Autolisp

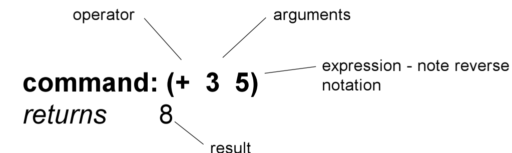
The programming environment available within AutoCAD has always been one of its main strengths and with advent of Rel. 14, the user can access three levels of capability.

1) **Command Scripts**
2) **Autolisp**
3) **Autocad Runtime eXtension (ARX)**

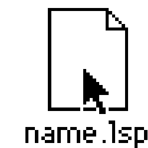
Command scripts are the simplest and are best suited to the operation of simple macros, although quite complex routines can be executed using this facility. To create a script, the user writes an external text file containing the required command steps in sequence. Any valid Acad command may be included in the script, which tracks from top to bottom of the text file in a simple linear fashion (toilet roll). Two additional commands may be used in a script ; **Delay** specifies a delay in milliseconds, and **Rscript** which allows a script to loop. Scripts also work outside the drawing editor and so are useful for plotting automation and can be used to close one drawing and open another. Scripts can be used for Acad slide presentation, cycling through a series of snapshots with predefined pauses. Scripts can be defined and called from Autolisp.

The **Autocad Runtime eXtension (ARX)** allows high level languages such as **C** and **JAVA** to interface with Acad. This interface consists of a library of functions which parallel Autolisp functions, and an ARX application is written as a set of external subroutines that can be loaded by and called from Autolisp, using the **(arxload)** function. The ARX environment is aimed at the commercial developer wishing to create Autocad applications.

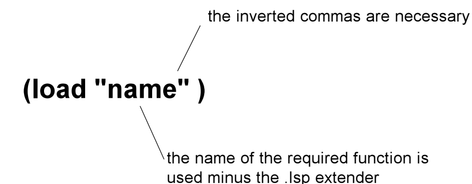
Autolisp is an implementation of the **lisp** programming language embedded in Autocad. Lisp is an old language taking its name from the acronym **LIS**t **P**rocessing. It is a relatively easy language to learn, and the user can produce elegant and efficient code with little effort. The heart of Autolisp's interaction with Acad is the lisp **evaluator**. This lurks behind the command line and intercepts user input, evaluating it and returning a result. If the input is not an Autolisp expression, then the evaluator passes it unaltered to Autocad. The simplest way to use Autolisp is to enter expressions directly at the command prompt



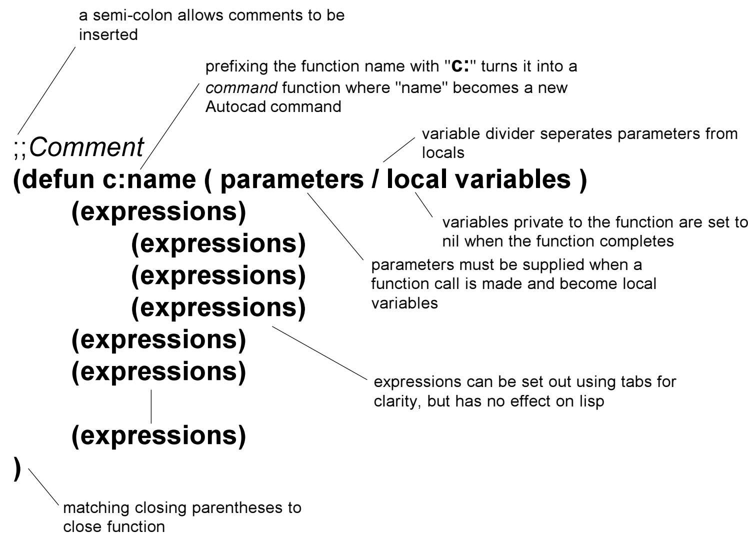
The scope of this method is limited so it is normal to create lisp routines in the form of text files which can be loaded into Acad for use. These files are created in text editors like Edit, Notepad, BBedit (mac), or PFE and should contain no formatting characters. The naming convention is that each file should have the extender *.lsp* after it i.e.



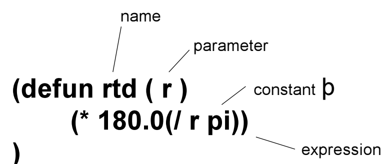
A lisp file is loaded into Acad using the *load* command or *drag+dropped* onto an open Acad session (you can also use the Application load menu option).



Lisp files all have the same basic structure, and are composed of functions. Each function has the following syntax:



A simple function called rtd.lsp, can be created which converts radians to degrees.



After loading rtd.lsp, the function remains active for the remainder of the drawing session, and the user can access it from the command line by entering

parameter

command:(rtd pi)
returns **180**

Variables or symbols (containers of information) are assigned values using the inbuilt **setq** function

(setq banana 5.0) returns 5.0
(setq fred "bloggs") returns "bloggs"

Variables are local if they are declared in a function header line, and are global otherwise. Global variables are accessible to any function during a drawing session, and local variables are private to their own function or its sub-functions, and are lost when the function terminates. All variables are lost at the end of a drawing session. Variables can be assigned several different value types.

Data Type	**Example**
Integer	24
Real Number	24.675
Text String	“Bananas are horrible”
Lists	(100 350 200)
Entity Name	<Entity name: 600000014h>
Selection sets	<Selection set: 168>
Symbols	point1

Non-Recursive 2D Tree Routine.

This example uses Autolisp's while loop construct to define the growth of a 2D tree and allows the user to specify the number of branches grown. The growth rules are simple - each branch splits into two similar branches growing at an arbitrary angle set symmetrically to the first branch.

Sub1

```
(defun atr(deg / rad)
  (setq rad(* deg(/ pi 180)))
)
```

The sub-function **atr** converts from Degrees to Radians. Hence the name "atr" (angles to radians).

Sub2

```
(defun grow(tot last / props ps pe vector vector1 vector2)
  (while(> tot 0)
    (setq props(entget last)
        ps(cdr(assoc 10 props))
        pe(cdr(assoc 11 props))
        vector(angle ps pe)
        vector1(- vector(atr 10))
        vector2(+ vector(atr 10)))
    (draw pe (polar pe vector1 100))
    (draw pe (polar pe vector2 100))
    (setq tot(- tot 2)
        last(entnext last))
  )
)
```

The **grow** sub-function cycles through a while loop until the required number of branches has been drawn (while *tot* is greater than zero). Each pass of the loop gets the properties of the last branch drawn and works out the vectors for two new branches to be drawn from the end of the last. The draw sub-function is then called to draw the new branches, and before restarting the loop, the total is decremented and the next entity selected.

Sub3

```
(defun draw(start finish)
  (command"line" start finish "")
)
```

The **draw** sub-function does the actual branch drawing, using the parameters passed to it, *start* and *finish*. These become the end points of the new branch.

Sub4

```
(defun rub( / all)
  (setq all(ssget"x"))
  (if all(command"erase"all""))
)
```

Rub acts as a clean-up function by erasing everything in the drawing, before creating a new tree.

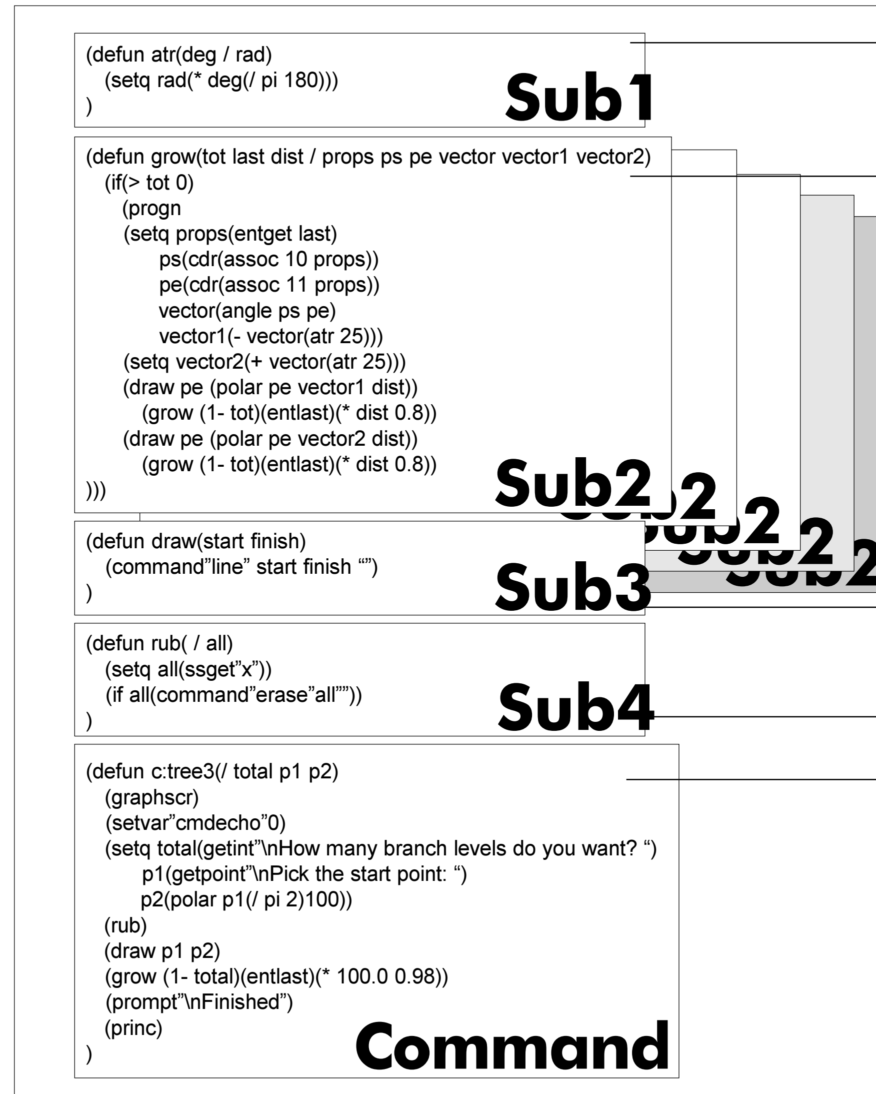
Command

```
(defun c:tree(/ total p1 p2)
  (graphscr)
  (setvar"cmdecho"0)
  (setq total(getint"\nHow many braches do you want? ")
      p1(getpoint"\nPick the start point: ")
      p2(polar p1(/ pi 2)100))
  (rub)
  (draw p1 p2)
  (grow (1- total)(entlast))
  (prompt"\nFinished")
  (princ)
)
```

The **command** function adds an extra command to Acad. This allows the user to call the routine by entering "**tree**" at the command prompt. The function first makes the graphics screen active and then sets the "cmdecho" variable to off. If this is not done, Acad spends a lot of time printing commands to the screen. The user is then prompted to input an integer value for the number of branches required, and to pick a start point for the growth of the tree. The end point of the first branch is then calculated and assigned to a variable ready for use. Before commencing, the **rub** sub-function is called to clear the drawing and **draw** is called to place the first branch. Grow is then called with the required parameters. Once execution has finished, the routine then prints "Finished" to the screen and exits cleanly.

Recursive 2D Tree Routine.

Similar in outline to the previous routine, this example uses Autolisp's ability to call itself recursively to define the growth of the tree in a more appropriate way.




```
(defun atr(deg / rad)
   (setq rad(* deg(/ pi 180)))
)

(defun grow(tot last dist / props ps pe vector vector1 vector2)
   (if(> tot 0)
     (progn
     (setq props(entget last)
           ps(cdr(assoc 10 props))
           pe(cdr(assoc 11 props))
           vector(angle ps pe)
           vector1(- vector(atr 25)))
     (setq vector2(+ vector(atr 25)))
     (draw pe (polar pe vector1 dist))
        (grow (1- tot)(entlast)(* dist 0.8))
     (draw pe (polar pe vector2 dist))
        (grow (1- tot)(entlast)(* dist 0.8))
)))

(defun draw(start finish)
   (command"line" start finish "")
)

(defun rub( / all)
   (setq all(ssget"x"))
   (if all(command"erase"all""))
)

(defun c:tree3(/ total p1 p2)
   (graphscr)
   (setvar"cmdecho"0)
   (setq total(getint"\nHow many branch levels do you want? ")
        p1(getpoint"\nPick the start point: ")
        p2(polar p1(/ pi 2)100))
   (rub)
   (draw p1 p2)
   (grow (1- total)(entlast)(* 100.0 0.98))
   (prompt"\nFinished")
   (princ)
)
```

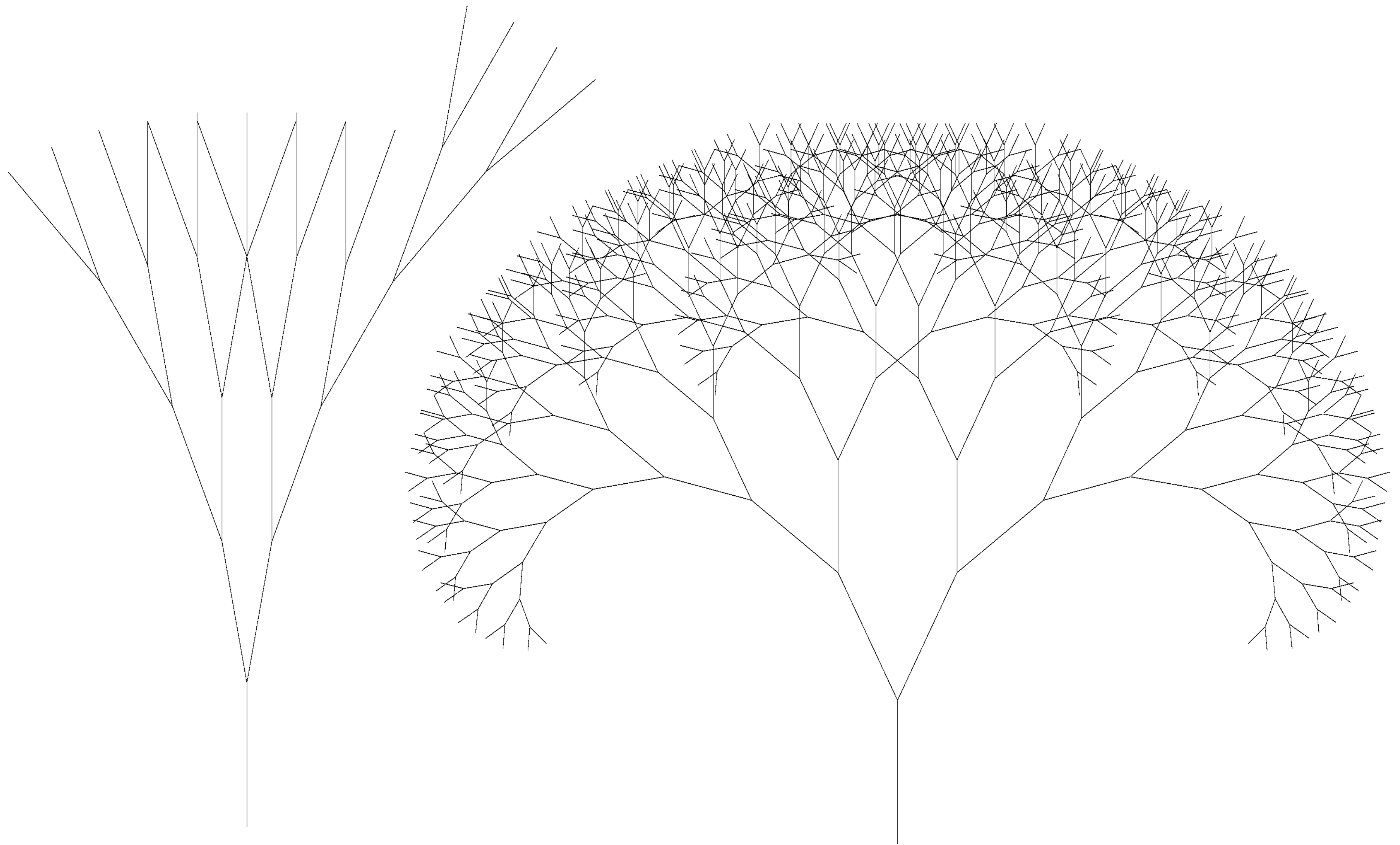
The sub-function **atr** converts from Degrees to Radians. Hence the name "atr" (angles to radians).

The **grow** sub-function for this routine is recursively defined, and this fundamentally effects the way that the tree grows. The while loop has been replaced with a simple conditional statement (if(> tot 0)). When the result is true, the sub-function executes, but when the result is false the sub-function skips. The first part of **grow** is similar to the previous example, but after calling **draw** to create the first branch, it immediately calls itself, passing a decremented counter, the name of the last entity and a reduced branch length as parameters. Each time **grow** calls itself in this way, a separate version of the sub-function is created in memory. The second version of **grow** proceeds to draw a branch and then call itself again, and so on until *tot* is zero. In this way **grow** steps down the levels of the tree drawing only the first branch in each case. Only when it has reached the condition where *tot* is zero will it draw a second branch. **Grow** will then step back up the hierarchy of the tree drawing the second branches from the previously drawn first branches, filling out the structure of the tree.

The **draw** sub-function does the actual branch drawing, using the parameters passed to it, *start* and *finish*. These become the end points of the new branch.

Rub acts as a clean-up function by erasing everything in the drawing, before creating a new tree.

The **command** function adds an extra command to Acad. This allows the user to call the routine by entering "**tree**" at the command prompt. The function first makes the graphics screen active and then sets the "cmdecho" variable to off. If this is not done, Acad spends a lot of time printing commands to the screen. The user is then prompted to input an integer value for the number of branches required, and to pick a start point for the growth of the tree. The end point of the first branch is then calculated and assigned to a variable ready for use. Before commencing, the **rub** sub-function is called to clear the drawing and **draw** is called to place the first branch. Grow is then called with the required parameters. Once execution has finished, the routine then prints "Finished" to the screen and exits cleanly.



Above - (left) a tree created by the non recursive routine and (right) one generated using the recursive lisp.

Recursive 3D Tree Routine.

The same principals can be used to create a 3D model of a tree of great complexity, with very few rules and only a small amount of coding.

```
(defun draw (en0 xscl zscl shrink crop / en1 en2 en3 en4)
        (if (> zscl small)(progn
                (command “ucs” “e” en0)
                (command “ucs” “o” (strcat “0,0,” (rtos (* (/ 1.0 crop) zscl) 2 0)))
                (command “ucs” “z” (rtos (* fan (rand 2 rness)) 2 0))
                (command “ucs” “y” (rtos fan 2 0))
                (branch xscl zscl)
                (setq en1 (entlast))
1       (draw en1 (* shrink xscl)(* crop zscl)(* srate shrink)(* crate crop))
        (command “ucs” “e” en1)
        (if (> brnch 1)(progn
                (command “ucs” “y” (strcat “-” (rtos fan 2 0)))
                (command “ucs” “z” (rtos (* fan (rand 2 rness)) 2 0))
                (command “ucs” “y” (rtos fan 2 0))
                (branch xscl zscl)
                (setq en2 (entlast))
2       (draw en2 (* shrink xscl)(* crop zscl)(* srate shrink)(* crate crop))
        (command “ucs” “e” en2)
        (if (> brnch 2)(progn
                (command “ucs” “y” (strcat “-” (rtos fan 2 0)))
                (command “ucs” “z” (rtos (* fan (rand 2 rness)) 2 0))
                (command “ucs” “y” (rtos fan 2 0))
                (branch xscl zscl)
                (setq en3 (entlast))
3       (draw en3 (* shrink xscl)(* crop zscl)(* srate shrink)(* crate crop))
        (command “ucs” “e” en3)));end if
        ));end if
                (command “ucs” “y” (strcat “-” (rtos fan 2 0)))
                (if (or(< (rand 0 100) cent)(> zscl (* 0.7 long)))(progn
                        (branch xscl zscl)
                        (setq en4 (entlast))
4               (draw en4 (* shrink xscl)(* crop zscl)(* srate shrink)(* crate crop))
                        );end progn
                );end if
                );end progn
        );end if
)
```

```
(defun rand (bot top / x z r rn)
        (if (not seed)( setq seed 757))
        (setq x (1+ (* seed 2197.0))
                z (fix (/ x 4096.0))
                seed (fix (- x (* z 4096.0)))
                r (1+ (* (/ seed 4096.0) (- (- top bot) 1.0)))
                rn (+ bot r))
)

(defun branch (xscl zscl)
        (command “insert” “cylind” “0,0” “xyz” xscl xscl zscl “0”)
        (setq cnt (1+ cnt))
)

(defun c:tree
        (/ ss z srate crate xscl zscl shrink crop fan small cnt en rness cent brnch long)
        (setvar “cmdecho” 0)
        (setvar “highlight” 0)(command”layer””s””2””f””hidden”””)
        (setq ss (ssget “x”) cnt 0)
        (if (not ss)(princ “\nNothing to erase: “) (command “erase” ss “”))
        (command “redrawall”)
        (command “ucs” “w”)
        (setq zscl 1400 xscl 200 shrink 0.650 crop 0.800 fan 30 small 300)
        (setq rness 5 cent 40 brnch 3 srate 0.910 crate 0.990 )
        (setq long zscl)
        (branch xscl zscl)
        (setq en (entlast))
        (draw en (* shrink xscl)(* crop zscl)(* 0.91 shrink)(* 0.99 crop))
        (command “ucs” “w”)
        (prompt (strcat “\n” (rtos (1+ cnt) 2 0) “ Branches grown. “))(princ))
```