uniquement des nombres ou des chaînes.
`(= 2.0 2 2)` retourne : `T`
`(= 3 12)` retourne : `nil`
`(= "a" "a")` retourne : `T`
`(= "a" "A")` retourne : `nil`

© Gilles Chanteau

**eq** et **equal** n'acceptent que deux arguments à comparer qui peuvent être des expressions LISP de tous types.
Pour les nombres (ou listes de nombres), **equal** accepte un troisième argument (optionnel) : une tolérance dans la comparaison.
`(eq '(1.12345 12.0) '(1.12346 12.0))` retourne : `nil`
`(equal '(1.12345 12.0) '(1.12346 12.0))` retourne : `nil`
`(equal '(1.12345 12.0) '(1.12346 12.0) 0.00001)` retourne : `T`

Petite subtilité dans la comparaison de listes affectées à des variables: **eq** évalue si le même objet est affecté aux variables, **equal** compare le contenu des variables.
`(setq l1 '(1 2 3) l2 '(1 2 3) l3 l2)`
`(eq l1 l2)` retourne : `nil`
`(eq l1 '(1 2 3))` retourne : `nil`
`(eq l2 l3)` retourne : `T`
`(equal l1 l2)` retourne : `T`

Comme **=**, les opérateurs d'inégalité :
**/=** : différent de
**<** : inférieur à
**<=** : inférieur ou égal à
**>** : supérieur à
**>=** : supérieur ou égal à
acceptent un nombre indéterminé de nombres ou de chaînes comme arguments.

**zerop** évalue si le nombre qui lui est passé comme argument est égal à 0.

**minusp** évalue si le nombre qui lui est passé comme argument est strictement inférieur à 0.

**wcmatch** compare une chaîne à un modèle contenant des caractères génériques.
Les caractères génériques utilisables sont :
**#** (dièse) : caractère numérique seul
**@** (arobase) : caractère alphabétique seul
**.** (point) : caractère non alphanumérique seul
***** (astérisque) : séquence de tout type de caractères
**?** (point d'interrogation) : tout type de caractère seul
**~** (tilde) : si placé au début du modèle, tout caractère excepté le modèle
**[…]** : chacun des caractères inclus dans les crochets
**[~…]** : tout caractère seul non inclus dans les crochets
**-** (tiret) : utilisé pour spécifier une série à l'intérieur des crochets
**,** (virgule) ; sépare deux modèles
**`** (apostrophe inversé) : lire littéralement le caractère spécial suivant

Exemples
Est-ce que la chaîne commence par un L ?
`(wcmatch "LISP" "L*")` retourne : `T`
Est-ce que la chaîne contient un caractère numérique ?
`(wcmatch "LISP" "*#*")` retourne `nil`

## 12.2 Opérateurs logiques (and or not)

**and** et **or** acceptent un nombre indéterminé d'expressions comme argument et retournent le AND ou OR logique pour ces expressions.
**and** évalue tour à tour les expressions qui lui sont passées. Si l'une d'elle retourne **nil**, l'évaluation s'arrête et **nil** est retourné, sinon, **T** est retourné.
**or** évalue tour à tour les expressions qui lui sont passées. Si l'une d'elle retourne un résultat non **nil** l'évaluation s'arrête et **T** est retourné, sinon, **nil** est retourné.

**not** retourne **T** si l'expression qui lui est passée comme argument retourne **nil**, sinon **nil**.

© Gilles Chanteau

## 12.3 Types de données (atom listp null vl-consp numberp)

D'autres fonctions prédicat permettent de vérifier le type de donnée.

**atom** retourne T si l'expression est un atome, nil sinon.

**listp** retourne T si l'expression est une liste, nil sinon.

**null** fonctionne comme **not** mais est généralement utilisé pour vérifier qu'une liste est vide.

Mais, comme vu plus haut :
`(atom nil)` retourne **T**
`(listp nil)` retourne **T** aussi puisque **nil** symbolise à la fois une liste vide et une variable non affectée. La fonction **vl-consp** permet de vérifier que l'expression est une liste non vide.

```
(setq a '(1 2 3) b "test" c nil)
```

`(atom a)` retourne : **nil**
`(atom b)` retourne : **T**
`(atom c)` retourne : **T**

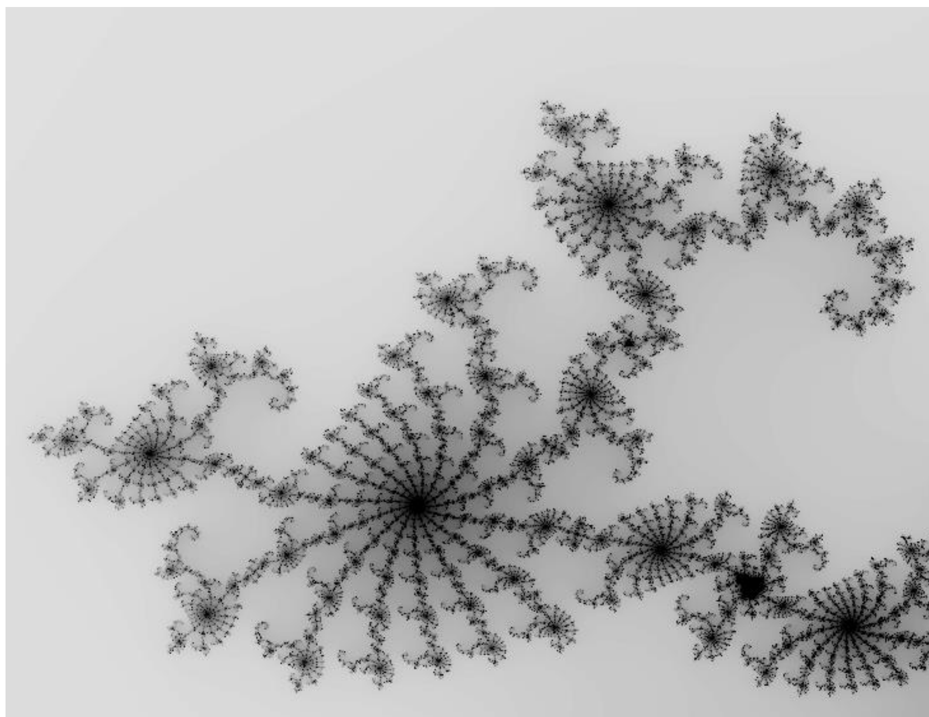
`(listp a)` retourne : **T**
`(listp b)` retourne : **nil**
`(listp c)` retourne : **T**

`(vl-consp a)` retourne : **T**
`(vl-consp b)` retourne : **nil**
`(vl-consp c)` retourne : **nil**

**numberp** vérifie si l'expression est un nombre (entier ou réel).

Comme il n'existe pas de fonction prédicat pour tous les types de données, on peut utiliser la fonction **type** avec une fonction de comparaison.

`(numberp "12")` retourne : **nil**
`(= (type "12") 'STR)` retourne **T**



# 13 Procédures itératives et récursives

Un des buts de la programmation étant de simplifier les taches répétitives, tous les langages de programmation fournissent des fonctions permettant de répéter des instructions en boucle.

## 13.1 Boucles et incrémentation (repeat while 1+ 1-)

On appelle boucle ou itération le fait de répéter un processus (évaluation d'une ou plusieurs expressions) Lorsque le nombre de fois que le processus doit se répéter est déterminé par avance, on peut utiliser la fonction **repeat** :

```
(repeat int [expression …])
```

Cette fonction évalue chaque expression le nombre de fois spécifié et retourne le résultat de la dernière évaluation.

Souvent, le nombre d'itérations dépend d'une condition et ne peut être déterminé, on utilise alors la fonction **while** :

```
(while expression_test [expression …])
```

Chaque expression est évaluée tant que l'expression test ne retourne pas nil. Il est donc impératif qu'à un moment ou à un autre cette expression retourne **nil**, sinon on entre dans une boucle sans fin*. C'est à l'intérieur même du processus itéré que des changements de données doivent permettre la condition d'arrêt (que l'expression test retourne **nil**).

```
(while (setq pt (getpoint "\nSpécifiez le centre du cercle: "))
  (command "_circle" "_non" pt 10.0)
)
```

Tant que l'utilisateur spécifie un point (pt est non nil), un cercle de centre pt et de rayon 10.0 est dessiné. Si l'utilisateur fait Entrée, Espace ou clic droit pt sera nil et boucle s'arrêtera.

Les processus itératifs nécessitent souvent l'incrémentation d'un nombre entier (un indice par exemple). Deux opérateurs AutoLISP® sont prévus pour incrémenter ou décrémenter une valeur entière : **1+** et **1-**.

```
(repeat (setq n 10)
  (setq n (1- n) lst (cons n lst))
)
```

Retourne :

```
(0 1 2 3 4 5 6 7 8 9)
```

*Comme il arrive parfois, qu'en phase de débogage on entre quand même dans une boucle sans fin, il faut pouvoir en sortir. Depuis la fenêtre AutoCAD, il suffit de faire Echap. Depuis l'éditeur Visual LISP, menu "Débogage" > "Abandonner l'interprétation " et répondre "Oui" à la boite de dialogue "Abandonner l'interprétation en cours ?".

## 13.2 Traitements de listes (foreach mapcar lambda apply vl-every vl-some vl-member-if(-not) vl-remove-if(-not) vl-sort)

AutoLISP® fournit bien sûr des fonctions pour traiter tous les éléments d'une liste. À part **foreach**, il ne s'agit pas à proprement parler de fonctions itératives.

**foreach** évalue la (ou les) expression(s) pour chaque élément de la liste.

```
(foreach nom liste [expression …])
```

`nom` est la variable à laquelle seront affectés tour à tour chaque élément de `liste`. **foreach** retourne le résultat de la dernière évaluation.

```
(foreach s '("Ceci" "est" "un" "test")
  (princ s)
  (princ " ")
)
```

écrit : `Ceci est un test` sur la ligne commande et retourne : `" "`.

Une des fonctions LISP les plus puissantes est la fonction **mapcar**.
`(mapcar 'fonction liste1 … liste n)`
**mapcar** requiert comme arguments une fonction et une ou plusieurs listes.
Elle retourne la liste qui est le résultat de l'application de la fonction à chacun des éléments de la (ou des) liste(s).

`(mapcar '1+ '(10 20 30))` retourne : `(11 21 31)` comme `(list (1+ 10) (1+ 20) (1+ 30))`

`(mapcar '+ '(1 2 3) '(4 5 6)` retourne : `(5 7 9)` comme `(list (+ 1 4) (+ 2 5) (+ 3 6))`

La fonction requise comme argument peut être une fonction prédéfinie (comme ci-dessus), une fonction définie avec **defun** ou une fonction anonyme (**lambda**).
Si, par exemple, on veut convertir une liste de nombres en les multipliant par 10, il n'existe pas de fonction prédéfinie qui le fasse. On peut donc définir cette fonction avec **defun** et l'utiliser ensuite avec **mapcar**.

```
(defun x10 (x) (* x 10))
```

`(mapcar 'x10 '(10 20 30))` retourne `(100 200 300)`

La fonction x10 ne présente que peu d'intérêt en dehors de cette utilisation. Quand la création d'une fonction avec **defun** ne se justifie pas, on peut définir une fonction anonyme avec **lambda**.
Une fonction **lambda** est semblable à une fonction définie avec **defun** à ceci près qu'elle n'a pas de nom (et donc, ne peux pas être appelée), qu'elle est exécutée et retourne son résultat à l'endroit même de sa définition.

`(mapcar '(lambda (x) (* x 10)) '(10 20 30)` retourne : `(100 200 300)`

On peut noter que la fonction lambda ci-dessus, est construite comme la fonction x10.

Plusieurs autres fonctions de traitement des listes requièrent une fonction comme argument.
Toutes peuvent utiliser indifféremment le nom d'une fonction prédéfinie ou définie avec **defun** ou encore une fonction **lambda**.

**apply** passe une liste d'arguments à une fonction et retourne le résultat de l'évaluation
`(apply 'fonction liste)`

`(apply '+ '(1 2 3))` retourne : `6` (l'expression est équivalente à `(+ 1 2 3)`

`(apply '< '(1 3 2))` retourne : `nil` (la liste n'es pas triée en ordre croissant)

**vl-every** et **vl-some** sont des fonctions prédicat (elles retournent T ou nil) qui requièrent comme arguments une fonction prédicat et une liste.

`(vl-every 'numberp '(1 25.4 256))`
retourne : `T` (tous les éléments de la liste sont des nombres).

`(vl-some '(lambda (x) (= (type x) 'REAL)) '(1 25.4 256))`
retourne : `T` (au moins un élément de la liste est un nombre réel).

**vl-sort** retourne une liste triée en fonction d'une fonction de comparaison.

`(vl-sort '(9 3 4 8 1 2) '<)`
retourne : `(1 2 3 4 8 9)` la liste triée en ordre croissant.

```
(vl-sort '((20 12) (13 85) (7 25))
        '(lambda (x1 x2) (> (car x1) (car x2)))
)
```
retourne : `((20 12) (13 85) (7 25))` la liste triée suivant ordre décroissant des premiers éléments.

**vl-member-if** (ou **vl-member-if-not)** fonctionne un peu comme **member** mais au lieu de comparer chaque élément de la liste avec une expression, elle évalue si chaque élément retourne T (ou nil) lorsqu'il est passé comme argument à la fonction prédicat.

```
(vl-member-if
  '(lambda (x) (= (car x) 5))
  '((1 . "test") (40 . 25.4) (5 . "1D9") (70 . 4))
)
```

retourne : `((5 . "1D9") (70 . 4))`

**vl-remove-if** (ou **vl-remove-if-not**) retourne la liste après avoir supprimé les éléments qui retourne T (ou nil) quand il sont passé à la fonction prédicat.

```
(vl-remove-if-not '(lambda (x) (= (type x) 'INT)) '(1 25.4 256))
```

retourne : `(1 256)`

## 13.3 Fonctions récursives

Une fonction récursive est une fonction qui fait appel à elle-même dans sa propre définition.
Ce type de procédure permet souvent d'écrire un code plus concis et de résoudre simplement certains problèmes complexes (imbrications multiples dans une arborescence, par exemple). Ceci nécessite forcément la définition d'une fonction dans laquelle on trouvera un (ou plusieurs) appel(s) à cette même fonction.
L'exemple classique pour illustrer la récursivité est la fonction **fact** qui retourne la factorielle d'un nombre :

```
(defun fact (n)
  (if (zerop n)
    1
    (* n (fact (1- n)))
  )
)
```

Une fonction récursive doit toujours contenir au moins une **condition d'arrêt**, ici : `(zerop n)` qui permet de sortir des **appels récursifs** : `(* n (fact (1- n)))`.
À chaque appel récursif, au moins un des arguments passés à la fonction est modifié de telle sorte qu'à un moment la condition d'arrêt soit remplie. Ici, n est décrémenté de 1.
L'évaluation d'une fonction récursive se fait en deux phases appelées "empilement" (jusqu'à ce que la condition d'arrêt soit atteinte et "dépilement" (jusqu'au résultat final).

Les procédures récursives peuvent aussi être utilisées dans le traitement des listes (**mapcar**, par exemple est définie récursivement).
La fonction **remove_doubles** ci-dessous supprime tous les doublons d'une liste.

```
(defun remove_doubles (lst)
  (if lst
    (cons (car lst) (remove_doubles (vl-remove (car lst) lst)))
  )
)
```

La condition d'arrêt est ici implicite : si **lst** est non **nil** un appel récursif survient, sinon la fonction retourne **nil**.
La condition d'arrêt est donc : **lst** égal **nil**.
L'appel récursif est :

```
(cons (car lst) (remove_doubles (vl-remove (car lst) lst)))
```

Soit : ajouter le premier élément de **lst** à la liste retournée par le résultat de l'évaluation de la fonction **remove_doubles** avec comme argument la liste **lst** de laquelle on aura supprimé toutes les occurrences du premier élément.

Si on décompose l'évaluation de l'expression (remove_doubles '(1 2 1 3 3)), on fera apparaître les phases d'empilement et de dépilement.

Empilement :

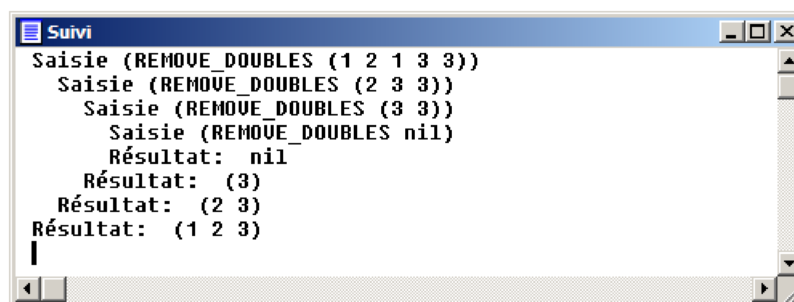
```
(remove_doubles '(1 2 1 3 3))
(cons 1 (remove_doubles '(2 3 3)))
(cons 1 (cons 2 (remove_doubles '(3 3))))
(cons 1 (cons 2 (cons 3 (remove_doubles nil))))
```

Dépilement :

```
(cons 1 (cons 2 (cons 3 nil)))
(cons 1 (cons 2 '(3)))
(cons 1 '(2 3))
(1 2 3)
```

La fonction LISP **trace** permet de suivre ce processus dans la fenêtre de texte d'AutoCAD® et dans la fenêtre de Suivi de l'éditeur Visual LISP.
Trace accepte plusieurs fonctions comme argument.

```
(trace remove_doubles)
```




Utiliser `(untrace remove_doubles)` pour désactiver le suivi à chaque appel de la fonction.

Exemples de fonctions récursives

```
;;; TRUNC
;;; Retourne la liste tronquée à partir de la première occurrence
;;; de l'expression (liste complémentaire de celle retournée par MEMBER)
(defun trunc (expr lst)
  (if (and lst
           (not (equal (car lst) expr))
      )
    (cons (car lst) (trunc expr (cdr lst)))
  )
)
```

`(trunc 5 '(1 2 3 4 5 6 7 8 9))` retourne : `(1 2 3 4)`

```
;;; BITSLIST
;;; Retourne la liste des codes binaires dont un nombre entier est la somme
(defun BitsList (n / b)
  (if (/= 0 n)
    (cons (setq b (expt 2 (fix (/ (log n) (log 2)))))
          (BitsList (- n b))
    )
  )
)
```

`(bitslist 27)` retourne : `(16 8 2 1)`

# *14 Chaînes de caractères et fichiers ASCII*

AutoLISP® permet de manipuler les chaînes, et de lire et écrire des fichiers texte.

## 14.1 Chaînes de caractères (strcat strlen strcase substr vl-string* read)

**strcat** `(strcat [str [str] …])`
retourne une chaîne qui est la concaténation de plusieurs chaînes.

**strlen** (strlen str)
retourne le nombre de caractères de la chaîne.

**strcase** `(strcase str [bas])`
retourne une chaîne dont tous les caractères sont passée en majuscules (si `bas` est absent ou nil) ou en minuscules (si `bas` est présent et non **nil**).

**substr** `(substr str start [length])`
Retourne la portion de la chaîne débutant à `start` (premier caractère = 1) et contenant `length` caractères. Si `length` est absent, la chaîne est retournée jusqu'au bout.

De nombreuses fonctions commençant par vl-string ont été ajoutées aux fonctions AutoLISP® originelles, en voici quelques exemples.
**vl-string-position**, **vl-string-search** retourne la position d'un caractère ou d'un modèle dans la chaîne.
**vl-string-left-trim**, **vl-string- right-trim**, **vl-string-trim** suppriment les caractères spécifiés respectivement au début de la chaîne, à la fin de la chaîne ou les deux.
**vl-string-subst** substitue une chaîne par une autre à l'intérieur d'une chaîne.
**vl-string-translate** remplace les caractères dans une chaîne par ceux spécifiés.
**vl-princ-to-string** et **vl-prin1-to-string** retournent la représentation sous forme de chaîne des données LISP telles qu'affichées par les fonctions **princ** et **prin1**.

Une autre fonction fondamentale des langages LISP est la fonction **read**. Cette fonction retourne le premier atome ou la première liste contenue dans la chaîne qui lui est passée comme argument.
`(read "test")` retourne : `TEST`
`(read "(1 2 3) (4 5 6)")` retourne : `(1 2 3)`
`(read "25.8600")` retourne : `25.86`

## 14.2 Conversions (itoa atoi rtos atof distof angtos angtof ascii chr vl-string->list vl-list->string float fix cvunit)

Il est souvent nécessaire de convertir des données numériques en données alphabétiques ou inversement.
**itoa** convertit un nombre entier en son équivalent en chaîne **atoi** est la fonction inverse.

**rtos** `(rtos nombre [mode [prec]])`
Convertit un nombre en son équivalent en chaîne. Cette fonction, en plus du nombre, accepte deux arguments optionnels : `mode`, le format du nombre (même valeurs que pour la variable système LUNITS) et `prec`, le nombre de décimales (voir LUPREC). Si ces arguments ne sont pas spécifiés rtos utilise les valeurs courantes des variables. La fonction inverse est **atof**.
Une autre fonction : **distof** sert aussi à la conversion des chaînes en nombres réels, l'argument `nombre` peut avoir tous les formats de chaînes acceptés par AutoCAD® pour les entrées au clavier (dont les fractions) et on peut spécifier un format de sortie (valeurs de LUNITS).

**angtos** `(angtos angle [unit [prec]])`
Convertit un nombre exprimant une valeur d'angle en radians en une chaîne exprimant cette valeur dans l'unité angulaire courante ou dans celle spécifié par l'argument `unit` (mêmes valeurs que la variable système AUNITS) le nombre de décimales, s'il n'est pas spécifié est celui de la variable AUPREC. La fonction inverse **angtof** accepte aussi l'argument `unit`.

La fonction **ascii** retourne le code ASCII (nombre entier) du caractère qui lui est passé comme argument. La fonction inverse est **chr**.
La fonction **vl-string->list** retourne la liste des codes ascii de tous les caractères d'une chaîne. La fonction inverse est : **vl-list->string**.

D'autres fonctions de conversion ne concernant pas les chaînes de caractères existent aussi.
**float** convertit un nombre entier en nombre réel, **fix** retourne la partie entière d'un nombre réel.

**cvunit** permet de convertir des valeurs d'un système d'unité à un autre. Les arguments requis pour les systèmes d'unités sont réunis dans le fichier personnalisable acad.unt (dossier Support courant).

Deux petites routines utiles :

```
;; str2lst © Gilles Chanteau
;; Transforme un chaîne avec séparateur en liste de chaînes
;;
;; Arguments
;; str : la chaîne à transformer en liste
;; sep : le séparateur

(defun str2lst (str sep / pos)
  (if (setq pos (vl-string-search sep str))
    (cons (substr str 1 pos)
          (str2lst (substr str (+ (strlen sep) pos 1)) sep)
    )
    (list str)
  )
)
```

`(str2lst "a b c" " ")` retourne : `("a" "b" "c")`
`(str2lst "1,2,3" ",")` retourne : `("1" "2" "3")`
`(mapcar 'read (str2lst "1,2,3" ","))` retourne : `(1 2 3)`

```
;; lst2str © Gilles Chanteau
;; Concatène une liste et un séparateur en une chaîne
;;
;; Arguments
;; lst : la liste à transformer en chaîne
;; sep : le séparateur

(defun lst2str (lst sep)
  (if (cdr lst)
    (strcat (vl-princ-to-string (car lst))
            sep
            (lst2str (cdr lst) sep)
    )
    (vl-princ-to-string (car lst))
  )
)
```

`(lst2str '(1 2 3) ",")` retourne : `"1,2,3"`
`(lst2str '("a" "b" "c") " ")` retourne : `"a b c"`

## 14.3 Fichiers ASCII (findfile open close read-char read-line write-char write-line)

AutoLISP® offre la possibilité de lire et écrire des fichiers texte ou fichiers ascii (.txt, .csv, .lsp…)
Avant d'essayer d'ouvrir un fichier, on peut s'assurer qu'il existe sur le poste. La fonction **findfile** retourne le chemin complet d'un fichier si le fichier est trouvé, **nil** sinon.
L'argument requis est le nom du fichier à ouvrir, si le chemin complet n'est pas spécifié, la recherche se limite aux répertoires du chemin de recherche, s'il est spécifié, la recherche est limitée à ce chemin.

```
(findfile "3darray.lsp")
```

Retourne : `"C:\\Program Files\\AutoCAD® 2007\\support\\3darray.lsp"`

Pour ouvrir le fichier, on utilise la fonction **open** qui requiert, outre le chemin complet du fichier, un argument mode d'ouverture. Cet argument peut avoir les valeurs suivantes :
"r" : (read) lecture
"w" : (write) écriture (écrase l'éventuel texte existant)
"a" : (append) écriture (ajoute à la suite du texte existant)
**open** retourne un pointeur vers le fichier, s'il n'existe pas, il est créé. Il est impératif de refermer le fichier après utilisation avec la fonction **close**.

Une fois le fichier ouvert, on  peut le lire avec la fonction **read-line**. Cette fonction retourne successivement chaque ligne du fichier jusqu'à ce qu'il n'y en ait plus. On l'utilise généralement dans une boucle avec **while**.
La fonction **write-line** permet d'écrire une ligne dans un fichier ouvert en écriture.

Les fonctions **write-char** et **read-char** fonctionnent comme **write-line** et **read-line** mais au lieu d'écrire ou de retourner une chaîne par ligne, elles écrivent ou retournent un code ascii par caractère.

L'exemple suivant définit deux commandes :
- **GetpointToFile** qui demande à l'utilisateur de spécifier des points et les écrit dans le fichier "C:\Points.txt" sous le format : "x,y,z" (cette commande appelle la routine **lst2str** ci-dessus).
- **DrawFromFile** qui lit le fichier "C:\Points.txt et dessine chaque point.

```
;; GetPointToFile
;; Ecrit les points saisis par l'utilisateur dans le fichier C:\Points.txt
;; NOTA : utilise la routine lst2str
(defun c:GetPointToFile (/ file pt lst)
  (while (setq pt (getpoint "\nSpécifiez un point: "))
    (setq lst (cons pt lst))
  )
  (setq file (open "C:\\Points.txt" "w"))
  (foreach p (reverse lst)
    (write-line (lst2str p ",") file)
  )
  (close file)
  (princ)
)
;; DrawFromFile
;; Dessine les points à partir du fichier Points.txt
(defun c:DrawFromFile (/ file line)
  (if (findfile "C:\\Points.txt")
    (progn
      (setq file (open "C:\\Points.txt" "r"))
      (while (setq line (read-line file))
        (command "_point" "_non" line)
      )
      (close file)
    )
    (princ "\nLe fichier \"C:\\Points.txt\" est introuvable")
  )
  (princ)
)
```

## 15 Gestion des erreurs (*error* vl-catch-all-apply …)

Durant l'exécution d'un programme LISP, des erreurs peuvent survenir du fait d'un manque de contrôle sur le type de certaines valeurs, par exemple, ou plus simplement, si l'utilisateur quitte le programme prématurément avec la touche Echap.
Suivant le moment où survient l'erreur dans l'exécution du programme, certaines variables système ont pu être modifiées, un ficher peut être ouvert, etc…
Pour pouvoir restaurer l'environnement initial avant le lancement du programme, AutoLISP® fournit une fonction : ***error*** qui peut être redéfinie. Il est impératif de limiter la portée de la redéfinition de ***error*** à la fonction à laquelle elle s'applique afin que celle-ci retrouve aussi sa définition initiale si elle a été lancée.
***error*** requiert un argument : le message affiché en cas d'erreur. La fonction ***error*** redéfinie peut décider de l'affichage ou non de ce message.

Par exemple, dans la commande GetpointToFile (ci-dessus), l'auteur a pris la précaution de faire une première boucle pour récupérer les points et les stocker dans une liste, avant d'ouvrir le fichier, traiter la liste et refermer le fichier. Si l'utilisateur quittait la commande pendant la saisie des points cela n'aurait pas d'incidence puisque le fichier ne serait pas encore ouvert.
Mais on pourrait préférer tout faire dans la même boucle et écrire les points au fur et à mesure qu'ils sont saisis. Il faudrait alors pouvoir refermer le fichier au cas où l'utilisateur annule la commande. Ce que nous allons faire en redéfinissant la fonction ***error***.
Deux méthodes sont couramment employées.
Dans les deux cas la redéfinition de ***error*** n'affichera le message que s'il est différent de "Fonction annulée" et fermera le fichier s'il est ouvert.
La première consiste à définir une fonction de gestion des erreurs à l'extérieur de la routine principale et à affecter cette définition à ***error*** après avoir sauvegardé la définition initiale dans une variable globale. La définition initiale sera restaurée en fin de routine ainsi qu'à la fin de la fonction de gestion des erreurs.

```
;; Fonction de gestion des erreurs
(defun GPTF_err (msg)
  (if (/= msg "Fonction annulée")
    (princ (strcat "\nErreur: " msg))
  )
  (if file
    (close file)
  )
  (setq *error* m:err                             ; restauration de *error*
        m:err nil
  )

  (princ)
)

;; Fonction principale
(defun c:GetPointToFile   (/ file pt lst)
 (setq  m:err *error*                             ; sauvegarde de *error*
       *error* GPTF_err                           ; nouvelle affectation
  )
  (setq file (open "C:\\Points.txt" "w"))
  (while (setq pt (getpoint "\nSpécifiez un point: "))
    (write-line (lst2str pt ",") file)
  )
  (close file)
  (setq *error* m:err                             ; restauration de *error*
        m:err nil
  )
  (princ)
)
```

La seconde méthode (préférable, à mon avis) consiste à redéfinir ***error*** localement, c'est-à-dire à l'intérieur de la fonction principale en ayant soin de déclarer ***error*** comme une variable locale.

```
(defun c:GetpointToFile (/ *error* file pt lst)

  (defun *error* (msg)
    (if (/= msg "Fonction annulée")
      (princ (strcat "\nErreur: " msg))
    )
    (if file
      (close file)
    )
    (princ)
  )

  (setq file (open "C:\\Points.txt" "w"))
  (while (setq pt (getpoint "\nSpécifiez un point: "))
    (write-line (lst2str pt ",") file)
  )
  (close file)
  (princ)
)
```

AutoLISP® fournit un autre moyen de gérer les erreurs. La fonction **vl-catch-all-apply** permet de capturer une erreur sans interrompre le processus en cours.
`(vl-catch-all-apply 'fonction liste)`
Comme pour la fonction **apply**, l'argument `fonction` peut être une fonction prédéfinie, définie avec **defun** ou une fonction **lambda**. L'argument `liste` étant la liste des arguments à passer à `fonction`.
**vl-catch-all-apply** retourne le résultat de l'appel de fonction ou, si une erreur survient, un objet de type VL-CATCH-ALL-APPLY-ERROR.

La fonction prédicat **vl-catch-all-error-p** retourne **T** si l'argument qui lui est passé est une erreur survenue dans un appel **vl-catch-all-apply**.

La fonction **vl-catch-all-error-message** retourne le message d'erreur de l'objet VL-CATCH-ALL-APPLY-ERROR qui lui est passé comme argument.

```
(defun catch_div (lst / catch)
  (setq catch (vl-catch-all-apply '/ lst))
  (if (vl-catch-all-error-p catch)
    (progn
      (alert (vl-catch-all-error-message catch))
      nil
    )
    catch
  )
)
```

`(catch_div '(15 3))` retourne : `5`

`(catch_div '(15 0))` retourne : `nil` et affiche `division par zéro`

`(catch_div '("a" "b"))` retourne : `nil` et affiche `type d'argument incorrect: numberp: "a"`

# 16 Accès aux objets du dessin

Un fichier dwg est une base de donnée constituée d'entité graphiques (lignes, cercles, références de bloc, etc…) et d'objets non graphiques (calques, styles de texte ou de cote, définitions de bloc, etc..).
AutoLISP® fournit des fonctions permettant d'accéder à ces objets pour les lire ou les modifier.

## 16.1 Entité unique

**entsel** `(entsel [msg])`
Invite l'utilisateur à sélectionner une entité unique en cliquant un point. Un message d'invite peut être passé comme argument à la fonction, s'il n'est pas spécifié, le message "Choix de l'objet: " est affiché. **entsel** retourne une liste dont le premier élément est le nom d'entité (type ENAME) de l'objet et le second, les coordonnées (SCU) du point cliqué (ce point n'est pas obligatoirement strictement sur l'objet). Si aucun objet n'est sélectionné **entsel** retourne **nil**.

```
(setq ent (car (entsel)))
```
retourne **`<Nom d'entité: 7efa1648>`** ou **`nil`** si aucun objet n'a été sélectionné

Certaines entités graphiques sont dites "complexes" parce qu'elles sont composées de sous entités (références de bloc, polylignes 3d…) les fonctions **nentsel** et **nentselp** permettent de sélectionner ces sous entités.
**nentsel** `(nentsel [msg])`
**nentselp** `(nentselp [msg] [pt])`
Elles acceptent aussi un message comme argument. **nentselp** accepte en plus un argument point (optionnel) qui permet d'éviter l'intervention de l'utilisateur.
Si l'entité sélectionnée par ces fonctions est une entité simple, la liste retournée est la même qu'avec **entsel**.
Si l'entité est un segment de polyligne 3d ou une référence d'attribut, la liste est semblable à celle retournée par **entsel** mais le nom d'entité retourné est celui du sommet (VERTEX) de départ du segment de polyligne 3d ou de la référence d'attribut (ATTRIB).
S'il s'agit d'un composant de bloc autre qu'un attribut, ces fonctions retournent une liste constituée de :
- le nom d'entité de la sous entité sélectionnée
- les coordonnées du point cliqué
- la matrice de transformation décrivant toutes les transformations (échelle, rotation…) subie par la sous entité
- la liste de noms d'entité de tous les ascendants de la sous entité du plus profondément imbriqué à la référence insérée.

**entlast** `(entlast)`
Retourne le nom d'entité de la dernière entité créée dans le dessin.

**entnext** `(entnext [ename])`
Appelé sans argument, **entnext** retourne le nom d'entité de la première entité non effacée de la base de données. Sinon **entnext** retourne le nom d'entité de l'entité suivant `ename` dans la base de données. Si `ename` est une entité complexe, **entnext** retourne le nom d'entité de la première sous entité composant `ename`. Ce nom d'entité peut être repassé comme argument à **entnext** pour obtenir la sous entité suivante et ainsi de suite jusqu'à rencontrer un objet "SEQEND" qui marque la fin de la séquence de sous entités.

**entdel** `(entdel ename)`
Supprime une entité, ou restaure une entité précédemment supprimée.

**handent** `(handent handle)`
Le nom d'entité des objets graphiques d' AutoCAD® n'est pas enregistré dans la base de donnée et  il est susceptible de changer entre deux ouvertures du même fichier, par contre, une donnée est immuablement liée à chaque objet (tant qu'il n'est pas définitivement supprimé) , son "maintien" (*handle* en anglais). C'est une chaîne exprimant un nombre hexadécimal. Nous verrons plus loin comment récupérer cette donnée. La fonction **handent** retourne le nom d'entité courant d'un objet d'après son maintien.

## 16.2 Données DXF des objets (entget entmake entmakex entmod entupd)

Avec AutoLISP®, la base de données des objets est accessible via les données DXF.
Si on ouvre un fichier DXF avec un éditeur de texte, on peut voir que chaque donnée est précédée d'un nombre entier : le code groupe.
Pour pouvoir accéder aisément aux données AutoLISP organise les données DXF en liste d'association dont le premier élément de chaque paire pointée (ou liste) est un code de groupe. Ainsi certains codes de groupes sont valables pour tous les types d'objets : -1 (nom d'entité de l'objet), 0 (type d'objet), 5 (maintien –*handle*–), etc. d'autres sont spécifiques à un type d'entité ou ont une signification différente suivant les types d'objet.
Toutes les références DXF sont décrites dans l'aide aux développeurs d'AutoCAD®, rubrique "Référence DXF" (en français pour les versions françaises).

**entget** `(entget ename [app])`
Retourne la liste DXF de l'entité.
L'argument facultatif `app` est une liste de noms d'applications enregistrées (clés pour les données étendues –xdata–). Si cette argument est spécifié, la liste retournée contiendra les données étendues pour ces applications (spécifier `'("*")` pour toutes les applications).

Exemple :
`(command "_circle" '(5 8) 10)` crée un cercle de centre (5 8) et de rayon 10.
`(setq ent (entlast))` retourne le ENAME du cercle : **<Nom d'entité: 7efcc4f0>**
`(entget ent)` retourne la liste suivante :

```
((-1 . <Nom d 'entité: 7efcc4f0>)        ename de l'entité
  (0 . "CIRCLE")                         type d'entité
  (330 . <Nom d 'entité: 7efc9cf8>)      ename de l'objet propriétaire
  (5 . "2076")                           maintien (handle)
  (100 . "AcDbEntity")                   marqueur de classe
  (67 . 0)                               absent ou 0 = espace objet, 1 = espace papier
  (410 . "Model")                        nom de la présentation (ou "Model")
  (8 . "0")                              nom du calque
  (100 . "AcDbCircle")                   marqueur de sous classe
  (10 5.0 8.0 0.0)                       centre du cercle (coordonnées SCO)
  (40 . 10.0)                            rayon du cercle
  (210 0.0 0.0 1.0)                      direction d'extrusion
)
```

**entmake** `(entmake dxflst)`
Crée un nouvel objet dans le dessin. **entmake** peut créer des entités graphiques ou des objets non graphiques.
L'argument `dxflst` est une liste du type de celle retournée par **entget**. La liste doit contenir toutes les données nécessaires pour créer l'objet. Si des données optionnelles ne sont pas spécifiées l'objet prendra les propriétés courantes.
Suivant le type d'entité certains groupes (groupe 100 par exemples) peuvent être requis ou optionnels.
Si la liste ne contient pas toutes les données nécessaires, **entmake** retourne **nil** et l'objet n'est pas créé, sinon **entmake** retourne `dxflst`.

L'utilisation de **entmake** à la place de **command** pour créer des entités présente plusieurs avantages. **entmake** est plus rapide, est insensible aux accrochages aux objets et permet de spécifier directement des propriétés autre que les propriétés courantes.
Pour créer le même cercle que ci-dessus sur le calque "TEST" (si le calque n'existe pas, il est créé avec les paramètres par défaut), on aurait pu faire :
`(entmake '((0 . "CIRCLE") (8 . "TEST") (10 5.0 8.0 0.0) (40 . 10.0)))`
retourne : `((0 . "CIRCLE") (8 . "TEST") (10 5.0 8.0 0.0) (40 . 10.0)))`

**entmakex** fonctionne comme **entmake**, mais retourne le nom de l'entité crée ou nil.

`(setq line (entmakex '((0 . "LINE") (10 -5.0 8.0 0.0) (11 15.0 8.0 0.0))))`
retourne : **<Nom d'entité: 7efa16f8>**

**entmod** `(entmod dxflst)`
Modifie la base de données d'un objet (graphique ou non)
L'argument `dxflst` est la liste DXF de l'objet modifiée. Pratiquement, on récupère la liste DXF avec **entget**, on la modifie avec les fonctions de traitement de listes (notamment la fonction **subst**) et on passe la liste modifiée comme argument à **entmod**.
Par exemple, pour mettre la ligne line précédemment créée avec entmakex sur le calque "TEST",
On peut faire:

```
(setq elst (entget line))
(setq elst (subst '(8 . "TEST") (assoc 8 elst) elst)
(entmod elst)
```

NOTA : **entmod** ne fonctionne pas avec les entités VPORT (fenêtres).

**entupd** `(entupd ename)`
Met à jour l'affichage à l'écran des entités complexes modifiées avec **entmod**.

## 16.3 Jeu de sélection (ssget ssadd ssdel sslength ssmemb ssname ssnamex ssgetfirst sssetfirst)

Un jeu de sélection est un objet AutoCAD® (type PICKSET) qui contient des entités graphiques.
Ce type d'objet peut être passé comme argument de certaines commandes d'édition. Le plus souvent il doit être parcouru avec la fonction ssname (voir ci-dessous) pour obtenir les noms d'entité (type ENAME) des éléments qui le composent.
Le nombre de jeux de sélection ouverts est limité à 128. Il est donc important d'affecter les jeux de sélection à des variables et de veiller à ce que ces variables retournent à **nil** après utilisation.

Il est possible d'inviter l'utilisateur à sélectionner plusieurs entités en une seule fois (ou de faire une sélection sans son intervention) avec la fonction **ssget**.
**ssget** `(ssget [mode] [pt1 [pt2]] [pt-lst] [filtre])`
Retourne un jeu de sélection (type PICKSET) contenant les entités sélectionnées.
Utilisé sans argument, **ssget** affiche le message "Choix des objets: " et invite l'utilisateur à faire une sélection classique (comme la commande SELECT). L'utilisateur peut utiliser les options de modes de sélection (CP, SP, T, P, etc…).

L'argument `mode` est utilisé pour forcer un mode de sélection, la plupart de ces modes permettent de faire des sélections sans intervention de l'utilisateur*.
`"_A"` : toute la base de données (excepté les entités sur les calques gelés, comme l'option "TOUT")
`"_C"` : capture, les arguments `pt1` et `pt2` (2 points) sont requis
`"_CP"` : capture polygonale, l'argument `pt-lst` (liste de points) est requis
`"_F"` : trajet, l'argument `pt-lst` (liste de points) est requis
`"_I"` : implicite (les objets sélectionnés si PICKFIRST = 1)
`"_L"` : dernier objet visible ajouté à la base de données
`"_P"` : précédent
`"_W"` : fenêtre, les arguments `pt1` et `pt2` (2 points) sont requis
`"_WP"` : fenêtre polygonale, l'argument `pt-lst` (liste de points) est requis
`"_X"`: toute la base de données (y compris les objets sur des calques gelés)
Exemple :
`(ssget "_W" '(2 5) '(30 18))` sélection par fenêtre du point (2 5) au point (30 18)

Certains modes servent à forcer une méthode de sélection par l'utilisateur
`"_:E"` : tous les objets situés sous la cible de sélection (ni fenêtre, ni capture)
`"_:S"` : sélection unique (une seule fenêtre ou capture)
`"_:L"` : exclusion des objets sur les calques verrouillés

* L'emploi d'un tiret bas (*underscore*) étant nécessaire ou non suivant les versions et/ou les modes, il est préférable de le systématiser.

Si l'argument `pt1` est utilisé sans mode de sélection, la sélection est faite comme si l'utilisateur cliquait un seul point.

L'argument `filtre` permet de filtrer la sélection (mêmes possibilités qu'avec la commande FILTER). Il s'agit d'une liste d'association stipulant les propriétés des objets que doivent posséder les objets pour pouvoir être sélectionnés (voir 16.4).
Exemple :
`(ssget "_X" '((0 . "LINE")))` sélection de toutes les lignes de la base de donnée.

**ssadd** `(ssadd [ename [jsel]])`
Utilisé sans argument, **ssadd** crée un jeu de sélection vide.
Si seul l'argument `ename` est spécifié, **ssadd** crée un jeu de sélection contenant cette entité.
Si les deux arguments sont spécifiés, l'entité `ename` est ajoutée au jeu de sélection `jsel`.
Retourne le jeu de sélection ou **nil** si l'opération a échoué.

**ssdel** `(ssdel ename jsel)`
Supprime l'entité du jeu de sélection.
Retourne le jeu de sélection ou **nil** si l'entité n'était pas dans le jeu de sélection.

**sslength** `(sslength jsel)`
Retourne le nombre d'entités contenues dans `jsel` (entier).

**ssmemb** `(ssmemb ename jsel)`
Retourne `ename` s'il est contenu dans `jsel`, **nil** sinon.

**ssname** `(ssname jsel ind)`
Retourne le nom d'entité (ENAME) de l'élément de `jsel` à l'indice `ind` (l'indice du premier élément est 0).
S'il n'y a aucun élément à l'indice spécifié, **ssname** retourne **nil**.

**ssnamex** `(ssnamex jsel)`
Retourne une liste contenant des informations sur la façon dont a été crée un jeu de sélection.

**ssgetfirst** `(ssgetfirst)`
Retourne une liste dont le premier élément (obsolète) est toujours **nil** et le second est un jeu de sélection contenant tous les objets actuellement sélectionnés dont les "grips" sont activés.

**sssetfirst** `(sssetfirst [gripset [jsel]])`
L'argument `gripset` est ignoré (obsolète) et généralement spécifié **nil**.
Utilisé sans arguments ou avec `gripset` et `jsel` à **nil**, **sssetfirst** désélectionne les objets actuellement sélectionnés/grippés.
Si l'argument `jsel` est spécifié et non **nil**, les entités contenues dans `jsel` sont sélectionnées et grippées.

Un jeu de sélection (type PICKSET) peut être utilisé directement avec certaines commandes, mais souvent il est nécessaire de le parcourir pour accéder à chacune des entités qu'il contient.
Pour ce faire, on utilise la fonction **ssname** en incrémentant l'argument indice dans une boucle avec **repeat** ou **while**.

Exemple

```
(setq n 0)                                   ; initialisation de l'indice
(if (setq ss (ssget "_X" '((0 . "HATCH"))))
  (while (setq ent (ssname ss n))
    (setq elst (entget ent)                  ; liste DXF de l'objet
          n    (1+ n)                        ; incrémentation de l'indice
    )
    (entmod (subst '(8 . "HACHURES")         ; modification du claque
                   (assoc 8 elst)
                   elst
            )
    )
  )
)
```

## 16.4 Filtre de sélection

Les filtres de sélection sont des listes d'association utilisant les codes de groupe DXF à l'exception des groupes -1 (nom d'entité), 5 (maintien), et des codes de groupe supérieurs à 1000 (données étendues).

Les groupes utilisant des chaînes : 0 (type d'entité), 8 (calque), etc. acceptent les caractères génériques et peuvent contenir plusieurs données séparés par des virgules.
(ssget '((0 . "*TEXT,DIMENSION"))) filtre les objets TEXT, MTEXT, RTEXT et DIMENSION.

Les groupes utilisant de données numériques (entiers, réels, points, vecteurs) supportent des tests relationnels. Les opérateurs relationnels sont utilisés avec le code groupe -4 :
(ssget "_X" '((0 . "CIRCLE") (-4 . "=") (40 . 20.0))) sélectionne tous les cercle dont de rayon (code de groupe 40) est égal à 20.0.

Les tests relationnels sont :
"*" : toute valeur
"=" : égal à
"!=" ou "/=" ou "<>" : différent de
"<=" : inférieur ou égal
"<" : strictement inférieur
">=" : supérieur ou égal
"<" : strictement supérieur
"&" : AND logique (entiers uniquement, au moins un bit du filtre est aussi dans le groupe)
"&=" : AND logique exclusif (entiers uniquement, tous les bits du filtre sont aussi dans le groupe)

(ssget "_X" '((0 . "POINT") (-4 . "*,*,!=") (10 0.0 0.0 0.0))) sélectionne tous les points dont le Z est différent de 0

(ssget "_X" '((0 . "POLYLINE") (-4 . "&=") (70 . 9))) sélectionne toutes les polylignes 3d fermées (8 = polyligne 3d et 1 = fermée)

Les filtres peuvent aussi contenir des groupes logiques imbriqués construits avec les opérateurs AND, OR, XOR et NOT de la manière suivante :
"<AND" un ou plusieurs opérandes "AND>"
"<OR" un ou plusieurs opérandes "OR>"
"<XOR" deux opérandes "XOR>"
"<NOT" un opérande "NOT>"

```
(ssget (list
        '(-4 . "<OR")
        '(0 . "CIRCLE")
        '(-4 . "<AND")
        '(0 . "ELLIPSE")
        '(41 . 0.0)
        (cons 42 (* 2 pi))
        '(-4 . "AND>")
        '(-4 . "<AND")
        '(0 . "LWPOLYLINE")
        '(-4 . "&")
        '(70 . 1)
        '(-4 . "AND>")
        '(-4 . "OR>")
       )
)
```

Filtre les cercles, les ellipses fermées et les polylignes fermées.

## 16.5 Tables (tblnext tblsearch tblobjname)

Les tables regroupent des objets non graphiques en fonction de leur type. Les entrées des différentes tables sont manipulables avec les fonctions **entdel**, **entmake**, **handent** et **entmod** (sauf les entités VPORT pour cette dernière).
Chaque table à un nom. Ce nom est utilisé comme argument avec les fonctions **tblnext**, **tblsearch** et **tblobjname** .

| | |
|---|---|
| LAYER : | calques |
| LTYPE : | types de lignes |
| VIEW : | vues enregistrées |
| STYLE : | styles de texte |
| BLOCK : | définitions de blocs |
| UCS : | systèmes de coordonnées utilisateur enregistrés |
| APPID : | applications enregistrées (xdata) |
| DIMSTYLE : | styles de cote |
| VPORT : | fenêtres de présentation |

**tblnext** `(tblnext nom_table [prem])`
Trouve l'élément suivant dans la table. Si l'entée existe **tblnext** retourne une liste d'association de données DXF, s'il n'y a plus d'entrée, **tblnext** retourne nil.
Si l'argument `prem` est spécifié et non **nil**, **tblnext** retourne les données de la première entrée.
`(tblnext "LAYER" T)` retourne :
`((0 . "LAYER") (2 . "0") (70 . 0) (62 . 7) (6 . "Continuous"))`

**tblsearch** `(tblsearch nom_table sym [setnext])`
Cherche l'entrée `sym` dans la table. Si une entrée nommée `sym` existe, **tblsearch** retourne sa liste de données DXF, si aucune entrée n'est trouvée **tblsearch** retourne nil.
Si l'argument `setnext` est spécifié et non **nil** et si une entrée est trouvée, le compteur pour **tnblnext** est réinitialisé à partir de cette entrée.
`(tblsearch "LTYPE" "Continuous")` retourne :
`((0 . "LTYPE") (2 . "Continuous") (70 . 0) (3 . "Solid line") (72 . 65) (73 . 0) (40 . 0.0))`

**tblobjname** `(tblobjname nom_table sym)`
Cherche l'entrée `sym` dans la table et retourne son nom d'entité si elle existe, **nil** sinon.
On obtient la liste DXF complète d'une entrés avec : `(entget (tblobjname nom_table sym))`

## 16.6 Données étendues, dictionnaires (regapp, xdroom, xdsize, dictnext dictsearch dictadd dictremove dictrename namedobjdict)

### 16.6.1 Données étendues

Les données étendues (xdatas) sont un moyen pour lier des données à un objet. Par exemple, AutoCAD® utilise les utilise pour conserver la liste des calques gelés uniquement dans une fenêtre de présentation.
Ces données appartiennent toujours à une application enregistrée qui sert aussi de clé pour y accéder.
Avant d'affecter des xdatas à un objet, il faut enregistrer l'application à laquelle elles appartiendront. La fonction LISP **regapp** sert à créer une nouvelle application enregistrée.
`(regapp application)`
L'argument `application` est le nom donné à l'application, ce nom doit être unique. S'il est valide **regapp** retourne le nom, sinon **nil**.
Les xdatas sont ajoutés en fin de liste DXF dans une liste unique dont le code de groupe est -3 (sentinelle).
Cette liste contient une sous liste par application dont le premier élément est le nom de l'application et suivants des paires pointées contenant les données.
Ces paires pointées utilisent des codes de groupe supérieurs ou égaux à 1000.
Les principaux codes de groupe utilisés sont :
1000 : chaîne de caractère
1003 : nom de calque
1005 : maintien (*handle*)

1010 : point
1040 : nombre réel
1041 : distance
1042 : facteur d'échelle
1070 : entier (16 bits)
1071 : entier long (32 bits)
Comme la taille de la mémoire des données étendues affectée à une entité est limité à 16 ko, il existe une fonction qui retourne la mémoire disponible dans une entité : **xdroom**. Une autre fonction : **xdsize** retourne la mémoire nécessaire pour une liste.
Exemple :

```
;; créer un cercle
(setq circle (entmakex '((0 . "CIRCLE") (10 20.0 15.0 0.0) (40 . 10.0))))

;; créer une application
(setq app (regapp "MON_APPLI"))

;; créer une liste de xdata
(setq xdata '(-3 ("MON_APPLI" (1000 . "circle") (1040 . 10.0))))

;; lier les xdatas au cercle
(entmod (append (entget circle) (list xdata)))

;; récupérer les xdatas
(cadr (assoc -3 (entget circle '("MON_APPLI"))))
```

Retourne : `("MON_APPLI" (1000 . "circle") (1040 . 10.0))`

### 16.6.2 Dictionnaires

Les dictionnaires fournissent un autre moyen de lier des données au dessin ou à des entités graphique.
Ils peuvent contenir des données arbitraires et leur taille n'est pas limitée.
Les données dans un dictionnaire sont stockées dans un objet XRECORD sous forme de paires pointées, les codes de groupes étant les même que pour les données DXF.
Un dictionnaire peut contenir plusieurs entrées XRECORD mais aussi d'autres dictionnaires.
Tout fichier DWG possède un "dictionnaire des objets nommés" qui est la racine de tous les objets non graphiques du dessin.
Par ailleurs tout objet graphique ou non peut avoir un "dictionnaire des extensions".

AutoLISP® fournit des fonctions qui permettent d'accéder à ces dictionnaires et à leurs données, mais aussi d'en créer de nouveaux.

**namedobjdict** ne requiert aucun argument et retourne le nom d'entité du dictionnaire d'objets nommés du dessin courant. Ce dictionnaire est la racine de tous les objets non graphiques du dessin.

**dictadd** `(dictadd ename sym nouv)`
Ajoute un objet DICTIONARY ou XRECORD à un dictionnaire.
`ename` est le nom d'entité du dictionnaire auquel est ajouté l'objet.
`sym` est le nom de l'entée du nouvel objet, ce nom doit être unique.
`nouv` est le nom d'entité de l'objet ajouté.

```
;; Créer un objet dictionnaire
(setq xname (entmakex '((0 . "DICTIONARY") (100 . "AcDbDictionary"))))
;; Ajouter le dicitonnaire au dictionnaire des objets nommés
(dictadd (namedobjdict) "MON_SUPER_DICO" xname)
;; créer un objet XRECORD avec des données
(setq xrec (entmakex '((0 . "XRECORD")
                       (100 . "AcDbXrecord")
                       (1 . "Ceci est un test")
                       (40 . 3.14159)
                      )
           )
)
;; ajouter l'objet XRECORD au dictionnaire
(dictadd xname "SUPER_DATA_1" xrec)
```

**dictsearch** `(dictsearch ename sym [setnext])`
Cherche l'entrée spécifiée dans le dictionnaire, et retourne sa liste DXF s'il existe (sinon **nil**).
`ename` : le nom d'entité du dictionnaire
`sym` : le nom de l'entrée
`setnext` : si spécifié et non nil, initialise dictnext avec cette entrée.

```
;; retrouver le dictionnaire
(setq dict  (dictsearch (namedobjdict) "MON_SUPER_DICO"))

;; retrouver les données
(setq data (dictsearch (cdr (assoc -1 dict)) "SUPER_DATA_1"))
```

`(cdr (assoc 1 data))` retourne : `"Ceci est un test"`
`(cdr (assoc 40 data))` retourne : `3.14159`

**dictnext** `(dictnext ename [premier])`
Retourne la liste DXF de l'entrée suivante du dictionnaire si elle existe, sinon **nil**.
`ename` le nom d'entité du dictionnaire propriétaire
Si l'argument `premier` est spécifié et non **nil**, **dictnext** retourne la première entrée.

Ajoutons une nouvelle entrée à notre dictionnaire

```
(setq xname (cdr (assoc -1 (dictsearch (namedobjdict) "MON_SUPER_DICO")))
      xrec  (entmakex '((0 . "XRECORD")
                        (100 . "AcDbXrecord")
                        (70 . 256)
                       )
            )
)
(dictadd xname "SUPER_DATA_2" xrec)
```

`(cdr (member '(280 . 1) (dictnext xname T)))` Retourne : `((70 . 256))`

`(cdr (member '(280 . 1) (dictnext xname)))`
Retourne : `((1 . "Ceci est un test") (40 . 3.14159))`

**dictremove** `(dictremove ename sym)`
Supprime l'entrée `sym` du dictionnaire et retourne le nom d'entité de l'objet supprimé ou nil si un des arguments n'est pas valide.

**dictrename** `(dictrename ename ancien nouveau)`
Renomme une entrée d'un dictionnaire. Si tous les arguments sont valides, l'entrée `ancien` est renommée `nouveau` et `nouveau` est retourné, sinon **nil** est retourné.

Tout objet AutoCAD® peut avoir un dictionnaire qui lui est lié : son dictionnaire d'extensions. Ce dictionnaire est unique et peut avoir été créé par AutoCAD ou une autre application.
Comme la fonction **dictadd** ne peut ajouter des entrées qu'à un dictionnaire existant, si un objet n'a pas de dictionnaire d'extensions, on l'ajoute en modifiant les données de l'objet avec la fonction **entmod**.