



November 30 - December 3, 2004 ♦ Las Vegas, Nevada

Things Your Mother Never Told You About AutoLISP®

Phil Kreiker - Looking Glass Microproducts

CP41-4 Now that you are acquainted with AutoLISP, we'll introduce you to some of the more subtle and advanced features of a language (LISP) that has become the darling of the Artificial Intelligence community, as well as the basis for many AutoCAD® applications. With the aid of over 200 code snippets, you'll be better equipped to customize AutoCAD with AutoLISP, and you'll be familiar (or at least acquainted) with the read/evaluate/print loop, symbol naming, data types, assignment functions, numeric functions, string literals, string manipulation functions, list functions, association lists, apply, mapcar, user-defined functions, arguments, scoping, conditional functions, and recursion. After this class you'll know more about AutoLISP than most if not all of your co-workers ... and you'll make your mother proud!

About the Speaker:

Phil is Megabyte Master at Looking Glass Microproducts, an Autodesk® Registered Developer, where he is responsible for AutoCAD® training, customization, and support. He taught AutoCAD at the Colorado School of Mines from 1998 to 2003. Phil wrote the —CAD Cookbook— column for 10 years for CADalyst, and was technical editor-at-large for CADENCE magazine from 1996 to 1997. He is the author of several books including his latest Visual LISP®: A Guide to Artful Programming which is part of the Programmer's Series from Autodesk® Press. He received his Bachelor of Engineering degree from Cooper Union for the Advancement of Science and Art and his M.S. degree from Lowell Technological Institute.
phil@lookingglassmicro.com

Part 1 - Strings and Lists

Introduction

Numbers are not all AutoLISP can manipulate. In this section, we'll explore the manipulation of strings and lists using AutoLISP.

Objectives

- Be able to manipulate strings with AutoLISP
- Be able to manipulate lists with AutoLISP

I've got the World on a String

If numbers are the language of computers, strings are the language of people.

String Literals

A string literal is a sequence of zero or more characters enclosed in double quotes. Thus

```
"'Curiouser and curiouser!' cried Alice."
```

is a string literal, as are

```
"Curiouser", "and", "curiouser!", "cried", and "Alice"
```

If you enter a string literal at the Visual LISP Console, the Visual LISP Interpreter evaluates it, and prints its value:

```
_$ "'Curiouser and curiouser!' cried Alice."  
"'Curiouser and curiouser!' cried Alice."
```

If you try to include a double quote (") inside a string literal, AutoLISP will treat it as the terminating double quote. This is typically not what you're looking for.

```
_$ ""Curiouser and curiouser!" cried Alice."  
""  
nil  
#<SUBR @02532adc AND>  
nil  
" cried Alice."
```

If you wish to include a double quote (") inside a string literal, you'll have prefix it with a backslash (\), so AutoLISP doesn't treat it as the ending double quote.

```
_$ "\"Curiouser and curiouser!\" cried Alice."  
"\"Curiouser and curiouser!\" cried Alice."
```

Even though the diad \" looks like two characters, it's really just one character, as far as AutoLISP is concerned.

```
_$ "15 \"characters\""  
"15 \"characters\""  
_$ (strlen "15 \"characters\"")  
15
```

By using the backslash in a string literal, we're telling the Visual LISP Interpreter not to take the next character literally. AutoLISP thus lets us express a number of interesting characters using the backslash, as shown in Table 1.

What You See	What AutoLISP Sees
<code>\"</code>	" character
<code>\e</code>	Escape character
<code>\n</code>	Newline character
<code>\r</code>	Return character
<code>\t</code>	Tab character
<code>\nnn</code>	Character whose octal code is nnn.
<code>\\</code>	\ character

Table 1 -- AutoLISP Backslash Sequences

If you want a backslash (\) in a string literal, you must type it as two backslashes (\\).

To check what a string will look like when displayed to the user, (followed by what it looked like to the programmer, use the princ function as shown

```

_$ "15 \"characters\"
"15 \"characters\"
_$ (princ "15 \"characters\"")
15 "characters""15 \"characters\"

```

The Tab character is used for creating tabular listings, and the Newline character for starting a new line:

```

_$ (princ "How\tCan\tJust\tOne\nCalorie\tTaste\tSo\tGood?")
How   Can   Just   One
Calorie   Taste So   Good? "How\tCan\tJust\tOne\nCalorie\tTaste\tSo\tGood?"

```

As you can see, the princ function returns a value, which is the original string.

Here's a princ function which doesn't.

```

_$ (defun ap-princ (astring) (princ astring) (princ))
AP-PRINC
_$ (ap-princ "15 \"characters\"")
15 "characters"
_$ (ap-princ "How\tCan\tJust\tOne\nCalorie\tTaste\tSo\tGood?")
How   Can   Just   One
Calorie   Taste So   Good?

```

String Functions

AutoLISP gives you just about everything you'll need to pull some strings.

1. `ascii`

Returns the ASCII character code of the first character of a string.

(*ascii String*)

Examples

```
_$ (ascii "a")
```

```
97
```

```
_$ (ascii "A")
```

```
65
```

```
_$ (ascii "Alice")
```

```
65
```

```
_$ (ascii "")
```

```
0
```

```
_$ (ascii (chr 11))
```

```
11
```

An empty string returns zero.

`ascii` and `chr` are inverse functions

- `ascii` is the (approximate) inverse of the `chr` function.

2. `chr`

Returns the single character string based on the supplied ASCII character code.

(*chr Integer*)

- The supplied Integer must be in the range $0 \leq \text{Integer} \leq 255$
- `chr` is the inverse of the `ascii` function.

Examples

```
_$ (chr 97)
```

```
"a"
```

```
_$ (chr 65)
```

```
"A"
```

```
_$ (chr 0)
```

```
""
```

```
_$ (chr (ascii "L"))
```

```
"L"
```

Returns an empty string

`ascii` and `chr` are inverse functions.

Funny you should ASCII

From time to time I wonder just how many people can live normal, productive, happy lives, without the knowledge that ASCII (pronounced as though it unlocked something unpleasant) stands for the American Standard Code for Information Interchange.

I suppose knowing what it stands for is of no utility in using it. Let's just say it's a code (perhaps a code with two pairs of plans) that maps the numbers 0-255 into a like number of printing and non-printing characters (tab, escape, etc.). ASCII isn't the only standard code out there.

EBCDIC (pronounced eb'-sa-dick) as short for the somewhat unrenowned Expanded Binary Coded Decimal Interchange Code, that was quite popular among the mainframes.

If you must perform math on characters, here are some things you should be able to count on, even without looking for an ASCII table.

```

_ $ (chr (+ 0 (ascii "A")))
"A"
_ $ (chr (+ 1 (ascii "A")))
"B"
_ $ (chr (+ 25 (ascii "A")))
"Z"
_ $ (chr (+ 0 (ascii "a")))
"a"
_ $ (chr (+ 1 (ascii "a")))
"b"
_ $ (chr (+ 25 (ascii "a")))
"z"
_ $ (chr (+ 0 (ascii "0")))
"0"
_ $ (chr (+ 1 (ascii "0")))
"1"
_ $ (chr (+ 9 (ascii "0")))
"9"
_ $ (setq lower (- (ascii "a") (ascii "A")))
"32"
_ $ (chr (+ (ascii "J") lower))
"j"

```

Never Never put 32 in here, however tempting it may seem.

String Me Along

By converting a string to a list of codes, you can process the string as a list of character codes.

3. vl-string->list

Converts a string to a list of ASCII codes integers.

(vl-string->list String)

Examples

```

_ $ (vl-string->list "Mad Hatter")
'(77 97 100 32 72 97 116 116 101 114))
_ $ (vl-string->list "")
nil
_ $ (setq a (vl-string->list "Mad Hatter"))
'(77 97 100 32 72 97 116 116 101 114))
_ $ (vl-list->string a)
"Mad Hatter"
_ $ (vl-list->string (reverse a))
"rettaH daM"

```

A empty string returns an empty list.

4. vl-list->string

Converts a list of ASCII codes integers into a string.

(vl-list->string List)

Each element of the supplied list must be a integer in the range $0 \leq \text{element} \leq 255$.

Examples

```
_$ (vl-list->string '(77 97 100 32 72 97 116 116 101 114))  
"Mad Hatter"  
_$ (vl-list->string nil)  
""
```

A empty list returns an empty string.

5. vl-string-elt (string element)

Returns the ASCII code of the character at a specified position in a string

(vl-string-elt String Position)

- The first character of a string is position 0.

Examples

```
_$ (vl-string->list "Mad Hatter")  
(77 97 100 32 72 97 116 116 101 114))  
_$ (vl-string-elt "Mad Hatter" 4)  
72  
_$ (vl-string-elt "Mad Hatter" 44)  
; error: bad argument value: string position out of range 44
```

6. strcase (string case)

If ToLower is absent or nil, strcase converts the given string entirely to uppercase. If ToLower is non-nil, strcase converts the given string entirely to lowercase,

(strcase String [ToLower])

Examples

```
_$ (strcase "Speak roughly to your Little Boy,")  
"SPEAK ROUGHLY TO YOUR LITTLE BOY,"  
_$ (strcase "Speak roughly to your Little Boy," T)  
"speak roughly to your little boy,"
```

7. strcat (string concatenate)

Strings together the supplied strings.

(strcat [String]...)

This is your basic put-all-these-strings-together function.

Examples

```
_$ (strcat "And " "beat " "him " "when " "he "  
"sneezes:")
```

```
"And beat him when he sneezes:"
_ $ (strcat)
""
```

Makes some sense.

8. strlen (string length)

Returns the length in characters of the specified string.

(strlen String)

Examples

```
_ $ (strlen "This string has exactly forty-five characters")
45
_ $ (strlen "This string has almost forty-five characters")
44
```

9. substr (substring)

Returns a portion of the specified string, starting at the specified position.

(substr String StartPos [Length])

- If Length is supplied, substr returns Length characters (unless, of course, it runs out of characters).
- If Length is not supplied, substr returns the rest of the string.
- If StartPos is greater than the length of the string, an empty string is returned.

WARNING: substr starts counting at string position 1, while every vl-string function starts counting at position 0.

Examples

```
_ $ (setq astring "He only does it to annoy,")
"He only does it to annoy,"
_ $ (substr astring 9)
"does it to annoy,"
_ $ (substr astring 9 7)
"does it"
_ $ (substr astring 30)
""
```

Just a Trim, Please

You're never sure, when getting a string from a user, or an Attribute, or an external file, that some extra white space has crept in. The next three functions remove unwanted characters from either or both ends of a string.

10. vl-string-left-trim

Removes the specified characters from the start of a string.

(vl-string-left-trim Characters String)

Example

```
_ $ (vl-string-left-trim " \t" " Mad Hatter ")  
"Mad Hatter "
```

Removes white space from the start of a string.

11. vl-string-right-trim

Removes the specified characters from the end of a string.

(vl-string-right-trim Characters String)

Example

```
_ $ (vl-string-right-trim " \t" " Mad Hatter ")  
" Mad Hatter"
```

Removes white space from the end of a string.

12. vl-string-trim

Removes the specified characters from both ends of a string.

(vl-string-trim Characters String)

Example

```
_ $ (vl-string-trim " \t" " Mad Hatter ")  
"Mad Hatter"
```

Removes white space from both ends of a string.

13. vl-string-mismatch

Returns the length of the longest common prefix for two strings, starting at specified positions.

(vl-string-mismatch String1 String2 [Pos1 [Pos2 [Ignore-case-p]])

- The first character of each string is position 0.
- If Pos1 or Pos2 are not specified, or nil, they default to 0.

Example

```
_ $ (setq file1 "C:\\Program Files\\AutoCAD 2005\\Sample\\colorwh.dwg")  
"C:\\Program Files\\AutoCAD 2005\\Sample\\colorwh.dwg"  
_ $ (setq file2 "C:\\Program Files\\AutoCAD 2005\\Express\\acadinfo.lsp")  
"C:\\Program Files\\AutoCAD 2005\\Express\\acadinfo.lsp"  
_ $ (setq m (vl-string-mismatch file1 file2))  
30  
_ $ (substr file1 1 m)  
"C:\\Program Files\\AutoCAD 2005\\"  
•
```


14. vl-string-position

Returns the position of a character in a string, or nil if it's not found.

(vl-string-position Char-code String [Start-pos [From-end-p]])

- The first character of the string is position 0.
- If Start-pos is not specified, or nil, it defaults to 0.
- If From-end-p is specified, and not nil, AutoLISP will search from the end of the string to Start-pos, otherwise, it will search from Start-pos to the end of the string.

Examples

```

_$ (setq a "'You are old, Father William,' the young man said,")
"'You are old, Father William,' the young man said,"
_$ (vl-string-position (ascii "'") a)
0
The first single-quote is at position 0.

_$ (vl-string-position (ascii "'") a 1)
29
The next single-quote is at position 29.

_$ (vl-string-position (ascii "'") a 30)
nil
And there are none after that.

_$ (vl-string-position (ascii "'") a nil T)
29
The last single-quote is in position 29.

_$ (vl-string-position (ascii "Z") a)
nil
There are no Z's here.

```

15. vl-string-search

Returns the position of a pattern in a string, or nil if it's not found.

(vl-string-search Pattern String [Start-pos])

Examples

```

_$ (setq a "'And your hair has become very white;")
"'And your hair has become very white;"
_$ (vl-string-position "hair" a)
10
The first 'hair' is at position 10.

_$ (vl-string-position "Hair" a)
nil
The search is case sensitive.

_$ (vl-string-position "" a)
0
An empty string will always be found.

```

16. ap-string-search

Returns the position of a pattern in a string, ignoring case, or nil if it's not found.

(vl-string-search Pattern String Start-pos)

- The first character of the string is position 0.

Things Your Mother Never Told You About AutoLISP®

- If Start-pos is nil, it defaults to 0.

Examples

```
_$ (defun ap-string-search (Pattern Astring Start-pos)
    (vl-string-search (strcase Pattern) (strcase Astring) Start-pos)
  )
```

AP-STRING-SEARCH

```
_$ (setq a "And yet you incessantly stand on your head")
```

```
"And yet you incessantly stand on your head"
```

```
_$ (ap-string-search "head" a nil)
```

```
38
```

The first 'head' is at position 38.

```
_$ (ap-string-search "Head" a nil)
```

```
38
```

The search is case-insensitive.

```
_$ (ap-string-search "" a nil)
```

```
0
```

An empty string will always be found..

17. vl-string-subst

Substitutes *New-string* for *Old-string*, within the specified *String*

(vl-string-subst New-string Old-string String [Start-pos])

- The first character of the string is position 0.
- The search is case-sensitive
- Only the first occurrence of the Old-string will be replaced.
- If Start-pos is not specified, or nil, it defaults to 0.

Examples

```
_$ (setq a "Do you think, at your age, it is right?")
```

```
"Do you think, at your age, it is right?"
```

```
_$ (vl-string-subst "You" "you" a)
```

```
"Do You think, at your age, it is right?"
```

Replaces the first occurrence of 'you'.

Tip: An empty string will always be found, allowing you to insert a string into another at the specified location.

```
_$ (vl-string-subst "<really> " "" a 7)
```

```
"Do you <really> think, at your age, it is right?"
```

Inserts a string after the 7th character, counting from 1.

18. vl-string-translate

Replaces one set of characters in a string with another.

(vl-string-translate Source-set Dest-set String)

Examples

```
_$ (setq a "Surrounded with 'Quotes'")
```

```
"Surrounded with 'Quotes'"
```

```
_$ (vl-string-translate "'" "\" a)
"Surrounded with \"Quotes\""
```

Replaces every single quote with a double-quote.

```
_$ (vl-string-translate ",." ".," "1,234.56")
"1.234,56"
```

Swaps commas and periods.

I'm Making a List

Lists comprise the only structured data type in Visual LISP. There are no arrays, no points, no complex numbers, no records, no variants, no structures, just lists. Learn how to manipulate lists, and you've got it all. If you have any experience with programming languages other than LISP, you must put them aside, forget them, and concentrate on manipulating lists.

List Functions

The **car** function returns the first element in a list. The **cdr** function (pronounced cudder) returns a list without its first element.

The names **car** and **cdr** refer to the names of hardware registers used in early implementations of LISP. The names are totally meaningless today, and might be better called **first** and **rest**, or **head** and **tail**, respectively, insofar as they return the first element of a list and the rest of the list. While **car** and **cdr** have been replaced by **first** and **rest** in some dialects of LISP, AutoLISP LISP uses **car** and **cdr**, so you might as well get used to it.

19. car

Returns the first element of a list.

(car List)

20. cdr

Returns a list less its first element

(cdr List)

Examples

```
_$ (setq x '(How doth the little crocodile Improve his shining tail))
(HOW DOTH THE LITTLE CROCODILE IMPROVE HIS SHINING TAIL)
```

```
_$ (car x)
HOW
```

The car function returns the first element of a list.

```
_$ (car nil)
nil
```

The first element of an empty list is nil

```
_$ (cdr x)
(DOTH THE LITTLE CROCODILE IMPROVE HIS SHINING TAIL)
```

The cdr function returns a list without its first element.

```
_$ (cdr nil)
nil
```

The cdr of an empty list is nil.

Things Your Mother Never Told You About AutoLISP®

```
_$_ (car (cdr x))  
DOTH
```

The car of the cdr is the second element of a list.

```
_$_ (car (cdr (cdr x)))  
THE
```

The car of the cdr of the cdr is the third element of a list.

```
_$_ (car (cdr (cdr (cdr x))))  
LITTLE
```

The car of the cdr of the cdr of the cdr is the fourth element of a list.

Put enough cars and cdrs together, and you can get to any element of a list. Because this is done quite often, there are a shortcuts available for the combinations of two through four cars and cdrs, as shown in Table 2.

The Composite	The Meaning
(caar x)	The cAr of the cAr of x
(cadr x)	The cAr of the cDr of x
(cdar x)	The cDr of the cAr of x
(cddr x)	The cDr of the cDr of x
(caaar x)	The cAr of the cAr of the cAr of x
(caadr x)	The cAr of the cAr of the cDr of x
(cadar x)	The cAr of the cDr of the cAr of x
(caddr x)	The cAr of the cDr of the cDr of x
(cdaar x)	The cDr of the cAr of the cAr of x
(cdadr x)	The cDr of the cAr of the cDr of x
(cddar x)	The cDr of the cDr of the cAr of x
(cdddr x)	The cDr of the cDr of the cDr of x
(caaaar x)	The cAr of the cAr of the cAr of the cAr of x
(caaaadr x)	The cAr of the cAr of the cAr of the cDr of x
(caadar x)	The cAr of the cAr of the cDr of the cAr of x
(caaddr x)	The cAr of the cAr of the cDr of the cDr of x
(cadaar x)	The cAr of the cDr of the cAr of the cAr of x

(cadadr x)	The cAr of the cDr of the cAr of the cDr of x
(caddar x)	The cAr of the cDr of the cDr of the cAr of x
(cadddr x)	The cAr of the cDr of the cDr of the cDr of x
(cdaaar x)	The cDr of the cAr of the cAr of the cAr of x
(cdaadr x)	The cDr of the cAr of the cAr of the cDr of x
(cdadar x)	The cDr of the cAr of the cDr of the cAr of x
(cdaddr x)	The cDr of the cAr of the cDr of the cDr of x
(cddaar x)	The cDr of the cDr of the cAr of the cAr of x
(cddadr x)	The cDr of the cDr of the cAr of the cDr of x
(cdddar x)	The cDr of the cDr of the cDr of the cAr of x
(cdddr x)	The cDr of the cDr of the cDr of the cDr of x

Table 2 -- car and cdr Composites

From examining the table, you may rightly conclude that each **a** stands for car, and each **d** stands for cdr.

In AutoLISP, points and vectors are represented as lists of 2 or 3 numbers. You can easily extract the X, Y, and Z components of a point or vector as follows:

```

_$ (setq p '(1.0 2.0 3.0))
(1.0 2.0 3.0)
_$ (car p)
1.0
_$ (cadr p)
2.0
_$ (caddr p)
3.0

```

The car function returns the X component.

The cadr function returns the Y component.

The caddr function returns the Z component.

21. ap-rdc

Returns a list less its last element

(ap-rdc List)

Examples

```
_ $ (defun ap-rdc (alist) (reverse (cdr (reverse alist))))  
AP-RDC
```

```
_ $ (setq x '(How doth the little crocodile Improve his shining tail))  
(HOW DOTH THE LITTLE CROCODILE IMPROVE HIS SHINING TAIL)
```

```
_ $ (cdr x)  
(DOTH THE LITTLE CROCODILE IMPROVE HIS SHINING TAIL)
```

The cdr function returns a list without its first element.

```
_ $ (ap-rdc x)  
(HOW DOTH THE LITTLE CROCODILE IMPROVE HIS SHINING)
```

The ap-rdc function returns a list without its last element.

22. cons

Adds an element to the start of a list.

(cons Element List)

The cons function is your basic list constructor. It adds an element to the start of a list. cons and cdr are complementary functions

nil is said to be the foundation of all lists. If **consing** to an empty list (nit), the cons function creates a single-element list.

Examples

```
_ $ (setq poem nil)  
nil
```

You gotta start somewhere.

```
_ $ (setq poem (cons "Nile" poem))  
("Nile")
```

Add an element.

```
_ $ (setq poem (cons "the" poem))  
("the" "Nile")
```

Add an element.

```
_ $ (setq poem (cons "of" poem))  
("of" "the" "Nile")
```

Add an element.

```
_ $ (setq poem (cons "waters" poem))  
("waters" "of" "the" "Nile")
```

Add an element.

```
_ $ (setq poem (cdr poem))  
("of" "the" "Nile")
```

Remove an element.

```
_ $ (setq poem (cdr poem))  
("the" "Nile")
```

Remove an element.

```
_ $ (setq poem (cdr poem))  
("Nile")
```

Remove an element.

```
_$ (setq poem (cdr poem))
nil
```

Remove an element.

Notice that items added to the head of our list with the **cons** function, and removed from the head of the list with the **cdr** function.

Going Dotty

There's a special kind of list in AutoLISP called a dotted pair. Rather than an oddly decorated fruit, the dotted pair is the result of consing an element onto an atom. As you might guess (or might not, come to think of it), the cdr of a dotted pair is an atom. An example or two might help.

Examples

```
_$ (setq bartlett (cons 10 20))
(10 . 20)
```

A dotted pair is shown with a dot between the pair.

```
_$ (car bartlett)
10
```

```
_$ (cdr bartlett)
20
```

```
_$ (setq sandpear '(sand . pear))
(sand . pear)
```

```
_$ (car sandpear)
SAND
```

```
_$ (cdr sandpear)
PEAR
```

Many functions that accept a list as an argument will not accept a dotted pair. The primary use of dotted pairs is in association lists, which will be covered layer.

23. list

Returns a list of the evaluated expressions.

(list [Expression...])

- Don't confuse the list function with the quote function. The list function evaluates its arguments, and the quote function does not.

Examples

```
_$ (setq a "You are old")
"You are old"
```

```
_$ (setq b "said the youth")
"said the youth"
```

```
_$ (setq c "as I mentioned before ")
"as I mentioned before"
```

```
_$ (list a b c)
```

```
("You are old" "said the youth" "as I mentioned before")
```

```
_$ (quote a b c)
(A B C)
```

Quote does not evaluate its arguments

```
_$ '(a b c)
```

Quote does not

Things Your Mother Never Told You About AutoLISP®

```
(A B C)
_ $ (list)
nil
```

evaluate its arguments
An empty list is nil.

24. append

Joins a number of lists into single list

(append [List...])

- Appending nil (an empty list) to a list has no effect.

Examples

```
_ $ (setq a (list "And" "have" "grown"))
("And" "have" "grown")
_ $ (setq b "most")
"most"
_ $ (setq c (list "uncommonly" "fat"))
("uncommonly" "fat")
_ $ (append a c)
("And" "have" "grown" "uncommonly" "fat")
_ $ (append a b c)
; error: bad argument type: listp "most"
_ $ (append a (list b) c)
("And" "have" "grown" "most" "uncommonly" "fat")
_ $ (append)
nil
```

"most" is not a list.

but (list b) is.

nothing plus nothing is
nothing.

25. last

Returns the last element of a list

(last List)

Examples

```
_ $ (setq fall '("Yet you turned a" "back-somersault in" "at the door"))
("Yet you turned a" "back-somersault in" "at the door")
_ $ (last fall)
"at the door"
_ $ (last '())
nil
```

The last element of an
empty list isn't.

26. length

Returns the number of elements in a list

(length List)

This function generate an error if called with a dotted pair. Use vl-list-length if this is a problem.

Examples

```

_$ (setq a '("Pray" "what is" "the reason" "of that?"))
("Pray" "what is" "the reason" "of that?")
_$ (length a)
4
_$ (length nil)
0
_$ (length '(1 . 2))
; error: bad list: 2

```

Generates an error if called with a dotted pair.

27. vl-list-length

Returns the number of elements in a list

(vl-list-length List)

This function is identical to the list function but does not generate an error when called with a dotted pair.

Examples

```

_$ (setq a '("Pray" "what is" "the reason" "of that?"))
("Pray" "what is" "the reason" "of that?")
_$ (vl-list-length a)
4
_$ (vl-list-length nil)
0
_$ (vl-list-length '(1 . 2))
nil

```

Returns nil if supplied with a dotted pair.

28. nth

Returns the nth element of a list

(nth Index List)

List elements are counted starting at zero. This means that the first element of a list is the 0th, the second is the 1st, and the fifth the 4th. Strangely enough, many programmers consider this normal.

Examples

```

_$ (setq a ('('In my youth,' "said the sage," "as he shook" "his grey"
"locks,")
('('In my youth,' "said the sage," "as he shook" "his grey" "locks,")
_$ (nth 0 a)
'('In my youth,'
_$ (nth 1 a)
"said the sage,"
_$ (nth 2 a)
"as he shook"
_$ (nth 3 a)

```

Equivalent to (car a).

Equivalent to (cadr a).

Equivalent to (caddr a).

Equivalent to (caddrd

Things Your Mother Never Told You About AutoLISP®

```
"his grey "  
_ $ (nth 4 a)  
"locks,"
```

a).

Nope, no equivalent

Why use `car`, `cadr`, `caddr`, and `caddr`, when `nth` is more readable? Because they're faster, and speed is one of the names of the game.

Why use `nth` when `car`, `cadr`, `caddr`, and `caddr` are faster? When the index of the element you wish to return is based on an expression.

That said, allow me to suggest that you don't want to manipulate lists by accessing their individual elements. As you'll discover, there are faster, simpler ways of doing this.

29.reverse

Reverses the elements of a list

(reverse list)

Example

```
_ $ (setq a ('("I" "kept" "all" "my" "limbs" "very" "supple")))  
("I" "kept" "all" "my" "limbs" "very" "supple")  
_ $ (reverse a)  
("supple" "very" "limbs" "my" "all" "kept" "I")
```

Sounds like Yoda,
doesn't it?

Surprisingly enough, I use the `reverse` function quite often when building lists.

As we've seen consing an element onto a list creates a list in the reverse order to which they were consed. Reverse the resulting list, and it's in the (non-reverse) order in which the elements were consed.

I've tried using `append` to accomplish the same thing, but it's slower, since `append` is slower than `cons`, and using `append` means using `list` as well.

Example

```
_ $ (setq a nil)  
nil  
_ $ (setq a (cons "By" a))  
("By")  
_ $ (setq a (cons "use" a))  
("use" "By")  
_ $ (setq a (cons "of" a))  
("of" "use" "By")  
_ $ (setq a (cons "this ointment" a))  
("this ointment" "of" "use" "By")  
_ $ (reverse a)  
("By" "use" "of" "this ointment")
```

Let's try it first with
`cons`.

A total of four `conses`
and one `reverse`.

```
_ $ (setq a nil)  
nil  
_ $ (setq a (append a (list "one")))
```

And here it is with
`append`.

```

("one")
_ $ (setq a (append a (list "shilling")))
("one" "shilling")
_ $ (setq a (append a (list "the")))
("One" "shilling" "the")
_ $ (setq a (append a (list "box")))
("One" "shilling" "the" "box")

```

A total of four appends
and four lists.

30. member

Returns the remainder of a list, starting at the first occurrence of the specified element

(member Element List)

- The element can be the result of any AutoLISP expression.

Examples

```

_ $ (setq a '("Allow" "me" "to" "sell" "you" "a" "couple?"))
("Allow" "me" "to" "sell" "you" "a" "couple?")
_ $ (member "sell" a)
("sell" "you" "a" "couple?")
_ $ (member (strcat "Al" "low") a)
("Allow" "me" "to" "sell" "you" "a" "couple?")
_ $ (member "buy" a)
nil
_ $ (member 1 '(0 1 1 2 3 5 8))
1

```

Returns the remainder
of the list starting at
the first occurrence of
the specified element

31. vl-position

Returns the first position of an element in a list.

(vl-position Element List)

- The element can be the result of *any* AutoLISP expression.
- The first element of the list is in position 0.

If the specified element doesn't exist, the function returns nil.

Examples

```

_ $ (setq a '("Allow" "me" "to" "sell" "you" "a" "couple?"))
("Allow" "me" "to" "sell" "you" "a" "couple?")
_ $ (vl-position "sell" a)
3
_ $ (vl-position (strcat "Al" "low") a)
0
_ $ (vl-position "buy" a)
nil

```

The first element in
the list is 0.

Things Your Mother Never Told You About AutoLISP®

```
_$ (vl-position 1 '(0 1 1 2 3 5 8))  
1
```

Returns the first position of the specified element

32. subst

Replaces every occurrence of an element in a list with a new element.

(subst NewElement OldElement List)

Examples

```
_$ (subst 4 3 '(1 2 3 2 1))  
(1 2 4 2 1)  
_$ (subst 5 1 '(1 2 3 2 1))  
(5 2 4 2 5)
```

Remember, every occurrence of OldElement will be replaced

This last behavior warrants some attention. Specifically, you can't use subst to replace a single element in a list unless you are certain that there are no duplicates. I've seen published programs that do something like this

```
_$ (setq a '(1 2 3 1 2 3))  
(1 2 3 1 2 3)  
_$ (subst 4 (nth 2 a) a)  
(1 2 4 1 2 4)
```

Hoping to replace only the 2th element, but failing miserably.

To replace an indexed element in a list, use ap-set-nth.

33. vl-remove

Removes all instances of the specified element from a list.

(vl-remove Element List)

The key word here is all.

Example

```
_$ (vl-remove 'x '(x c x h x e x s x h x i x r x e x))  
(C H E S H I R E)
```

If I'm Mabel, I'll stay down here!

Although we have yet to explore conditional expressions, the vl-remove-if and vl-remove-if-not functions are such powerful list-manipulators, that they're worth discussing right now.

One of the many conditional functions of AutoLISP is the minusp function, which returns T (passes) if its argument is negative, and returns nil (fails) if its argument isn't negative. Let's take a quick look at it.

Examples

```
_$ (minusp 0)  
nil
```

0 isn't negative.

```

_ $ (minusp -1)          -1 is negative.
T
_ $ (minusp (- 6 4))    2 isn't negative.
nil
_ $ (minusp (- 4 6))    -2 is negative.
T

```

Got it? Good. Now let's look at vl-remove-if and vl-remove-if-not.

34. vl-remove-if

Removes all the elements of a list that pass the specified test function.

(vl-remove-if TestFunction List)

TestFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

We'll be examining the lambda and function functions in the next section.

Examples

```

_ $ (setq a '(0 1 -2 4 -8 16 -32 64 -128))
(0 1 -2 4 -8 16 -32 64 -128)
_ $ (vl-remove-if 'minusp a)
(0 1 4 16 64)

```

Removes all the negative elements from our list.

35. vl-remove-if-not

Removes all the elements of a list that fail the specified test function.

(vl-remove-if-not TestFunction List)

TestFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

We'll be examining the lambda and function functions in the next section.

Examples

```

_ $ (setq a '(0 1 -2 4 -8 16 -32 64 -128))
(0 1 -2 4 -8 16 -32 64 -128)
_ $ (vl-remove-if-not 'minusp a)
(-2 -8 -32 -128)

```

Removes all the non-negative elements from our list.

Feeling out of Sorts

AutoLISP provides us with a number of way for sorting lists.

36. acad_strlsort

Sorts a list of strings by alphabetical order.

(acad_strlsort Strings)

- Case-insensitive, alphabetical sort order.

Tip: This is the one you'll want to use for most, if not all, of your string sorting.

Examples

```
_$ (setq alist ("Teases" "It" "knows" "he" "Because"))
("Teases" "It" "knows" "he" "Because")
_$ (acad_strlsort alist)
("Because" "he" "It" "knows" "Teases")
_$ (< "he" "It")
nil
```

Alphabetical Order.

Standard AutoLISP string comparison order.

37. vl-sort

Sorts a list according to the given comparison function.

(vl-sort List BeforeFunction)

The BeforeFunction is used by the vl-sort function in determining if the first of two list elements should appear before the second in the sorted list.

BeforeFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

We'll be examining the lambda, function , and comparison functions in the next section.

Examples

```
_$ (setq alist '(0 1 -1 2 -2 3 -3 4 -4))
(0 1 -1 2 -2 3 -3 4 -4)
_$ (vl-sort alist '<)
(-4 -3 -2 -1 0 1 2 3 4)
_$ (vl-sort alist '>)
(4 3 2 1 0 -1 -2 -3 -4)
```

We're making a list,

And sorting it,

Twice.

According to the AutoLISP Reference for vl-sort, 'Duplicate elements may be eliminated from the list.'

Examples

```
_$ (vl-sort '(4 3 2 1 0 1 2 3 4) '<)
(0 1 2 3 4)
```

Note that duplicate elements were

removed.

Although I've yet to encounter a situation in which vl-sort didn't remove duplicate elements, the use of the word 'may' scares me into thinking that vl-sort might not always remove duplicate items from the sorted list.

Tip If you don't wish possibly remove duplicate elements, use ap-sort instead of vl-sort.

38. ap-sort

Sorts a list according to the given comparison function, without removing duplicates.

(ap-sort List BeforeFunction)

The BeforeFunction is used by the ap-sort function in determining if the first of two list elements should appear before the second in the sorted list.

BeforeFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

We'll be examining the lambda, function, and comparison functions in the next section.

Examples

```

_ $ (defun ap-sort (alist comparefunc)
      (mapcar '(lambda (index) (nth index alist))
              (vl-sort-i alist comparefunc))
      )
      )

```

AP-SORT

```

_ $ (ap-sort '(4 3 2 1 0 1 2 3 4) '<)
(0 1 1 2 2 3 3 4 4)

```

Duplicate elements were not removed.

```

_ $ (ap-sort '(4 3 2 1 0 1 2 3 4) '>)
(4 4 3 3 2 2 1 1 0)

```

Ditto

39. vl-sort-i

Sorts a list according to the given comparison function, returning the element index numbers.

(vl-sort List BeforeFunction)

The BeforeFunction is used by the vl-sort-i function in determining if the first of two list elements should appear before the second in the sorted list.

BeforeFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

We'll be examining the lambda, function, and comparison functions in the next section.

Examples

```
_ $ (vl-sort-i '("twinkle" "twinkle" "little" "bat") '<)  
(3 2 1 0)
```

Association Lists

If you're new to AutoLISP, you're probably new to association lists. An association list is a list of non-nil lists. A list of non-nil lists becomes an association list when you use the `assoc` function to retrieve one of the elements from the list.

You can think of an association list as collection of values that may be retrieved by keyword.

Association lists are order-insensitive. It's a lot easier to remember, for example that the radius parameter to function is

```
(ap-assoc 'radius parameters),
```

than it would be to remember that the radius is

```
(nth 3 parameters)
```

It also makes your code more readable.

I use association lists quite a bit. You can even set up hierarchical data structures using associations lists whose elements are, in turn, association lists. Let's take a look at association list.

40. `assoc`

Returns, from an association list, the first element whose car is equal to the key

(assoc Key AssociationList)

Examples

```
_ $ (setq a '(("april" "showers") ("may" "flowers") ("paris" "lovers")))  
(("april" "showers") ("may" "flowers") ("paris" "lovers"))  
_ $ (assoc "april" a)  
("april" "showers")  
_ $ (assoc "may" a)  
("may" "flowers")  
_ $ (assoc "paris" a)  
("paris" "lovers")  
_ $ (assoc "june" a)  
nil
```

If the specified key isn't found, `assoc` returns nil.

If you have any atoms in an association list, AutoLISP chokes when it gets to them.

```
_ $ (setq b '(("april" "showers") ("may" "flowers") "bugs" ))  
(("april" "showers") ("may" "flowers") "bugs")  
_ $ (assoc "may" b)  
("may" "flowers")  
_ $ (assoc "june" b)  
; error: bad association list: ("bugs")
```

In general, you're looking for the `cdr` of the retrieved list. That's where `ap-assoc` comes in.

41. ap-assoc

Returns, from an association list, the cdr of first element whose car is equal to the key

(ap-assoc Key AssociationList)

Examples

```

_ $ (defun ap-assoc (key alist) (cdr (assoc key alist)))
AP-ASSOC
_ $ (setq a '(("april" "showers") ("may" "flowers") ("paris" "lovers")))
(("april" "showers") ("may" "flowers") ("paris" "lovers"))
_ $ (ap-assoc "april" a)
("showers")
_ $ (ap-assoc "may" a)
("flowers")
_ $ (assoc "paris" a)
("lovers")

```

If you wish to associate an atom, instead of a list, with a key, use a dotted pair. An association list may consist of both dotted pairs and lists.

```

_ $ (setq c '(("april" "showers") ("may" . "flowers")))
(("april" "showers") ("may" . "flowers"))
_ $ (ap-assoc "april" c)
("showers")
_ $ (ap-assoc "may" c)
"flowers"

```

A list returns a list

A dotted pair returns an atom.

Just one More

To add an item to an association list, just cons it onto the list.

```

_ $ (cons (cons "mayflowers" "pilgrams") a)
(("mayflowers" . "pilgrams") ("april" "showers") ("may" . "flowers"))

```

And Now for Something Completely Different

Thus far, we've examined AutoLISP functions that operate on atoms, and AutoLISP functions that operate on lists. The functions that follow, although strictly speaking operate on lists, are of a different color.

Atoms Need Not Apply**42. apply**

Passes a list of arguments to, and executes, a specified function

(apply PredicateFunction List)

PredicateFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

Things Your Mother Never Told You About AutoLISP®

We'll be examining the lambda, and function functions in the next section.

The apply function allows us to use the elements of a list as the arguments of the specified function. The net result of applying a function to a list is that of consing the function onto the head of the list, and evaluating the resulting expression.

It may not seem like it now, but **apply** is one of the most powerful functions in AutoLISP.

The key point here is that it is not necessary to iterate through a list to process its members. The apply function can do it for you.

Examples

As you recall, the min function accepts any number of arguments, and returns the smallest value. How do we find the smallest value in a list?

```
_ $ (setq alist '(0 1 -1 2 -2 3 -3 4 -4))
(0 1 -1 2 -2 3 -3 4 -4)
_ $ (apply 'min alist)
-4
_ $ (min 0 1 -1 2 -2 3 -3 4 -4)
-4
```

We're making a list

We apply 'min to alist.

Which is the same as evaluating this expression

The strcat function, as you recall, concatenates (joins) any number of strings into a single string.

```
_ $ (setq slist '("Our " "family " "always " "hated " "cats."))
("Our " "family " "always " "hated " "cats.")
_ $ (apply 'strcat slist)
"Our family always hated cats."
_ $ (strcat "Our " "family " "always " "hated " "cats.")
"Our family always hated cats."
```

We apply 'strcat to slist.

Which is the same as evaluating this expression.

We can apply any function that accepts numbers as arguments to a list of numbers.

```
_ $ (setq nlist '(1 2 3 4 5 6))
(1 2 3 4 5 6)
_ $ (apply '+ nlist)
21
_ $ (apply '* nlist)
720
```

Add `em up.

Times `em up.

The apply function even works with lists of lists.

```
_ $ (setq a '((I have answered) (three questions) (and that) (is enough)))
((I HAVE ANSWERED) (THREE QUESTIONS) (AND THAT) (IS ENOUGH))
_ $ (apply 'append a)
(I HAVE ANSWERED THREE QUESTIONS AND THAT IS ENOUGH)
```

Let Me See that Mapcar

43. mapcar

Returns a list of the result of executing a function with the individual elements of a list or lists are

supplied as arguments to the function

(mapcar PredicateFunction List... [List...])

PredicateFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

A Predicate Function is just a function that's not at the beginning of an expression.

We'll be examining the lambda, and function functions in the next section.

The **mapcar** function returns a list of the same length as the shortest list provided as an argument.

The **mapcar** function is the function of choice if you wish to create a list of the same order as the arguments list, or don't need to create a list at all.

It may not seem like it now, but **mapcar**, is one of the most powerful functions in AutoLISP.

The key point here is that it is not necessary to iterate through a list to process its members. The **mapcar** function can do it for you.

I'm sorry if this sound confusing, but some examples may help clear things up.

Examples

<pre>_\$ (setq alist '(0 1 -1 2 -2 3 -3))</pre>	We're making a list
<pre>(0 1 -1 2 -2 3 -3)</pre>	
<pre>_\$ (mapcar 'abs alist)</pre>	Find the absolute
<pre>(0 1 1 2 2 3 3)</pre>	values of the
	elements.

Note that the result is the same as applying the abs function to each of the elements of the list.

```
_$ (list (abs 0) (abs 1) (abs -1) (abs 2) (abs -2) (abs 3) (abs -3))
(0 1 1 2 2 3 3)
```

Let's try some more.

<pre>_\$ (mapcar '1+ alist)</pre>	
<pre>(1 2 0 3 -1 4 -2)</pre>	
<pre>_\$ (setq slist '("Our " "family " "always " "hated " "cats."))</pre>	
<pre>("Our " "family " "always " "hated " "cats.")</pre>	
<pre>_\$ (mapcar 'strcase slist)</pre>	Convert each
<pre>("OUR " "FAMILY " "ALWAYS " "HATED " "CATS.")</pre>	element to
	uppercase
<pre>_\$ (mapcar 'strlen slist)</pre>	Find the string
<pre>(4 7 7 6 5)</pre>	length of each
	element in slist.
<pre>_\$ (apply 'max (mapcar 'strlen slist))</pre>	Find the length of
<pre>7</pre>	the longest string in
	slist.

Things Your Mother Never Told You About AutoLISP®

```
_$ (setq bases '(0 1 2 3))
(0 1 2 3)
_$ (setq powers '(1 2 3 4))
(1 2 3 4)
_$ (mapcar '+ bases powers)
(1 3 5 7)
_$ (mapcar '* bases powers)
(0 2 6 12)
_$ (mapcar 'expt bases powers)
(0 1 8 81)
```

Add'em up

Multiply'em

Raise'em.
That's (0^1 1^2 2^3 3^4)

Part 2 - User-Defined Functions

Introduction

Previously, we have learned to apply AutoLISP's intrinsic functions to numbers, lists, and strings. In this section, we'll start using editor windows in the Visual LISP Integrated Development Environment to write our own AutoLISP functions.

Objectives

- Be able to create your own functions with AutoLISP.
- Understand scope, arguments, and local variables.
- Understand conditional functions and conditional evaluation
- Be able to write recursive functions.

Defun Has Just Begun

44. defun

Defines a function with the specified name, arguments, and local variables

(defun funcname ([argument...] [/ [variable...]] expression [expression...])

45. sqr

Returns the square of the argument

(sqr number)

```
_$ (defun sqr (x) (* x x))
SQR
```

Now, let's dissect the code.

I Came Here for an Argument

The only line of code for `sqr` is a `defun` statement. The `defun` function defines a function.

```
(defun sqr (x) (* x x))
```

The first parameter of the `defun` statement defines the name of the function; in this case, the name of the function is `sqr`.

```
(defun sqr (x) (* x x))
```

The second parameter of the defun function specifies that the sqr function will have one formal parameter, or argument, called x.

```
(defun sqr (x) (* x x))
```

Whenever you evoke the sqr function, AutoLISP evaluates whatever expression follows the name of the function, and binds that value to the symbol x. For example, when you pass the expression

```
(sqr (+ 2 3))
```

to the Visual LISP Interpreter, it evaluates the expression (+ 2 3), and assigns that value, 5, to x.

The remainder of the expressions in the defun statement are evaluated, not now, but each time you evoke the sqr function.

```
(defun sqr (x) (* x x))
```

The **sqr** function returns, as its value, the last expression evaluated. In this case (* x x), or x².

Examples

```
_ $ (sqr (+ 2 3))
```

```
25
```

```
_ $ (sqr -3)
```

```
9
```

That's better.

Scope - Once in the Morning

Our first user-defined function, sqr, accepts a single parameter (or argument), x.

'But what,' you ask, 'has this x to do with any other x in AutoLISP?'

'At the moment,' I reply, with a smirk on my face, 'absolutely nothing.'

Examples

```
_ $ (setq x (sqrt 2))
```

```
1.41421
```

The value of x is displayed to 5 decimal places.

```
_ $ x
```

```
1.41421
```

Just checking...

```
_ $ (sqr x)
```

```
2.0
```

Cool.

```
_ $ (sqr 5)
```

```
25
```

No surprises, here.

```
_ $ x
```

```
1.41421
```

x is unchanged by the sqr function..

Scope - The Persistence of Binding

When we evoke the sqr function, AutoLISP binds the value of its argument (in this case, 5) to x. But this binding is in effect only so long as we are evaluating the sqr function, or any function evoked by the sqr function. When we're done evaluating the sqr function, the binding is discarded, and any existing bindings to x are restored.

We thus say that an argument to a function is *local* to that function, and *global* to any function evoked by that function.

46. Sqr redux

Returns the square of the argument

(sqr number)

Here, we've explicitly assigned a new value to `x`, specifically the square of the original value. Nevertheless, this assignment won't affect `x` once we've exited the `sqr` function.

Examples

```
_$ (defun sqr (x) (setq x (* x x)) x)
SQR
_$ x                                     Just checking...
1.41421
_$ (sqr x)                               Cool.
2.0
_$ x                                     x is unchanged by the
1.41421                                 sqr function..
```

Any symbols not declared as parameters by the `defun` statement are treated as global variables; any changes made to them will persist after the function exits.

47. Sqr re-redux

Returns the square of the argument

(sqr number)

Here, we've explicitly assigned a value to `b`, specifically the square of the original value of `x`. Unlike assignment to the argument `x`, this assignment will affect `b` after we've exited the `sqr` function.

```
_$ (defun sqr (x) (setq b (* x x)) b)
SQR
_$ x                                     Just checking...
1.41421
_$ b                                     Just checking...
nil
_$ (sqr x)                               Cool.
2.0
_$ x                                     x is unchanged by the
1.41421                                 sqr function..
_$ b                                     Oops. b is changed by
2.0                                     the sqr function.
```

A Little off the Side

The modification of a global variable by a function is referred to as a side-effect.

Side-effects can produce some unexpected results, and should, in general be avoided.

Examples

```
_$ (setq b 2)
2
```

```
_ $ (+ (sqr b) b)
8
```

Not quite what we expected..

```
_ $ (+ b (sqr b))
20
```

Even worse..

You expected 6 each time, didn't you. So did I.

How do we avoid side effects? By declaring every variable occurring in a function definition to be local to that function. Here's how.

48. Sqr re-re-redux

Returns the square of the argument

(sqr number)

Not a very big change. Here, we've followed the arguments with a slash (/) to indicate the end of the arguments. Following the slash are the symbols to be declared local to our function, in this case, b.

This prevents modifying the variable b outside of our sqr function, thus avoiding any unwanted side-effects.

Examples

```
_ $ (defun sqr (x / b) (setq b (* x x)) b)
```

```
SQR
```

```
_ $ (setq b 2)
```

```
2
```

```
_ $ (+ (sqr b) b)
```

```
6
```

This looks better.

```
_ $ (+ b (sqr b))
```

```
6
```

And so does this.

If, for some reason, side-effects can not be avoided, be sure to document them in your function headers.

I'll grant you that the use of the local variable b in the preceding example is somewhat contrived. Let's look at a slightly more complicated example.

49. Inscribed-Radius

Returns the radius of a circle inscribed within a triangle

(inscribed radius a b c)

The radius of a circle inscribed in any triangle, whose sides are a, b, and c, is given by the formula

$$r = \frac{\sqrt{s(s-a)(s-b)(s-c)}}{s}, \text{ where } s = \frac{1}{2}(a+b+c).$$

The following code defines a function to return the radius of a circle inscribed in a triangle.

Notice the use of the local variable s in our function.

Things Your Mother Never Told You About AutoLISP®

Note also that we didn't divide $(+ a b c)$ by 2 for two important reasons. Firstly, a , b , and c might be integers, and AutoLISP truncates integer division. I could have divided by 2.0 instead of multiplying by 0.5, but, in general, multiplication is faster than division.

Examples

```
_$ (defun inscribed-radius (a b c / s)
      (setq s (* 0.5 (+ a b c)))
      (/ (sqrt (* s (- s a) (- s b) (- s c))) s)
    )
INSCRIBED-RADIUS
_$ (inscribed-radius 3 4 5)
1.0
```

Be sure to not include a symbol both as an argument and as a local variable the same function. The results can be quite unpredictable.

Lambda Functions

50. lambda

Defines an anonymous function with the specified arguments and local variables

(lambda ([argument...] [/ [variable...]]) expression [expression...])

Lambda (λ) is the 11th letter of the Greek alphabet, which, in my mind, has absolutely nothing to do with why **lambda** functions are called **lambda** functions.

Lambda functions are typically used with the mapcar and apply functions perform a function on a list, when, in the words of the AutoLISP reference, 'the overhead of defining a new function is not justified.'

I think this means something like 'I need to use a short, little function here, and I don't feel like thinking up a name for it.

Suppose we wanted to add 1 to each element of a list. The 1+ function would serve nicely.

Examples

```
_$ (setq alist '(1 1 2 3 5 8))
(1 1 2 3 5 8)
_$ (mapcar '1+ alist)
(2 2 3 4 6 9)
```

But what if we wanted to add 10 to each element of a list. There isn't (yet) a 10+ function, so we'd have to write one before it could be used with the mapcar function.

```
_$ (defun 10+ (a) (+ a 10))
10+
_$ (mapcar '10+ alist)
(11 11 12 13 15 18)
```

This same functionality could be accomplished with the lambda function.

```
_$ (mapcar '(lambda (a) (+ a 10)) alist)
(11 11 12 13 15 18)
```

If we wanted to multiply every element in the list by 2, we could do it as shown:


```
_$ (mapcar '(lambda (a) (* 2 a)) alist)
(2 2 4 6 10 16)
```

We're not limited to defining lambda functions that accept only one argument; anything we can defun, we can lambda.

AutoLISP represents 3D points as lists of three reals. Here's an expression to find the midpoint between two points:

```
_$ (setq p1 '(1.0 1.0 0.0))           The point 1,1,0
(1.0 1.0 0.0)
_$ (setq p2 '(3.0 1.0 0.0))           The point 3,1,0
(3.0 1.0 0.0)
_$ (mapcar '(lambda (x1 x2) (* 0.5 (+ x1 x2))) p1 p2) The midpoint between
(1.0 1.0 0.0)                          p1 and p2.
```

Include this in a defun, and you have a quick and easy midpoint function:

51. midpoint

Returns a point half-way between two points

(midpoint p1 p2)

Examples

```
_$ (defun midpoint (p1 p2)
      (mapcar '(lambda (x1 x2) (* 0.5 (+ x1 x2))) p1 p2)
    )
MIDPOINT
_$ (midpoint p1 p2)           The midpoint between
(1.0 1.0 0.0)                  p1 and p2.
```

Unfortunately, the lambda functions has one drawback when used with AutoLISP: the Visual LISP compiler can't optimize (make more efficient in size and execution time) a lambda function. It can, however, optimize a function function.

52. function

Tells the Visual LISP compiler to link and optimize an argument as if it were a built-in function

(function symbol|lambda-expression)

In order for the Visual LISP compiler to optimize a lambda function, we 'quote' it using the function function instead of the quote function. Here's how we would use the function function with mapcar in the examples we used for the lambda function.

Suppose we wanted to add 1 to each element of a list. The 1+ function would serve nicely.

```
_$ (setq alist '(1 1 2 3 5 8))
(1 1 2 3 5 8)
_$ (mapcar (function 1+) alist)
(2 2 3 4 6 9)
```

But what if we wanted to add 10 to each element of a list. There isn't (yet) a 10+ function, so we'd have to write one before it could be used with the mapcar function.

Things Your Mother Never Told You About AutoLISP®

```
_$ (defun 10+ (a) (+ a 10))
10+
_$ (mapcar (function 10+) alist)
(11 11 12 13 15 18)
```

This same functionality could be accomplished with the lambda function.

```
_$ (mapcar (function (lambda (a) (+ a 10))) alist)
(11 11 12 13 15 18)
```

If we wanted to multiply every element in the list by 2, we could do it as shown:

```
_$ (mapcar (function (lambda (a) (* 2 a))) alist)
(2 2 4 6 10 16)
```

For all practical purposes (impractical purposes are not worth considering), use the function function in place of the quote function, whenever specifying the name of a function.

Part 3 - Logical Expressions and Conditional Evaluation

It's the conditional evaluation of expressions that differentiates an AutoLISP program from script file or a player piano roll. The conditional evaluation of expressions is controlled by logical expressions.

Logical Expressions

What is Truth?

AutoLISP has no concept of true and false. As far as conditional expressions are concerned, things are either **nil**, or they aren't. You could say that **nil** is considered false, and anything else is considered true, but that wouldn't be quite true; it just wouldn't be **nil**. Since every expression either does or does not evaluate to **nil**, you may rightfully conclude that

Every AutoLISP expression may be used as a conditional expression, insofar as it must return a value that is nil or not-nil

If there is no Truth, What is T?

According to the Visual LISP help file, the predefined variable T is 'defined as the constant T. This is used as a non-nil value.'

I chewed on this one for a real long time. I asked my cat, Midnight, about this.:

Phil: What is T?

Midnight: T isn't nil.

Phil: Okay, but what *is* it.

Midnight: What T *is* isn't important. What *is* important about T is what it *isn't*, and what it *isn't*, is nil.

Phil: Then why have it? Why not just use '1' instead.

Midnight: Because that you lead you (or others) to believe that it was to be numeric. Using T is just a way of saying 'I don't care what it is, as long as it isn't nil, and *neither* should you.

With that, she vanished quite slowly, beginning with the end of the tail, and ending with the grin, which remained some time after the rest of it had gone.

53. and (logical and)

Returns T if none of the expressions evaluates to nil, nil otherwise.

*(and [Expression]...)***Examples**

```

_$ (and)
T
_$ (setq a nil p 2 c "spot run")
"spot run"
_$ (and a)
nil
_$ (and p)
T
_$ (and a p)
nil
_$ (and p c)
T

```

Unlike most AutoLISP functions, (and) stops evaluating arguments as soon as one of them evaluates to nil. See Control Structures for more on this.

54. or (logical or)

Returns T if any of the expressions evaluates to a non-nil value, nil otherwise.

*(or [Expression]...)***Examples**

```

_$ (or)
nil
_$ (setq a nil p nil c "spot run")
"spot run"
_$ (or c)
T
_$ (or a p)
nil
_$ (or p c)
T

```

Unlike most AutoLISP functions, (or) stops evaluating arguments as soon as one of them evaluates to a non-nil value. See Control Structures for more on this.

Tying Nots

Given that the (not) and (null) functions return identical results with identical arguments, it's not surprising that most AutoLISPers (myself included) tend to use them interchangeably.

The intent here, though, is that the (not) function returns the logical inverse of its argument, and the (null) function return T no value (or a value of nil) has been bound to a symbol.

While it won't make any difference to your programs, using these two functions as intended should clarify your *intent*, answering the question 'What was I testing for?'

55. not (logical inverse)

Returns T if the expression evaluates to nil, and nil otherwise.

(not Expression)

Examples

```
_$ (setq a nil b T)
T
_$ (not a)
T
_$ (not b)
nil
_$ (not (or a b))
nil
```

56. null

Returns T if the argument is bound to nil, and nil otherwise.

(null Argument)

Examples

```
_$ (setq a nil b '(1 2 3) c "spot run")
"spot run"
_$ (null a)
T
_$ (null b)
nil
_$ (null c)
nil
```

57. boundp

Returns T if the expression is bound to anything but nil, and nil otherwise.

(boundp Expression)

Examples

```
_$ (setq a nil b '(1 2 3) c "spot run")
"spot run"
_$ (boundp a)
nil
_$ (boundp b)
T
_$ (boundp c)
T
```

Any conditional expression will behave exactly the same if you replace the expression you're testing with **(boundp expression)**.

The Meaning of Equality

I've been tripped up more than once by AutoLISP's equality functions. This may be because AutoLISP has unequal three tests for equality.

58. equal

Returns T if the two expressions are equal [within *Fuzz*], nil otherwise.

(equal Expression₁ Expression₂ [Fuzz])

If you specify a value for Fuzz, two numbers (either as atoms or as members of a list) will be considered equal if they are within that value of each other.

Tip: (equal) is the *only* function to use when comparing lists, or when comparing floating-point numbers (reals) for equality.

Examples

```

_ $ (equal '(1 2) '(1 2))
T
_ $ (equal '(1.0 2.0) '(1.01 2.01))
nil
_ $ (equal '(1.0 2.0) '(1.01 2.01) 0.2)
T

```

When two floating-point numbers are computed via different methods, the numbers may not be precisely equal, even if they are theoretically equal.

Examples

_ \$ (setq a (sin (/ pi 4)))	sin 45°
0.707107	
_ \$ (setq b (/ (sqrt 2) 2))	$\sqrt{2}/2$
0.707107	
_ \$ _ \$ (equal a b)	Not quite equal
nil	
_ \$ (- a b)	
-1.11022e-016	
_ \$ (equal a b 1e-10)	But equal to within 10 ⁻¹⁰
T	is good enough for me

Tip: Always use (equal) with a fuzz value whenever comparing reals for equality.

The following two functions don't work with lists. They work with everything else. How come they don't work with lists?

To make matters worse, if you attempt to use them to compare lists, they don't even give you a bad argument type error. You don't get any error at all.

Presumably, these functions are faster for comparing atoms than is the (equal) function. By all means use them, but don't use them with lists.

59. = (equal to)

Returns T if all the arguments are equal, nil otherwise.

(= [expression] [expression]...)

Examples

```
_$ (= 2 2)
```

```
T
```

```
_$ (= 2 3)
```

```
nil
```

```
_$ (= 2 2.0)
```

```
T
```

```
_$ (= 2)
```

```
T
```

```
_$ (= 2 2 2)
```

```
T
```

```
_$ (= 2 2 3)
```

```
nil
```

```
_$ (= "human" "human")
```

```
T
```

```
_$ (= "human" "HUMAN")
```

```
nil
```

Arguments are promoted if required.

A number is equal to itself.

String comparisons are case-sensitive

This function does not work with lists, but gives you no warning.

```
_$ (setq a '(1 2))
```

```
'(1 2)
```

```
_$ (= a '(1 2))
```

```
nil
```

Huh?.

Tip: When comparing reals, always use the (equal) function.

```
_$ (setq a (sin (/ pi 4)))
```

```
0.707107
```

```
_$ (setq b (/ (sqrt 2) 2))
```

```
0.707107
```

```
_$ (= a b)
```

```
nil
```

```
_$ (equal a b 1e-10)
```

```
T
```

sin 45°

$\sqrt{2}/2$

Not quite equal

60. /= (not equal to)

Returns T if no two adjacent arguments are equal, nil otherwise.

(/= Expression [Expression]...)

Examples

```
_$ (/= 1 1)
```

```

nil
_ $ (/= 1 2)
T
_ $ (/= 1)
T

```

A number not equal to nothing.

This function does not work with lists, but gives you no warning. When comparing lists, always use the (equal) function.

```

_ $ (setq a '(1 2))
'(1 2)
_ $ (/= a '(1 2))
T

```

Huh?.

Tip: When comparing reals, always use the (equal) function.

```

_ $ (setq a (sin (/ pi 4)))
0.707107
_ $ (setq b (/ (sqrt 2) 2))
0.707107
_ $ (/= a b)
T
_ $ (not (equal a b 1e-10))
nil

```

sin 45°
 $\sqrt{2}/2$
 Not quite equal

The behavior of the (/=) function with more than two arguments is a bit esoteric.

```

_ $ (/= 1 2 1)
T
_ $ (/= 2 1 1)
nil

```

No two adjacent arguments are equal.
 Any two adjacent arguments are equal.

Expressed in infix notation, (/= a b c d...) is equivalent to (a ≠ b) ∧ (b ≠ c) ∧ (c ≠ d)...

Of all the AutoLISP applications I've written thus far, I've correctly used (/=) only when comparing exactly two arguments.

You could use (/=) to determine if there are no duplicates in a sorted list.

```

_ $ (setq a '("That" "depends" "a" "good" "deal" "on"
             "where" "you" "want" "to" "get" "to"))
("That" "depends" "a" "good" "deal" "on" "where" "you"
 "want" "to" "get" "to")
_ $ (setq b (acad_strlsort a))
("a" "deal" "depends" "get" "good" "on" "That" "to"
 "to" "want" "where" "you")
_ $ (apply '/= b)
nil

```

Sorts the list alphabetically.
 Returns nil because there's at least one duplicate.

61. eq

Returns T if both expressions are bound to the same object, nil otherwise.

(eq expression₁ expression₂)

Despite its name the (eq) function does not compare two expressions for equality; it checks to see if they're bound to the same object. Confused? Just wait.

Examples

```
_ $ (setq a '(1 2))  
(1 2)
```

```
_ $ (setq b '(1 2))  
(1 2)
```

```
_ $ (setq c b)  
(1 2)
```

```
_ $ (eq a c)  
nil
```

Returns nil because a and b, while containing the same value, do not refer to the same list.

```
_ $ (eq b c)  
T
```

Returns T because b and c do refer to the same list.

```
_ $ (eq '(1 2) '(1 2))  
nil
```

Two lists that (equal) determines to be the same may be found to be different according to the (eq) function.

I haven't used (eq) myself, except by accident. Each time I've done so, I've spent a lot of time trying to figure out why things weren't working the way I expected.

I'm sure that someone knows a case for which this function would return something truly meaningful, but I've yet to find one myself. If you do, be sure to use it. Unless you do, be sure you don't.

62. < (less than)

Returns T if each argument is less than the argument to its right, nil otherwise.

(< Expression [Expression]...)

This function works only with numbers and strings as arguments.

Expressed in infix notation, (< a b c...) is equivalent to (a < b) ^ (b < c)...

Examples

```
_ $ (< 1)  
T
```

There may be some truth here, but I don't know what it is.

```
_ $ (setq a 5)  
5
```

```
_ $ (< 1 a)  
T
```

(1 < 7)? Yes.

```
_ $ (< a 7)  
nil
```

(7 < 7)? No


```
_ $ (< 5 a 10)
T
```

(5 < 7) ^ (7 < 10)?
Yes.

63. <= (less than or equal to)

Returns T if each argument is less than or equal to the argument to its right, nil otherwise.
(<= Expression [Expression]...)

This function works only with numbers and strings as arguments.

Expressed in infix notation, (<= a b c...) is equivalent to $(a \leq b) \wedge (b \leq c)$...

Examples

```
_ $ (<= 1)
T
```

There may be some truth here, but I don't know what it is.

```
_ $ (setq a 5)
5
```

```
_ $ (<= 1 a)
T
```

(1 ≤ 7)? Yes.

```
_ $ (<= a 7)
T
```

(7 ≤ 7)? Yes.

```
_ $ (<= 5 a 10)
T
```

(5 ≤ 7) ^ (7 ≤ 10)?
Yes.

I've found this last construct useful in determining if a number falls within a specified range.

64. > (greater than)

Returns T if each argument is greater than the argument to its right, nil otherwise.
(> Expression [Expression]...)

This function works only with numbers and strings as arguments.

Expressed in infix notation, (> a b c...) is equivalent to $(a > b) \wedge (b > c)$...

Examples

```
_ $ (> 1)
T
```

There may be some truth here, but I don't know what it is.

```
_ $ (setq a 5)
5
```

```
_ $ (> 1 a)
nil
```

(1 > 7)? No.

```
_ $ (> a 7)
nil
```

(7 > 7)? No.

```
_ $ (> 10 a 5)
T
```

(10 > 7) ^ (7 > 5)?
Yes.

65. >= (greater than or equal to)

Returns T if each argument is greater than or equal to the argument to its right, nil otherwise.

(>= Expression [Expression]...)

This function works only with numbers and strings as arguments.

Expressed in infix notation, (>= a b c...) is equivalent to $(a \geq b) \wedge (b \geq c) \dots$

Examples

```
_$ (>= 1)  
T
```

There may be some truth here, but I don't know what it is.

```
_$ (setq a 5)  
5
```

```
_$ (>= 1 a)  
nil
```

(1 ≥ 7)? No.

```
_$ (>= a 7)  
nil
```

(7 ≥ 7)? No.

```
_$ (>= 10 a 5)  
T
```

**(10 ≥ 7) □ (7 ≥ 5)?
Yes.**

66. minusp

Returns T if the argument is less than zero, and nil otherwise.

(minusp Number)

Examples

```
_$ (minusp -1)  
T
```

```
_$ (minusp 0)  
nil
```

```
_$ (minusp 1)  
nil
```

67. zerop

Returns T if the argument is zero, and nil otherwise.

(zerop Number)

Examples

```
_$ (zerop -1)  
nil
```

```
_$ (zerop 0)  
T
```

```
_$ (zerop 1)  
nil
```

68. numberp

Returns T if the argument is a number, nil otherwise.

(numberp expression)

Examples

<code>_\$ (numberp 1)</code>	An integer is a number.
<code>T</code>	
<code>_\$ (numberp 1.0)</code>	And so is a real.
<code>T</code>	
<code>_\$ (numberp '(1.0 2.0))</code>	A list of numbers isn't a
<code>nil</code>	number.

69. atom

Returns T if the argument is an atom, nil otherwise.

(atom expression)

Note that nil is both an atom and a list. See vl-consp.

Examples

<code>_\$ (atom 1)</code>	An integer is an atom.
<code>T</code>	
<code>_\$ (atom 'alice)</code>	And so is a symbol.
<code>T</code>	
<code>_\$ (atom nil)</code>	nil is both an atom and
<code>T</code>	a list.
<code>_\$ (atom '(1.0 2.0))</code>	A non-nil list isn't an
<code>nil</code>	atom.

70. listp

Returns T if the argument is a list, nil otherwise.

(listp expression)

Examples

<code>_\$ (listp 1)</code>	An integer is not a list.
<code>nil</code>	
<code>_\$ (listp nil)</code>	nil is both an atom and
<code>T</code>	a list.
<code>_\$ (listp '(1.0 2.0))</code>	A list is a list.
<code>T</code>	

71. vl-consp

Returns T if the argument is non-nil list, nil otherwise.

(vl-consp expression)

Examples

```
_$ (vl-consp 1)
nil
```

An integer is not a non-nil list.

```
_$ (vl-consp nil)
nil
```

nil is not a non-nil list.

```
_$ (vl-consp '(1.0 2.0))
T
```

A non-nil list is a non-nil list.

72. ap-stringp

Returns T if the argument is a string, nil otherwise.

(ap-stringp expression)

Examples

```
_$ (defun ap-stringp (a) (= 'STR (type a)))
AP-STRINGP
```

```
_$ (ap-stringp "twine")
T
```

"twine" is a string

```
_$ (ap-stringp not)
nil
```

But not is not.

73. vl-every

Returns T if the results of executing a function are non-nil for every set of elements in the lists.

(vl-every PredicateFunction List... [List...])

PredicateFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

The time will come when you want to see if some condition is true (or at least, non-nil) for every set of element in one or more lists.

Examples

Suppose we wished to know if every element in a list is a number.

```
_$ (setq alist '(1 2 cat 4))
(1 2 CAT 4)
_$ (setq blist '(1 2 3 4))
(1 2 3 4)
```

We could check each element individually

```
_$ (mapcar 'numberp alist)
(T T nil T)
```

Is each element of the list a number?

```
_$ (mapcar 'numberp blist)
(T T T T)
```

and then see if they're all non-nil.

```
_$ (apply 'and (mapcar 'numberp alist))
nil
```

Not all the elements of alist are numbers.

```
_$ (apply 'and (mapcar 'numberp blist))
T
```

But all the members of blist are.

Or, we can use vl-every to accomplish just about the same thing.

```
_$ (vl-every 'numberp alist)
nil
```

Not all the elements of alist are numbers.

```
_$ (vl-every 'numberp blist)
T
```

But all the members of blist are.

There's one very important difference, however, between these two methods:

vl-every stops evaluating your predicate function as soon as it returns a nil value, while mapcar evaluates your predicate function for every set of elements in your lists.

This results in shorter execution times for vl-every than with mapcar.

If your predicate function has any side effects, be aware that it will not be evaluated after it once evaluates nil.

74. vl-some

Returns T if the results of executing a function are non-nil for any set of elements in the lists.

(vl-some PredicateFunction List... [List...])

PredicateFunction may be any of the following:

- A quoted function name
- A quoted lambda function
- A function function

The time will come when you want to see if some condition is true (or at least, non-nil) for any set of element in one or more lists.

Examples

Suppose we wished to know if any element in a list is a number.

```
_$ (setq alist '(dormouse mad_hatter march_hare))
(DORMOUSE MAD_HATTER MARCH_HARE)
_$ (setq blist '(dormouse mad_hatter 3 march_hare))
(DORMOUSE MAD_HATTER 3 MARCH_HARE)
```

We could check each element individually

```
_$ (mapcar 'numberp alist)
(nil nil nil nil)
```

Is each element of the list a number?

Things Your Mother Never Told You About AutoLISP®

```
_ $ (mapcar 'numberp blist)
(nil nil T nil)
```

and then see if any are non-nil.

```
_ $ (apply 'or (mapcar 'numberp alist))
nil
```

None of the elements of alist is a number.

```
_ $ (apply 'or (mapcar 'numberp blist))
T
```

But some of the members of blist are.

Or, we can use vl-every to accomplish just about the same thing.

```
_ $ (vl-some 'numberp alist)
nil
```

None of the elements of alist is a number.

```
_ $ (vl-some 'numberp blist)
T
```

But some of the members of blist are.

There's one very important difference, however, between these two methods:

vl-some stops evaluating your predicate function as soon as it returns a non-nil value, while mapcar evaluates your predicate function for every set of elements in your lists.

This results in shorter execution times for vl-some than with mapcar.

If your predicate function has any side effects, be aware that it will not be evaluated after it once evaluates to a non-nil value.

Conditional Evaluation

It's the conditional evaluation of expressions that differentiates an AutoLISP program from script file or a player piano roll. The conditional evaluation of expressions is controlled by logical expressions. Perhaps the simplest conditional function is the `if` function.

75. if

Conditionally evaluates expressions.

(if TestExpression ThenExpression [ElseExpression])

The *TestExpression* is evaluated, and if non-nil, the *ThenExpression* is evaluated and returned by the if function. If the *TestExpression* is nil, the optional *ElseExpression* is evaluated and returned. If there is no *ElseExpression*, nil is returned.

Unlike its counterparts in such languages as BASIC, Pascal, FORTRAN, C++, the AutoLISP if returns the value of one of the predicate expressions.

There's program flow control here, insofar as only one of the expressions is evaluated.

Examples

Suppose we wished assign the absolute value of A to B, and for pedagogical purposes we didn't wish to use the abs function.

Non-AutoLISP programmers tend to write conditional expressions like this:

```
_ $ (setq a -2)
-2
_ $ (if (> a 0) (setq b a) (setq b (- a)))
```

2

It took me a while to start writing conditional expressions like this:

```
_ $ (setq b (if (> a 0) a (- a)))
2
```

This saves some typing, and really confounds the Visual BASIC Programmers.

Suppose we just wanted to be grammatical:

```
_ $ (setq n 1)
1
_ $ (strcat (itoa n) " object" (if (= n 1) "" "s") " found.")
"1 object found."
_ $ (setq n 2)
2
_ $ (strcat (itoa n) " object" (if (= n 1) "" "s") " found.")
"2 objects found."
```

What happens if we wish to evaluate more than one expression? That's where `progn` comes in.

76. progn

Evaluates each of the supplied expressions, and returns the value of the last expression.

(progn [Expression...])

The `progn` function lets you use a compound expression whenever a simple expression is called for. Let me say that again. The `progn` function lets you use a compound expression whenever a simple expression is called for.

In this respect, it's a lot like `{` and `}` in C++, `begin` and `end` in Pascal.

```
(if conditional_expression
  (progn
    expression_1
    expression_2
    ...
    expression_n
  )
)
```

In this example, if **conditional_expression** is not **nil** (I didn't say 'true'), then **expression_1** through **expression_n** are evaluated.

Don't forget that **progn** returns the value of the last expression evaluated; in this case **expression_n**, which itself is returned by the **if** function. I've found this last property of **progn** most useful.

77. cond

The primary conditional function in AutoLISP

(cond

```
(test1 expression_1_1 ... expression_1_n)
(test2 expression_2_1 ... expression_2_n)
(test3 expression_3_1 ... expression_3_n)
```

...

```
(testn expression_n_1 ... expression_n_n)  
)
```

Examples

```
_ $ (defun xgetint (prmpnt default)  
      (cond  
        ((getint (strcat prmpnt " <" (itoa default) ">: "))  
         (default))  
        )  
      )  
)  
  
XGETINT  
_ $ (xgetint "\nPick a number" 99)  
Pick a number <99>: 12  
12  
_ $ (xgetint "\nPick a number" 99)  
Pick a number <99>:  
99
```

78. and (logical and)

Returns T if none of the expressions evaluates to nil, nil otherwise.

(and [Expression]...)

Unlike most AutoLISP functions, (and) stops evaluating arguments as soon as one of them evaluates to nil. See Control Structures for more on this.

Examples

```
_ $ (defun c:points ( / p1 p2)  
      (and (setq p1 (getpoint "\nSpecify p1: "))  
           (setq p2 (getpoint "\nSpecify p2: "))  
           (princ "\nTwo points specified."))  
      )  
      (princ)  
    )  
)
```

C:POINTS

Command: points

Press Enter

Specify p1:

Command:

Command: points

Specify p1, then press
Enter

Specify p1:

Specify p2:

Command:

Command: points

Specify p1 and p2.

Specify p1:

Specify p2:

Two points specified.

Command

79. or (logical or)

Returns T if any of the expressions evaluates to a non-nil value, nil otherwise.

*(or [Expression]...)**Unlike most AutoLISP functions, (or) stops evaluating arguments as soon as one of them evaluates to a non-nil value. See Control Structures for more on this.***Examples**

```

_$ (defun c:pick (/ x)
    (or
      (setq x (getint "\nAn integer: "))
      (setq x (getreal "\nA real: "))
      (setq x (getpoint "\nA point: "))
    )
    (print x)
    (princ)
  )
C:PICK
Command: pick
An integer: 2
2
Command: pick
An integer:
A real: 5
5.0
Command: pick
An integer:
A real:
A point: 1,1
(1.0 1.0 0.0)
Command: pick
An integer:
A real:
A point:
Nil

```

80. repeat

Evaluates a list of expressions a specified number of times, and returns the value of the last expression evaluated.

*(repeat [IntegerExpression][Expression...])***Examples**

```

_$ (defun oneton (n / a)
    (repeat n

```

Things Your Mother Never Told You About AutoLISP®

```
(setq a (cons n a)
      n (1- n)
)
)
a
)
ONETON
_$ (oneton 5)
(1 2 3 4 5)
```

81. repeat

Repeats a list of expressions a specified number of times, and returns the value of the last expression.

(repeat [IntegerExpression][Expression...])

Sorry, I couldn't resist.

82. while

Repeats a list of expressions as long as a test condition is true, and returns the value of the last expression.

(while [TestExpression][Expression...])

Examples

```
_$ (defun myreverse (a / b)
    (while a
      (setq b (cons (car a) b))
      (setq a (cdr a))
    )
    b
  )
MYREVERSE
_$ (myreverse '(a b c d))
(D C B A)
```

Part 4 - Recursion: See Recursion.

Recursion is the essence of programming, because it trades execution speed for description.

If this were not the case, programs wouldn't have loops or branches.

83. generic recursive function

A recursive function is a function which calls itself, on the premise that it has already been written.

A recursive function is a function which calls itself, on the premise that it has already been written.

Every recursive function has two parts:

A terminal condition, which eventually terminates the function.

A (recursive) call to the function with some modified values of the arguments.

Together, these should guarantee that the function terminates.

Examples

```
(defun <recursive function> (args)
  (cond
    (<terminal condition> <terminal value>)
    ((<recursive function><function of args>))
  )
)
```

84. factorial

Returns the product of the numbers 1..number.

(factorial number)

Examples

```
_ $ (defun factorial (n)
  (cond
    ((= n 1) 1)
    ((* n (factorial (1- n))))
  )
)
FACTORIAL
_ $ (factorial 2)
2
_ $ (factorial 3)
6
_ $ (factorial 4)
24
_ $ (trace factorial)
FACTORIAL
_ $ (factorial 4)
24
Entering (FACTORIAL 4)
  Entering (FACTORIAL 3)
    Entering (FACTORIAL 2)
      Entering (FACTORIAL 1)
        Result: 1
      Result: 2
    Result: 6
  Result: 24
```

85. mergesort

Sorts a list according to the given comparison function.

(mergesort List BeforeFunction)

The BeforeFunction is used by the vl-sort function in determining if the first of two list elements should appear before the second in the sorted list.

BeforeFunction may be any of the following:

- A quoted function name
- A quoted **lambda** function
- A **function** function

Examples

```
_ $ (defun mergesort (a before / b c d)
  (cond
    ((null (cdr a)) a)
    (T
     (setq b (split a)
           c (car b)
           d (cadr b)
           )
     (merge (mergesort c before) (mergesort d before) before)
     )
  )
)
```

MERGESORT

Note the all important terminating condition: When a is a single-element (or null element) list, it's sorted.

Otherwise, we split the list to be sorted in two, and merge the two sorted halves. It doesn't look like it should work, but it does. The key to writing a recursive routine is to assume the routine is already written, then write it.

86. split

Split a list into two equal segments.

(split List)

Examples

```
_ $ (defun split (a / b)
  (repeat (/ (length a) 2)
    (setq b (cons (car a) b))
    (setq a (cdr a))
  )
  (list (reverse b) a)
)
```

```

)
SPLIT
_ $ (split '(1 2 3 4 5 6))
((1 2 3) (4 5 6))

```

87.merge

Merge two sorted lists.

(merge List List)

Examples

```

_ $ (defun merge (a b before / m)
  (while (and a b)
    (if (eval (list before (car a) (car b)))
      (setq m (cons (car a) m)
            a (cdr a))
      (setq m (cons (car b) m)
            b (cdr b))
    )
  )
  (append (reverse m) a b)
)
MERGE
_ $ (merge '(2 4 5) '(1 3 6) '<)
(1 2 3 4 5 6)

```

Things Your Mother Never Told You About AutoLISP®

```
_$_ (mergesort '(6 1 5 2 4 3) '<)  
(1 2 3 4 5 6)  
Entering (MERGESORT (6 1 5 2 4 3) <)  
  Entering (MERGESORT (6 1 5) <)  
    Entering (MERGESORT (6) <)  
    Result: (6)  
  Entering (MERGESORT (1 5) <)  
    Entering (MERGESORT (1) <)  
    Result: (1)  
    Entering (MERGESORT (5) <)  
    Result: (5)  
  Result: (1 5)  
Result: (1 5 6)  
Entering (MERGESORT (2 4 3) <)  
  Entering (MERGESORT (2) <)  
  Result: (2)  
  Entering (MERGESORT (4 3) <)  
    Entering (MERGESORT (4) <)  
    Result: (4)  
    Entering (MERGESORT (3) <)  
    Result: (3)  
  Result: (3 4)  
Result: (2 3 4)  
Result: (1 2 3 4 5 6)
```