XLISP 2.0 LANGUAGE REFERENCE

by

Tim I Mikkelsen

December 11, 1989

This document is intended to serve as a reference for the XLISP 2.0 dialect of LISP. It includes a description of each symbol, function, special form and keyword available in XLISP. This reference is not a complete and extensive introduction to LISP programming.

If you find problems with the reference or find that I have left something out, drop me a line. If you find this useful, I would be interested in hearing that as well. If you are into 'pretty' looking documents (as oppossed to plain ASCII text), I have a TeX version of the reference.

Tim Mikkelsen
4316 Picadilly Drive
Fort Collins, Colorado 80526

Each entry is a symbol of some variety that the XLISP system will recognize. The parts of each reference entry include:

Name This top line gives the name or symbol of the entry. The reference has the entries in alphabetical order.

Type The entry type may be one of the following:

- function (subr)
- predicate function (subr)
- special form (fsubr)
- reader expansion
- defined function (closure)
- defined macro (closure)
- system variable
- system constant
- keyword
- object
- message selector

Location This line specifies if the entry is built-in to the system or an extension.

Source file This line specifies the source file where the routine or code associated with the entry resides. If the entry is an extension, it specifies the source file (usually "init.lsp").

Common LISP compatable This line specifies whether the entry is compatable with the defintion of Common LISP. There are four levels:

yes - compatable with Common LISP.
similar - compatable, some differences.
related - related, major differences.
no - not compatable.

Supported on This line specifies machine dependencies. A few features are available only on PCs or on Macintoshes. (Note that I have not included the Macintosh specific graphics commands.)

Syntax This area defines the syntax or usage of the entry. It is also used to specify the arguments. Items that are enclosed between a < and a > are arguments. Items enclosed between [and] are optional entries. Items that have ... (elipses) indicate that there can be one or many of the item. Items enclosed between { and } which are separated by | indicate that one of the items should be included.

Description This defines the entry, necessary conditions,

results, defaults, etc.

Examples This area shows example uses of the entry.

Comments This area includes additional information such as compatability notes, bugs, usage notes, potential problems, keystroke equivalences, etc.

*

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(* <expr1> ... )
     <exprN>          -     integer or floating point number/expression
```

DESCRIPTION

The multiply (*) function multiplies a list of numbers together and returns the result.

EXAMPLES

```
     (* 1)                        ; returns 1
     (* 1 2)                          ; returns 2
     (* 1 2 3)                      ; returns 6
     (* 1 2 3 4)                    ; returns 24

     (print (+ 1 2 (* 3.5 (/ 3.9 1.45))))    ; returns and prints 12.4138
```

*

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

*

DESCRIPTION

The * variable is set to the result of the previously evaluated expression.

EXAMPLES

```
(setq a 'b)                ; returns B
*                          ; returns B
*                          ; returns B
```

NOTE:
It is best not to use this variable in a program.

**

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

**

DESCRIPTION

The ** variable is set to the result of the next to the last evaluated expression.

EXAMPLES

```
(setq fee 'fi)                    ; returns FI
(setq fo 'fum)                    ; returns FUM
**                              ; returns FI
**                              ; returns FUM
**                              ; returns FI
```

NOTE:
It is best not to use this variable in a program.

***

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

***

DESCRIPTION

The *** variable is set to the result of the second to the last evaluated expression.

EXAMPLES

```
(setq a 'eenie)                  ; returns EENIE
(setq b 'meenie)               ; returns MEENIE
(setq c 'beanie)               ; returns BEANIE
***                            ; returns EENIE
***                            ; returns MEENIE
***                            ; returns BEANIE
***                            ; returns EENIE
```

NOTE:
It is best not to use this variable in a program.

+

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(+ <expr1> ... )
     <exprN>          -     integer or floating point number/expression
```

DESCRIPTION

The add (+) function adds a list of numbers together and returns the result.

EXAMPLES

```
     (+ 1)                        ; returns 1
     (+ 1 2)                           ; returns 3
     (+ 1 2 3)                    ; returns 6
     (+ 1 2 3 4)                  ; returns 10

     (print (+ 1 2 (* 3.5 (/ 3.9 1.45))))    ; returns and prints 12.4138
```

+

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

+

DESCRIPTION

The + variable is set to the most recent input expression.

EXAMPLES

```
(setq hi 'there)            ;returns THERE
+                           ;returns (SETQ HI (QUOTE THERE))
+                           ;returns +
```

NOTE:
It is best not to use this variable in a program.

++

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

++

DESCRIPTION

The ++ variable is set to the next to the last input expression.

EXAMPLES

```
(setq fee 'fi)                    ; returns FI
(setq fo 'fum)                    ; returns FUM
++                              ; returns (SETQ FEE (QUOTE FI))
++                              ; returns (SETQ FO (QUOTE FUM))
++                              ; returns ++
```

NOTE:
It is best not to use this variable in a program.

+++

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

+++

DESCRIPTION

The +++ variable is set to the second to the last input expression.

EXAMPLES

```
(setq a 'eenie)                  ;returns EENIE
(setq b 'meenie)              ;returns MEENIE
(setq c 'beanie)              ;returns BEANIE
+                             ;returns (SETQ C (QUOTE BEANIE))
++                            ;returns (SETQ C (QUOTE BEANIE))
+++                           ;returns (SETQ C (QUOTE BEANIE))
+                             ;returns +
```

NOTE:
It is best not to use this variable in a program.

-

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(- <expr1> ...)
<exprN> - integer or floating point number/expression

DESCRIPTION

The subtract (-) function subtracts a list of numbers from the first number in the list and returns the result. If there is only one number as an argument, it is negated.

EXAMPLES

```
(- 1)                          ; returns -1
(- 1 2)                             ; returns -1
(- 1 2 3)                      ; returns -4
(- 1 2 3 4)                    ; returns -8

(print (+ 1 2 (* 3.5 (/ 3.9 1.45))))    ; returns and prints 12.4138
```

-

---

type: variable
location: built-in
source file: xlinit.c xlisp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

-

DESCRIPTION

The - variable is set to the expression currently being evaluated.

EXAMPLES

```
-                           ; returns -
(setq a -)                  ; returns (SETQ A -)
a                           ; returns (SETQ A -)
```

NOTE:
It is best not to use this variable in a program.

/

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(/ <expr1> ...)
<exprN> - integer or floating point number/expression

DESCRIPTION

The divide (/) function divides the first number in the list by the rest of the numbers in the list and returns the result. If all the expressions are integers, the division is integer division. If any expression is a floating point number, then the division will be floating point division.

EXAMPLES

```
(/ 1)                       ; returns 1
(/ 1 2)                          ; returns 0 (integer division)
(float (/ 1 2))                  ; returns 0 (integer division)
(/ (float 1) 2)                  ; returns 0.5
(/ 1 1.0 2)                 ; returns 0.5     (short cut)
(/ (float 1) 2 3)           ; returns 0.166667
(/ 1 1.0 2 3 4)                  ; returns 0.0416667

(print (+ 1 2 (* 3.5 (/ 3.9 1.45))))    ; returns and prints 12.4138
```

COMMON LISP COMPATABILITY:
Common LISP supports a ratio data type. This means that (/ 3 4 5) will result in the value 3/20. In XLISP (/ 3 4 5) will result in 0 (because of integer values). (/ 3.0 4 5) will result in 0.15 for both XLISP and Common LISP.

NOTE:
An easy way to force a sequence of integers to be divided as floating point numbers is to insert the number 1.0 after the first argument in the list of arguments to the divider function.

/=

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(/= <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The /= (NOT-EQUAL) operation takes an arbitrary number of numeric arguments. It checks to see if all the numeric arguments are different. T is returned if the arguments are numerically not equivalent, NIL is returned otherwise.

EXAMPLES

```
      (/= 1 1)                    ; returns NIL
      (/= 1 2)                    ; returns T
      (/= 1 1.0)                  ; returns NIL
      (/= 1 2 3)                  ; returns T
      (/= 1 2 2)                  ; returns NIL

      (/= "a" "b")                      ; error: bad argument type
      (setq a 1) (setq b 12.4)          ; set up A and B with values
      (/= a b)                    ; returns NIL
```

BUG:
The XLISP /= (NOT-EQUAL) function checks to see if the each argument is different from the next in the list. This means that (/= 1 2 3) returns T as it is supposed to, but that (/= 1 2 3 2 1) returns T when it should return NIL. This is only a problem for the /= (NOT-EQUAL) function.

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(1+ <expr>)
      <expr>   -   integer or floating point number/expression

DESCRIPTION

The increment (1+) function adds one to a number and returns the result.

EXAMPLES

```
      (1+ 1)                          ; returns 2
      (1+ 99.1)                    ; returns 100.1
      (1+ 1 2)                     ; error: too many arguments
```

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(1- <expr>)
      <expr>   -   integer or floating point number/expression

DESCRIPTION

The decrement (1-) function subtracts one from a number and returns the result.

EXAMPLES

```
      (1- 1)                          ; returns 0
      (1- 99.6)                    ; returns 98.6
      (1- 1 2)                     ; error: too many arguments
```

<

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(< <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The < (LESS-THAN) operation takes an arbitrary number of numeric arguments. It checks to see if all the numbers are monotonically increasing. T is returned if the arguments are numerically, monotonically increasing, , NIL is returned otherwise. In the case of two arguments, this has the effect of testing if <expr1> is less than <expr2>.

EXAMPLES

```
      (< 1 2)                          ; returns T
      (< 1 1)                          ; returns NIL
      (< -1.5 -1.4)                    ; returns T
      (< 1 2 3 4)                  ; returns T
      (< 1 2 3 2)                  ; returns NIL

      (< "a" "b")                  ; error: bad argument type
      (setq a 12) (setq b 13.99)       ; set up A and B with values
      (< a b)                          ; returns T
      (< b a)                          ; returns NIL
```

<=

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(<= <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The <= (LESS-THAN-OR-EQUAL) operation takes an arbitrary number of numeric arguments. It checks to see if all the numbers are monotonically non-decreasing. T is returned if the arguments are numerically, monotonically non-decreasing, NIL is returned otherwise. For two arguments, this has the effect of testing if <expr1> is less than or equal to <expr2>.

EXAMPLES

```
(<= 1 1)                    ; returns T
(<= 1 2)                    ; returns T
(<= 2.0 1.99)                    ; returns NIL
(<= 1 2 3 3)                     ; returns T
(<= 1 2 3 3 2)                   ; returns NIL

(<= "aa" "aa")                   ; error: bad argument type
(setq a 12) (setq b 999.999)     ; set up A and B with values
(<= a b)                    ; returns T
(<= b a)                    ; returns NIL
```

=

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(= <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The = (EQUALITY) operation takes an arbitrary number of numeric arguments. It checks to see if all the numbers are equal. T is returned if all of the arguments are numerically equal to each other, NIL is returned otherwise.

EXAMPLES

```
      (= 1 1)                          ; returns T
      (= 1 2)                          ; returns NIL
      (= 1 1.0)                   ; returns T
      (= 1 1.0 1 (+ 0 1))              ; returns T
      (= 1 1.0 1.00001)           ; returns NIL

      (= "a" "b")                 ; error: bad argument type
      (setq a 1) (setq b 1.0)                ; set up A and B with values
      (= a b)                          ; returns T
```

>

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(> <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The > (GREATER-THAN) operation takes an arbitrary number of numeric arguments. It checks to see if all the numbers are monotonically decreasing. T is returned if the arguments are numerically, monotonically decreasing, NIL is returned otherwise. For two arguments, this has the effect of testing if <expr1> is greater than <expr2>.

EXAMPLES

```
(> 1 1)                          ; returns NIL
(> 1 2)                          ; returns NIL
(> 2.0 1.99)                     ; returns T
(> 3 2 1)                   ; returns T
(> 3 2 2)                   ; returns NIL

(> "aa" "aa")                    ; error: bad argument type
(setq a 12) (setq b 999.999)     ; set up A and B with values
(> a b)                          ; returns NIL
(> b a)                          ; returns T
```

>=

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(>= <expr1> <expr2> ...)
      <exprN>   -   a numeric expression

DESCRIPTION

The >= (GREATER-THAN-OR-EQUAL) operation takes an arbitrary number of numeric arguments. It checks to see if all the numbers are monotonically non-increasing. T is returned if <expr1> is the arguments are numerically, monotonically non-increasing, NIL is returned otherwise. For two arguments, this has the effect of testing if <expr1> is greater than or equal to <expr2>.

EXAMPLES

```
(>= 1 2)                    ; returns NIL
(>= 1 1)                    ; returns T
(>= -1.5 -1.4)                   ; returns NIL
(>= 3 2 1)                  ; returns T
(>= 3 2 2)                  ; returns T
(>= 3 2 3)                  ; returns NIL

(>= "aa" "abc")                  ; error: bad argument type
(setq a 12) (setq b 13.99)       ; set up A and B with values
(>= a b)                    ; returns NIL
(>= b a)                    ; returns T
```

abs

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(abs <expr>)
     <expr> - integer or floating point number/expression

DESCRIPTION

The ABS function finds the absolute value of a number and returns the result.

EXAMPLES

```
(abs 1)                          ; returns 1
(abs -99)                   ; returns 99
(abs -99.9)                 ; returns 99.9
(abs -32768)                     ; returns 32768
```

COMMON LISP COMPATABILITY:
Common LISP supports a complex number data type which is not supported in XLISP.

address-of

---

type: function (subr)
location: built-in
source file: xlsys.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(address-of <expr>)
     <expr>   -   an expression

DESCRIPTION

The ADDRESS-OF function returns the internal memory address of the XLISP node that corresponds to <expr>. The value returned is an integer.

EXAMPLES

```
     (setq var 0)                      ; set up VAR with 0
     (address-of var)             ; returns 123224
     (address-of 'var)            ; returns 182638
     (peek (address-of var))                ; returns 83951616
     (peek (1+ (address-of var)))      ; returns 16777216
     (peek (+ 2 (address-of var)))          ; returns 0  <-- value of
VAR
     (setq var 14)                     ; change the value to 14
     (peek (+ 2 (address-of var)))          ; returns 14
     (setq var 99)                     ; change the value to 99
     (peek (+ 2 (address-of var)))          ; returns 99
```

CAUTION:
Be careful when modifying the internal state of XLISP. If you have modified it, it would be a good idea to exit XLISP and re-enter before doing any work you really want to retain.

alloc

type: function (subr)
location: built-in
source file: xldmem.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(alloc <size>)
<size> - an integer expression

DESCRIPTION

The ALLOC function changes the number of memory nodes allocated per segment whenever memory is expanded. The previous number of nodes allocated per segment is the value returned as the result. The power up default if 1000 nodes per segment. Note that ALLOC does not, itself, expand memory. You need to execute the EXPAND function to do the expand operation.

EXAMPLES

```
(room)                          ; prints  Nodes:       4000
                                ;       Free nodes:  1669
                                ;       Segments:    4
                                ;         Allocate:    1000
                                ;       Total:       52570
                                ;       Collections: 8
                                ; returns NIL
(alloc 2000)                    ; returns 1000
(room)                          ; prints  Nodes:       4000
                                ;       Free nodes:  1655
                                ;       Segments:    4
                                ;       Allocate:    2000
                                ;       Total:       52570
                                ;       Collections: 8
                                ; returns NIL
```

and

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(and  [ <expr1> ... ] )
     <exprN>          -     an expression
```

DESCRIPTION

The AND special form evaluates a sequence of expressions and returns the effect of a logical AND on the expressions. If, at any point, an expression is NIL, NIL is returned as AND's result. If all of the expressions have a non-NIL value, the last expression's value is returned as the result. Evaluation of the expressions will stop when an expression evaluates to NIL, none of the subsequent expressions will be evaluated. If there are no expressions, AND returns T as its result.

EXAMPLES

```
(and T "boo" "hiss" T "rah")      ; returns "rah"
(and T T T T)                     ; returns T
(and)                        ; returns T
(and (princ "hi") NIL (princ "ho"))     ; prints  hi and returns NIL
(and (princ "hi") (princ " de ")  ; prints  hi de ho
             (princ "ho"))        ;   returns "ho"

(setq a 5)   (setq b 6)                ; set up A and B
(if (and (numberp a)              ; if      A is a number
       (numberp b)                ;     and B is a number
       (< a b) )             ;     and A<B
  (print "A is less than B")      ;   THEN do this
  (print "something else happened"))    ;   ELSE do this
                              ; prints "A is less than B"
```

:answer

---

type: message selector
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(send <class> :answer <message> <args> <code> )
      <class>          -     an existing class
      <message>   -     the message symbol
      <args>           -     formal argument list to the <msg> method
                       of the same form as a lambda argument list
      <code>           -     a list containing the method code
```

DESCRIPTION

The :ANSWER message selector adds or changes a method in the specified <class>. This method consists of the <message> selector symbol, the <arg> formal argument list and the executable code associated with the <message>.

EXAMPLES

```
      (setq myclass (send class :new '(var))) ; create MYCLASS with VAR
      (send myclass :answer :isnew '()  ; set up initialization
            '((setq var nil) self))
      (send myclass :answer :set-it '(value)  ; create :SET-IT message
            '((setq var value)))
      (send myclass :answer :mine '()   ; create :MINE message
            '((print "hi there")))
      (setq my-obj (send myclass :new)) ; create MY-OBJ of MYCLASS
      (send my-obj :set-it 5)                ; VAR is set to 5
      (send my-obj :mine)               ; prints  "hi there"
```

NOTE:
When you define a <message> in a <class>, the <message> is only valid for instances of the <class> or its sub-classes. You will get an error if you try to send the <message> to the <class> where it was first defined. If you wish to add a <message> to the <class>, you need to define it in the super-class of <class>.

MESSAGE STRUCTURE:
The normal XLISP convention for a <message> is to have a valid symbol preceeded by a colon like :ISNEW or :MY-MESSAGE. However, it is possible to define a <message> that is a symbol without a colon, but this makes the code less readable.

append

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(append [<expr1> ...])
     <exprN>   -   a list or list expression

DESCRIPTION

The APPEND function takes an arbitrary number of lists and splices them together into a single list. This single list is returned. If an empty list NIL is appended, it has no effect - it does not appear in the final list. (Remember that '(NIL) is not an empty list.) If an atom is is appended, it also has no effect and will not appear in the final list.

EXAMPLES

```
(append)                      ; returns NIL
(append 'a 'b)                     ; returns B
(append '(a) '(b))                 ; returns (A B)
(append 'a '(b))              ; returns (B)
(append '(a) 'b)              ; returns (A . B)
(append '(a) nil)             ; returns (A)
(append (list 'a 'b) (list 'c 'd)); returns (A B C D)
(append '(a (b)) '(c (d)))         ; returns (A (B) C (D))
(append '(a) nil nil nil '(b))          ; returns (A B)
(append '(a) '(nil) '(b))          ; returns (A NIL B)
```

apply

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(apply <function> <args>)
      <function> -   the function or symbol to be applied to <args>
      <args>   -   a list that contains the arguments to be
passed
                       to <function>

DESCRIPTION

APPLY causes <function> to be evaluated with <args> as the parameters. APPLY returns the result of <function>. <args> must be in the form of a list.

EXAMPLES

```
      (defun my-add (x y)               ; create MY-ADD function
            (print "my add")
            (+ x y))
      (my-add 1 2)                      ; prints "my add" returns 3
      (apply 'my-add '(2 4))            ; prints "my add" returns 6
      (apply 'my-add 1 2)               ; error: bad argument type
      (apply 'my-add '(1 2 3))          ; error: too many arguments

      (apply (function +) '(9 10))      ; returns 19
      (apply '+ '(4 6))           ; returns 10
      (apply 'print '("hello there"))         ; prints/returns "hello
there"
      (apply 'print "hello there")      ; error: bad argument type
```

NOTE:
Note that when using APPLY to cause the evaluation of a system function, you can use the quoted name of the function (like 'PRINT in the examples). You can also use the actual function (like (FUNCTION +) in the examples).

applyhook

---

type: system variable
location: built-in
source file: xlglob.c (not implemented)
Common LISP compatible: similar
supported on: all machines

SYNTAX

applyhook

DESCRIPTION

APPLYHOOK is a system variable that exists and is initialized to NIL. It is a hook that is intended to contain a user function that is to be called whenever a function is applied to a list of arguments. It is not, however, implemented in XLISP 2.0 - it only exists as a dummy hook.

EXAMPLES

```
      *applyhook*                  ; returns NIL
```

COMMON LISP COMPATABILITY:
APPLYHOOK is defined in Common LISP and is often used to implement function stepping functionality in a debugger.

aref

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(aref <array> <element> )
     <array>    -    specified array
     <element>  -    the element number to be retrieved
```

DESCRIPTION

AREF returns the specified element out of a previously created array. Array elements may be any valid lisp data type - including lists or arrays. Arrays made by MAKE-ARRAY and accessed by AREF are base 0. This means the first element is accessed by element number 0 and the last element is accessed by element number n-1 (where n is the array size). Array elements are initialized to NIL.

EXAMPLES

```
     (setq my-array '#(0 1 2 3 4))          ; make the array
     (aref my-array 0)           ; return 0th (first) element
     (aref my-array 4)           ; return 4th (last)  element
     (aref my-array 5)           ; error: non existant element
     my-array                    ; look at array

     (setq new (make-array 4))         ; make another array
     (setf (aref new 0) (make-array 4)); make new[0] an array of 4
     (setf (aref (aref new 0) 1) 'a)         ; set new[0,1] = 'a
     (setf (aref new 2) '(a b c))      ; set new[2] = '(a b c)
     new                         ; look at array
```

READ MACRO:
There is a built-in read-macro for arrays - # (the hash symbol). This allows you to create arbitrary arrays with initial values without going through a MAKE-ARRAY function.

NOTE:
This function returns the value of an array element. However, there is no equivalent direct function to set the value of an array element to some value. To set an element value, you must use the SETF function. The SETF function is a generalized function that allows you to set the value of arbitrary lisp entities.

COMMON LISP COMPATABILITY:
XLISP only supports one-dimensional arrays. Common LISP supports multi-dimension arrays.

arrayp

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(arrayp <expr>)
      <expr>   -   the expression to check

DESCRIPTION

The ARRAYP predicate checks if an <expr> is an array. T is returned if <expr> is an array, NIL is returned otherwise.

EXAMPLES

```
      (arrayp #(0 1 2))           ; returns T - array
      (setq a #(a b c))           ;
      (arrayp a)                  ; returns T - evaluates to array

      (arrayp '(a b c))           ; returns NIL - list
      (arrayp 1)                  ; returns NIL - integer
      (arrayp 1.2)                      ; returns NIL - float
      (arrayp 'a)                 ; returns NIL - symbol
      (arrayp #\a)                      ; returns NIL - character
      (arrayp NIL)                      ; returns NIL - NIL
```

assoc

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(assoc <expr> <a-list> [ { :test | :test-not } <test> ] )
     <expr>          -    the expression to find - an atom or list
     <a-list>   -    the association list to search
     <test>          -    optional test function (default is EQL)
```

DESCRIPTION

An association list is a collection of list pairs of the form ((key1 item1) (key2 item2) ... (keyN itemN)). ASSOC searches through an association list <a-list> looking for the key (a CAR in an association pair) that matches the search <expr>. If a match is found, that association pair (keyN itemN) is returned as the result. If no match is found, a NIL is returned. You may specify your own test with the :TEST and :TEST-NOT keywords followed by the test you which to perform.

EXAMPLES

```
     (setq mylist '((a . my-a)  (b . his-b)  ; set up an association
                 (c . her-c) (d . end)))    ;   list
     (assoc 'a mylist)           ; returns (A . MY-A)
     (assoc 'b mylist)           ; returns (B . HIS-B)
     (assoc 1 mylist)            ; returns NIL

     (setq agelist '((1 (bill bob))          ; set up another
                (2 (jane jill))  ;   association list
                (3 (tim tom))          ;
                (5 (larry daryl daryl))     ;
                 ))             ;
     (assoc 1 agelist)           ; returns (1 (BILL BOB))
     (assoc 3 agelist :test '>=)       ; returns (1 (BILL BOB))
     (assoc 3 agelist :test '<)        ; returns (5 (LARRY DARYL DARYL))
     (assoc 3 agelist :test '<=)       ; returns (3 (TIM TOM))
     (assoc 3 agelist :test-not '>=)         ; returns (5 (LARRY DARYL
DARYL))

     (assoc '(a b) '( ((c d) e) ((a b) x) )  ; use a list as the search
                :test 'equal)          ;   note the use of EQUAL
                               ;   returns ((A B) X)
```

NOTE:
The ASSOC function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST

keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
Common LISP supports the use of the :KEY keyword which specifies a function that is applied to each element of <a-list> before it is tested. XLISP does not support this.

atom

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(atom <expr> )
     <expr>           -     the expression to check
```

DESCRIPTION

The ATOM predicate checks if the <expr> is an atom. T is returned if <expr> is an atom, NIL is returned otherwise.

EXAMPLES

```
     (atom 'a)                    ; returns T - symbol
     (atom #'atom)                     ; returns T - subr - function
     (atom "string")                   ; returns T - string
     (atom 4)                     ; returns T - integer
     (atom 4.5)                   ; returns T - float
     (atom object)                     ; returns T - object
     (atom #(1 2 3))                   ; returns T - array
     (atom #'quote)                    ; returns T - fsubr
     (atom *standard-output*)          ; returns T - stream
     (atom '())                   ; returns T - NIL is an atom
     (atom #'defvar)                   ; returns T - closure - macro
     (atom (lambda (x) (print x)))          ; returns T - closure -
lambda

     (atom '(a b c))                   ; returns NIL - list

     (setq a '(a b))                   ; set up A with value (A B)
     (atom a)                     ; returns NIL -
                                  ;   value of A is not an atom
```

NOTE:
NIL or '() is used in many places as a list-class or atom-class expression. Both ATOM and LISTP, when applied to NIL, return T.

&aux

type: keyword
location: built-in
source file: xleval.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
&aux [ <aux-var> | ( <aux-var> <aux-value> ) ] ...
      <aux-var>   -     auxiliary variable
      <aux-value> -     auxiliary variable initialization
```

DESCRIPTION

In XLISP, there are several times that you define a formal argument list for a body of code (like DEFUN, DEFMACRO, :ANSWER and LAMBDA). The <aux-var> variables are a mechanism for you to define variables local to the function or operation definition. If there is an optional <aux-value>, they will be set to that value on entry to the body of code. Otherwise, they are initialized to NIL. At the end of the function or operation execution, these local symbols and their values are are removed.

EXAMPLES

```
      (defun my-add                       ; define function MY-ADD
        (num1 &rest num-list &aux sum)  ;   with 1 arg, rest, 1 aux var
        (setq sum num1)             ;   clear SUM
        (dotimes (i (length num-list) ) ;   loop through rest list
           (setq sum (+ sum (car num-list)))  ;      add the number to sum
           (setq num-list (cdr num-list)))    ;      and remove num from
list
        sum)                                ;   return sum when finished
      (my-add 1 2 3 4)              ; returns 10
      (my-add 5 5 5 5 5)                  ; returns 25

      (defun more-keys              ; define MORE-KEYS
           ( a                      ;   with 1 parameter A
             &aux b                 ;   with local var B
                  (c 99)                ;        local var C= 99
                  (d T)  )              ;        local var D= T
           (format T "a=~a " a)         ;   body of the function
           (format T "b=~a " b)         ;
           (format T "c=~a " c)         ;
           (format T "d=~a " d))        ;
      (more-keys "hi")              ; prints a=hi b=NIL c=99 d=T
```

backquote

---

type: special form (fsubr)
location: built-in
source file: xlcont.c and xlread.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(backquote <expr>)
<expr> - an expression which is not evaluated except for comma and comma-at portions

DESCRIPTION

BACKQUOTE returns the <expr> unevaluated - like QUOTE. The difference is that portions of the <expr> may be evaluated when they are preceeded by a COMMA (,) or COMMA-AT (,@). COMMA will evaluate the portion of the expression the comma preceeds. If the portion is an atom or a list, it is placed as is within the expression. COMMA-AT will evaluate the portion of the expression that the comma-at preceeds. The portion needs to be a list. The list is spliced into the expression. If the portion is not a list, COMMA-AT will splice in nothing.

EXAMPLES

```
(setq box 'stuff-inside)          ; BOX contains STUFF-INSIDE
(print box)                  ; prints STUFF-INSIDE
(quote (i have the box))          ; returns (I HAVE THE BOX)
(backquote (i have the box))      ; returns (I HAVE THE BOX)
(backquote (I have (comma box)))  ; returns (I HAVE STUFF-INSIDE)
(backquote (I have the ,@box))          ; returns (I HAVE THE)

(setq automobile '(a van))        ; set up AUTOMOBILE
(backquote (I have automobile))         ; returns (I HAVE
AUTOMOBILE)
(backquote (I have (comma automobile))) ; returns (I HAVE (A VAN))
(backquote (I have ,@automobile)) ; returns (I HAVE A VAN)
`(I have ,@automobile)            ; returns (I HAVE A VAN)
```

READ MACRO:
XLISP supports the normal read macro of a single reverse quote (`) as a short-hand method of writing the BACKQUOTE special form.

NOTE:
BACKQUOTE and COMMA and COMMA-AT are very useful in defining macros via DEFMACRO.

baktrace

---

type: function (subr)
location: built-in
source file: xldbug.c and xlsys.c
Common LISP compatible: related
supported on: all machines

SYNTAX

(baktrace [<level>])
<level> - an optional integer expression

DESCRIPTION

The BAKTRACE function is used to examine the system execution stack from within the break look. It shows the nested forms that got the system to the current state. The break loop can be entered by a system error, ERROR, CERROR or BREAK functions. If the <levels> parameter is not specified, all the nested forms will be shown back to the main loop form that started the execution. If <level> is specified the most recent <level> nested forms will be shown.

EXAMPLES

```
(defun out (x) (print x) (mid 99)); define OUT
(defun mid (x) (print x) (in 999)); define MID
(defun in (x) (print x) (break "in" x)) ; define IN with a BREAK
(out 9)                              ; prints  9
                                     ;         99
                                     ;         999
                                     ; break: in - 999
(baktrace)        ; this is done from within break loop
                  ; prints  Function: #<Subr-BAKTRACE: #22cb4>
                  ;         Function: #<Subr-BREAK
                  ;         Arguments:
                  ;         "in"
                  ;         999
                  ;         Function: #<Closure-IN: #2bc44>
                  ;         Arguments:
                  ;         999
                  ;         Function: #<Closure-MID: #2bd20>
                  ;         Arguments:
                  ;           99
                  ;         Function: #<Closure-OUT: #2bec4>
                  ;         Arguments:
                  ;           9
                  ;         NIL
```

COMMON LISP COMPATABILITY:
Common LISP has a similar function called BACKTRACE. For XLISP, BAKTRACE is spelled with no 'c'.

block

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(block  <name> [ <body> ... ] )
      <name>           -     an unevaluated symbol for the block name
      <body>           -     a series of expressions
```

DESCRIPTION

The BLOCK special form specifies a 'named-block' construct. The last expression in <body> will be returned by the BLOCK construct as its result unless a RETURN or RETURN-FROM is executed within BLOCK. The RETURN exit will exit the nearest (inner-most) BLOCK. The RETURN-FROM exit will exit the specified block.

EXAMPLES

```
      (block out                    ; outer BLOCK
         (print "outer")                 ;
         (block in                       ; inner BLOCK
            (print "inner")              ;
            (return-from out "all done") ;
            (print "won't get here")     ;
         )                          ;
      )                             ; prints "outer"
                                    ; prints "inner"
                                    ; returns "all done"
```

both-case-p

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

(both-case-p <char>)
     <char> - a character expression

DESCRIPTION

The BOTH-CASE-P predicate checks if the <char> expression is an alphabetic character. If <char> is an alphabetic (either an upper or lower case) character a T is returned, otherwise a NIL is returned. Upper case characters are 'A' (ASCII decimal value 65) through 'Z' (ASCII decimal value 90). Lower case characters are 'a' (ASCII decimal value 97) through 'z' (ASCII decimal value 122).

EXAMPLES

```
(both-case-p #\A)          ; returns T
(both-case-p #\a)          ; returns T
(both-case-p #\1)          ; returns NIL
(both-case-p #\[)          ; returns NIL
```

boundp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(boundp <symbol>)
<symbol> - the symbol expression to check for a value

DESCRIPTION

The BOUNDP predicate checks to see if <symbol> is a symbol with a value bound to it. T is returned if <symbol> has a value, NIL is returned otherwise. Note that <symbol> is a symbol expression - it is evaluated and the resulting expression is the one that is checked.

EXAMPLES

```
(setq a 1)                  ; set up A with value 1
(boundp 'a)                 ; returns T - value is 1

(defun foo (x) (print x))       ; set up function FOO
(boundp 'foo)                   ; returns NIL - value is closure
(boundp 'defvar)            ; returns NIL - value is closure
(boundp 'car)                   ; returns NIL - value is closure

(print myvar)                   ; error: unbound variable
(BOUNDP 'myvar)                 ; returns NIL
(setq myvar 'abc)           ; set up MYVAR with a value
(BOUNDP 'myvar)                 ; returns T - because of SETQ

(setq myvar 'qq)            ; set up MYVAR to have value QQ
(BOUNDP myvar)                  ; returns NIL - because QQ has
                            ;               no value yet
(setq qq 'new-value)            ; set QQ to have value NEW-VALUE
(BOUNDP myvar)                  ; returns T
```

break

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(break [<err-msg> [<arg>]])
<err-msg> - a string expression for the error message
<arg> - an optional expression

DESCRIPTION

The BREAK function allows the entry into the break loop with a continuable error. The continuable error generated by BREAK does not require any corrective action. The form of the message generated is:

break: <err-msg> - <arg>
if continued: return from BREAK

The default for <err-msg> is "**BREAK**". From within the break-loop, if a CONTINUE form is evaluated then a NIL is returned from BREAK. If desired, the CLEAN-UP and TOP-LEVEL functions may be evaluated to abort out of the break loop.

EXAMPLES

```
(break)                          ; break: **BREAK**

(break "out")                    ; break: out

(break "it" "up")           ; break: it - "up"
```

COMMON LISP COMPATIBILITY:

Common LISP and XLISP have the same basic form and style for BREAK. However, the <err-msg> string in Common LISP is sent to FORMAT. FORMAT is a output function that takes in format strings that include control information. Although, XLISP does have the FORMAT function, it is not used for error messages. Porting from XLISP to Common LISP will work fine. When porting from Common LISP to XLISP, you will need to check for this embedded control information in the messages.

breakenable

type: system variable
location: built-in
source file: xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

breakenable

DESCRIPTION

BREAKENABLE is a system variable that controls entry to the break loop and the trapping of errors. If *BREAKENABLE* is set to NIL, then no errors from the system, ERROR and CERROR will be trapped. If *BREAKENABLE* is non-NIL, the break loop will handle these errors. The BREAK function is not affected by *BREAKENABLE* and will always force entry to the break loop. The INIT.LSP initialization file sets *BREAKENABLE* to T, which allows errors to be trapped by the break loop.

The DEBUG function causes *BREAKENABLE* to be set to T. NODEBUG causes *BREAKENABLE* to be set to NIL.

EXAMPLES

```
(setq *breakenable* NIL)          ; disable break loop
(defun foo (x) (+ x x))                ; define FOO
(foo "a")                    ; error: bad argument type
                             ;   but did NOT enter break loop
(setq *breakenable* T)             ; enable break loop
(foo "a")                    ; error: bad argument type
                             ;   entered break loop
```

car

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(car <expr>)
     <expr> - a list or list expression

DESCRIPTION

CAR returns the first element of the expression. If the first expression is itself a list, then the sublist is returned. If the list is NIL, NIL is returned.

EXAMPLES

```
(car '(a b c))                  ; returns A
(car '((a b) c d))              ; returns (A B)
(car NIL)                  ; returns NIL
(car 'a)                   ; error: bad argument type

(setq bob '(1 2 3))             ; set up variable BOB
(car bob)                  ; returns 1
```

caar cadr

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(caar <expr> )
(cadr <expr> )
     <expr>           -     a list or list expression
```

DESCRIPTION

The CAAR and CADR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CADR does a CDR on the expression, followed by a CAR. If at any point the list is NIL, NIL is returned. If at any point a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
     (setq mylist '( (a1 a2)          ; make a 2-level list
                 (b1 b2)
                 (c1 c2)
                 (d1 d2) ) )
     (caar mylist)                    ; returns A1
     (cadr mylist)                    ; returns (B1 B2)
     (cdar mylist)                    ; returns (A2)
     (cddr mylist)                    ; returns ((C1 C2) (D1 D2))

     (caar 'a)                   ; error: bad argument
     (caar nil)                  ; returns NIL
     (cadr nil)                  ; returns NIL
     (cdar nil)                  ; returns NIL
     (cddr nil)                  ; returns NIL
```

caaar caadr cadar caddr

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(caaar <expr>)
(caadr <expr>)
(cadar <expr>)
(caddr <expr>)
<expr> - a list or list expression

DESCRIPTION

The CAAAR, CAADR, CADAR and CADDR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CADDR does a CDR on the expression, followed by a CDR, followed by a CAR. If at any point the list is NIL, NIL is returned. If at any point a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
(setq mylist '( ( (a b) (c d) (e f) )   ; make a 3-level list
                ( (g h) (i j) (k l) )
              ( (m n) (o p) (q r) )
                  ( (s t) (u v) (w x) )
              ) )
(caaar mylist)                    ; returns A
(caadr mylist)                    ; returns (G H)
(cadar mylist)                    ; returns (C D)
(caddr mylist)                    ; returns ((M N) (O P) (Q R))
(cdaar mylist)                    ; returns (B)
(cdadr mylist)                    ; returns ((I J) (K L))
(cddar mylist)                    ; returns ((E F))
(cdddr mylist)                    ; returns (((S T) (U V) (W X)))
```

caaaar caaadr ... caddar cadddr

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(caaaar <expr> )
(caaadr <expr> )
(caadar <expr> )
...
(caddar <expr> )
(cadddr <expr> )
     <expr>          -     a list or list expression
```

DESCRIPTION

The CAAAAR, CAAADR ... CADDAR, CADDDR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CAADDR does a CDR on the expression, followed by a CDR, followed by a CAR, followed by another CAR. If at any point the list is NIL, NIL is returned. If at anypoint a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
     (setq mylist '( ( (a b) (c d) (e f) )   ; make a 3-level list
                   ( (g h) (i j) (k l) )
                ( (m n) (o p) (q r) )
                     ( (s t) (u v) (w x) )
                ) )
     (caaadr mylist)                    ; returns G
     (caadar mylist)                    ; returns C
     (cdadar mylist)                    ; returns (D)
     (cadadr mylist)                    ; returns (I J)
     (cdaddr mylist)                    ; returns ((O P) (Q R))
     (cadddr mylist)                    ; returns ((S T) (U V) (W X))
```

case

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(case  <expr>  [ ( <value> <action> ) ... ]  )
     <expr>           -     an expression
     <value>          -     an unevaluated expression or list of
unevaluated
                      expressions to compare against <expr>
     <action>    -    an expression
```

DESCRIPTION

The CASE special form is a selection control form. CASE evaluates <expr>. This value is then compared against all the <value> entries. If <value> is a single atom, the atom is compared against <expr>. If <value> is a list, each of the elements of the list are compared against <expr>. The <action> associated with the first <value> that matches <expr> is evaluated and returned as CASE's result. If no <value> matches, a NIL is returned. If the last <value> is the T symbol and no other <value> has matched <expr>, then CASE will evaluate the <action> associated with T. If there are multiple T entries, the first is considered to be the end of the CASE.

EXAMPLES

```
(case 'a ('a "a"))                ; returns "a"
(case 'a ('b "b"))                ; returns NIL
(case 9 ( 1 "num") (t "ho") (t "hi"))   ; returns "ho"
(case 'a ((1 2 3 4) "number")           ;
      ( (a b c d) "alpha"))       ; returns "alpha"

(case 'a)                    ; returns NIL
(case)                            ; returns NIL

(defun print-what (parm)          ; define a function
  (case (type-of parm)            ;  check PARM type
      ( flonum   (print "float"))  ;
      ( fixnum   (print "integer"))     ;
      ( string   (print "string")) ;
      ( cons     (print "list"))   ;
      ( T        (print "other"))) ;  otherwise
  NIL)                            ;  and always return NIL
(print-what 1.2)             ; prints  "float"   returns NIL
(print-what 3)                    ; prints  "integer" returns NIL
(print-what "ab")            ; prints  "string"  returns NIL
(print-what '(a b))               ; prints  "list"    returns NIL
```

```
(print-what 'a)                    ; prints  "other"   returns NIL
```

NOTE:
The CASE special form does not work with a list or string as the <expr>. This is because CASE defines the test used to be the EQL test which does not work with lists or strings, only symbols and numbers.

COMMON LISP COMPATABILITY:
In XLISP, you can use the value T as the last value to get the effect of 'otherwise'. Common LISP uses the symbol OTHERWISE and T for this. If you are porting in code from Common LISP, be careful to make sure T is used instead of OTHERWISE in CASE statements. (Note that no error will be generated, XLISP will just try to match the <expr> to OTHERWISE.

catch

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(catch  <tag-symbol>  [ <expr> ... ]  )
      <tag-symbol>     -     an expression that evaluates to a symbol
      <expr>           -     an optional series of expressions to be
                       evaluated
```

DESCRIPTION

The CATCH and THROW special forms allow for non-local exits and traps without going through the intermediate evaluations and function returns. If there is a CATCH for a <sym-tag> that has no THROW performed to it, CATCH returns the value returned from <expr>. If there is no <expr>, a NIL is returned. If a THROW is evaluated with no corresponding CATCH, an error is generated - "error: no target for THROW". If, in the calling process, more than one CATCH is set up for the same <tag-symbol>, the most recently evaluated <tag-symbol> will be the one that does the actual catching.

EXAMPLES

```
      (catch 'mytag)                    ; returns NIL    - no THROW
      (catch 'mytag (+ 1 (+ 2 3)))      ; returns 6      - no THROW
      (catch 'mytag (+ 1 (throw 'mytag)))     ; returns NIL    - caught it
      (catch 'mytag (+ 1 (throw 'mytag 55)))  ; returns 55     - caught it
      (catch 'mytag (throw 'foo))       ; error: no target for THROW

      (defun in (x)                     ; define IN
         (if (numberp x) (+ x x)        ;   if number THEN double
                     (throw 'math 42)))  ;             ELSE throw 42
      (defun out (x)                    ; define OUT
        (princ "<")                     ;
        (princ  (* (in x) 2))                ;   double via multiply
        (princ ">") "there")            ;
      (defun main (x)                   ; define MAIN
        (catch 'math (out x)))          ;   with CATCH
      (in 5)                            ; returns 10
      (out 5)                           ; prints  <20>   returns "there"
      (main 5)                    ; prints  <20>   returns "there"
      (main 'a)                   ; prints  < returns 42
```

NOTE:
Although CATCH and THROW will accept a <tag-symbol> that is not a symbol, it will not find this improper <tag-symbol>. An error will be generated - "error: no target for THROW".

cdr

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(cdr <expr>)
     <expr> - a list or list expression

DESCRIPTION

CDR returns the remainder of a list or list expression after first element of the list is removed. If the list is NIL, NIL is returned.

EXAMPLES

```
     (cdr '(a b c))                  ; returns (B C)
     (cdr '((a b) c d))              ; returns (C D)
     (cdr NIL)                  ; returns NIL
     (cdr 'a)                   ; error: bad argument type
     (cdr '(a))                 ; returns NIL

     (setq ben '(a b c))             ; set up variable BEN
     (cdr ben)                  ; returns (B C)
```

cdar cddr

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(cdar <expr> )
(cddr <expr> )
     <expr>           -     a list or list expression
```

DESCRIPTION

The CDAR and CDDR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CDAR does a CAR on the expression, followed by a CDR. If at any point the list is NIL, NIL is returned. If at any point a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
     (setq mylist '( (a1 a2)          ; make a 2-level list
                 (b1 b2)
                 (c1 c2)
                 (d1 d2) ) )
     (cdar mylist)                    ; returns (A2)
     (cddr mylist)                    ; returns ((C1 C2) (D1 D2))
     (caar mylist)                    ; returns A1
     (cadr mylist)                    ; returns (B1 B2)

     (caar 'a)                   ; error: bad argument
     (caar nil)                  ; returns NIL
     (cadr nil)                  ; returns NIL
     (cdar nil)                  ; returns NIL
     (cddr nil)                  ; returns NIL
```

cdaar cdadr cddar cdddr

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(cdaar <expr> )
(cdadr <expr> )
(cddar <expr> )
(cdddr <expr> )
     <expr>          -     a list or list expression
```

DESCRIPTION

The CDAAR, CDADR, CDDAR and CDDDR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CDDAR does a CAR on the expression, followed by a CDR, followed by another CDR. If at any point the list is NIL, NIL is returned. If at any point a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
     (setq mylist '( ( (a b) (c d) (e f) )   ; make a 3-level list
                   ( (g h) (i j) (k l) )
                ( (m n) (o p) (q r) )
                   ( (s t) (u v) (w x) )
                ) )
     (cdaar mylist)                    ; returns (B)
     (cdadr mylist)                    ; returns ((I J) (K L))
     (cddar mylist)                    ; returns ((E F))
     (cdddr mylist)                    ; returns (((S T) (U V) (W X)))
     (caaar mylist)                    ; returns A
     (caadr mylist)                    ; returns (G H)
     (cadar mylist)                    ; returns (C D)
     (caddr mylist)                    ; returns ((M N) (O P) (Q R))
```

cdaaar cdaadr ... cdddar cddddr

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(cdaaar <expr> )
(cdaadr <expr> )
...
(cdddar <expr> )
(cddddr <expr> )
      <expr>           -     a list or list expression
```

DESCRIPTION

The CDAAAR, CDAADR .... CDDDAR, CDDDDR functions go through the list expression and perform a sequence of CAR/CDR operations. The sequence of operations is performed from right to left. So CDDAAR does a CAR on the expression, followed by a CAR, followed by a CDR, followed by another CDR. If at any point the list is NIL, NIL is returned. If at anypoint a CAR operation is performed on an atom (as opposed to a list) an error is reported - "error: BAD ARGUMENT".

EXAMPLES

```
      (setq mylist '( ( (a b) (c d) (e f) )   ; make a 3-level list
                    ( (g h) (i j) (k l) )
                  ( (m n) (o p) (q r) )
                      ( (s t) (u v) (w x) )
                  ) )
      (cdadar mylist)                   ; returns (D)
      (cdaddr mylist)                   ; returns ((O P) (Q R))
      (caaadr mylist)                   ; returns G
      (cadadr mylist)                   ; returns (I J)
      (caadar mylist)                   ; returns C
      (cadddr mylist)                   ; returns ((S T) (U V) (W X))
```

cerror

---

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(cerror <cont-msg> <err-msg> [<arg>])
<cont-msg> - a string expression for the continue message
<err-msg> - a string expression for the error message
<arg> - an optional expression

DESCRIPTION

The CERROR function allows the generation of a continuable error. A continuable error is one that can be corrected by some action within the XLISP break loop. The form of the message generated is:

error: <err-msg> - <arg>
if continued: <cont-msg>

From within the break-loop, if a CONTINUE form is evaluated then a NIL is returned from CERROR. From within the break loop, forms can be evaluated to correct the error. If desired, the CLEAN-UP and TOP-LEVEL forms may be evaluated to abort out of the break loop.

EXAMPLES

```
(cerror "fee" "fi" "fo")          ; CERROR generates the message -
                                  ; error: fi - "fo"
                                  ; if continued: fee

(cerror "I will do better"        ; CERROR generates the message -
      "There's a problem, Dave")  ;
                                  ; error: There's a problem, Dave
                                  ; if continued: I will do better

                                  ; example of system generated
                                  ; correctable error -
(symbol-value 'f)                 ; error: unbound variable - F
                                  ; if continued:
                                  ;   try evaluating symbol again
```

COMMON LISP COMPATIBILITY:
Common LISP and XLISP have the same basic form and style for CERROR. However, the <err-msg> and <cont-msg> string in Common LISP is sent to FORMAT. FORMAT is a output function that takes in format strings that include control information. Although, XLISP does have the FORMAT function, it does not print the CERROR <err-msg> with FORMAT. So, Porting from XLISP to Common LISP will work fine. When porting from

Common LISP to XLISP, you will need to check for this embedded control information in the error messages.

NOTE:
In the XLISP system, the only correctable system errors have to do with the value of a symbol being unbound. In these cases, you can do a SETQ or SET from within the break loop and then CONTINUE.

NOTE:
Remember that *BREAKENABLE* needs to non-NIL for ERROR and CERROR and system errors to be caught by the normal system break loop. If *BREAKENABLE* is NIL, ERROR and CERROR and system errors can be caught by an ERRSET form. If there is no surrounding ERRSET, no error message is generated and the break loop is not entered.

char

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char <string> <position>)
      <string>   -   a string expression
      <position> -   an integer expression

DESCRIPTION

The CHAR function returns the ASCII numeric value of the character at the specified <position> in the <string>. A <position> of 0 is the first character in the string.

EXAMPLES

```
      (char "12345" 0)            ; returns #\1
      (char "12 45" 2)            ; returns #\Space

      (string (char "1234" 3))          ; returns "4"
      (char "1234" 9)                   ; error: index out of range
```

char/=

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char/= <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR/= (character-NOT-EQUAL) function takes one or more character arguments. It checks to see if all the character arguments are different values. T is returned if the arguments are of different ASCII value. In the case of two arguments, this has the effect of testing if <char1> is not equal to <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char/= #\a #\b)            ; returns T
     (char/= #\a #\b #\c)             ; returns T
     (char/= #\a #\a)            ; returns NIL
     (char/= #\a #\b #\b)             ; returns NIL
     (char/= #\A #\a)            ; returns T
     (char/= #\a #\A)            ; returns T
```

NOTE:
Be sure that the CHAR/= function is properly typed in. The '/' is a forward slash. It is possible to mistakenly type a '\' (backslash). This is especially easy because the character mechanism is '#\a'. If you do use the backslash, no error will be reported because backslash is the single escape character and the LISP reader will evaluate 'CHAR\=' as 'CHAR='. No error will be reported, but the sense of the test is reversed.

char<

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(char< <char1> <charN> ... )
     <char1>          -     a character expression
     <charN>          -     character expression(s) to compare
```

DESCRIPTION

The CHAR< (character-LESS-THAN) function takes one or more character arguments. It checks to see if all the character arguments are monotonically increasing. T is returned if the arguments are of increasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is less than <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char< #\a #\b)                  ; returns T
     (char< #\b #\a)                  ; returns NIL
     (char< #\a #\b #\c)              ; returns T
     (char< #\a #\a)                  ; returns NIL
     (char< #\a #\b #\b)              ; returns NIL
     (char< #\A #\a)                  ; returns T
     (char< #\a #\A)                  ; returns NIL
```

char<=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char<= <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR<= (character-LESS-THAN-OR-EQUAL) function takes one or more character arguments. It checks to see if all the character arguments are monotonically non-decreasing. T is returned if the arguments are of non-decreasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is less than or equal to <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char<= #\a #\b)            ; returns T
     (char<= #\b #\a)            ; returns NIL
     (char<= #\a #\b #\c)             ; returns T
     (char<= #\a #\a)            ; returns T
     (char<= #\a #\b #\b)             ; returns T
     (char<= #\A #\a)            ; returns T
     (char<= #\a #\A)            ; returns NIL
```

char=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char= <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR= (character-EQUALITY) function takes one or more character arguments. It checks to see if all the character arguments are equivalent. T is returned if the arguments are of the same ASCII value. In the case of two arguments, this has the effect of testing if <char1> is equal to <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char= #\a #\b)                  ; returns NIL
     (char= #\b #\a)                  ; returns NIL
     (char= #\a #\b #\c)              ; returns NIL
     (char= #\a #\a)                  ; returns T
     (char= #\a #\a #\a)              ; returns T
     (char= #\a #\a #\b)              ; returns NIL
     (char= #\A #\a)                  ; returns NIL
     (char= #\a #\A)                  ; returns NIL
```

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char> <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR> (character-GREATER-THAN) function takes one or more character arguments. It checks to see if all the character arguments are monotonically decreasing. T is returned if the arguments are of monotonically decreasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is greater than <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char> #\a #\b)                  ; returns NIL
     (char> #\b #\a)                  ; returns T
     (char> #\c #\b #\a)              ; returns T
     (char> #\a #\a)                  ; returns NIL
     (char> #\c #\a #\b)              ; returns NIL
     (char> #\A #\a)                  ; returns NIL
     (char> #\a #\A)                  ; returns T
```

char>=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char>= <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR>= (character-GREATER-THAN-OR-EQUAL) function takes one or more character arguments. It checks to see if all the character arguments are monotonically non-increasing. T is returned if the arguments are of monotonically non-increasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is greater than or equal to <char2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

EXAMPLES

```
     (char>= #\a #\b)            ; returns NIL
     (char>= #\b #\a)            ; returns T
     (char>= #\c #\b #\a)             ; returns T
     (char>= #\a #\a)            ; returns T
     (char>= #\c #\a #\b)             ; returns NIL
     (char>= #\A #\a)            ; returns NIL
     (char>= #\a #\A)            ; returns T
```

characterp

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(characterp <expr>)
<expr> - the expression to check

DESCRIPTION

The CHARACTERP predicate checks if an <expr> is a character. T is returned if <expr> is a character, NIL is returned otherwise.

EXAMPLES

```
(characterp #\a)            ; returns T - character
(setq a #\b)                      ;
(characterp a)                    ; returns T - evaluates to char

(characterp "a")            ; returns NIL - string
(characterp '(a b c))             ; returns NIL - list
(characterp 1)                    ; returns NIL - integer
(characterp 1.2)            ; returns NIL - float
(characterp 'a)                   ; returns NIL - symbol
(characterp #(0 1 2))             ; returns NIL - array
(characterp NIL)            ; returns NIL - NIL
```

char-code

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
versions: all machines

SYNTAX

(char-code <char>)
     <char>   -   a character expression

DESCRIPTION

The CHAR-CODE function returns the value of the <char> expression.

EXAMPLES

```
     (char-code #\0)                  ; returns 48
     (char-code #\A)                  ; returns 65
     (char-code #\a)                  ; returns 97
     (char-code #\[)                  ; returns 91
     (char-code #\newline)            ; returns 10

     (char-code (code-char 127))      ; returns 127
     (char-code (int-char 255))       ; returns 255
```

COMMON LISP COMPATABILITY:
Common LISP supports the concept of a complex character that includes not only the ASCII code value, but also fonts and bits. The bits allow for more than 8 bits per character (16 bits is especially useful in oriental languages). The fonts allow for up to 128 different fonts. This is interesting and neat stuff, however, XLISP does not support fonts and bits.

NOTE:
Because XLISP does not support fonts and bits (as discussed above), CHAR-CODE and CHAR-INT are identical in use.

char-downcase

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

```
(char-downcase <char> )
     <char>           -     a character expression
```

DESCRIPTION

The CHAR-DOWNCASE function converts the <char> expression to lower case. The lower case equivalent of <char> is returned. If the <char> is not alphabetic ('a' thru 'z' or 'A' thru 'Z'), the character is returned unchanged.

EXAMPLES

```
     (char-downcase #\0)              ; returns #\0
     (char-downcase #\A)              ; returns #\a
     (char-downcase #\a)              ; returns #\a
     (char-downcase #\[)              ; returns #\[
     (char-downcase #\+)              ; returns #\+
```

char-equal

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char-equal <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR-EQUAL function takes one or more character arguments. It checks to see if all the character arguments are equivalent. T is returned if the arguments are of the same ASCII value. In the case of two arguments, this has the effect of testing if <char1> is equal to <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
     (char-equal #\a #\b)              ; returns NIL
     (char-equal #\b #\a)              ; returns NIL
     (char-equal #\a #\b #\c)          ; returns NIL
     (char-equal #\a #\a)              ; returns T
     (char-equal #\a #\a #\a)          ; returns T
     (char-equal #\a #\a #\b)          ; returns NIL
     (char-equal #\A #\a)              ; returns T
     (char-equal #\a #\A)              ; returns T
```

NOTE:
The CHAR-EQUAL function is listed in the documentation that
comes with XLISP as CHAR-EQUALP.

char-greaterp

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char-greaterp <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR-GREATERP function takes one or more character arguments. It checks to see if all the character arguments are monotonically decreasing. T is returned if the arguments are of monotonically decreasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is greater than <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
     (char-greaterp #\a #\b)                ; returns NIL
     (char-greaterp #\b #\a)                ; returns T
     (char-greaterp #\c #\b #\a)       ; returns T
     (char-greaterp #\a #\a)                ; returns NIL
     (char-greaterp #\c #\a #\b)       ; returns NIL
     (char-greaterp #\A #\a)                ; returns NIL
     (char-greaterp #\a #\A)                ; returns NIL
```

char-int

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

(char-int <char>)
     <char> - a character expression

DESCRIPTION

The CHAR-INT function returns the ASCII value of the <char> expression.

EXAMPLES

```
     (char-int #\0)                    ; returns 48
     (char-int #\A)                    ; returns 65
     (char-int #\a)                    ; returns 97
     (char-int #\[)                    ; returns 91
     (char-int #\newline)              ; returns 10

     (char-int (code-char 127))        ; returns 127
     (char-int (int-char 255))         ; returns 255
```

NOTE:
CHAR-CODE and CHAR-INT are identical in use. See CHAR-CODE for additional information.

char-lessp

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(char-lessp <char1> <charN> ... )
     <char1>          -     a character expression
     <charN>          -     character expression(s) to compare
```

DESCRIPTION

The CHAR-LESSP function takes one or more character arguments. It checks to see if all the character arguments are monotonically increasing. T is returned if the arguments are of increasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is less than <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
     (char-lessp #\a #\b)              ; returns T
     (char-lessp #\b #\a)              ; returns NIL
     (char-lessp #\a #\b #\c)          ; returns T
     (char-lessp #\a #\a)              ; returns NIL
     (char-lessp #\a #\b #\b)          ; returns NIL
     (char-lessp #\A #\a)              ; returns NIL
     (char-lessp #\a #\A)              ; returns NIL
```

char-not-equal

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char-not-equal <char1> <charN> ...)
     <char1>   -   a character expression
     <charN>   -   character expression(s) to compare

DESCRIPTION

The CHAR-NOT-EQUAL function takes one or more character arguments. It checks to see if all the character arguments are different values. T is returned if the arguments are of different ASCII value. In the case of two arguments, this has the effect of testing if <char1> is not equal to <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
     (char-not-equal #\a #\b)          ; returns T
     (char-not-equal #\a #\b #\c)      ; returns T
     (char-not-equal #\a #\a)          ; returns NIL
     (char-not-equal #\a #\b #\b)      ; returns NIL
     (char-not-equal #\A #\a)          ; returns NIL
     (char-not-equal #\a #\A)          ; returns NIL
```

NOTE:
The CHAR-NOT-EQUAL function is listed in the documentation that comes with XLISP as CHAR-NOT-EQUALP. It functions properly in the XLISP code as CHAR-NOT-EQUAL.

char-not-greaterp

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(char-not-greaterp <char1> <charN> ... )
     <char1>          -     a character expression
     <charN>          -     character expression(s) to compare
```

DESCRIPTION

The CHAR-NOT-GREATERP function takes one or more character arguments. It checks to see if all the character arguments are monotonically non-decreasing. T is returned if the arguments are of non-decreasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is less than or equal to <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
     (char-not-greaterp #\a #\b)       ; returns T
     (char-not-greaterp #\b #\a)       ; returns NIL
     (char-not-greaterp #\a #\b #\c)         ; returns T
     (char-not-greaterp #\a #\a)       ; returns T
     (char-not-greaterp #\a #\b #\b)         ; returns T
     (char-not-greaterp #\A #\a)       ; returns T
     (char-not-greaterp #\a #\A)       ; returns T
```

char-not-lessp

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(char-not-lessp <char1> <charN> ...)
<char1> - a character expression
<charN> - character expression(s) to compare

DESCRIPTION

The CHAR-NOT-LESSP function takes one or more character arguments. It checks to see if all the character arguments are monotonically non-increasing. T is returned if the arguments are of monotonically non-increasing ASCII value. In the case of two arguments, this has the effect of testing if <char1> is greater than or equal to <char2>. This test is case insensitive - the character #\a is considered to be the same ASCII value as #\A.

EXAMPLES

```
(char-not-lessp #\a #\b)          ; returns NIL
(char-not-lessp #\b #\a)          ; returns T
(char-not-lessp #\c #\b #\a)      ; returns T
(char-not-lessp #\a #\a)          ; returns T
(char-not-lessp #\c #\a #\b)      ; returns NIL
(char-not-lessp #\A #\a)          ; returns T
(char-not-lessp #\a #\A)          ; returns T
```

char-upcase

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

(char-upcase <char>)
     <char>   -   a character expression

DESCRIPTION

The CHAR-UPCASE function converts the <char> expression to upper case. The upper case equivalent of <char> is returned. If the <char> is not alphabetic ('a' thru 'z' or 'A' thru 'Z'), the character is returned unchanged.

EXAMPLES

```
     (char-upcase #\0)           ; returns #\0
     (char-upcase #\A)           ; returns #\A
     (char-upcase #\a)           ; returns #\A
     (char-upcase #\[)           ; returns #\[
     (char-upcase #\+)           ; returns #\+
```

class

---

type: object
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

class

DESCRIPTION

CLASS is the built-in object class that is used to build other classes. Classes are, essentially, the template for defining object instances.

EXAMPLES

```
(setq myclass (send class :new '(var))) ; create MYCLASS with VAR
(send myclass :answer :isnew '()  ; set up initialization
      '((setq var nil) self))
(send myclass :answer :set-it '(value)  ; create :SET-IT message
      '((setq var value)))
(setq my-obj (send myclass :new)) ; create MY-OBJ of MYCLASS
(send my-obj :set-it 5)                 ; VAR is set to 5
```

CLASS DEFINITION:
The internal definition of the CLASS object instance looks like:

```
Object is #<Object: #23fe2>, Class is #<Object: #23fe2>
  MESSAGES = ((:ANSWER . #<Subr-: #23e48>)
              (:ISNEW . #<Subr-: #23e84>)
              (:NEW . #<Subr-: #23ea2>))
  IVARS = (MESSAGES IVARS CVARS CVALS SUPERCLASS IVARCNT IVARTOTAL)
  CVARS = NIL
  CVALS = NIL
  SUPERCLASS = #<Object: #23fd8>
  IVARCNT = 7
  IVARTOTAL = 7
#<Object: #23fe2>
```

The class of CLASS is CLASS, itself. The superclass of CLASS is OBJECT. Remember that the location information (like #23fe2) varies from system to system, yours will probably look different.

BUILT-IN METHODS:
The built in methods in XLISP include:

<message>	operation
:ANSWER	Add a method to an object.
:CLASS	Return the object's class.

```
:ISNEW          Run initialization code on object.
:NEW       Create a new object (instance or class).
:SHOW      Show the internal state of the object.
```

MESSAGE STRUCTURE:
The normal XLISP convention for a <message> is to have a valid symbol preceeded by a colon like :ISNEW or :MY-MESSAGE. However, it is possible to define a <message> that is a symbol without a colon, but this makes the code less readable.

:class

type: message selector
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(send <object> :class)
      <object>   -   an existing object

DESCRIPTION

The :CLASS message selector will cause a method to run that will return the object which is the class of the specified <object>. Note that the returned value is an object which will look like "#<Object: #18d8c>". The <object> must exist or an error will be generated - "error: bad argument type".

EXAMPLES

```
      (send object :class)              ; returns the CLASS object
      (send class :class)               ; returns the CLASS object
      (setq new-cls (send class :new '(var))) ; create NEW-CLS
      (setq new-obj (send new-cls :new)); create NEW-OBJ of NEW-CLS
      (send new-obj :class)             ; returns the NEW-CLS object
      (send new-cls :class)             ; returns the CLASS object
```

clean-up

---

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(clean-up)

DESCRIPTION

The CLEAN-UP function aborts one level of the break loop. This is valid for BREAKs, ERRORs and CERRORs (continuable errors). If CLEAN-UP is evaluated while not in a break loop, an error is generated - "error: not in a break loop". This error does not cause XLISP to go into a break loop. CLEAN-UP never actually returns a value.

EXAMPLES

```
(clean-up)                  ; [back to previous break level]

(break "out")                    ; break: out
(clean-up)                  ; to exit out of break loop
```

KEYSTROKE EQUIVALENT:
In the IBM PC and MS-DOS versions of XLISP, a CTRL-g key sequence has the same effect as doing a (CLEAN-UP). On a Macintosh, this can be accomplished by a pull-down menu or a COMMAND-g.

close

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(close <file-ptr>)
<file-ptr> - a file pointer expression

DESCRIPTION

The CLOSE function closes the file specified through <file-ptr>. If the file close was successful, then a NIL is returned as the result. For the file close to be successful, the <file-ptr> has to point to a valid file. If the file close was not successful, an error is generated - "error: file not open").

EXAMPLES

```
(close (open 'f :direction :output))    ; returns NIL

(setq myfile                          ; create MYFILE
   (open 'mine :direction :output))     ;
(print "hi" myfile)                    ; returns "hi"
(close myfile)                        ; returns NIL
                                ; file contains <hi>   <NL>
(setq myfile                          ; open MYFILE for input
   (open 'mine :direction :input));
(read myfile)                         ; returns "hi"
(close myfile)                        ; returns NIL
```

COMMON LISP COMPATABILITY:
Common LISP has an XLISP compatable CLOSE function. Common LISP does support an :ABORT keyword, which is not supported in XLISP.

code-char

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
versions: all machines

SYNTAX

(code-char <code>)
     <code> - a numeric expression

DESCRIPTION

The CODE-CHAR function returns a character which is the result of turning <code> expression into a character. If a <code> cannot be made into a character, NIL is returned. The range that <code> produces a valid character is 0 through 127.

EXAMPLES

```
(code-char 48)                    ; returns #\0
(code-char 65)                    ; returns #\A
(code-char 97)                    ; returns #\a
(code-char 91)                    ; returns #\[
(code-char 10)                    ; returns #\Newline
(code-char 128)                   ; returns NIL
(code-char 999)                   ; returns NIL
```

COMMON LISP COMPATABILITY:
Common LISP allows for some optional arguments in CODE-CHAR because it supports the concept of a complex character that includes not only the ASCII code value, but also fonts and bits. The bits allow for more than 8 bits per character (16 bits is especially useful in oriental languages). The fonts allow for up to 128 different fonts. This is interesting and neat stuff, however, XLISP does not support fonts and bits or the optional parameters associated with them.

NOTE:
Unlike the CHAR-CODE and CHAR-INT functions, CODE-CHAR and INT-CHAR are not identical in use. CODE-CHAR accepts 0..127 for its range and then produces NIL results. INT-CHAR accepts 0..255 for its range and then produces errors.

comma

---

type: reader expansion
location: built-in
source file: xlcont.c and xlread.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(comma <expr>)
<expr> - an expression which is evaluated within a BACKQUOTEd expression

DESCRIPTION

A BACKQUOTE special form returns an expression unevaluated, except that portions of the expression may be evaluated when they are preceeded by a COMMA (,) or COMMA-AT (,@). COMMA will evaluate the portion of the expression the comma preceeds. If the portion is an atom or a list, it is placed as is within the expression.

EXAMPLES

```
(setq box 'stuff-inside)          ; BOX contains STUFF-INSIDE
(print box)                  ; prints STUFF-INSIDE
(quote (i have the box))          ; returns (I HAVE THE BOX)
(backquote (i have the box))      ; returns (I HAVE THE BOX)
(backquote (I have (comma box)))  ; returns (I HAVE STUFF-INSIDE)
(backquote (I have the ,@box))          ; returns (I HAVE THE)

(setq automobile '(a van))        ; set up AUTOMOBILE
(backquote (I have automobile))         ; returns (I HAVE
AUTOMOBILE)
(backquote (I have ,automobile))  ; returns (I HAVE (A VAN))
(backquote (I have ,@automobile)) ; returns (I HAVE A VAN)
`(I have ,@automobile)            ; returns (I HAVE A VAN)
```

READ MACRO:
XLISP supports the normal read macro of a comma (,) as a short-hand method of writing the COMMA read-expansion.

NOTE:
BACKQUOTE and COMMA and COMMA-AT are very useful in defining macros via DEFMACRO.

comma-at

---

type: reader expansion
location: built-in
source file: xlcont.c and xlread.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(comma-at <expr>)
<expr> - an expression which is evaluated within a BACKQUOTEd expression

DESCRIPTION

A BACKQUOTE special form returns an expression unevaluated, except that portions of the expression may be evaluated when they are preceeded by a COMMA (,) or COMMA-AT (,@). COMMA-AT will evaluate the portion of the expression that the comma-at preceeds. The portion needs to be a list. The list is spliced into the expression. If the portion is not a list, COMMA-AT will splice in nothing.

EXAMPLES

```
(setq box 'stuff-inside)          ; BOX contains STUFF-INSIDE
(print box)                  ; prints STUFF-INSIDE
(quote (i have the box))          ; returns (I HAVE THE BOX)
(backquote (i have the box))      ; returns (I HAVE THE BOX)
(backquote (I have (comma box)))  ; returns (I HAVE STUFF-INSIDE)
(backquote (I have the ,@box))          ; returns (I HAVE THE)

(setq automobile '(a van))        ; set up AUTOMOBILE
(backquote (I have automobile))         ; returns (I HAVE
AUTOMOBILE)
(backquote (I have (comma automobile))) ; returns (I HAVE (A VAN))
(backquote (I have ,@automobile)) ; returns (I HAVE A VAN)
`(I have ,@automobile)            ; returns (I HAVE A VAN)
```

READ MACRO:
XLISP supports the normal read macro of a comma (,@) as a short-hand method of writing the COMMA-AT read-expansion.

NOTE:
BACKQUOTE and COMMA and COMMA-AT are very useful in defining macros via DEFMACRO.

cond

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(cond  [ ( <pred1> <expr1> ) [ ( <pred2> <expr2> ) ... ] ] )
      <predN>          -     a predicate (NIL/non-NIL) expression
      <exprN>          -     an expression
```

DESCRIPTION

The COND special form evaluates a series of predicate / expression pairs. COND will evaluate each predicate in sequential order until it finds one that returns a non-NIL value. The expression that is associated with the non-NIL value is evaluated. The resulting value of the evaluated expression is returned by COND. If there are no predicates that return a non-NIL value, NIL is returned by COND. Only one expression is evaluated - the first one with a non-NIL predicate. Note that the predicate can be a symbol or expression.

EXAMPLES

```
      (cond                              ; sample CONDitional
        ((not T) (print "this won't print"))  ;
        ( NIL    (print "neither will this")) ;
        ( T      (print "this will print"))   ;
        ( T      (print "won't get here")))   ; prints "this will print"

      (defun print-what (parm)
       (cond                             ; start of COND
        ((numberp parm) (print "numeric"))    ;  check for number
        ((consp parm)   (print "list")) ;  check for list
        ((null parm)     (print "nil")) ;  check for NIL
        (T          (print "something"))) ;  catch-all
       NIL)                         ;  always return
                                    ;
      (print-what 'a)                    ; prints "something"
      (print-what 12)                    ; prints "numeric"
      (print-what NIL)              ; prints "nil"
      (print-what '(a b))                ; prints "list"
```

cons

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(cons <expr-car> <expr-cdr> )
     <arg>      -    description
     <expr-car> -    an expression
     <expr-cdr> -    an expression
```

DESCRIPTION

The CONS function takes two expressions and constructs a new list from them. If the <expr-cdr> is not a list, then the result will be a 'dotted-pair'.

EXAMPLES

```
     (cons 'a 'b)                    ; returns (A . B)
     (cons 'a nil)                   ; returns (A)
     (cons 'a '(b))                  ; returns (A B)
     (cons '(a b) '(c d))            ; returns ((A B) C D)
     (cons '(a b) 'c)           ; returns ((A B) . C)

     (cons (- 4 3) '(2 3))           ; returns (1 2 3)
```

consp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(consp <expr>)
<expr> - the expression to check

DESCRIPTION

The CONSP predicate checks if the <expr> is a non-empty list. T is returned if <expr> is a list, NIL is returned otherwise. Note that if the <expr> is NIL, NIL is returned.

EXAMPLES

```
(consp '(a b))                    ; returns T - list
(consp '(a . b))             ; returns T - dotted pair list

(consp #'defvar)             ; returns NIL - closure - macro
(consp (lambda (x) (print x)))          ; returns NIL - closure -
lambda
(consp NIL)                  ; returns NIL - NIL
(consp #(1 2 3))             ; returns NIL - array
(consp *standard-output*)         ; returns NIL - stream
(consp 1.2)                  ; returns NIL - float
(consp #'quote)                   ; returns NIL - fsubr
(consp 1)                    ; returns NIL - integer
(consp object)                    ; returns NIL - object
(consp "str")                     ; returns NIL - string
(consp #'car)                     ; returns NIL - subr
(consp 'a)                   ; returns NIL - symbol
```

NOTE:
When applied to CONSP, NIL - the empty list - returns a NIL. NIL or '() is used in many places as a list-class or atom-class expression. Both ATOM and LISTP, when applied to NIL, return T. If you wish to check for a list where an empty list is still considered a valid list, use the LISTP predicate.

:constituent

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

:constituent

DESCRIPTION

:CONSTITUENT is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :CONSTITUENT keyword means that the specified character is to be used, as is, with no further processing. The system defines that the following characters are :CONSTITUENT characters:

```
0123456789  !$%&*+-./  :<=>?@[]^_{}~
ABCDEFGHIJKLMNOPQRSTUVWXYZ  abcdefghijklmnopqrstuvwxyz
```

EXAMPLES

```
(defun look-at (table)            ; define a function to
 (dotimes (ch 127)                ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
    (case entry             ;   entries with a function
      (:CONSTITUENT                 ;
          (princ (int-char ch)))    ;
      (T       NIL))))      ;
 (terpri))                  ;
(look-at *readtable*)               ;  prints  !$%&*+-./0123456789
                            ;        :<=>?@ABCDEFGHIJKLM
                            ;        NOPQRSTUVWXYZ[]^_ab
                            ;        cdefghijklmnopqrstu
                            ;        vwxyz{}~
```

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

continue

---

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(continue)

DESCRIPTION

The CONTINUE function attempts to continue from the break loop. This is valid only for CERRORs (continuable errors). If CONTINUE is evaluated while not in a break loop, an error is generated - "error: not in a break loop". This error does not cause XLISP to go into a break loop. CONTINUE never actually returns a value.

EXAMPLES

```
(continue)                  ; error: not in a break loop

(break "out")                    ; break: out
(continue)                  ; to continue from break loop
                            ; BREAK returns NIL
```

KEYSTROKE EQUIVALENT:
In the IBM PC and MS-DOS versions of XLISP, a CTRL-p key sequence has the same effect as doing a (CONTINUE). On a Macintosh, this can be accomplished by a pull-down menu or a COMMAND-p.

cos

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(cos <expr>)
<expr> - floating point number/expression

DESCRIPTION

The cos function returns the cosine of the <expr>. The <expr> is in radians.

EXAMPLES

```
(cos 0.0)                   ; returns 1
(cos (/ 3.14159 2))             ; returns 1.32679e-06 (almost 0)
(cos .5)                    ; returns 0.877583
(cos 0)                         ; error: bad integer operation
(cos 1.)                    ; error: bad integer operation
```

debug

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: no
supported on: all machines

SYNTAX

(debug)

DESCRIPTION

The DEBUG function sets *BREAKENABLE* to T. This has the effect of turning on the break loop for errors. DEBUG always returns T. The default is DEBUG enabled.

EXAMPLES

```
(debug)                          ; returns T
(+ 1 "a")                      ; error: bad argument type
                               ; enters break-loop
(clean-up)                     ; from within the break-loop
(nodebug)                      ; returns NIL
(+ 1 "a")                      ; error: bad argument type
                               ; but doesn't enter break-loop
```

NOTE:
The functions DEBUG and NODEBUG are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

debug-io

---

type: system variable
location: built-in
source file: xlinit.c xlio.c xldbug.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

debug-io

DESCRIPTION

DEBUG-IO is a system variable that contains a file pointer that points to the stream where all debug input/output goes to and from. The default file for *DEBUG-IO* is the system standard error device - normally the keyboard and screen.

EXAMPLES

```
      *debug-io*                    ; returns #<File-Stream: #243de>
```

NOTE:
TRACE-OUTPUT, *DEBUG-IO* and *ERROR-OUTPUT* are normally all set to the same file stream - STDERR.

defconstant

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(defconstant  <symbol>  <init-value> )
      <symbol>    -     an expression evaluating to a symbol
      <init-value>      -     an initial value expression
```

DESCRIPTION

The DEFCONSTANT macro defines a user constant with the name <symbol>. The <symbol> is created with the initial value <init-value> expression. If <symbol> did exist, its previous value will be overwritten. DEFCONSTANT returns the <symbol> as its result.

EXAMPLES

```
      (boundp 'mvyar)                    ; returns NIL - doesn't exist
      (defconstant myvar 7)              ; returns MYVAR
      (boundp 'myvar)                    ; returns T
      myvar                         ; returns 7
```

BUG:
In Common LISP, the definition of DEFCONSTANT is such that it returns the <symbol> as its result. XLISP returns the value of <symbol>.

COMMON LISP COMPATABILITY:
In Common LISP, any change to the value of the DEFCONSTANT <symbol> is supposed to generate an error. XLISP treats it like any user symbol and allows it to change.

COMMON LISP COMPATABILITY:
Common LISP supports an additional optional parameter. This parameter is a documentation string. XLISP does not support this.

NOTE:
The functions DEFVAR, DEFPARAMETER and DEFCONSTANT are created in the INIT.LSP file. If it does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

defmacro

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(defmacro <symbol> <arg-list> <body> )
     <symbol>    -     The name of the macro being defined
     <arg-list>  -     A list of the formal arguments to the macro
                       of the form:     ( [ <arg1> ... ]
                                        [ &optional <oarg1> ... ]
                                        [ &rest <rarg> ]
                                        [ &key ... ]
                                        [ &aux <aux1> ... ] )
     <body>            -     A series of LISP forms (expressions)
```

DESCRIPTION

DEFMACRO defines a macro expansion. When the <symbol> name of the macro expansion is encountered (similar to a function invocation), the <body> of code that was defined in the DEFMACRO is expanded and replaces the macro invocation.

All of the <argN> formal arguments that are defined are required to appear in the invocation of the macro expansion. If there are any &OPTIONAL arguments defined, they will be filled in order. If there is a &REST argument defined, and all the required formal arguments and &OPTIONAL arguments are filled, any and all further parameters will be passed into the function via the <rarg> argument. Note that there can be only one <rarg> argument for &REST. If there are insufficient parameters for any of the &OPTIONAL or &REST arguments, they will contain NIL. The &AUX variables are a mechanism for you to define variables local to the DEFMACRO execution. At the end of the function execution, these local symbols and their values are are removed.

EXAMPLES

```
     (defmacro plus (num1 num2)        ; define PLUS macro
       `(+ ,num1 ,num2))               ;   which is a 2 number add
     (plus 1 2)                  ; returns 3
     (setq x 10)                 ; set x to 10
     (setq y 20)                 ; set y to 20
     (plus x y)                  ; returns 30

     (defmacro betterplus (num &rest nlist)  ; define a BETTERPLUS macro
       `(+ ,num ,@nlist))              ;   which can take many numbers
     (betterplus 1)                  ; returns 1
     (betterplus 1 2 3)              ; returns 6
     (betterplus 1 2 3 4 5)          ; returns 15
```

```
(defmacro atest (x &optional y &rest z) ; define ATEST macro
  (princ " x: ") (princ x)        ;   \
  (princ " y: ") (princ y)        ;    print out the parameters
  (princ " z: ") (princ z) (terpri)     ;   /      (un-evaluated)
  `(print (+ ,x ,y ,@z)) )        ;   add them together (eval'ed)
                                  ;
(atest 1)                         ; prints - x: 1 y: NIL z: NIL
                                  ;   error: bad argument type
                                  ; because (+ 1 NIL) isn't valid
(atest 1 2)                             ; prints - x: 1 y: 2 z: NIL
                                  ;   returns 3
(atest 1 2 3)                           ; prints - x: 1 y: 2 z: (3)
                                  ;   returns 6
(atest 1 2 3 4 5)                       ; prints - x: 1 y: 2 z: (3 4 5)
                                  ;   returns 15
                                  ;
(setq a 99)                       ; set A to 99
(setq b 101)                            ; set B to 101
(atest a b)                             ; prints - x: A y: B z: NIL
                                  ;   returns 200
(atest a b 9 10 11)                     ; prints - x: A y: B z: (9 10 11)
                                  ;   returns 230
```

COMMON LISP COMPATABILITY:
Common LISP supports an optional documentation string as the first form in the <body> of a DEFMACRO or DEFUN. XLISP will accept this string as a valid form, but it will not do anything special with it.

defparameter

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: similar
supported on: all machines

SYNTAX

(defparameter <symbol> <init-value>)
<symbol> - an expression evaluating to a symbol
<init-value> - an initial value expression

DESCRIPTION

The DEFPARAMETER macro defines a user parameter (variable) with the name <symbol>. A user parameter is supposed to be a variable that should not change but is allowed to change. The <symbol> is created with the initial value <init-value> expression. If <symbol> did exist, its previous value will be overwritten. DEFPARAMETER returns the <symbol> as its result.

EXAMPLES

```
(boundp 'mvyar)                    ; returns NIL - doesn't exist
(defparameter myvar 7)             ; returns MYVAR
(boundp 'myvar)                    ; returns T
myvar                         ; returns 7
```

BUG:
In Common LISP, the definition of DEFPARAMETER is such that it returns the <symbol> as its result. XLISP returns the value of <symbol>.

COMMON LISP COMPATABILITY:
Common LISP supports an additional optional parameter. This parameter is a documentation string. XLISP does not support this.

NOTE:
The functions DEFVAR, DEFPARAMETER and DEFCONSTANT are created in the INIT.LSP file. If it does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

defun

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(defun <symbol> <arg-list> <body> )
      <symbol>    -     The name of the function being defined
      <arg-list>  -     A list of the formal arguments to the function
                        of the form:     ( [ <arg1> ... ]
                                         [ &optional <oarg1> ... ]
                                         [ &rest <rarg> ]
                                         [ &key ... ]
                                         [ &aux <aux1> ... ] )
      <body>            -     A series of LISP forms (expressions) that
                        are executed in order.
```

DESCRIPTION

DEFUN defines a new function or re-defines an exisiting function. The last form in <body> that is evaluated is the value that is returned when the function is executed.

All of the <argN> formal arguments that are defined are required to appear in a call to the defined function. If there are any &OPTIONAL arguments defined, they will be filled in order. If there is a &REST argument defined, and all the required formal arguments and &OPTIONAL arguments are filled, any and all further parameters will be passed into the function via the <rarg> argument. Note that there can be only one <rarg> argument for &REST. If there are insufficient parameters for any of the &OPTIONAL or &REST arguments, they will contain NIL. The &AUX variables are a mechanism for you to define variables local to the function definition. At the end of the function execution, these local symbols and their values are are removed.

EXAMPLES

```
      (defun my-add                     ; define function MY-ADD
        (num1 num2)                     ;   with 2 formal parameters
        (+ num1 num2))            ;   that adds the two paramters
      (my-add 1 2)                      ; returns 3

      (defun foo                  ; define function FOO
        (a b &optional c d &rest e)           ;   with some of each
argument
        (print a) (print b)             ;
        (print c) (print d)             ;   print out each
        (print e))                      ;
      (foo)                       ; error: too few arguments
```

```
      (foo 1)                          ; error: too few arguments
      (foo 1 2)                   ; prints 1 2 NIL NIL NIL
      (foo 1 2 3)                 ; prints 1 2 3 NIL NIL
      (foo 1 2 3 4)                    ; prints 1 2 3 4 NIL
      (foo 1 2 3 4 5)                  ; prints 1 2 3 4 (5)
      (foo 1 2 3 4 5 6 7 8 9)               ; prints 1 2 3 4 (5 6 7 8 9)

      (defun my-add                    ; define function MY-ADD
        (num1 &rest num-list &aux sum)  ;   with 1 arg, rest, 1 aux var
        (setq sum num1)            ;   clear SUM
        (dotimes (i (length num-list) ) ;   loop through rest list
           (setq sum (+ sum (car num-list)))  ;      add the number to sum
           (setq num-list (cdr num-list)))    ;      and remove num from
list
        sum)                             ;   return sum when finished
      (my-add 1 2 3 4)            ; returns 10
      (my-add 5 5 5 5 5)               ; returns 25
```

COMMON LISP COMPATABILITY:
Common LISP supports an optional documentation string as the first form in the <body> of a DEFMACRO or DEFUN. XLISP will accept this string as a valid form, but it will not do anything special with it.

defvar

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: similar
supported on: all machines

SYNTAX

(defvar <symbol> [<init-value>])
<symbol> - an expression evaluating to a symbol
<init-value> - an optional initial value expression

DESCRIPTION

The DEFVAR macro defines a user variable with the name <symbol>. If <symbol> did not already exist, the <symbol> is created with the initial value NIL. If the optional <init-value> expression is present, the new <symbol> will be set to the <init-value>. If <symbol> did exist, its previous value will be left untouched. DEFVAR returns the <symbol> as its result.

EXAMPLES

```
(boundp 'mvyar)                  ; returns NIL - doesn't exist
(defvar myvar)                   ; returns MYVAR
(boundp 'myvar)                  ; returns T
(setq myvar 7)                   ; returns 7
(defvar myvar)                   ; returns MYVAR
myvar                        ; returns 7 - was not initialized
(defvar myvar 99)            ; returns MYVAR
myvar                        ; returns 7 - was not initialized
```

BUG:
In Common LISP, the definition of DEFVAR is such that it returns the <symbol> as its result. XLISP returns the value of <symbol>.

COMMON LISP COMPATABILITY:
Common LISP supports an additional optional parameter. This parameter is a documentation string. XLISP does not support this.

NOTE:
The functions DEFVAR, DEFPARAMETER and DEFCONSTANT are created in the INIT.LSP file. If it does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

delete

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(delete <expr> <list>  [ { :test | :test-not } <test> ] )
     <expr>           -     the expression to delete from <list>
     <list>           -     the list to DESTRUCTIVELY modify
     <test>           -     optional test function (default is EQL)
```

DESCRIPTION

DELETE destructively modifies the <list> by removing the <expr>. The destructive aspect of this operation means that the actual symbol value is used in the list-modifying operations - not a copy. If <expr> appears multiple times in the <list>, all occurances will be removed. <list> must evaluate to a valid list. An atom for <list> will result in an error. Having NIL for <list> will return a NIL as the result. You may specify your own test with the :TEST and :TEST-NOT keywords.

EXAMPLES

```
     (delete 'b NIL)                    ; returns NIL
     (delete 'b '(a b b c b))           ; returns (A C)

     (setq a '(1 2 3)) (setq b a)       ; set up A and B
     (delete '2 a)                      ; returns (1 3)
     (print a)                   ; prints (1 3)    A IS MODIFIED!
     (print b)                   ; prints (1 3)    B IS MODIFIED!

     (delete '(b) '((a)(b)(c)))         ; returns ((A) (B) (C))
                                 ;   EQL doesn't work on lists
     (delete '(b) '((a)(b)(c)) :test 'equal) ; returns ((A) (C))
```

NOTE:
The DELETE function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

delete-if

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(delete-if <test> <list>)
     <test>   -   the test function to be performed
     <list>   -   the list to delete from

DESCRIPTION

DELETE-IF destructively modifies the <list> by removing the elements of the <list> that pass the <test>. The destructive aspect of this operation means that the actual symbol value is used in the list-modifying operations - not a copy. <list> must evaluate to a valid list. An atom for <list> will result in an error. Having NIL for <list> will return a NIL as the result.

EXAMPLES

```
     (setq mylist '(1 2 3 4 5 6 7 8))  ; set up a list
     (delete-if 'oddp mylist)          ; returns (2 4 6 8)
     (print mylist)                    ; prints  (2 4 6 8)
                                  ;   note that MYLIST is affected

     (setq mylist '(a nil b nil c))          ; set up a list
     (delete-if 'null mylist)          ; returns (A B C)
```

BUG:
DELETE-IF will return the proper value, but it does not always properly modify the symbol containing the value. This seems to be true if the first element of the <list> passes the test (and should be deleted).

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

delete-if-not

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(delete-if-not <test> <list>)
     <test>   -   the test function to be performed
     <list>   -   the list to delete from

DESCRIPTION

DELETE-IF-NOT destructively modifies the <list> by removing the elements of the <list> that fail the <test>. The destructive aspect of this operation means that the actual symbol value is used in the list-modifying operations - not a copy. <list> must evaluate to a valid list. An atom for <list> will result in an error. Having NIL for <list> will return a NIL as the result.

EXAMPLES

```
     (setq mylist '(1 2 3 4 5 6 7 8))  ; set up a list
     (delete-if-not 'oddp mylist)      ; returns (1 3 5 7)
     (print mylist)                    ; prints  (1 3 5 7)
                                  ;   note that MYLIST is affected

     (setq mylist '(a nil b nil c))          ; set up a list
     (delete-if-not 'null mylist)      ; returns (NIL NIL)
```

BUG:
DELETE-IF-NOT will return the proper value, but it does not always properly modify the symbol containing the value. This seems to be true if the first element of the <list> fails the test (and should be deleted).

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

digit-char

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
versions: all machines

SYNTAX

(digit-char <int>)
     <int>   -   an integer expression

DESCRIPTION

The DIGIT-CHAR function takes an integer expression <int> and converts it into a decimal digit character. So, an integer value of 0 produces the character #\0. An integer value of 1 produces the character #\1 and so on. If a valid character can be produce it is returned, otherwise a NIL is returned.

EXAMPLES

```
     (digit-char 0)                    ; returns #\0
     (digit-char 9)                    ; returns #\9
     (digit-char 10)                   ; returns NIL
```

COMMON LISP COMPATABILITY:
Common LISP supports the use of an optional radix parameter. This option specifies numeric base. This allows the DIGIT-CHAR to function properly for hexadecimal digits (for example). Common LISP supports up to base 36 radix systems. XLISP does not support this radix parameter. Common LISP also supports a font parameter which XLISP does not.

digit-char-p

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
versions: all machines

SYNTAX

(digit-char-p <char>)
     <char>   -   a character expression

DESCRIPTION

The DIGIT-CHAR-P predicate checks if the <char> expression is a numeric digit. If <char> is numeric digit a T is returned, otherwise a NIL is returned. Decimal digits are '0' (ASCII decimal value 48) through '9' (ASCII decimal value 57).

EXAMPLES

```
     (digit-char-p #\0)                ; returns T
     (digit-char-p #\9)                ; returns T
     (digit-char-p #\A)                ; returns NIL
     (digit-char-p #\a)                ; returns NIL
     (digit-char-p #\.)                ; returns NIL
     (digit-char-p #\-)                ; returns NIL
     (digit-char-p #\+)                ; returns NIL
```

NOTE:
Other non-digit characters used in numbers are NOT included: plus (+), minus (-), exponent (e or E) and decimal point (.).

COMMON LISP COMPATABILITY:
Common LISP supports the use of an optional radix parameter. This option specifies numeric base. This allows the DIGIT-CHAR-P to function properly for hexadecimal digits (for example). Common LISP supports up to base 36 radix systems. XLISP does not support this radix parameter.

do

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(do  ( [ <binding> ... ]  ) ( <test-expr> [ <result> ] ) [ <expr> ... ]  )
     <binding>  -     a variable binding which is can take one of
                      the following forms:
                           <symbol>
                           ( <symbol> <init-expr> [<step-expr>] )
     <symbol>   -     a symbol
     <init-expr> -    an initialization expression for <symbol>
     <step-expr> -    an expression that <symbol> symbol is updated
                      at the end of each loop
     <test-expr> -    an expression to test for loop termination
     <result>   -     an optional expression for the returned result
     <expr>           -    expressions comprising the body of the loop
                      which may contain RETURNs, GOs or tags for GO
```

DESCRIPTION

The DO special form is basically a 'while' looping construct that contains symbols (with optional initializations and updates), a loop test (with an optional return value) and a block of code (expressions) to evaluate. The DO form evaluates its initializations and updates in no specified order (as opposed to DO* which does it in sequential order). The sequence of these events is:

```
<init-expr> execution
while <test-expr> do
      loop code execution
      <step-expr> execution
end-while
return <result>
```

The first form after the DO is the 'binding' form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr> and an optional <step-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. There is no specification as to the order of execution of the bindings or the step expressions - except that they happen all together.

The DO form will go through and create and initialize the symbols. This is followed by evaluating the <test-expr>. If <test-expr> returns a non-NIL value, the loop will terminate. If <test-expr> returns a NIL value then the DO will sequentially execute the <expr>'s. After execution of the loop <expr>'s, the <symbol>'s are set to the <step-expr>'s (if the <step-expr>'s exist). Then, the <test-expr> is

re-evaluated, and so on.... The value of the <result> expression is evaluated and returned. If no <result> is specified, NIL is returned. When the DO is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
(do (i)                        ; DO loop with var I
  ((eql i 0) "done")                ;   test and result
  (print i) (setq i 0) (print i)) ;   prints  NIL     0
                               ;   returns "done"

(do (i)                        ; DO loop with var I
  ((eql i 0))                       ;   test but no result
  (print i) (setq i 0) (print i)) ;   prints  NIL     0
                               ;   returns NIL

(do                            ; DO loop
   ((i 0 (setq i (1+ i)))            ;   var I=0  increment by 1
    (j 10 (setq j (1- j)))   )       ;   var J=10 decrement by 1
   ((eql i j)  "met in the middle" )    ;   test and result
   (princ i) (princ " ")             ;   prints  0 10
   (princ j) (terpri))               ;         1 9
                               ;         2 8
                               ;         3 7
                               ;         4 6
                               ;   returns "met in the middle"
```

do*

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(do* ([<binding> ...]) (<test-expr> [<result>]) [<expr> ...]
)

<binding>	-	a variable binding which is can take one of the following forms: <symbol> (<symbol> <init-expr> [<step-expr>])
<symbol>	-	a symbol
<init-expr>	-	an initialization expression for <symbol>
<step-expr>	-	an expression that <symbol> symbol is updated at the end of each loop
<test-expr>	-	an expression to test for loop termination
<result>	-	an optional expression for the returned result
<expr>	-	expressions comprising the body of the loop which may contain RETURNs, GOs or tags for GO

DESCRIPTION

The DO* special form is basically a 'while' looping construct that contains symbols (with optional initializations and updates), a loop test (with an optional return value) and a block of code (expressions) to evaluate. The DO* form evaluates its initializations and updates in sequential order (as opposed to DO which doesn't). The sequence of these events is:

```
<init-expr> execution
while <test-expr> do
      loop code execution
      <step-expr> execution
end-while
return <result>
```

The first form after the DO* is the 'binding' form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr> and an optional <step-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. There is no specification as to the order of execution of the bindings or the step expressions - except that they happen all together.

The DO* form will go through and create and initialize the symbols. This is followed by evaluating the <test-expr>. If <test-expr> returns a non-NIL value, the loop will terminate. If <test-expr> returns a NIL value then the DO* will sequentially execute the <expr>'s. After

execution of the loop <expr>'s, the <symbol>'s are set to the <step-expr>'s (if the <step-expr>'s exist). Then, the <test-expr> is re-evaluated, and so on.... The value of the <result> expression is evaluated and returned. If no <result> is specified, NIL is returned. When the DO* is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
(do                             ; DO example - won't work
  ((i 0)                        ;   var I=0
     (j i) )                      ;   var J=I (won't work)
  ( (eql i j) "done")             ;   test and result
  (print "looping"))              ; error: unbound variable - I
                                ;
(do*                              ; DO* example - will work
  ((i 0)                        ;   var I=0
     (j i) )                      ;   var J=I (proper exec. order)
  ( (eql i j) "done")             ;   test and result
  (print "looping"))              ;   returns "done"

(do* (i)                        ; DO* loop with var I
  ((eql i 0) "done")              ;   test and result
  (print i) (setq i 0) (print i)) ;   prints  NIL     0
                                ;   returns "done"

(do* (i)                        ; DO* loop with var I
  ((eql i 0))                     ;   test but no result
  (print i) (setq i 0) (print i)) ;   prints  NIL     0
                                ;   returns NIL

(do*                            ; DO* loop
   ((i 0 (setq i (1+ i)))         ;   var I=0  increment by 1
    (j 10 (setq j (1- j)))   )    ;   var J=10 decrement by 1
   ((eql i j)  "met in the middle" )    ;   test and result
   (princ i) (princ " ")          ;   prints  0 10
   (princ j) (terpri))            ;         1 9
                                ;         2 8
                                ;         3 7
                                ;         4 6
                                ;   returns "met in the middle"
```

dolist

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(dolist  ( <symbol> <list-expr> [ <result> ] ) [ <expr> ... ]  )
      <symbol>    -     a symbol
      <list-expr> -     a list expression
      <result>    -     an optional expression for the returned result
      <expr>            -     expressions comprising the body of the loop
                        which may contain RETURNs, GOs or tags for GO
```

DESCRIPTION

The DOLIST special form is basically a list-oriented 'for' looping construct that contains a loop <symbol>, a <list-expr> to draw values from, an optional return value and a block of code (expressions) to evaluate. The sequence of execution is:

```
            <symbol>  := CAR of <list-expr>
            temp-list := CDR of <list-expr>
            while  temp-list is not empty
                  loop code execution
                  <symbol>  := CAR of temp-list
                  temp-list := CDR of temp-list
            end-while
            return <result>
```

The main loop <symbol> will take on successive values from <list-expr>. The DOLIST form will go through and create and initialize the <symbol>. After execution of the loop <expr>'s, the <symbol> is set to the next value in the <list-expr>. This continues until the <list-expr> has been exhausted. The value of the <result> expression is evaluated and returned. If no <result> is specified, NIL is returned. When the DOLIST is finished execution, the <symbol> that was defined will no longer exist or retain its value. If the <list-expr> is an empty list, then no loop execution takes place and the <result> is returned.

EXAMPLES

```
      (dolist (i () "done")               ; DOLIST with I loop variable
            (print "here"))               ;   an empty list
                                    ;   and a return value
                                    ;   returns "done"

      (dolist (x '(a b c) "fini")         ; DOLIST with X loop variable
            (princ x))              ;   a list with (A B C)
                                    ;   and a return value
```

```
                                ;   prints  ABC   returns "fini"

(dolist (y '(1 2 3))                  ; DOLIST with Y loop variable
     (princ (* y y)))           ;   a list with (1 2 3)
                                ;   and no return value
                                ;   prints  149   returns NIL
                                ;   returns "met in the middle"
```

dotimes

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(dotimes  ( <symbol> <end-expr> [ <result> ] ) [ <expr> ... ]  )
     <symbol>    -     a symbol
     <end-expr>  -     an integer expression
     <result>    -     an optional expression for the returned result
     <expr>            -     expressions comprising the body of the loop
                       which may contain RETURNs, GOs or tags for GO
```

DESCRIPTION

The DOTIMES special form is basically a 'for' looping construct that contains a loop <symbol>, an <end-expr> to specify the final value for <symbol>, an optional return value and a block of code (expressions) to evaluate. The sequence of execution is:

```
           <symbol>  := 0
           while  <symbol> value is not equal to <end-expr> value
                 loop code execution
                 <symbol>  := <symbol> + 1
           end-while
           return <result>
```

The main loop <symbol> will take on successive values from zero to (<end-expr> - 1). The DOTIMES form will go through and create and initialize the <symbol> to zero. After execution of the loop <expr>'s, the <symbol> value is incremented. This continues until the <symbol> value is equal to <end-expr>. The value of the <result> expression is evaluated and returned. If no <result> is specified, NIL is returned. When the DOTIMES is finished execution, the <symbol> that was defined will no longer exist or retain its value. If the <end-expr> is zero or less, then there will be no execution of the loop body's code.

EXAMPLES

```
     (dotimes (i 4 "done") (princ i))  ; prints  0123   returns "done"
     (dotimes (i 4)        (princ i))  ; prints  0123   returns NIL
     (dotimes (i 1)        (princ i))  ; prints  0      returns NIL
     (dotimes (i 0)        (princ i))  ; returns NIL
     (dotimes (i -9)       (princ i))  ; returns NIL
```

dribble

---

type: function (subr)
location: built-in
source file: xlisp.c xlsys.c msstuff.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(transcript [<file-str>])
     <file-str> -   a string expression for a file name

DESCRIPTION

The DRIBBLE function, when called with a <file-str> argument, opens the specified file and records a transcript of the XLISP session. When DRIBBLE is called with no <file-str> argument, it closes the current transcript file (if any). DRIBBLE will return T if the specified <file-str> was successfully opened. It will return a NIL if the <file-str> was not opened successfully or if DRIBBLE was evaluated to close a transcript.

EXAMPLES

```
     (dribble "my-trans-file")          ; open file "my-trans-file"
                                  ; for a session transcript
     (+ 2 2)
     (dribble)                    ; close the transcript
```

NOTE:
It is also possible to start a transcript when invoking XLISP. To start xlisp with a transcript file of 'myfile' type in "xlisp -tmyfile".

NOTE:
The DRIBBLE function works in XLISP 2.0 for MS-DOS systems. However, depending on the sources you use - or where you got XLISP 2.0, the generic (non-DOS) systems might not have the appropriate code for DRIBBLE to work properly.

endp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(endp <list>)
     <list>   -   the list to check

DESCRIPTION

The ENDP predicate checks to see if <list> is an empty list. T is returned if the list is empty, NIL is returned if the <list> is not empty. The <list> has to be a valid list. An error is returned if it is not a list.

EXAMPLES

```
     (endp '())                  ; returns T - empty list
     (endp ())                   ; returns T - still empty
     (endp '(a b c))                  ; returns NIL

     (setq a NIL)                     ; set up a variable
     (endp a)                    ; returns T - value = empty list

     (endp "a")                  ; error: bad argument type - "a"
     (endp 'a)                   ; error: bad argument type - A
```

NOTE:
The ENDP predicate is different from the NULL and NOT predicates in that it requires a valid list.

eq

type: predicate function (subr)
location: built-in
source file: xllist.c and xlsubr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(eq <expr1> <expr2>)
     <exprN>   -   an expression to compare

DESCRIPTION

The EQ predicate checks to see if <expr1> and <expr2> are identical. T is returned if they are exactly the same internal value, NIL is returned otherwise.

EXAMPLES

```
(eq 'a 'a)                  ; returns T
(eq 1 1)                    ; returns T
(eq 1 1.0)                  ; returns NIL
(eq 1.0 1.0)                     ; returns NIL
(eq "a" "a")                     ; returns NIL
(eq '(a b) '(a b))               ; returns NIL
(eq 'a 34)                  ; returns NIL

(setq a '(a b))                  ; set value of A to (A B)
(setq b a)                  ; set B to point to A's value
(setq c '(a b))                  ; set value of C to dif. (A B)
(eq a b)                    ; returns T
(eq a c)                    ; returns NIL
```

eql

type: predicate function (subr)
location: built-in
source file: xllist.c and xlsubr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(eql <expr1> <expr2>)
      <exprN>   -   an expression to compare

DESCRIPTION

The EQL predicate checks to see if <expr1> and <expr2> are identical (in the EQ test sense - the expression values being the same exact internal values) or if they have the same value when the expressions are numbers. T is returned if they are identical or have the same numeric value, NIL is returned otherwise.

EXAMPLES

```
      (eql 'a 'a)                ; returns T
      (eql 1 1)                  ; returns T
      (eql 1 1.0)                ; returns NIL
      (eql 1.0 1.0)                   ; returns T
      (eql "a" "a")                   ; returns NIL
      (eql '(a b) '(a b))             ; returns NIL
      (eql 'a 34)                ; returns NIL

      (setq a '(a b))                 ; set value of A to (A B)
      (setq b a)                 ; set B to point to A's value
      (setq c '(a b))                 ; set value of C to dif. (A B)
      (eql a b)                  ; returns T
      (eql a c)                  ; returns NIL
```

equal

type: predicate function (subr)
location: built-in
source file: xllist.c and xlsubr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(equal <expr1> <expr2>)
     <exprN>   -   an expression to compare

DESCRIPTION

The EQUAL predicate checks to see if <expr1> and <expr2> are structurally equivalent. T is returned if they are equivalent, NIL is returned otherwise.

EXAMPLES

```
     (equal 'a 'a)                    ; returns T
     (equal 1 1)                 ; returns T
     (equal 1 1.0)                    ; returns NIL
     (equal 1.0 1.0)                  ; returns T
     (equal "a" "a")                  ; returns T
     (equal '(a b) '(a b))            ; returns T
     (equal 'a 34)                    ; returns NIL

     (setq a '(a b))                  ; set value of A to (A B)
     (setq b a)                  ; set B to point to A's value
     (setq c '(a b))                  ; set value of C to dif. (A B)
     (equal a b)                 ; returns T
     (equal a c)                      ; returns T

     (equal '(a b) '(A B))            ; returns T
     (equal '(a b) '(c d))            ; returns NIL
     (equal "a" "A")                  ; returns NIL
     (equal "abc" "abcD")             ; returns NIL
```

NOTE:
A way to view EQUAL is that if <expr1> and <expr2> were printed (via PRINT or PRINC), if they look the same, then EQUAL will return T.

error

---

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(error <err-msg> [<arg>])
<err-msg> - a string expression for the error message
<arg> - an optional expression

DESCRIPTION

The ERROR function allows the generation of a non-correctable error. A non-correctable error requires evaluation of a CLEAN-UP or TOP-LEVEL function from within the XLISP break loop to return to normal execution. The form of the message generated is:

error: <err-msg> - <arg>

From within the break-loop, if a CONTINUE function is evaluated then a an error message is generated - "error: this error can't be continued". There is no return from the ERROR function.

EXAMPLES

```
(error "fee" "fi")                ; ERROR generates the message -
                                  ; error: fee - "fi"
(error "can't get" "there")       ; ERROR generates the message -
                                  ; error: Can't get - "there"
```

COMMON LISP COMPATIBILITY:
Common LISP and XLISP have the same basic form and style for ERROR. However, the <err-msg> string in Common LISP is sent to FORMAT. FORMAT is a output function that takes in format strings that include control information. Although, XLISP does have the FORMAT function, it is not used with error messages. Porting from XLISP to Common LISP will work fine. When porting from Common LISP to XLISP, you will need to check for this embedded control information in the error messages.

NOTE:
Remember that *BREAKENABLE* needs to non-NIL for ERROR and CERROR and system errors to be caught by the normal system break loop. If *BREAKENABLE* is NIL, ERROR and CERROR and system errors can be caught by an ERRSET form. If there is no surrounding ERRSET, no error message is generated and the break loop is not entered.

error-output

---

type: system variable
location: built-in
source file: xlinit.c xlio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

error-output

DESCRIPTION

ERROR-OUTPUT is a system variable that contains a file pointer that points to the file where all error output goes to. The default file for *ERROR-OUTPUT* is the system standard error device - normally the screen.

EXAMPLES

```
     *error-output*                    ; returns #<File-Stream: #243de>
```

NOTE:
TRACE-OUTPUT, *DEBUG-IO* and *ERROR-OUTPUT* are normally all set to the same file stream - STDERR.

errset

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(errset <expr> [<print-flag>])
<expr> - an expression to be evaluated
<print-flag> - an optional expression (NIL or non-NIL)

DESCRIPTION

The ERRSET special form is a mechanism that allows the trapping of errors within the execution of <expr>. *BREAKENABLE* must be set to NIL for the ERRSET form to function. If *BREAKENABLE* is non-NIL, the normal break loop will handle the error. For ERRSET, if no error occurs within <expr>, the value of the last expression is CONSed with NIL. If an error occurs within <expr>, the error is caught by ERRSET and a NIL is returned from ERRSET. If <print-flag> is NIL, the error message normally generated by <expr> will not be printed. If <print-flag> is non-NIL or not present in the ERRSET form, the error message will be printed.

Errors from ERROR and CERROR and system errors will be handled by ERRSET. Note that the CERROR message will only include the error message portion, not the continue message portion. BREAK is not intercepted by ERRSET.

EXAMPLES

```
(nodebug)                       ; sets *BREAKENABLE* to NIL
(errset (error "hi" "ho"))          ; prints  error: hi - "ho"
                                ; returns NIL
(errset (cerror "hi" "ho" "he"))  ; prints  error: ho - "he"
                                ; returns NIL
(errset (error "hey" "ho") NIL)           ; returns NIL
(errset (break "hey"))              ; break: hey
(errset (+ 1 5) )             ; returns (6)
(errset (+ 1 "a") NIL )                   ; returns NIL
(debug)                             ; re-enable break-loop on errors
```

NOTE:
Be sure to set *BREAKENABLE* to NIL before using ERRSET and to non-NIL after using ERRSET. If you don't reset *BREAKENABLE*, no errors will be reported.

eval

---

type: function (subr)
location: built-in
source file: xlbfun.c and xleval.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(eval <expression>)
     <expression>   -   An arbitrary expression

DESCRIPTION

EVAL evaluates the <expression> and returns the resulting value.

EXAMPLES

```
     (eval '(+ 2 2))                  ; returns 4
     (eval (cons '+ '(2 2 2)))        ; returns 6
     (eval (list '+ '2 '3 ))               ; returns 5

     (setq a 10)                 ; set up A with value 10
     (setq b 220)                     ; set up B with value 220
     (eval (list '+ a b ))            ; returns 230 because
                                 ;  (list '+ a b) => '(+ 10 220)
     (eval (list '+ 'a b))            ; returns 230 because
                                 ;  (list '+ 'a b) => '(+ A 220)
                                 ;  and A has the value 10
```

evalhook

---

type: function (subr)
location: built-in
source file: xlbfun.c and xleval.c
Common LISP compatible: related
supported on: all machines

SYNTAX

```
(evalhook  <expr> <eval-expr> <apply-expr> [ <env> ] )
     <expr>           -     an expression to evaluate
     <eval-expr> -     an expression for the evaluation routine
     <apply-expr>     -     an expression for APPLY - not used
     <env>       -     an environment expression - default is NIL
```

DESCRIPTION

EVALHOOK is a function that performs evaluation. The routine specified by <eval-expr> is called with the <expr> and <env> parameters. If <eval-expr> is NIL, then the normal system evaluator is called. The <apply-hook> is a dummy parameter that is not used in the current XLISP system. The <expr> contains the expression to be evaluated. If the <env> argument to EVALHOOK is not specified, NIL is used, which specifies to use the current global environment. The <env>, if specified, is a structure composed of dotted pairs constructed of the symbol and its value which have the form ((((<sym1> . <val1>) (<sym2> . <val2>) ...))).

EXAMPLES

```
     (setq a 100)     (setq b 200)          ; set up global values
     (evalhook '(+ a b) NIL NIL)       ; returns 300    - no <env>
     (evalhook '(+ a b) NIL NIL        ; eval with a=1 and b=2
            '((((a . 1)(b . 2)))))     ;   returns 3


     (defun myeval (exp env)                ; define MYEVAL routine
        (princ "exp: ") (print exp)         ;
        (princ "env: ") (print env)         ;
        (evalhook exp #'myeval NIL env))    ;
     (defun foo (a) (+ a a))                ; create simple function
     (setq *evalhook* #'myeval)        ; and install MYEVAL as hook
     (foo 1)                           ; prints
                                  ; exp: (FOO 1) env:NIL
                                  ; exp: 1       env:NIL
                                  ; exp: (+ A A) env:((((A . 1))))
                                  ; exp: A       env:((((A . 1))))
                                  ; exp: A       env:((((A . 1))))
                                  ; returns 2
     (top-level)                  ; to clean up *evalhook*)
```

NOTE:

The EVALHOOK function and *EVALHOOK* system variable are very useful in the construction of debugging facilities within XLISP. The TRACE and UNTRACE functions use EVALHOOK and *EVALHOOK* to implement their functionality. The other useful aspect of EVALHOOK and *EVALHOOK* is to help in understanding how XLISP works to see the expressions, their environment and how they are evaluated.

CAUTION:
Be careful when using *EVALHOOK* and EVALHOOK. If you put in a 'bad' definition into *EVALHOOK*, you might not be able to do anything and will need to exit XLISP.

UNUSUAL BEHAVIOUR:
The EVALHOOK function and *EVALHOOK* system variable, by their nature, cause some unusual things to happen. After you have set *EVALHOOK* to some non-NIL value, your function will be called. However, when you are all done and set *EVALHOOK* to NIL or some other new routine, it will never be set. This is because the XEVALHOOK function (in the xlbfun.c source file) saves the old value of *EVALHOOK* before calling your routine, and then restores it after the evaluation. The mechanism to reset *EVALHOOK* is to execute the TOP-LEVEL function, which sets *EVALHOOK* to NIL.

evalhook

---

type: system variable
location: built-in
source file: xleval.c
Common LISP compatible: related
supported on: all machines

SYNTAX

evalhook

DESCRIPTION

EVALHOOK is a system variable whose value is user code that will intercept evaluations either through normal system evaluation or through calls to EVALHOOK. The default value for *EVALHOOK* is NIL, which specifies to use the built in system evaluator. If *EVALHOOK* is non-NIL, the routine is called with expression and environment parameters. If the environment argument is NIL, then the the current global environment is used. The environment, if non-NIL, is a structure composed of dotted pairs constructed of the symbol and its value which have the form ((((<sym1> . <val1>) (<sym2> . <val2>) ...))).

EXAMPLES

```
(defun myeval (exp env)                 ; define MYEVAL routine
   (princ "exp: ") (print exp)          ;
   (princ "env: ") (print env)          ;
   (evalhook exp #'myeval NIL env))     ;
(defun foo (a) (+ a a))                 ; create simple function
(setq *evalhook* #'myeval)        ; and install MYEVAL as hook
(foo 1)                           ; prints
                                ; exp: (FOO 1) env:NIL
                                ; exp: 1       env:NIL
                                ; exp: (+ A A) env:((((A . 1))))
                                ; exp: A       env:((((A . 1))))
                                ; exp: A       env:((((A . 1))))
                                ; returns 2
(top-level)                     ; to clean up *evalhook*)
```

NOTE:
The EVALHOOK function and *EVALHOOK* system variable are very useful in the construction of debugging facilities within XLISP. The TRACE and UNTRACE functions use EVALHOOK and *EVALHOOK* to implement their functionality. The other useful aspect of EVALHOOK and *EVALHOOK* is to help in understanding how XLISP works to see the expressions, their environment and how they are evaluated.

CAUTION:
Be careful when using *EVALHOOK* and EVALHOOK. If you put in a 'bad' definition into *EVALHOOK*, you might not be able to do anything and will need to exit XLISP.

UNUSUAL BEHAVIOUR:
The EVALHOOK function and *EVALHOOK* system variable, by their nature, cause some unusual things to happen. After you have set *EVALHOOK* to some non-NIL value, your function will be called. However, when you are all done and set *EVALHOOK* to NIL or some other new routine, it will never be set. This is because the XEVALHOOK function (in the xlbfun.c source file) saves the old value of *EVALHOOK* before calling your routine, and then restores it after the evaluation. The mechanism to reset *EVALHOOK* is to execute the TOP-LEVEL function, which sets *EVALHOOK* to NIL.

evenp

---

type: predicate function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(evenp <expr>)
<expr> - the integer numeric expression to check

DESCRIPTION

The EVENP predicate checks to see if the number <expr> is even. T is returned if the number is even, NIL is returned otherwise. A bad argument type error is generated if the <expr> is not a numeric expression. A bad floating point operation is generated if the <expr> is a floating point number. Zero is an even number.

EXAMPLES

```
(evenp 0)                    ; returns T
(evenp 1)                    ; returns NIL
(evenp 2)                    ; returns T
(evenp -1)                   ; returns NIL
(evenp -2)                   ; returns T

(evenp 14.0)                      ; error: bad flt. pt. op.
(evenp 'a)                   ; error: bad argument type
(setq a 2)                   ; set value of A to 2
(evenp a)                    ; returns T
```

exit

---

type: function (subr)
location: built-in
source file: xlsys.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(exit)

DESCRIPTION

The EXIT function causes the current XLISP session to be terminated. It never returns.

EXAMPLES

```
(exit)                          ; never returns
```

KEYSTROKE EQUIVALENT:
In the IBM PC and MS-DOS versions of XLISP, a CTRL-z key sequence has the same effect as doing a (EXIT). On a Macintosh, this can be accomplished by a pull-down menu or a COMMAND-q.

NOTE:
When XLISP is EXITed, any TRANSCRIPT file is automatically closed. However, other open files are not closed, and so may lose some information.

exp

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(exp <power>)
     <power>   -   floating point number/expression

DESCRIPTION

The EXP function calculates e (2.7128) raised to the specified <power> and returns the result.

EXAMPLES

```
     (exp 0.0)                  ; returns 1
     (exp 1.0)                  ; returns 2.71828  (e)
     (exp 2.0)                  ; returns 7.38906
     (exp 10.0)                 ; returns 22026.5
     (exp 0)                         ; error: bad integer operation
```

NOTE:
EXP with a large <power> like 1000.0 causes an incorrect value to be generated, with no error. The returned value will be a very large floating point number near the computer's limit (something like 1.79000e+308).

expand

---

type: function (subr)
location: built-in
source file: xlsys.c and xldmem.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(expand <segments>)
<segments> - an integer expression

DESCRIPTION

The EXPAND function expands memory by the specified number of <segments>. The expression <segments> is returned as the result. The power up default is 1000 nodes per segment. Note that ALLOC allows you to change the number of nodes per segment.

EXAMPLES

```
(room)                          ; prints  Nodes:       8000
                                ;       Free nodes:  5622
                                ;       Segments:    6
                                ;       Allocate:    1000
                                ;       Total:       92586
                                ;       Collections: 8
                                ; returns NIL
(expand 2)                      ; add more nodes
(room)                              ; prints  Nodes:       10000
                                ;       Free nodes:  7608
                                ;       Segments:    8
                                ;       Allocate:    1000
                                ;       Total:       112602
                                ;       Collections: 8
                                ; returns NIL
```

NOTE:
When GC is called or an automatic garbage collection occurs, if the amount of free memory is still low after the garbage collection, the system attempts to add more segments (an automatic EXPAND).

expt

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(expt <expr> [ <power> ... ] )
     <expr>          -    floating point number/expression
     <power>         -    integer or floating point number/expression
```

DESCRIPTION

The EXPT function raises the <expr> to the specified <power> and returns the result. If there is no <power> specified, the <expr> is returned. If there are multiple <power>'s, they will be applied sequentially to <expr>.

EXAMPLES

```
     (expt 2.0 2)                    ; returns 4
     (expt 2.0 10)                   ; returns 1024
     (expt 2 2)                 ; error: bad integer operation
     (expt 99.9)                ; returns 99.9
     (expt 2.0 2.0 2.0)              ; returns 16
```

NOTE:
EXPT with a large values like (expt 999.9 999.9) causes an incorrect value to be generated, with no error. The returned value will be a very large floating point number near the computer's limit (something like 1.79000e+308).

fboundp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(fboundp <symbol>)
      <symbol>   -   the symbol expression to check for a value

DESCRIPTION

The FBOUNDP predicate checks to see if <symbol> is a symbol with a function definition (closure) bound to it. T is returned if <symbol> has a function value, NIL is returned otherwise. Note that <symbol> is a symbol expression - it is evaluated and the resulting expression is the one that is checked.

EXAMPLES

```
(defun foo (x) (print x))          ; set up function FOO
(fboundp 'foo)                     ; returns T - value is closure
(fboundp 'defvar)            ; returns T - value is closure
(fboundp 'car)                     ; returns T - value is closure

(setq myvar 'goo)            ; set up MYVAR to have value GOO
(FBOUNDP myvar)                    ; returns NIL - because GOO has
                                   ;               no value yet
(defmacro goo () (print "hi"))          ; define GOO macro
(FBOUNDP myvar)                    ; returns T

(fboundp 'a)                       ; returns NIL
(fboundp '1)                       ; error: bad argument type - 1
(fboundp "hi")                     ; error: bad argument type - "hi"
```

first

type: function (subr)
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(first <expr>)
     <expr> - a list or list expression

DESCRIPTION

FIRST returns the first element of the expression. If the first expression is itself a list, then the sublist is returned. If the list is NIL, NIL is returned.

EXAMPLES

```
(first '(a b c))            ; returns A
(first '((a b) c d))             ; returns (A B)
(first NIL)                 ; returns NIL
(first 'a)                  ; error: bad argument type

(setq children '(amanda ben))          ; set up variable CHILDREN
(first children)            ; returns AMANDA
```

flatc

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(flatc <expr>)
<expr> - an expression

DESCRIPTION

The FLATC function determines the character length that would be printed if the <expr> were printed using PRINC. This means that the <expr> would be printed without a new-line. If <expr> is a string, it would not be printed with quotes around the string. The print character length is returned as the result.

EXAMPLES

```
(flatc 1234)                    ; returns 4
(flatc '(a b c))              ; returns 7
(flatc "abcd")                  ; returns 4
(flatc 'mybigsymbol)            ; returns 11
```

flatsize

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(flatsize <expr>)
<expr> - an expression

DESCRIPTION

The FLATSIZE function determines the character length that would be printed if the <expr> were printed using PRIN1. This means that the <expr> would be printed without a new-line. If <expr> is a string, it would be printed with quotes around the string. The print character length is returned as the result.

EXAMPLES

```
(flatsize 1234)                    ; returns 4
(flatsize '(a b c))                ; returns 7
(flatsize "abcd")            ; returns 6
(flatsize 'mybigsymbol)                 ; returns 11
```

flet

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(flet  ( [ <function> ... ]  ) <expr> ... )
     <function>  -     a function definition binding which is of the
                       form  ( <symbol> <arg-list> <body> )
     <symbol>    -     the symbol specifying the function name
     <arg-list>  -     the argument list for the function
     <body>            -     the body of the function
     <expr>            -     an expression
```

DESCRIPTION

The FLET special form is basically a local block construct that allows local <function> definitions followed by a block of code to evaluate. The first form after the FLET is the 'binding' form. It contains a series of <functions>. The FLET form will go through and define the <symbol>s of the <functions> and then sequentially execute the <expr>'s. The value of the last <expr> evaluated is returned. When the FLET is finished execution, the <symbol>'s that were defined will no longer exist.

EXAMPLES

```
     (flet ( (fozz (x) (+ x x) ))      ; an FLET with FOZZ local func.
         (fozz 2))                     ; returns 4
                                 ; FOZZ no longer exists
     (fozz 2)                    ; error: unbound function - FOZZ

                                 ; an empty flet
     (flet () (print 'a))              ; prints A
```

NOTE:
FLET does not allow recursive definitions of functions. The LABEL special form does allow this.

float

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(float <expr>)
<expr> - integer or floating point number/expression

DESCRIPTION

The FLOAT function takes a numeric expression and returns the result which is forced to be a floating point number.

EXAMPLES

```
(/ 1 2)                          ; returns 0 (integer division)
(/ (float 1) 2)                  ; returns 0.5
(float (/ 1 2))                  ; returns 0 (integer division)
(/ 1 2 3)                    ; returns 0 (integer division)
(/ (float 1) 2 3)            ; returns 0.166667
```

float-format

---

type: system variable
location: built-in
source file: xlprin.c
Common LISP compatible: no
supported on: all machines

SYNTAX

float-format

DESCRIPTION

FLOAT-FORMAT is a system variable that allows a user to specify how floating point numbers are to be printed by XLISP. The value of *FLOAT-FORMAT* should be set to one of the string expressions "%e", "%f" or "%g". These format strings are similar to C-language floating point specifications.

format	name	description
%e	exponential	The number is converted to decimal notation of the form [-]m.nnnnnnE[+-]xx. There is one leading digit. There are 6 digits after the decimal point.
%f	decimal	The number is converted to decimal notation of the form [-]mmmmmm.nnnnnn . There are as many digits before the decimal point as necessary. There are 6 digits after the decimal point.
%g	shortest	The number is converted to either the form of %e or %f, whichever produces the shortest output string. Non-significant zeroes are not printed.

The default value for *FLOAT-FORMAT* is the string "%g".

EXAMPLES

```
(setq *float-format* "%e")        ; exponential notation
(print 1.0)                  ; prints 1.000000e+00
(print -9e99)                     ; prints -9.000000e+99

(setq *float-format* "%f")        ; decimal notation
(print 1.0)                  ; prints 1.000000
(print 1.0e4)                     ; prints 10000.000000
(print -999.99e-99)               ; prints -0.000000

(setq *float-format* "%g")        ; shortest notation
(print 1.0)                  ; prints 1
```

```
(print 1.0e6)                    ; prints 1000000
(print 1.0e7)                    ; prints 1e+07
(print -999.999e99)              ; prints -9.99999e+101

(setq *float-format* "SOMETHING") ; bad notation
(print 1.0)                  ; prints SOMETHING
(setq *float-format* "%g")       ; reset to shortest notation
```

NOTE:
There can be other characters put in the string, but in general, this will not produce particularly desirable behaviour. There is no error checking performed on the format string.

floatp

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(floatp <expr>)
<expr> - the expression to check

DESCRIPTION

The FLOATP predicate checks if an <expr> is a floating point number. T is returned if <expr> is a floating point number, NIL is returned otherwise.

EXAMPLES

```
(floatp 1.2)                    ; returns T - float
(floatp '1.2)                   ; returns T - still a float
(setq a 1.234)                  ;
(floatp a)                  ; returns T - evaluates to float
(floatp 0.0)                    ; returns T - float zero

(floatp 0)                  ; returns NIL - integer zero
(floatp 1)                  ; returns NIL - integer
(floatp #x034)                  ; returns NIL - integer readmacro
(floatp 'a)                 ; returns NIL - symbol
(floatp #\a)                    ; returns NIL - character
(floatp NIL)                    ; returns NIL - NIL
(floatp #(0 1 2))           ; returns NIL - array
```

fmakunbound

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: yes
supported on: all machines

SYNTAX

(fmakunbound <symbol>)
<symbol> - an expression evaluating to a symbol

DESCRIPTION

The FMAKUNBOUND function makes a symbol's function definition unbound. The <symbol> must be a valid symbol, but it does not need to have a definition. The FMAKUNBOUND function returns the symbol as its result.

EXAMPLES

```
(defun myfn () (print "hi"))      ; define MYFN
(myfn)                            ; prints "hi"
(fmakunbound 'myfn)               ; returns MYFN
(myfn)                            ; error: unbound function - MYFN
```

NOTE:
FMAKUNBOUND is not misspelled - there is no 'e' in it.

NOTE:
The FMAKUNBOUND works on functions (closures) in the same way that MAKUNBOUND works on variables. Be sure to use the correct one for what you are unbinding. These functions do not generate an error if you try to unbind the wrong type. This is because of the definition of these functions and the fact that the function and variable name spaces are separate. You can have both a function called FOO and a variable called FOO.

NOTE:
The function FMAKUNBOUND is created in the INIT.LSP file. If it does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

format

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(format <destination> <format>  [ <expr1> ... ]  )
     <destination>    -     a required destination - must be a file
                            pointer, a stream, NIL (to create a string)
                            or T (to print to *standard-output*)
     <format>    -     a format string
     <exprN>           -     an expression
```

DESCRIPTION

The FORMAT function prints the specified expressions (if any) to the specified <destination> using the <format> string to control the print format. If the <destination> is NIL, a string is created and returned with the contents of the FORMAT. If the <destination> is T, the printing occurs to *STANDARD-OUTPUT*. FORMAT returns a NIL, if the <destination> was non-NIL. The <format> string is a string (surrounded by double-quote characters). This string contains ASCII text to be printed along with formatting directives (identified by a preceeding tilde ~ character). The character following the tilde character is not case sensitive (~a and ~A will function equivalently).

EXAMPLES

```
     (format T "Now is the time for")  ; prints  Now is the time for
     (format T "all ~A ~S to" 'good 'men)    ; prints  all GOOD MEN to
     (format T "come to the")          ; prints  come to the
     (format T "~A of their ~S"        ; prints  aid of their "party"
           "aid" "party")              ;

     (format *standard-ouput* "Hello there") ; prints  Hello there
     (format nil "ho ho ~S" 'ho)       ; returns "ho ho HO"

     (format T "this is ~%a break")          ; prints  this is
                                ;         a break
     (format T "this is a long ~
                    string")           ; prints  this is a long string
```

SUPPORTED FORMAT DIRECTIVES:
The <format> string in XLISP supports the following format directives:

directive	name	action
~A	ASCII	Print the <expr>.

If it is a string print it without quotes. This is like the PRINC function.

~S S-EXPR Print the <expr>.

If it is a string print it with quotes. This is like the PRIN1 function.

~% NEW-LINE Print a new line.

~~ TILDE Print a single tilde ~ character.

~<new-line> CONTINUE Continue the <format> string on the next line.

This signals a line break in the format. The FORMAT will ignore all white-space (blanks, tabs, newlines). This is useful when the <format> string is longer than a program line. Note that the new-line character must immediately follow the tilde character.

COMMON LISP COMPATABILITY:
The FORMAT function in Common LISP is quite impressive. It includes 26 different formatting directives. XLISP, as shown above, does not include most of these. The more difficult ones that you might encounter are the Decimal, Octal, heXidecimal, Fixed-format floating-point and Exponential floating-point. It is possible to print in octal and hexadecimal notation by setting *INTEGER-FORMAT*. It is possible to print in fixed format and exponential by setting *FLOAT-FORMAT*. However, neither of these system variables are supported in Common LISP and neither gives control over field size.

fourth

type: function (subr)
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(fourth <expr>)
     <expr> - a list or list expression

DESCRIPTION

FOURTH returns the fourth element of a list or list expression. If the list is NIL, NIL is returned.

EXAMPLES

```
     (fourth '(1 2 3 4 5))              ; returns 4
     (fourth NIL)                       ; returns NIL

     (setq kids '(junie vickie cindy chris)) ; set up variable KIDS
     (first kids)                       ; returns JUNIE
     (second kids)                      ; returns VICKIE
     (third kids)                       ; returns CINDY
     (fourth kids)                      ; returns CHRIS
     (rest kids)                 ; returns (VICKIE CINDY CHRIS)
```

NOTE:
This function is set to the same code as CADDDR.

funcall

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(funcall <function> [<arg1> ... ] )
      <function> -     the function or symbol to be called
      <argN>           -     an argument to be passed to <function>
```

DESCRIPTION

FUNCALL calls a function with a series of arguments. FUNCALL returns the result from <function>.

EXAMPLES

```
      (funcall '+ 1 2 3 4)              ; returns 10
      (funcall #'+ 1 2 3 4)             ; returns 10
      (funcall '+ '1 '2 '3)             ; returns 6

      (setq sys-add (function +))       ; returns #<Subr-+: #22c32>
      (setq a 99)                   ;
      (funcall sys-add 1 a)             ; 100
      (funcall sys-add 1 'a)            ; error: bad argument type
                                    ;   you can't add a symbol
                                    ;   only it's value

      (setq a 2) (setq b 3)         ; set A and B values
      (funcall (if (< a b) (function +) ; <function> can be computed
                     (function -))      ;
            a b)                    ; returns 5

      (defun add-to-list (arg list)           ; add a list or an atom
         (funcall (if (atom arg) 'cons  ;   to the front of a list
                                 'append)    ;
                  arg list))                 ;
      (add-to-list 'a '(b c))                ; returns (A B C)
      (add-to-list '(a b) '(b c))       ; returns (A B B C)
```

function

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(function <expr>)
     <expr>   -   an expression that evaluates to a function

DESCRIPTION

FUNCTION returns the function definition of the <expr>. Execution of the <expr> form does not occur. FUNCTION will operate on functions, special forms, lambda-expressions and macros.

EXAMPLES

```
     (function car)                    ; returns #<Subr-CAR: #23ac4>
     (function quote)             ; returns #<FSubr-QUOTE: #23d1c>
     #'quote                           ; returns #<FSubr-QUOTE: #23d1c>
     (function 'cdr)                   ; error: not a function

     (defun foo (x) (+ x x))                ; define FOO function
     (function foo)                    ; returns #<Closure-FOO: #2cfb6>
     (defmacro bar (x) (+ x x))        ; define FOOMAC macro
     (function bar)                    ; returns #<Closure-BAR: #2ceee>

     (setq my 99)                      ; define a variable
     (function my)                     ; error: unbound function
     (defun my (x) (print x))          ; define a function
     (function my)                     ; returns #<Closure-MY: #2cdd6>
                                  ;
                                  ; NOTE THAT THERE ARE 2 SYMBOLS
                                  ; A VARIABLE my AND A FUNCTION
                                  ; my.
```

READ MACRO:
XLISP supports the normal read macro of a hash and quote (#') as a short-hand method of writing the FUNCTION special form.

gc

---

type: function (subr)
location: built-in
source file: xldmem.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(gc)

DESCRIPTION

The GC function forces a garbage collection of the unused memory of XLISP. NIL is always returned.

EXAMPLES

```
(gc)                          ; force a garbage collection
```

NOTE:
The system will cause an automatic garbage collection if it runs out of free memory.

NOTE:
When GC is called or an automatic garbage collection occurs, if the amount of free memory is still low after the garbage collection, the system attempts to add more segments (an automatic EXPAND).

gcd

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(gcd [<int> ...])
     <int>   -   an integer expression

DESCRIPTION

The GCD function returns the greatest common divisor of a series of integers. If no arguments are given, a zero is returned. If only one argument is given, the absolute value of the argument is returned. The successful result is always a positive integer.

EXAMPLES

```
     (gcd 51 34)                ; returns 17
     (gcd 99 66 22)                  ; returns 11
     (gcd -99 66 -33)           ; returns 33
     (gcd -14)                  ; returns 14
     (gcd 0)                         ; returns 0
     (gcd)                      ; returns 0
     (gcd .2)                   ; error: bad argument type - 0.2
```

gc-flag

---

type: system variable
location: built-in
source file: xldmem.c
Common LISP compatible: no
supported on: all machines

SYNTAX

gc-flag

DESCRIPTION

GC-FLAG is a system variable that controls the printing of a garbage collection message. If *GC-FLAG* is NIL, no garbage collection messages will be printed. If *GC-FLAG* is non-NIL, a garbage collection message will be printed whenever a GC takes place. The default value for *GC-FLAG* is NIL. The message will be of the form:

```
[ gc: total 4000, 2497 free ]
```

EXAMPLES

```
*gc-flag*                   ; returns NIL
(gc)                        ; returns NIL
(setq *gc-flag* T)               ; set up for message
(gc)                        ; prints a gc message
```

gc-hook

---

type: system variable
location: built-in
source file: xldmem.c
Common LISP compatible: no
supported on: all machines

SYNTAX

gc-hook

DESCRIPTION

GC-HOOK is a system variable that allows a user function to be performed everytime garbage is collected (either explicitly with GC or automatically). The default value for *GC-HOOK* is NIL. When *GC-HOOK* is set to a non-NIL symbol, it is enabled to execute the specified user routine. The user routine can be a quoted symbol or a closure. There are two parameters to the user routine - the total number of nodes and current free nodes after the garbage collection.

EXAMPLES

```
      *gc-hook*                    ; returns NIL
      (gc)                         ; returns NIL

      (defun mygchook (&rest stuff)           ; define the hook
            (print stuff)                ;
            (print "my hook"))           ;
      (setq *gc-hook* 'mygchook)         ; set up *GC-HOOK*
      (gc)                         ; prints (2640 232)
                                   ;        "my hook"
                                   ; returns NIL
      (setq *gc-flag* T)                 ; turn on the system GC message
      (gc)                         ; prints
                                   ;   [ gc: total 2640, (2640 241)
                                   ;   "my hook"
                                   ;   236 free ]
                                   ; returns NIL
      (setq *gc-flag* NIL)               ; turn off GC message

      (setq *gc-hook* (lambda (x y)           ; enable user routine
                      (princ "\007")))   ;   that beeps at every GC
      (gc)                         ; beeps

      (defun expand-on-gc (total free)  ; define EXPAND-ON-GC
        (if (< (/ free 1.0 total) .1)          ; IF free/total < .10
            (progn (expand 2)                  ;    THEN expand memory
                   (princ "\007") )            ;         and beep
            )                              ;    ELSE do nothing
        )                              ; NOTE: XLISP already gets more
                                       ;       nodes automatically,
```

```
                                ;       this is just an example.
    (setq *gc-hook* 'expand-on-gc)          ; enable EXPAND-ON-GC
    (gc)                        ; beeps when low on nodes
```

NOTE:
The *GC-HOOK* and *GC-FLAG* facilities can interact. If you do printing in the *GC-HOOK* user form and enable *GC-FLAG*, the *GC-HOOK* printing will come out in the middle of the *GC-FLAG* message.

NOTE:
The *GC-HOOK* user form is evaluated after the execution of the actual garbage collection code. This means that if the user form causes an error, it does not prevent a garbage collection.

NOTE:
Since *GC-HOOK* is set to a symbol, the user defined form can be changed by doing another DEFUN (or whatever) to the symbol in *GC-HOOK*. Note also that you should define the symbol first and then set *GC-HOOK* to the symbol. If you don't, an automatic garbage collection might occur before you set *GC-HOOK* - generating an error and stopping your program.

gensym

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(gensym [<tag>])
      <tag>   -   an optional integer or string

DESCRIPTION

GENSYM generates and returns a symbol.

The default symbol form is as a character G followed by a number - Gn. The default numbering starts at 1. You can change what the generated symbol looks like. By calling GENSYM with a string <tag>, the default string is set to string parameter. If an integer number is the <tag>, the current number is set to the integer parameter.

EXAMPLES

```
      (gensym)                    ; first time => G1
      (gensym 100)                     ; returns G100
      (gensym "MYGENSYM")              ; returns MYGENSYM101
      (gensym 0)                  ; returns MYGENSYM0
      (gensym)                    ; returns MYGENSYM1
      (gensym "G")                     ; \
      (gensym 0)                  ; /  put it back to 'normal'
      (gensym)                    ; just like first time => G1
```

NOTE:
It takes 2 calls to GENSYM to set both portions of the GENSYM symbol.

NOTE:
Although it is possible to call GENSYM with numbers in the string (like "AB1"), this does generate an odd sequence. What will happen is you will get a sequence of symbols like .....AB18 AB19 AB110 AB111.....

get

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(get <symbol> <property>)
      <symbol>   -   the symbol with a property list
      <property> -   the property name being retrieved

DESCRIPTION

GET returns the value of the <property> from the <symbol>. If the <property> does not exist, a NIL is returned. The <symbol> must be an existing symbol. The returned value may be a single value or a list.

Property lists are lists attached to any user defined variables. The lists are in the form of (name1 val1 name2 val2 ....). Any number of properties may be attached to a single variable.

EXAMPLES

```
      (setq person 'bobby)              ; create a var with a value
      (putprop person 'boogie 'last-name)     ; add a LAST-NAME property
      (putprop person 'disc-jockey 'job); add a JOB property
      (get person 'last-name)                 ; retrieve LAST-NAME -
boogie
      (get person 'job)           ; retrieve JOB - disc-jockey
      (get person 'height)              ; non-existant - returns NIL
      (putprop person '(10 20 30) 'stats)     ; add STATS - a list
      (get person 'stats)               ;
```

NOTE:
You can set a property to the value NIL. However, this NIL value is indistinguishable from the NIL returned when a property does not exist.

COMMON LISP COMPATABILITY:
Common LISP allows for an optional default value, which XLISP does not support.

get-key

---

type: function (subr)
location: built-in
source file: msstuff.c and osdefs.h and osptrs.h
Common LISP compatible: no
supported on: MS-DOS compatibles

SYNTAX

(get-key)

DESCRIPTION

The GET-KEY function gets a single key stroke from the keyboard (as opposed to an entire line - as READ does).

EXAMPLES

```
(setq mychar (get-key))                ; get a character
```

NOTE:
This function is an extension of the XLISP system. It is provided in the MSSTUFF.C source code file. If your XLISP system is built for an IBM PC and compatibles or MS-DOS, this function will work. If your system is built on UNIX or some other operating system, it is unlikely that these functions will work unless you extend STUFF.C.

get-lambda-expression

type: function (subr)
location: built-in
source file: xlcont.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(get-lambda-expression <closure> )
     <closure>   -     a closure object from a previously defined
                       function.
```

DESCRIPTION

The GET-LAMBDA-EXPRESSION function takes the <closure> object and returns a reconstruction of a LAMBDA or a MACRO expression that defines the <closure>. The parameter must be a <closure> expression (of the the form #<Closure-FUNC #277e2>).

EXAMPLES

```
     (defun mine (a b) (print (+ a b))); define MINE defun
     (get-lambda-expression (function mine)) ; returns (LAMBDA (A B)
                                ;         (PRINT (+ A B)))

     (get-lambda-expression                ;
          (lambda (a) (print a))       ; returns (LAMBDA (A) (PRINT A))

     (defmacro plus (n1 n2) `(+ ,n1 ,n2))    ; define PLUS macro
     (get-lambda-expression (function plus)) ; returns
                                ;  (MACRO (N1 N2)
                                ;    (BACKQUOTE (+ (COMMA N1)
                                ;               (COMMA N2))))
```

get-macro-character

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: related
supported on: all machines

SYNTAX

(get-macro-character <char-num>)
     <char-num> -   an integer expression

DESCRIPTION

The GET-MACRO-CHARACTER function returns the code that will be executed when the specified character <char-num> is encountered by the XLISP reader. The returned value is taken from the *READTABLE* system variable which contains the reader table array. The table is 128 entries (0..127) for each of the 7-bit ASCII characters that XLISP can read. Each entry in the table must be one of NIL, :CONSTITUENT, :SESCAPE, :MESCAPE, :WHITE-SPACE, a :TMACRO dotted pair or a :NMACRO dotted pair. The GET-MACRO-CHARACTER function will return a NIL value if the table entry is NIL, :CONSTITUENT, :SESCAPE, :MESCAPE or :WHITE-SPACE. If the table entry is :TMACRO or :NMACRO, then the code associated with the entry is returned. :TMACRO is used for a terminating read-macro. :NMACRO is used for a non-terminating read-macro. GET-MACRO-CHARACTER does not differentiate whether the code returned is a :TMACRO or an :NMACRO. The function returned may be a built-in read-macro function or a user defined lambda expression. The function takes two parameters, an input stream specification, and an integer that is the character value. The <function> should return NIL if the character is 'white-space' or a value CONSed with NIL to return the value.

EXAMPLES

```
(get-macro-character #\( )        ; returns #<Subr-: #2401e>
(get-macro-character #\# )        ; returns #<Subr-: #24082>
(get-macro-character #\Space )          ; returns NIL
```

NOTE:
In the normal XLISP system the following characters have code associated with them in the *READTABLE*:

```
" # ' ( ) , ; `
```

NOTE:
The functions GET-MACRO-CHARACTER and SET-MACRO-CHARACTER are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

COMMON LISP COMPATABILITY:
The GET-MACRO-CHARACTER function is somewhat related to the Common LISP GET-DISPATCH-MACRO-CHARACTER function.

get-output-stream-list

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(get-output-stream-list <source>)
      <source>   -   an output stream expression

DESCRIPTION

The GET-OUTPUT-STREAM-LIST function empties the specified <source> and returns this data as a list. The output stream is emptied by this operation.

EXAMPLES

```
      (setq out (make-string-output-stream))  ; returns #<Unnamed-Stream:
#2d2cc>
      (format out "123")                ; add some data to output stream
      (get-output-stream-list out)      ; returns (#\1 #\2 #\3)
      (format out "123")                ; add some data to output stream
      (read out)                   ; returns 123
      (get-output-stream-list out)      ; returns NIL
```

get-output-stream-string

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(get-output-stream-string <source>)
     <source>   -   an output stream expression

DESCRIPTION

The GET-OUTPUT-STREAM-STRING function empties the specified <source> and returns this data as a single string. The output stream is emptied by this operation.

EXAMPLES

```
     (make-string-output-stream)       ; returns #<Unnamed-Stream:
#2d9c0>
     (setq out (make-string-output-stream))  ; returns #<Unnamed-Stream:
#2d95c>
     (format out "fee fi fo fum ")           ; \
     (format out "I smell the blood of ")    ;  fill up output stream
     (format out "Elmer Fudd")          ; /
     (get-output-stream-string out)          ; returns
                                ;     "fee fi fo fum I smell
                                ;       the blood of Elmer Fudd"
     (format out "~%now what")          ; add more to output stream
     (get-output-stream-string out)          ; returns "\nnow what"
     (get-output-stream-string out)          ; returns ""
     (format out "hello")               ; add more to output stream
     (read out)                  ; returns HELLO
```

go

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(go <tag-symbol>)
     <tag-symbol>   -   a symbol

DESCRIPTION

The GO special form allows 'go-to' style branching within 'block' constructs (DO, DO*, DOLIST, DOTIMES, TAGBODY, LOOP, PROG and PROG*). The <tag-symbol> is the 'label' and must exist somewhere within the 'block' that the GO occurs within. Otherwise an error will be generated - "error: no target for GO". GO never returns a value. If the <tag-symbol> exists, then the execution will continue immediately after the <tag-symbol>.

EXAMPLES

```
(defun foo (i j)            ; define FOO
  (prog ()                  ; with a PROG
       (print "begin")      ;
      start (print j)             ; tag - START
       (setq j (1- j))      ;
       (if (eql i j) (GO start)   ; 2-way branch
                  (GO end)) ;
       (print "hello")      ; won't ever be reached
   end   (print "done")                ; tag - END
       (return 42)))            ;
(foo 1 2)                   ; prints  "begin" 2 1 "done"
                            ;   returns 42
(foo 2 1)                   ; prints  "begin" 1 "done"
                            ;   returns 42
```

NOTE:
Although GO will accept a <tag-symbol> that is not a symbol, it will not find this improper <tag-symbol>. An error will be generated - "error: no target for GO".

hash

---

type: function (subr)
location: built-in
source file: xlbfun.c and xlsym.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(hash <name> <table-size> )
     <name>           -     a symbol or string expression
     <table-size>     -     an integer expression
```

DESCRIPTION

The HASH function computes and returns an integer index for a given symbol <name> and a given size of hash table <table-size>. The intention is for HASH to be used with tables made by MAKE-ARRAY and accessed by AREF.

EXAMPLES

```
     (hash "zzzz" 1000)                ; returns index 322
     (hash "ZZZZ" 1000)                ; returns index 626
     (hash 'ZZZZ  1000)                ; returns index 626

     (hash "hiho" 1000)                ; returns index 519
     (hash 'hiho  1000)                ; returns index 143
     (hash "abcd" 1000)                ; returns index 72

     (defun lookin (sym)               ; create a function to
        (aref *obarray*                ;   look inside *OBARRAY*
              (hash sym (length *obarray*))));   and look for a specific
                                ;   symbol - returns a list
     (lookin 'caar)                      ; returns the hash table entry
                                ;   (ZEROP CDDDDR CAAR HASH)
```

NOTE:
This is a useful function for creating and accessing tables. It is also useful for looking inside of XLISP's own symbol table *OBARRAY*.

if

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(if <test-expr>  <then-expr>   [ <else-expr> ]  )
      <test-expr> -     an expression
      <then-expr> -     the THEN-CLAUSE, an expression
      <else-expr> -     the ELSE-CLAUSE, an optional expression
```

DESCRIPTION

The IF special form evaluates the <test-expr>. If <test-expr> evaluates to a non-NIL value, then <then-expr> is evaluated and returned as the result. If <test-expr> evaluates to NIL and there is an <else-expr>, then the <else-expr> is evaluated and its result is returned. If there is no <else-expr> and <test-expr> evaluates to NIL, then NIL is returned as a result.

EXAMPLES

```
      (if T (print "will print")        ; prints  "will print"
            (print "won't print"))      ;

      (if NIL (print "won't print")           ;
              (print "will print"))           ; prints  "will print"

      (if 'a T NIL)                     ; returns T
      (if NIL 'nope 'yep)               ; returns YEP
```

int-char

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
versions: all machines

SYNTAX

(int-char <int>)
      <int>   -   an integer numeric expression

DESCRIPTION

The INT-CHAR function returns a character which is the result of turning the <int> expression into a character. If a <int> cannot be made into a character, an error is signalled. The range that <int> produces a valid character is 0 through 255.

EXAMPLES

```
      (int-char 48)                    ; returns #\0
      (int-char 65)                    ; returns #\A
      (int-char 97)                    ; returns #\a
      (int-char 91)                    ; returns #\[
      (int-char 10)                    ; returns #\Newline
      (int-char 999)                   ; error - character code out of
                                  ;       range - 999
```

COMMON LISP COMPATABILITY:
Common LISP specifies that INT-CHAR should return a NIL when there is no valid character for the integer value being passed in. XLISP generates an error in these cases. In some cases it is possible to substitue the CODE-CHAR function for INT-CHAR.

NOTE:
Unlike the CHAR-CODE and CHAR-INT functions, CODE-CHAR and INT-CHAR are not identical in use. CODE-CHAR accepts 0..127 for its range and then produces NIL results. INT-CHAR accepts 0..255 for its range and then produces errors.

integer-format

---

type: system variable
location: built-in
source file: xlprin.c
Common LISP compatible: no
supported on: all machines

SYNTAX

integer-format

DESCRIPTION

INTEGER-FORMAT is a system variable that allows a user to specify how integer numbers are to be printed by XLISP. The value of *INTEGER-FORMAT* should be set to one of the string expressions "%ld", "%lo" or "%lx". The character after the percent character is the alphabetic 'ell' character. These format strings are similar to C-language floating point specifications.

format	description
"%ld"	decimal
"%lu"	unsigned decimal
"%lo"	unsigned octal
"%lx"	unsigned hexadecimal

The default value for *INTEGER-FORMAT* is the string "%ld".

EXAMPLES

```
*integer-format*            ; returns "%ld"

(setq *integer-format* "%ld")          ; signed decimal
(print 1)                   ; prints 1
(print 1234)                    ; prints 1234
(print -1)                  ; prints -1
(print -1234)                   ; prints -1234

(setq *integer-format* "%lo")          ; octal notation
(print 1)                   ; prints 1
(print 1234)                    ; prints 2322
(print -1)                  ; prints 37777777777
(print -1234)                   ; prints 37777775456

(setq *integer-format* "%lx")          ; hexadecimal notation
(print 1)                   ; prints 1
(print -1)                  ; prints ffffffff
(print 1234)                    ; prints 4d2
(print -1234)                   ; prints fffffb2e

(setq *integer-format* "%u")       ; unsigned decimal
```

```
(print 1)                    ; prints 1
(print 1234)                      ; prints 1234
(print -1)                   ; prints 4294967295
(print -1234)                     ; prints 4294966062

(setq *integer-format* "hi")      ; a bad notation
(print 1)                    ; prints hi
(print 9999)                      ; prints hi

(setq *integer-format* "%ld")          ; reset to original "%ld"
```

NOTE:
There can be other characters put in the string, but in general, this will not produce particularly desirable behaviour. There is no error checking performed on the format string.

integerp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(integerp <expr>)
     <expr>   -   the expression to check

DESCRIPTION

The INTEGERP predicate checks if an <expr> is a integer number. T is returned if <expr> is a integer number, NIL is returned otherwise.

EXAMPLES

```
     (integerp 1)                    ; returns T - integer
     (integerp #x034)              ; returns T - integer readmacro
     (integerp '1)                   ; returns T - still an integer
     (setq a 14)                   ;
     (integerp a)                    ; returns T - evaluates to int.
     (integerp 0)                    ; returns T - integer zero

     (integerp 1.2)                  ; returns NIL - float
     (integerp 0.0)                  ; returns NIL - float zero
     (integerp 'a)                   ; returns NIL - symbol
     (integerp #\a)                  ; returns NIL - character
     (integerp NIL)                  ; returns NIL - NIL
     (integerp #(0 1 2))             ; returns NIL - array
```

intern

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(intern <name-str>)
      <name-str> -   a string expression

DESCRIPTION

The INTERN function takes a string name - <name-str> and creates a new interned symbol. What this means is that the symbol <name-str> will be placed into the symbol hash table *OBARRAY*. It's value will be unbound. It's property list will be NIL (empty). If the symbol already exists, no error or action is taken and the old values and property lists remain intact. The INTERN function returns the symbol as its result.

EXAMPLES

```
      (defun lookin (sym)               ; create a function to
         (aref *obarray*                ;   look inside *OBARRAY*
               (hash sym (length *obarray*))));   and look for a specific
                                 ;   symbol - returns a list

      (lookin "FINGERS")                  ; see if "FINGERS" is a symbol
                                 ;   returns (:START1) - it isn't
      (intern "FINGERS")                  ; intern "FINGERS" as a symbol
                                 ;   returns FINGERS
      (lookin "FINGERS")                  ; returns (FINGERS :START1)
      (print fingers)                     ; error: unbound variable
                                 ;   it exists, but has no value

      (lookin "TOES")                     ; returns NIL - doesn't exist
      toes                       ; error: unbound variable
      (lookin "TOES")                     ; returns (TOES)
                                 ;   the act of looking for a
                                 ;   value or using a symbol
                                 ;   causes it to be INTERNed

      (lookin "KNEECAPS")                 ; returns (MAX MAPLIST) -
                                 ;   KNEECAPS doesn't exist
      (setq kneecaps 'a-bone)                   ; create symbol with a value
      (lookin "KNEECAPS")                 ; returns (KNEECAPS MAX MAPLIST)
```

NOTE:
When you INTERN a symbol like "fingers", this gets placed in the *OBARRAY* symbol table as a lower case symbol. Note that this is

different from doing an INTERN on "FINGERS". "fingers" and "FINGERS" are two different symbols in *OBARRAY*. Remember also that normal symbols created by XLISP are upper case names. So, an intern of 'fingers or 'FINGERS are normal symbols, and will be the upper-case symbol FINGERS.

COMMON LISP COMPATABILITY:
Common LISP allows an optional package specification, which XLISP does not support.

:isnew

---

type: message selector
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(send <object> :isnew <args> )
     <object>    -     an existing object
     <args>            -     the arguments to be passed to the init. code
(send <class> :isnew <ivars> [ <cvars> [ <superclass> ] ] )
     <class>           -     an existing XLISP class
     <ivars>           -     list of instance variables for new class
     <cvars>           -     list of class variable symbols for new class
     <superclass>      -     superclass for new object
                       (the default is 'OBJECT')
```

DESCRIPTION

The :ISNEW message selector causes an instance to run its initialization method. If an :ISNEW message is sent to a class, the class definition and state will be reset as specified in the arguments of the message.

EXAMPLES

```
     (setq a-class                       ; create a new class A-CLASS
           (send class :new '(state)))   ;        with STATE
     (send a-class :answer :isnew '()    ; set up initialization
           '((setq state nil) self))     ;
     (send a-class :answer :set-it '(value)  ; create :SET-IT message
           '((setq state value)))        ;
     (setq an-obj (send a-class :new))   ; create AN-OBJ out of A-CLASS
     (send an-obj :show)                 ; returns object - STATE = NIL
     (send an-obj :set-it 5)                  ; STATE is set to 5
     (send an-obj :show)                 ; returns object - STATE = 5
     (SEND an-obj :ISNEW)                ; re-initialize AN-OBJ
     (send an-obj :show)                 ; returns object - STATE = NIL
```

&key

type: keyword
location: built-in
source file: xleval.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
&key <key-arg> ...
&key ( <key-arg> [ <key-value> [ <exist-symbol> ] ] ) ...
&key ( ( <key-symbol> <key-arg> ) [ <key-value> [ <exist-symbol> ] ] ) ...
      <key-arg>   -     keyword argument
      <key-symbol>      -     keyword argument symbol
      <key-value> -     keyword argument initialization
      <exist-symbol>    -     keyword argument existence symbol
```

DESCRIPTION

In XLISP, there are several times that you define a formal argument list for a body of code (like DEFUN, DEFMACRO, :ANSWER and LAMBDA). All of the formal arguments that are defined are required to appear in the invocation of the defined function or operation. If there are any &OPTIONAL arguments defined, they will be filled in order. There are other optional arguments called KEYWORD arguments. These arguments are not position dependent but can be specified in any order by a preceding keyword (a symbol with a leading ':'). If there is no <key-symbol> specified in the argument list, the keyword will be constructed from the <key-arg> name by adding a leading ':'. (For example a <key-arg> of FURTER will generate a keyword symbol of :FURTER).

Like the &OPTIONAL arguments, there can be initialization values provided via the <key-value> argument. If there is no <key-value> argument and no value is provided by the function call, the <key-arg> value will be NIL.

The <exist-symbol>, if it is specified, will contain a T if the <key-arg> value was supplied by the function call and a NIL if it was not supplied by the function call. This <exist-symbol> allows the programmer to test for an argument's existence. At the end of the function or operation execution, these local symbols and their values are are removed.

EXAMPLES

```
(defun foo                    ; define function FOO
  (a &key b c )                     ;   with some optional args
  (print a) (print b) (print c))    ;
(foo)                         ; error: too few arguments
(foo 1)                           ; prints 1 NIL NIL
(foo 1 2)                     ; prints 1 NIL NIL
(foo 1 :b 2 :c 3)             ; prints 1 2 3
```

```
(foo 1 :c 3 :b 2)           ; prints 1 2 3
(foo 1 :b 3 :b 2)           ; prints 1 3 NIL

(defun fee                  ; define function FEE
  (a &key (b 9 b-passed) )        ;   with some optional args
  (print a) (print b)             ;
  (if b-passed (print "b was passed")   ;
               (print "b not passed"))) ;
(fee)                       ; error: too few arguments
(fee 1)                          ; prints 1 9 "b not passed"
(fee 1 2)                   ; prints 1 9 "b not passed"
(fee 1 :b 2)                     ; prints 1 2 "b was passed"

(defun fi                   ; define function FI
  (a &key ((:mykey b) 9 b-passed) )     ;   with some optional args
  (print a) (print b)             ;
  (if b-passed (print "b was passed")   ;
               (print "b not passed"))) ;
(fi)                        ; error: too few arguments
(fi 1)                           ; prints 1 9 "b not passed"
(fi 1 2)                    ; prints 1 9 "b not passed"
(fi 1 :b 2)                 ; prints 1 9 "b not passed"
(fi 1 :mykey 2)                  ; prints 1 2 "b was passed"
```

NOTE:
There is a &ALLOW-OTHER-KEYS keyword in XLISP and Common LISP. In the case of XLISP, this keyword is extraneous since the default for keyword arguments is to allow other keys (without errors).

labels

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(labels  ( [ <function> ... ]  ) <expr> ... )
     <function> -     a function definition binding which is of the
                      form  ( <symbol> <arg-list> <body> )
     <symbol>   -     the symbol specifying the function name
     <arg-list> -     the argument list for the function
     <body>           -     the body of the function
     <expr>           -     an expression
```

DESCRIPTION

The LABELS special form is basically a local block construct that allows local <function> definitions followed by a block of code to evaluate. The first form after the labels is the 'binding' form. It contains a series of <functions>. LABELS allows the <functions> to be defined in a mutually recursive manner. (The similar FLET form does not allow this.) The LABELS form will go through and define the <symbol>s of the <functions> and then sequentially execute the <expr>'s. The value of the last <expr> evaluated is returned. When the LABELS is finished execution, the <symbol>'s that were defined will no longer exist.

EXAMPLES

```
     (labels ( (fozz (x) (+ x x) ))          ; a LABELS with FOZZ local
func.
         (fozz 2))                      ; returns 4
                                ; FOZZ no longer exists
     (fozz 2)                   ; error: unbound function - FOZZ

                                ; an empty LABELS
     (labels () (print 'a))          ; prints A

                                ; LABELS form including
     (labels ( (inc (arg) (est arg))         ;   INC definition using EST
             (est (var) (* .1 var)) )   ;   EST definition
        (inc 99) )                  ;   returns 9.9
                                ;
                                ; FLET form including
     (flet ( (inc (arg) (est arg))          ;   INC definition using EST
           (est (var) (* .1 var)) )     ;   EST definition
        (inc 99)                ; error: unbound function - EST
```

NOTE:
FLET does not allow recursive definitions of functions. The LABEL

special form does allow this.

lambda

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(lambda <arg-list> [ <body> ] )
      <arg-list>  -     A list of the formal arguments to the function
                        of the form:     ( [ <arg1> ... ]
                                         [ &optional <oarg1> ... ]
                                         [ &rest <rarg> ]
                                         [ &key ... ]
                                         [ &aux <aux1> ... ] )
      <body>            -     A series of LISP forms (expressions) that
                        are executed in order.
```

DESCRIPTION

LAMBDA returns a function definition - an executable function - that has no name.

All of the <argN> formal arguments that are defined are required to appear in a call to the defined function. If there are any &OPTIONAL arguments defined, they will be filled in order. If there is a &REST argument defined, and all the required formal arguments and &OPTIONAL arguments are filled, any and all further parameters will be passed into the function via the <rarg> argument. Note that there can be only one <rarg> argument for &REST. If there are insufficient parameters for any of the &OPTIONAL or &REST arguments, they will contain NIL. The &AUX variables are a mechanism for you to define variables local to the function definition. At the end of the function execution, these local symbols and their values are are removed.

EXAMPLES

```
      (funcall (lambda (a b) (* a b)) 4 8 )   ; evaluate a lambda function
                                    ;   returns 32
      (funcall (lambda '(a b) (+ a b)) 1 2)   ; evaluate another
                                    ;   returns 3
      (funcall (lambda (a b)                  ; evaluate a more complex
one
                (print "a no-name fnc") ;   prints "a no-name fnc"
                (* a b)) 3 8)       ;   and returns 24
```

NOTE:
Using a SETQ on a LAMBDA expression is not the same as a DEFUN. A SETQ on a LAMBDA will give the variable the value of the LAMBDA closure. This does not mean that the variable name can be used as a function.

last

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(last <list-expr>)
      <list-expr> -   a list or list expression

DESCRIPTION

The LAST function returns a list containing the last node or element of a list. If the last node is a sub-list, this is returned unaffected.

EXAMPLES

```
      (last NIL)                  ; returns NIL
      (last 'a)                   ; error: bad argument type
      (last '(A))                 ; returns (A)
      (last '(A B C D E))               ; returns (E)
      (last '( A (B C) (D E (F))))      ; returns ((D E (F)))

      (setq children '(junie vicki            ;
                  cindy chris))         ;
      (last children)                   ; returns CHRIS
```

length

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(length <expr>)
     <expr> - a list expression or string expression

DESCRIPTION

LENGTH returns the length of the <expr>. If the <expr> is a string, the number of characters is returned. If the <expr> is a list, the number of top level elements (atoms or sublists) is returned. If the list is NIL, a 0 is returned.

EXAMPLES

```
(length NIL)                        ; returns 0
(length 'a)                   ; error: bad argument type
(length '(a))                       ; returns 1
(length '(1 2 3 4 5 6))                  ; returns 6
(length '(a (b c) (d (e) f) g))           ; returns 4

(length "12345")              ; returns 5
```

let

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(let  ( [ <binding> ... ]  ) <expr> ... )
     <binding>  -     a variable binding which is of the form
                      <symbol>     or    ( <symbol> <init-expr> )
     <symbol>   -     a symbol
     <init-expr> -    an initialization expression for <symbol>
     <expr>           -     an expression
```

DESCRIPTION

The LET special form is basically a local block construct that contains symbols (with optional initializations) and a block of code (expressions) to evaluate. The first form after the LET is the 'binding' form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. There is no specification as to the order of execution of the bindings. The LET form will go through and create and initialize the symbols and then sequentially execute the <expr>'s. The value of the last <expr> evaluated is returned. When the LET is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
     (let (x y z)                        ; LET with local vars
        (print x) (print y) (print z)) ;  prints   NIL NIL NIL
     (let ((a 1) (b 2) (c 3))           ; LET with local vars & init.
       (print (+ a b c)))               ;  prints and returns 6
     (let ( (a 1) (b 2) (c (+ a b)))         ; LET with local vars &
init.
       (print (+ a b c)))               ;  error: unbound variable - A
                                ;  because (+ A B) init code
                                ;  depends on vars A and B
                                ;  which haven't been created
                                ;  yet - because of ordering
```

let*

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(let*  ( [ <binding> ... ]  ) <expr> ... )
     <binding>  -     a variable binding which is of the form
                      <symbol>     or    ( <symbol> <init-expr> )
     <symbol>   -     a symbol
     <init-expr> -    an initialization expression for <symbol>
     <expr>           -     an expression
```

DESCRIPTION

The LET* special form is basically a local block construct that contains symbols (with optional initializations) and a block of code (expressions) to evaluate. The first form after the LET* is the 'binding' form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. The execution of the bindings will occur from the first to the last binding. The LET* form will go through and create and initialize the symbols and then sequentially execute the <expr>'s. The value of the last <expr> evaluated is returned. When the LET* is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
     (let* (x y z)                       ; LET* with local vars
        (print x) (print y) (print z)) ;  prints   NIL NIL NIL
     (let* ((a 1) (b 2) (c 3))           ; LET* with local vars & init.
       (print (+ a b c)))                ;  prints and returns 6
     (let* ( (a 1) (b 2) (c (+ a b)))  ; LET* with local vars & init.
       (print (+ a b c)))                ;  prints and returns 6
                                ;  because (+ A B) init code
                                ;  depends on vars A and B
                                ;  which have been created -
                                ;  because of ordering of LET*
```

list

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(list [<expr1> ...])
     <exprN>   -   an expression

DESCRIPTION

The LIST function takes the expressions and constructs a list out of them. This constructed list is returned.

EXAMPLES

```
     (list)                          ; returns NIL
     (list NIL)                  ; returns (NIL)
     (list 'a)                   ; returns (A)
     (list 'a 'b)                    ; returns (A B)
     (list 'a 'b 'c)                 ; returns (A B C)
     (list 'a 'b NIL)            ; returns (A B NIL)
     (list '(a b) '(c d) '( (e f) ))         ; returns ((A B) (C D) ((E
F)))
     (list (+ 1 2) (+ 3 4))          ; returns (3 7)
```

listp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(listp <expr>)
     <expr>   -   the expression to check

DESCRIPTION

The LISTP predicate checks if the <expr> is a list. T is returned if <expr> is a list or an empty list (the NIL value), NIL is returned otherwise.

EXAMPLES

```
     (listp '(a b))                    ; returns T - list
     (listp NIL)                  ; returns T - NIL
     (listp '(a . b))             ; returns T - dotted pair list

     (listp (lambda (x) (print x)))          ; returns NIL - closure -
lambda
     (listp #(1 2 3))             ; returns NIL - array
     (listp *standard-output*)         ; returns NIL - stream
     (listp 1.2)                  ; returns NIL - float
     (listp #'quote)                   ; returns NIL - fsubr
     (listp 1)                    ; returns NIL - integer
     (listp object)                    ; returns NIL - object
     (listp "str")                     ; returns NIL - string
     (listp #'car)                     ; returns NIL - subr
     (listp 'a)                   ; returns NIL - symbol
```

NOTE:
NIL or '() is used in many places as a list-class or atom-class expression. Both ATOM and LISTP, when applied to NIL, return T. If you wish to check for a non-empty list, use the CONSP predicate.

load

---

type: function (subr)
location: built-in
source file: xlsys.c and xlread.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(load  <file>  [ :verbose <v-flag> ] [ :print <p-flag> ] ))
     <file>           -     a string expression or symbol
     <v-flag>    -     an optional key-word expression - default is T
     <p-flag>    -     an optional key-word expression - default is NIL
```

DESCRIPTION

The LOAD function opens the <file>, reads and evaluates all the forms within the <file>. <file> may be a string expression or a symbol. When <file> is a string, you may specify a complete file location or extensions (like "/usr/local/bin/myfile.lsp" or "A:\LISP\TIM.LSP"). If <file> is a string and includes a file type or an extension (like ".lsp"), then LOAD accesses the specified file. If there is no extension on <file>, it will add ".lsp". If the :VERBOSE keyword is present and <v-flag> is non-NIL, a load message of the form ; loading "xxxx.lsp" will be printed to *STANDARD-OUTPUT*. If the :PRINT keyword is present and <p-flag> is non-NIL, the resulting value of each top-level form in <file> will be printed to *STANDARD-OUTPUT*. If the file load was successful, then T is returned as the result. If the file load was not successful, a NIL is returned.

EXAMPLES

```
     (load 'gloop)                      ; prints  ; loading "GLOOP.lsp"
                                ; returns NIL   there is no file

     (defun foo (x) (print x))          ; create a function
     (savefun foo)                      ; create a file FOO.lsp
     (load 'foo)                ; prints  ; loading "FOO.lsp"
                                ; returns T
     (load 'foo :verbose NIL)           ; no printing    returns T
     (load 'foo :print T)               ; prints  FOO    returns T
     (load 'save :verbose T :print T)   ; prints  ; loading "FOO.lsp"
                                ; prints  FOO    returns T
     (load "foo")                       ; prints  ; loading "foo.lsp"
                                ; returns NIL    didn't work
                                ; because the file is "FOO.lsp"
     (load "FOO")                       ; prints  ; loading "FOO.lsp"
                                ; returns T did work
     (load "FOO.lsp")           ; prints  ; loading "FOO.lsp"
                                ; returns T did work
```

FILE NAMES:

In the PC and DOS world, all file names and extensions ("FOO.BAT") are automatically made uppercase. In using XLISP, this means you don't have to worry about whether the name is "foo.bat", "FOO.BAT" or even "FoO.bAt" - they will all work. However, in other file systems (UNIX in particular), uppercase and lowercase do make a difference. So, in UNIX if you do a (open 'foo-file :direction :output), this will create a file named FOO-FILE because XLISP uppercases its symbols. If you do a (open "foo-file" :direction :output), this will create a file named "foo-file" because UNIX doesn't uppercase its file names. Another case is if you do (savefun mydefun), this will create the file "MYDEFUN.lsp". So, if you are having trouble with opening and accessing files, check to make sure the file name is in the proper case.

COMMON LISP COMPATABILITY:
Common LISP has a LOAD function that is similar to XLISP's LOAD. The only difference is that Common LISP uses an optional keyword parameter :IF-DOES-NOT-EXIST which XLISP does not support.

NOTE:
In XLISP, the keyword parameters are order sensitive. If both :VERBOSE and :PRINT keywords are used, :VERBOSE must come first.

KEYSTROKE EQUIVALENT:
In the Macintosh version of XLISP, a COMMAND-l brings up a dialog box for loading a file. COMMAND-n operates similarly, except that the load is done with :VERBOSE and :PRINT flags set. These can also be accomplished by a pull-down menu selection.

logand

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(logand <expr1> ...)
     <exprN>   -   an integer expression

DESCRIPTION

The LOGAND function returns the logical (bitwise) AND of the list of expressions. If there is only one argument, it is returned unaltered. If there are two or more arguments, the LOGAND function performs the logical and operation successively applying the bitwise operation.

EXAMPLES

```
     (logand 0 0)                    ; returns 0
     (logand 0 1)                    ; returns 0
     (logand 1 0)                    ; returns 0
     (logand 1 1)                    ; returns 1
     (logand 55 #x0F)            ; returns 7
     (logand 7 #b0011)           ; returns 3
     (logand 1 2 4 8 16)             ; returns 0
     (logand 15 7 3)                 ; returns 3
```

NOTE:
XLISP does not check when read-macro expansions (like #x0FF) are out of bounds. It gives no error message and will just truncate the number to the low-order bits that it can deal with (usually 32 bits or 8 hex digits).

logior

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(logior <expr1> ...)
     <exprN>   -   an integer expression

DESCRIPTION

The LOGIOR function returns the logical (bitwise) INCLUSIVE-OR of the list of expressions. If there is only one argument, it is returned unaltered. If there are two or more arguments, the LOGIOR function performs the inclusive-or successively applying the bitwise operation.

EXAMPLES

```
     (logior 0 0)                      ; returns 0
     (logior 0 1)                      ; returns 1
     (logior 1 0)                      ; returns 1
     (logior 1 1)                      ; returns 1

     (logior 1 2 4 8 16 32 64)         ; returns 127
     (logior 5 #b010)             ; returns 7
     (logior 99 #x1FF)            ; returns 511
     (logior 99 #x400)            ; returns 1123
```

NOTE:
XLISP does not check when read-macro expansions (like #x0FF) are out of bounds. It gives no error message and will just truncate the number to the low-order bits that it can deal with (usually 32 bits or 8 hex digits).

lognot

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(lognot <expr>)
<expr> - an integer expression

DESCRIPTION

The LOGNOT function returns the logical (bitwise) INVERSION of the expression.

EXAMPLES

```
(lognot 255)                      ; returns -256
(lognot #xffff0000)               ; returns 65535
(lognot #x00000000)               ; returns -1
(lognot 1)                  ; returns -2

(logand (lognot 256) 65535)       ; returns 65279
(lognot #xFFFFFFFE)               ; returns 1
(lognot #xFFFFFFFC)               ; returns 3
```

NOTE:
XLISP does not check when read-macro expansions (like #x0FF) are out of bounds. It gives no error message and will just truncate the number to the low-order bits that it can deal with (usually 32 bits or 8 hex digits).

logxor

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(logxor <expr1> ...)
<exprN> - an integer expression

DESCRIPTION

The LOGXOR function returns the logical (bitwise) EXCLUSIVE-OR of the list of expressions. If there is only one argument, it is returned unaltered. If there are two or more arguments, the LOGXOR function performs the exclusive-or successively applying the bitwise operation.

EXAMPLES

```
(logxor 0 0)                    ; returns 0
(logxor 0 1)                    ; returns 1
(logxor 1 0)                    ; returns 1
(logxor 1 1)                    ; returns 0
(logxor #b0011 #b0101)          ; returns 6
(logxor 255 #xF0)           ; returns 15
(logxor 255 #x0F)           ; returns 240
(logxor 255 (logxor 255 99))    ; returns 99
```

NOTE:
XLISP does not check when read-macro expansions (like #x0FF) are out of bounds. It gives no error message and will just truncate the number to the low-order bits that it can deal with (usually 32 bits or 8 hex digits).

loop

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(loop  <body> ...  )
     <body>           -     a series of expressions
```

DESCRIPTION

The LOOP special form specifies a 'repeat-forever' construct. The expressions in <body> will be evaluated. When the last expression is evaluated in <body>, LOOP will then repeat the <body>. When a RETURN is evaluated within a LOOP, the specified value will be returned. LOOP itself does not generate a return value. Other exit mechanisms include GO, THROW, RETURN-FROM and errors.

EXAMPLES

```
     (setq i 65)                  ; initial value
     (loop                        ; LOOP
           (princ (int-char i) )        ;   print the character
           (if (= i 90) (return "done"))     ;   test for limit
           (setq i (1+ i) ) )           ;   increment and repeat
                                  ; prints
                                  ;   ABCDEFGHIJKLMNOPQRSTUVWXYZ
                                  ; returns "done"
```

NOTE:
If you create a LOOP with no exit mechanism, you will probably have to abort your XLISP session.

lower-case-p

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

(lower-case-p <char>)
     <char> - a character expression

DESCRIPTION

The LOWER-CASE-P predicate checks if the <char> expression is a lower case character. If <char> is lower case a T is returned, otherwise a NIL is returned. Lower case characters are 'a' (ASCII decimal value 97) through 'z' (ASCII decimal value 122).

EXAMPLES

```
(lower-case-p #\a)          ; returns T
(lower-case-p #\A)          ; returns NIL
(lower-case-p #\1)          ; returns NIL
(lower-case-p #\[)          ; returns NIL
```

macroexpand

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(macroexpand <form>)
     <form>   -   a macro form

DESCRIPTION

The MACROEXPAND function takes a <form> and recursively expands the macro definitions used in the <form>. The function returns the expansion. If the <form> does not contain a macro, the form is returned unaltered.

EXAMPLES

```
(defmacro plus (n1 n2) `(+ ,n1 ,n2))    ; define PLUS macro
(plus 1 2)                    ; returns 3
(macroexpand '(plus 3 4))          ; returns (+ 3 4)

(defmacro pl (p1 p2) `(plus ,p1 ,p2))   ; define PL macro using PLUS
(pl 3 4)                      ; returns 7
(macroexpand '(pl 3 4))                 ; returns (+ 3 4)
(macroexpand-1 '(pl 3 4))          ; returns (PLUS 3 4)
```

COMMON LISP COMPATABILITY:
Common LISP returns 2 values for its result of MACROEXPAND - the expanded form and a T or NIL value that indicates if the <form> was a macro. XLISP returns only the expanded form.

COMMON LISP COMPATABILITY:
Common LISP supports an optional argument in MACROEXPAND for the environment of the expansion. XLISP does not support this optional argument.

macroexpand-1

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(macroexpand-1 <form>)
     <form>   -   a macro form

DESCRIPTION

The MACROEXPAND-1 function takes a <form> and expands the first level of the macro definition used in the <form>. The function returns the expansion. If the <form> does not contain a macro, the form is returned unaltered.

EXAMPLES

```
(defmacro plus (n1 n2) `(+ ,n1 ,n2))    ; define PLUS macro
(plus 1 2)                    ; returns 3
(macroexpand '(plus 3 4))          ; returns (+ 3 4)
(macroexpand-1 '(plus 3 4))        ; returns (+ 3 4)

(defmacro pl (p1 p2) `(plus ,p1 ,p2))   ; define PL macro using PLUS
(pl 3 4)                      ; returns 7
(macroexpand '(pl 3 4))                 ; returns (+ 3 4)
(macroexpand-1 '(pl 3 4))          ; returns (PLUS 3 4)
```

COMMON LISP COMPATABILITY:
Common LISP returns 2 values for its result of MACROEXPAND-1 - the expanded form and a T or NIL value that indicates if the <form> was a macro. XLISP returns only the expanded form.

COMMON LISP COMPATABILITY:
Common LISP supports an optional argument in MACROEXPAND-1 for the environment of the expansion. XLISP does not support this optional argument.

macrolet

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(macrolet  ( [ <macro> ... ]  ) <expr> ... )
     <macro>          -     a macro definition binding which is of the
                      form  ( <symbol> <arg-list> <body> )
     <symbol>    -    the symbol specifying the function name
     <arg-list>  -    the argument list for the function
     <body>           -     the body of the function
     <expr>           -     an expression
```

DESCRIPTION

The MACROLET special form is basically a local block construct that allows local <macro> definitions followed by a block of code to evaluate. The first form after the macrolet is the 'binding' form. It contains a series of <macro>s. The MACROLET form will sequentially execute the <expr>'s after defining the <macro>s. The value of the last <expr> evaluated is returned. When the MACROLET is finished execution, the <symbol>'s that were defined will no longer exist.

EXAMPLES

```
                                ; a MACROLET form including
     (macrolet ((pls (n1 n2) `(+ ,n1 ,n2)))  ;  PLS macro
       (pls 4 5))                      ;  returns 9
                                ; the PLS macro no longer exists
     (pls 4 5)                  ; error: unbound function - PLS

                                ; an empty MACROLET
     (macrolet () (print 'a))          ; prints A
```

make-array

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(make-array <size>)
     <size>   -   the size (integer) of the array to be created

DESCRIPTION

MAKE-ARRAY creates an array of the specified size and returns the array. Array elements may be any valid lisp data type - including lists or arrays. Arrays made by MAKE-ARRAY and accessed by AREF are base 0. This means the first element is accessed by element number 0 and the last element is accessed by element number n-1 (where n is the array size). Array elements are initialized to NIL.

EXAMPLES

```
     (setq my-array (make-array 16))         ; make the array
     (aref my-array 0)            ; return 0th (first) element
     (aref my-array 15)                ; return 15th (last) element
     (aref my-array 16)                ; error: non existant element
     (dotimes (i 16)              ; set each element to its index
           (setf (aref my-array i) i)) ;     by the setf function

     (setq new (make-array 4))           ; make another array
     (setf (aref new 0) (make-array 4)); make new[0] an array of 4
     (setf (aref (aref new 0) 1) 'a)          ; set new[0,1] = 'a
     (setf (aref new 2) '(a b c))       ; set new[2] = '(a b c)
     my-array                     ; look at array
```

READ MACRO:
There is a built-in read-macro for arrays - # (the hash symbol). This allows you to create arbitrary arrays with initial values without going through a MAKE-ARRAY function. There is also the VECTOR function to create initialized arrays. For example:

```
     (aref #(0 1 2) 1)            ; returns 1
```

COMMON LISP COMPATABILITY:
XLISP only supports one-dimensional arrays. Common LISP supports multi-dimension arrays. Common LISP also supports various keyword parameters that are not supported in XLISP.

make-string-input-stream

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(make-string-input-stream <string> [<start-pos> [<end-pos>]])
<string> - a string expression
<start-pos> - an optional numeric expression, default value is 0 (giving the first character of the string)
<end-pos> - an optional numeric expression, default value is the length of the string

DESCRIPTION

The MAKE-STRING-INPUT-STREAM function creates an unnamed stream from the <string> expression. The stream can then be used as any other stream object. The optional <start-pos> expression specifies the starting offset of the <string> expression. A <start-pos> of 0 will start with the beginning of the <string>. The optional <end-pos> expression specifies the ending offset of the <string> expression. A <end-pos> of 4 will make the fourth character the last in the stream. If the function is successful, it returns the unnamed stream object. If the string is empty, an unnamed stream is still returned. Error conditions include <start-pos> and <end-pos> being out of bounds.

EXAMPLES

```
(make-string-input-stream "abcdefgh")   ; returns #<Unnamed-Stream:
#277e2>
(read (make-string-input-stream   ;
         "123456"))               ; returns 123456
(read (make-string-input-stream   ;
         "123456" 1))             ; returns 23456
(read (make-string-input-stream   ;
         "123456" 1 3))                ; returns 23

(read (make-string-input-stream   ;
         "123" 0))                ; returns 123
(read (make-string-input-stream   ;
         "123" 0 3))              ; returns 123
(read (make-string-input-stream   ;
         "123" 2 1))              ; returns NIL
(read (make-string-input-stream   ;
         "123" 0 4))              ; error: string index out of
                            ;        bounds - 4
```

make-string-output-stream

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(make-string-output-stream)

DESCRIPTION

The MAKE-STRING-OUTPUT-STREAM function creates and returns an unnamed output stream. The stream can then be used as any other stream object.

EXAMPLES

```
      (make-string-output-stream)       ; returns #<Unnamed-Stream:
#2d9c0>
      (setq out (make-string-output-stream))  ; returns #<Unnamed-Stream:
#2d95c>
      (format out "fee fi fo fum ")           ; \
      (format out "I smell the blood of ")    ;  fill up output stream
      (format out "Elmer Fudd")         ; /
      (get-output-stream-string out)          ; returns
                                ;     "fee fi fo fum I smell
                                ;       the blood of Elmer Fudd"
      (format out "~%now what")         ; add more to output stream
      (get-output-stream-string out)          ; returns "\nnow what"
      (format out "hello")              ; add more to output stream
      (read out)                  ; returns HELLO
```

make-symbol

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(make-symbol <symbol-str>)
     <symbol-str>   -   a string expression

DESCRIPTION

The MAKE-SYMBOL function takes a string name - <symbol-str> and creates a new symbol. This symbol is temporary and is not interned (placed) into the symbol hash table *OBARRAY*. If the symbol already exists, no error or action is taken and the old values and property lists remain intact. The MAKE-SYMBOL function returns the symbol as its result.

EXAMPLES

```
     (defun lookin (sym)               ; create a function to
        (aref *obarray*                ;   look inside *OBARRAY*
              (hash sym (length *obarray*))));   and look for a specific
                                ;   symbol - returns a list

     (lookin "FEE")                      ; returns (CHAR-INT NTH ++)
                                ;   FEE symbol doesn't exist
     (MAKE-SYMBOL "FEE")                 ; returns FEE symbol
     (lookin "FEE")                      ; returns (CHAR-INT NTH ++)
                                ;   FEE still doesn't exist
     (intern "FEE")                      ; intern FEE symbol
     (lookin "FEE")                      ; returns (FEE CHAR-INT NTH ++)
                                ;   FEE does now exist
```

NOTE:
When you MAKE-SYMBOL a symbol like "fingers", this is a lower case symbol. Note that this is different from doing a MAKE-SYMBOL on "FINGERS". "fingers" and "FINGERS" are two different symbols. Remember also that normal symbols created by XLISP are upper case names.

makunbound

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: yes
supported on: all machines

SYNTAX

(makunbound <symbol>)
     <symbol> - an expression evaluating to a symbol

DESCRIPTION

The MAKUNBOUND function makes a symbol's value unbound. The <symbol> must be a valid symbol, but it does not need to have a value. The MAKUNBOUND function returns the symbol as its result.

EXAMPLES

```
(makunbound 'florp)              ; returns FLORP
(setq myvar "hi")           ; setup MYVAR "hi"
myvar                       ; returns "hi"
(makunbound 'myvar)              ; returns MYVAR
myvar                       ; error: unbound variable
```

NOTE:
MAKUNBOUND is not misspelled - there is no 'e' in it.

NOTE:
The FMAKUNBOUND works on functions (closures) in the same way that MAKUNBOUND works on variables. Be sure to use the correct one for what you are unbinding. These functions do not generate an error if you try to unbind the wrong type. This is because of the definition of these functions and the fact that the function and variable name spaces are separate. You can have both a function called FOO and a variable called FOO.

NOTE:
The function MAKUNBOUND is created in the INIT.LSP file. If it does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

mapc

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(mapc <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPC applies the <function> to the succesive CARs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. The MAPC function returns a list that is equivalent to the first list <list1>. It's purpose is to perform operations that have side-effects. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (mapc 'princ '(hi there bob))           ; prints HITHEREBOB
                                   ;   returns (HI THERE BOB)

     (mapc '+ '(1 2 3) '(1 2 3))       ; returns (1 2 3)
                                   ;   there were no side effects

     (mapc (lambda (x y) (print (+ x y)))    ; define fun. with side
effects
           '(1 2 3) '(1 2 3))          ;   prints 2 4 6
                                   ;   returns (1 2 3)
```

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

mapcan

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(mapcan <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPCAN applies the <function> to the succesive CARs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. MAPCAN is similar to MAPCAR, except that the MAPCAN macro returns a list that is constructed via the destructive NCONC function from the results of the <function> applications. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (mapcar 'list '(1 2 3) '(a b c) ) ; returns ((1 A) (2 B) (3 C))
     (mapcan 'list '(1 2 3) '(a b c) ) ; returns (1 A 2 B 3 C)
     (mapcan 'list '(a b c)                  ; different length lists
             '(1 2 3 4 5 6))           ;   returns (A 1 B 2 C 3)
```

NOTE:
Remember that MAPCAN uses NCONC and so it deals with its list arguments destructively. It is often used when you want to remove NIL entries from the resulting list - because NCONC will take out the NILs.

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

NOTE:
The macros MAPCAN and MAPCON are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

mapcar

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(mapcar <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPCAR applies the <function> to the succesive CARs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. The MAPCAR function returns a list that is constructed from the results of the <function> applications. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (mapcar '+ '(1 2 3) '(1 2 3))           ; returns (2 4 6)
     (mapcar 'princ '(1 2 3))          ; prints 123
                                ;   returns (1 2 3)
     (mapcar '+ '(1 2 3)               ; different length lists
              '(1 2 3 4 5 6))          ;   returns (2 4 6)
```

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

mapcon

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(mapcon <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPCON applies the <function> to the successive CDRs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. The MAPCON macro is similar to the MAPLIST function, except that MAPCON returns a list that is constructed via the destructive NCONC function from the results of the <function> applications. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (maplist 'list '(a b))            ; returns (((A B)) ((B)))
     (mapcon  'list '(a b))            ; returns ((A B) (B))
```

NOTE:
Remember that MAPCON uses NCONC and so it destructively deals with its list arguments.

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

NOTE:
The macros MAPCAN and MAPCON are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

mapl

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(mapl <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPL applies the <function> to the successive CDRs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. The MAPL function returns a list that is equivalent to the first list <list1>. It's purpose is to perform operations that have side-effects. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (mapl 'print '(a b c))              ; prints (A B C)
                                   ;        (B C)
                                   ;        (C)
                                   ; returns (A B C)

     (mapl (lambda (x y) (princ x) (princ y) ; apply lambda fun. to list
                   (terpri))           ;
           '(a b c) '(1 2 3))          ; prints (A B C)(1 2 3)
                                   ;        (B C)(2 3)
                                   ;        (C)(3)
                                   ; returns (A B C)
```

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

maplist

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(maplist <function> <list1> [ <list2> ... ] )
     <function>  -     a function definition (like a LAMBDA)
                       or a function name
     <listN>           -     a list or list expression
```

DESCRIPTION

MAPLIST applies the <function> to the successive CDRs of each of the lists <listN>. Each of the lists supplies one of the arguments to <function>. The MAPLIST function returns a list that is constructed from the results of the <function> applications. If the lists are of different lengths, the shortest list will determine the number of applications of <function>.

EXAMPLES

```
     (maplist 'print '(a b c))          ; prints (A B C)
                                ;        (B C)
                                ;        (C)
                                ; returns ((A B C) (B C) (C))

     (maplist (lambda (x y)                   ; append the lists into one
                (length (append x y)))  ;   list and find it's length
              '(a b c d) '(1 2 3 4))           ; returns (8 6 4 2)
```

NOTE:
The use of the <function> will work properly when it is a quoted symbol (which is the name of the function), an unquoted symbol (whose value is a function) or a closure object (like a LAMBDA).

max

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(max <expr1> ...)
     <exprN>   -   integer or floating point number/expression

DESCRIPTION

The MAX function returns the largest numeric expression from the list of arguments.

EXAMPLES

```
     (max 1)                          ; returns 1
     (max 1 -5 9)                     ; returns 9

     (setq a '( 9 3 5 2))             ; set up a list - (9 3 5 2)
     (apply 'max a)                   ; returns 9
     (apply #'max a)                  ; returns 9
     (apply 'min a)                   ; returns 2
```

member

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(member <expr> <list-expr> [ { :test | :test-not } <test> ] )
     <expr>           -    the expression to find - an atom or list
     <list-expr> -     the list to search
     <test>           -    optional test function (default is EQL)
```

DESCRIPTION

MEMBER searches through <list-expr> for <expr>. If found, MEMBER returns the remainder of the <list-expr> starting with <expr>. If <expr> is not found, a NIL is returned. You may specify your own test with the :TEST and :TEST-NOT keywords followed by the test you which to perform.

EXAMPLES

```
     (member 'a '(1 2 3 4))            ; returns NIL
     (member '2 '(1 2 3 4))            ; returns (2 3 4)

     (setq mylist '(2 4 8 16 32 64 128 256)) ; make a numeric list
     (member 6 mylist :test '<)        ; returns (8 16 32 64 128 256)
     (member 6 (reverse mylist) :test-not '<); returns (4 2)
     (member '20 '(60 40 20 10) :test '> )   ; returns (10)

     (member '(a) '((see) (a) (cat))   ; returns ((A) (CAT))
                :test 'equal)          ;   note EQUAL as test
     (member "hi" '("a" "hi" "c")           ; returns ("hi" "c")
           :test 'string= )       ;   note STRING= as test
```

NOTE:
The MEMBER function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
Common LISP supports the use of the :KEY keyword which specifies a function that is applied to each element of <list-expr> before it is tested. XLISP does not support this.

:mescape

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

:mescape

DESCRIPTION

:MESCAPE is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :MESCAPE keyword means that the specified character is to be used as a multiple escape character. The system defines that the the vertical bar character | is the only :MESCAPE character.

EXAMPLES

```
(defun look-at (table)            ; define a function to
 (dotimes (ch 127)                ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
    (case entry             ;   entries with a function
      (:MESCAPE             ;
          (princ (int-char ch)))  ;
      (T       NIL))))      ;
 (terpri))                  ;
(look-at *readtable*)             ;  prints  |
```

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

min

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(min <expr1> ...)
     <exprN>   -   integer or floating point number/expression

DESCRIPTION

The MIN function returns the minimum (most negative or most nearly negative) numeric expression from the list of arguments.

EXAMPLES

```
(min 1)                          ; returns 1
(min 8 7 4 2)                    ; returns 2
(min 2 3 -1 -99)               ; returns -99
(setq a '( 9 3 5 2))             ; make a numeric list - (9 3 5 2)
(apply 'min a)                   ; returns 2
(apply #'min a)                  ; returns 2
(apply 'max a)                   ; returns 9
```

minusp

---

type: predicate function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(minusp <expr>)
     <expr>   -   the numeric expression to check

DESCRIPTION

The MINUSP predicate checks to see if the number <expr> is negative. T is returned if the number is negative (less than zero), NIL is returned otherwise. A bad argument type error is generated if the <expr> is not a numeric expression.

EXAMPLES

```
     (minusp 1)                  ; returns NIL
     (minusp 0)                  ; returns NIL
     (minusp -1)                 ; returns T
     (minusp -.000000005)             ; returns T
     (minusp #xFFFFFFFF)              ; returns T
     (minusp #x01)                    ; returns NIL

     (minusp 'a)                 ; error: bad argument type
     (setq a -3.5)                    ; set A to -3.5
     (minusp a)                  ; returns T
```

nconc

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(nconc [<list1> ...])
     <listN>   -   a list to DESTRUCTIVELY concatenate

DESCRIPTION

NCONC destructively concatenates a sequence of lists and returns the result of this concatentation. The destructive aspect of this operation means that the actual symbol values are used in the list-modifying operations - not copies. This means, for NCONC, that the lists are spliced together. <listN> must evaluate to a valid list. An atom for <listN> will result in an error. NIL is a valid <listN>.

EXAMPLES

```
     (setq a '(1 2 3))           ; set up A with (1 2 3)
     (setq b '(4 5 6))           ; set up B with (4 5 6)
     (setq c '(7 8 9))           ; set up C with (7 8 9)
     (NCONC a b c)                    ; returns (1 2 3 4 5 6 7 8 9)
     (setf (nth 8 a) 'end)            ; change last element of A
     (print a)                   ; prints (1 2 3 4 5 6 7 8 END)
     (print b)                   ; prints (4 5 6 7 8 END)
     (print c)                   ; prints (7 8 END)
```

:new

---

type: message selector
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(send <class> :new <args> )
      <class>          -     an existing XLISP class except for 'CLASS'
      <args>           -     the init. args for the new instance
(send class :new <ivars> [ <cvars> [ <superclass> ] ] )
      <ivars>          -     list of instance variables for new class
      <cvars>          -     list of class variable symbols for new class
      <superclass>     -     superclass for new object
                       (the default is 'OBJECT')
```

DESCRIPTION

The :NEW message selector exhibits 2 different behaviors. When you are creating an instance of a class you only need the :NEW message (consisting of the message selector and any data). When you are creating a new class with :NEW, you need to specify instance variables and optionally the class variables and superclass.

EXAMPLES

```
      (setq new-class              ; create NEW-CLASS with STATE
            (send class :new '(state)))  ;
      (setq new-obj (send new-class :new))    ; create NEW-OBJ of NEW-
CLASS
      (send new-obj :show)               ; shows the object
      (setq sub-class              ; create SUB-CLASS of NEW-CLASS
            (send class :new '(sub-state)     ;
                        '() new-class));
      (send sub-class :show)             ; show the SUB-CLASS
```

nil

---

type: system constant
location: built-in
source file: xlsym.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

nil

DESCRIPTION

The NIL constant represents the empty list or the false value - as oppossed to the true value (the symbol T). NIL can be writen as the 3 character symbol NIL or as the empty list ().

EXAMPLES

```
      (setq myvar NIL)            ; set MYVAR to False
      (setq myvar 'NIL)           ; NIL and 'NIL evaluate to NIL
      (setq myvar ())                  ; () is the empty list = NIL
      (setq myvar '())            ; () and '() evaluate to NIL
      (if nil (print "this won't print"); if/then/else
              (print "this will print"))
```

NOTE:
You can not change the value of NIL.

:nmacro

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(:nmacro . <function>)
<function> - a function

DESCRIPTION

:NMACRO is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :NMACRO keyword means that the specified character is the start of a non-terminal macro. For :NMACRO, the form of the *READTABLE* entry is a dotted pair like (:NMACRO . <function>). The <function> can be a built-in read-macro function or a user defined lambda expression. The <function> takes two parameters, an input stream specification, and an integer that is the character value. The <function> should return NIL if the character is 'white-space' or a value CONSed with NIL to return the value. The <function> will probably read additional characters from the input stream.

EXAMPLES

```
(defun look-at (table)            ; define a function to
 (dotimes (ch 127)                ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
        (if (and (consp entry)          ;   :NMACRO entries
                 (equal (car entry)     ;
                   ':NMACRO))     ;
          (princ (int-char ch)))));
 (terpri))                  ;
                            ;
(look-at *readtable*)             ;  prints  #
```

NOTE:
The system defines that the hash (#) character is a non-terminal. This is because the hash is used for a variety of 'read macro expansions' including FUNCTION, an ASCII code, and hexadecimal numbers.

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

nodebug

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: no
supported on: all machines

SYNTAX

(nodebug)

DESCRIPTION

The NODEBUG function sets *BREAKENABLE* to NIL. This has the effect of turning off the break loop for errors. NODEBUG always returns NIL. The default is DEBUG enabled.

EXAMPLES

```
(nodebug)                    ; returns NIL
(+ 1 "a")                    ; error: bad argument type
                             ; but doesn't enter break-loop
(debug)                           ; returns T
(+ 1 "a")                    ; error: bad argument type
                             ; enters break-loop
(clean-up)                   ; from within the break-loop
```

NOTE:
The functions DEBUG and NODEBUG are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

not

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(not <expr>)
     <expr>   -   the expression to check

DESCRIPTION

The NOT predicate checks to see if the <expr> is false. T is returned if the expression is NIL, NIL is returned otherwise.

EXAMPLES

```
     (not '())                  ; returns T - empty list
     (not ())                   ; returns T - still empty
     (setq a NIL)                    ; set up a variable
     (not a)                         ; returns T - value = empty list

     (not "a")                  ; returns NIL - not a list
     (not 'a)                   ; returns NIL - not a list
```

NOTE:
The NOT predicate is the same function as the NULL predicate.

nstring-downcase

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(nstring-downcase <string> [{ :start | :end } <offset>] ...)
<string> - a string expression
<offset> - an optional integer expression (for a keyword)

DESCRIPTION

The NSTRING-DOWNCASE function takes a string argument and makes it lower case. This function modifies the string (or string variable itself) - it does not just make a copy. The lower case string is returned.

The keyword arguments allow for accessing substrings within <string>. The keyword arguments require a keyword (:START or :END) first and a single integer expression second. The :START keyword specifies the starting offset for the NSTRING-DOWNCASE operation on <string>. A value of 0 starts the string at the beginning (no offset). The :END keyword specifies the end offset for the operation on <string>.

EXAMPLES

```
(nstring-downcase "ABcd+-12&[")         ; returns "abcd+-&["
(nstring-downcase "ABCDEFGH"            ;
            :start 2 :end 4); returns "ABcdEFGH"

(setq mystr "ABcdEFgh")                 ; set up variable
(nstring-downcase mystr)          ; returns "abcdefgh"
(print mystr)                     ; prints  "abcdefgh"
                              ; note that MYSTR is modified
```

nstring-upcase

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(nstring-upcase <string> [ { :start | :end } <offset> ] ... )
     <string>   -    a string expression
     <offset>   -    an optional integer expression (for a keyword)
```

DESCRIPTION

The NSTRING-UPCASE function takes a string argument and makes it upper case. This function modifies the string (or string variable itself) - it does not just make a copy. The upper case string is returned.

The keyword arguments allow for accessing substrings within <string>. The keyword arguments require a keyword (:START or :END) first and a single integer expression second. The :START keyword specifies the starting offset for the NSTRING-UPCASE operation on <string>. A value of 0 starts the string at the beginning (no offset). The :END keyword specifies the end offset for the operation on <string>.

EXAMPLES

```
     (nstring-upcase "ABcd+-12&[")          ; returns "ABCD+-&["
     (nstring-upcase "abcdefgh"        ;
                 :start 2 :end 4); returns "abCDefgh"

     (setq mystr "ABcdEFgh")                ; set up variable
     (nstring-upcase mystr)            ; returns "ABCDEFGH"
     (print mystr)                     ; prints  "ABCDEFGH"
                                ; note that MYSTR is modified
```

nth

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(nth <expr> <list-expr> )
     <expr>           -     an integer expression
     <list-expr> -     a list or list expression
```

DESCRIPTION

NTH returns the <expr>'th element of <list-expr>. If the <list-expr> is shorter than <expr>, a NIL is returned. The counting sequence is base zero - the first element is the 0th element.

EXAMPLES

```
     (nth 4 '(0 1 2 3 4 5 6))          ; returns 4
     (nth 3 '(a b))                    ; returns NIL

     (nth 4 'a)                   ; error: bad argument type
     (nth 3 "abcdefg")            ; error: bad argument type
```

nthcdr

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(nthcdr <expr> <list-expr> )
      <expr>           -     an integer expression
      <list-expr> -     a list or list expression
```

DESCRIPTION

NTHCDR returns the <expr>'th CDR of <list-expr>. If the <list-expr> is shorter than <expr>, a NIL is returned. The counting sequence is base zero - the first element is the 0th element.

EXAMPLES

```
      (nthcdr 4 '(0 1 2 3 4 5 6))       ; returns (4 5 6)
      (nthcdr 3 '(a b))           ; returns NIL

      (nthcdr 4 'a)                     ; error: bad argument type
```

null

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(null <expr>)
     <expr> - the expression to check

DESCRIPTION

The NULL predicate checks <expr> for an empty list. T is returned if the list is empty, NIL is returned otherwise. The <expr> does not have to be a valid list, but if it is not a list then NIL is returned as the result.

EXAMPLES

```
(null '())                  ; returns T - empty list
(null ())                   ; returns T - still empty
(setq a NIL)                     ; set up a variable
(null a)                    ; returns T - value = empty list

(null "a")                  ; returns NIL - not a list
(null 'a)                   ; returns NIL - not a list
```

NOTE:
The NULL predicate is the same function as the NOT predicate.

numberp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(numberp <expr>)
     <expr>   -   the expression to check

DESCRIPTION

The NUMBERP predicate checks if an <expr> is a number. T is returned if <expr> is an integer or floating point number, NIL is returned otherwise.

EXAMPLES

```
     (numberp 1)                ; returns T - integer
     (numberp 1.2)                   ; returns T - float
     (numberp '1)                    ; returns T - still an integer
     (numberp #x034)                 ; returns T - readmacro produces
                                ;             an integer

     (numberp 'a)                    ; returns NIL - symbol
     (numberp #\a)                   ; returns NIL - character
     (numberp NIL)                   ; returns NIL - NIL
     (numberp #(0 1 2))              ; returns NIL - array
```

obarray

---

type: system variable
location: built-in
source file: xlsym.c
Common LISP compatible: no
supported on: all machines

SYNTAX

obarray

DESCRIPTION

OBARRAY is the system variable that contains the system symbol table. This symbol table is an XLISP array that is constructed out of lists.

EXAMPLES

```
(defun lookin (sym)                ; create a function to
   (aref *obarray*                 ;   look inside *OBARRAY*
         (hash sym (length *obarray*))));   and look for a specific
                                   ;   symbol - returns a list
                                   ;
(lookin "CAR")                     ; returns (TEST PEEK CAR)
(lookin "car")                     ; returns NIL
```

NOTE:
When looking into *OBARRAY* or INTERNing symbols, remember that "car" and "CAR" are two different symbols in *OBARRAY*. Remember also that normal symbols created by XLISP are upper case names. So, if you type in "car" as a normal symbol, it will be the symbol "CAR" after this normal upper-casing operation.

object

---

type: object
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

object

DESCRIPTION

OBJECT is an object class. An object is a composite structure that contains internal state information, methods (which respond to messages), a pointer to the object's class and a pointer to the object's super-class. XLISP contains two built in objects: OBJECT and CLASS. OBJECT is the superclass for the CLASS object.

EXAMPLES

```
      (send object :show)               ; look at the object definition

                                  ; example use of objects
      (setq my-class                      ; new class MY-CLASS with STATE
            (send class :new '(state)))  ;
      (send my-class :answer :isnew '() ; set up initialization
            '((setq state nil) self))    ;
      (send my-class :answer :set-it '(value) ; create :SET-IT message
            '((setq state value)))       ;
      (setq my-obj (send my-class :new)); create MY-OBJ out of MY-CLASS
      (send my-obj :set-it 5)                 ; STATE is set to 5
```

OBJECT DEFINITION:
The internal definition of the OBJECT object instance is:

```
      Object is #<Object: #23fd8>, Class is #<Object: #23fe2>
        MESSAGES = ((:SHOW . #<Subr-: #23db2>)
                    (:CLASS . #<Subr-: #23dee>)
                    (:ISNEW . #<Subr-: #23e2a>))
        IVARS = NIL
        CVARS = NIL
        CVALS = NIL
        SUPERCLASS = NIL
        IVARCNT = 0
        IVARTOTAL = 0
      #<Object: #23fd8>
```

The class of OBJECT is CLASS. There is no superclass of OBJECT. Remember that the location information (like #23fd8) varies from system to system, yours will probably look different.

BUILT-IN METHODS:
The built in methods in XLISP include:

<message>	operation
:ANSWER	Add a method to an object.
:CLASS	Return the object's class.
:ISNEW	Run initialization code on object.
:NEW	Create a new object (instance or class).
:SHOW	Show the internal state of the object.

MESSAGE STRUCTURE:
The normal XLISP convention for a <message> is to have a valid symbol preceeded by a colon like :ISNEW or :MY-MESSAGE. However, it is possible to define a <message> that is a symbol without a colon, but this makes the code less readable.

objectp

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(objectp <expr>)
<expr> - the expression to check

DESCRIPTION

The OBJECTP predicate checks if the <expr> is an object. T is returned if <expr> is an object, NIL is returned otherwise.

EXAMPLES

```
(objectp object)            ; returns T
(objectp class)                  ; returns T
(objectp NIL)                    ; returns NIL
(objectp '(a b))            ; returns NIL
```

oddp

type: predicate function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(oddp <expr>)
<expr> - the integer numeric expression to check

DESCRIPTION

The ODDP predicate checks to see if the number <expr> is odd. T is returned if the number is odd, NIL is returned otherwise. A bad argument type error is generated if the <expr> is not a numeric expression. A bad floating point operation is generated if the <expr> is a floating point number. Zero is an even number.

EXAMPLES

```
(oddp 0)                    ; returns NIL
(oddp 1)                    ; returns T
(oddp 2)                    ; returns NIL
(oddp -1)                   ; returns T
(oddp -2)                   ; returns NIL

(oddp 13.0)                 ; error: bad flt. pt. op.
(oddp 'a)                   ; error: bad argument type
(setq a 3)                  ; set value of A to 3
(oddp a)                    ; returns T
```

open

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(open  <file> [ :direction <in-out> ]  )
     <file>           -     a string expression or symbol
     <in-out>    -     an optional keyword symbol that must either
                       :INPUT or :OUTPUT.  The default is :INPUT.
```

DESCRIPTION

The OPEN function opens the <file> for input or output. The <file> may be a string expression or a symbol. Following the <file>, there is an optional keyword, :DIRECTION. The argument following this is either :INPUT or :OUTPUT which specifies the direction of the file. If no :DIRECTION is specified, the default is :INPUT. When <file> is a string, you may specify a complete file location or extensions (like "/usr/local/bin/myfile.lsp" or "A:\LISP\TIM.BAT"). If the file open was successful, then a file pointer of the form #<File: #99999> is returned as the result. If the file open was not successful, a NIL is returned. For an input file, the file has to exist, or an error will be signaled.

EXAMPLES

```
     (setq f (open 'mine :direction :output)); create file MINE
     (print "hi" f)                    ; returns "hi"
     (close f)                   ; file contains <hi>   <NL>
     (setq f (open 'mine :direction :input)) ; open MYFILE for input
     (read f)                    ; returns "hi"
     (close f)                   ; close it
```

FILE NAMES:
In the PC and DOS world, all file names and extensions ("FOO.BAT") are automatically made uppercase. In using XLISP, this means you don't have to worry about whether the name is "foo.bat", "FOO.BAT" or even "FoO.bAt" - they will all work. However, in other file systems (UNIX in particular), uppercase and lowercase do make a difference. So, in UNIX if you do a (open 'foo-file :direction :output), this will create a file named "FOO-FILE" because XLISP uppercases its symbols. If you do a (open "foo-file" :direction :output), this will create a file named "foo-file" because UNIX doesn't uppercase its file names. Another case is if you do (savefun mydefun), this will create the file "MYDEFUN.lsp". So, if you are having trouble with opening and accessing files, check to make sure the file name is in the proper case.

COMMON LISP COMPATABILITY:
Common LISP supports bidirectional files. So, porting Common LISP code

may be difficult to port if it uses these other file types.

&optional

type: keyword
location: built-in
source file: xleval.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
&optional [ <opt-arg> | ( <opt-arg> [ <opt-value> [ <opt-symbol> ] ] ) ]
...
      <opt-arg>   -     optional argument
      <opt-value> -     optional argument initialization
      <exist-symbol>    -     optional argument existence symbol
```

DESCRIPTION

In XLISP, there are several times that you define a formal argument list for a body of code (like DEFUN, DEFMACRO, :ANSWER and LAMBDA). All of the formal arguments that are defined are required to appear in the invocation of the defined function or operation. If there are any &OPTIONAL arguments defined, they will be filled in order. If there are insufficient parameters for the &OPTIONAL arguments, they will contain NIL, unless the arguments have an <opt-value> initial value specified. The <exist-symbol>, if it is specified, will contain a T if the <opt-arg> was supplied by the function call and a NIL if it was not supplied by the function call. This <exist-symbol> allows the programmer to test for an arguments existence. At the end of the function or operation execution, these local symbols and their values are are removed.

EXAMPLES

```
(defun foo                    ; define function FOO
  (a &optional b (c 1) )           ;   with some optional args
  (print a) (print b) (print c))   ;
(foo)                         ; error: too few arguments
(foo 1)                            ; prints 1 NIL 1
(foo 1 2)                     ; prints 1 2 1
(foo 1 2 3)                   ; prints 1 2 3

(defun fee                    ; define function FEE
  (a &optional (b 9 b-passed) )          ;   with some optional args
  (print a) (print b)              ;
  (if b-passed (print "b was passed")   ;
               (print "b not passed"))) ;
(fee 1)                            ; prints 1 9 "b not passed"
(fee 1 2)                     ; prints 1 2 "b was passed"
```

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(or  [ <expr1> ... ] )
     <exprN>          -     an expression
```

DESCRIPTION

The OR special form evaluates a sequence of expressions and returns the effect of a logical INCLUSIVE-OR operation on the expressions. If all of the expressions are NIL, NIL is returned as OR's result. Evaluation of the expressions will stop when an expression evaluates to something other than NIL, none of the subsequent expressions will be evaluated. If there are no expressions, OR returns NIL as its result.

EXAMPLES

```
(or NIL NIL NIL)             ; returns NIL
(or NIL T NIL)                    ; returns T
(or NIL (princ "hi") (princ "ho")); prints  hi  and returns "hi"
(or T T T)                   ; returns T
(or)                         ; returns NIL

(setq a 5)  (setq b 6)            ; set up A and B
(if (or (< a b) (< b a))          ; if
   (print "not equal")            ;  then
   (print "equal"))               ;  else
                             ; prints  "not equal"
```

peek

---

type: function (subr)
location: built-in
source file: xlsys.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(peek <address>)
      <address> -   an integer expression

DESCRIPTION

The PEEK function returns the internal memory value at the <address>. The returned value is an integer.

EXAMPLES

```
      (setq var 0)                        ; set up VAR with 0
      (address-of var)              ; returns 123224
      (address-of 'var)             ; returns 161922
      (peek (address-of var))                   ; returns 83951616
      (peek (1+ (address-of var)))      ; returns 16777216
      (peek (+ 2 (address-of var)))           ; returns 0  <-- value of
VAR
      (setq var 14)                       ; change the value to 14
      (peek (+ 2 (address-of var)))           ; returns 14
      (setq var 99)                       ; change the value to 99
      (peek (+ 2 (address-of var)))           ; returns 99
```

CAUTION:
Be careful when modifying the internal state of XLISP. If you have modified it, it would be a good idea to exit XLISP and re-enter before doing any work you really want to retain.

ADDITIONAL CAUTION:
It is possible to PEEK and POKE not just XLISP's memory put other parts of your computer's memory. Be very careful when doing this. Also, in some computers, just looking at a memory location can cause things to happen - I/O locations fall in this category.

peek-char

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(peek-char [<skip-flag> [<source>]])
<skip-flag> - an optional expression - default is NIL
<source> - an optional source - must be a file pointer
or stream, the default is *standard-input*

DESCRIPTION

The PEEK-CHAR function looks at a single character from the specified <source>. The character looked-at is returned as an integer value for the result. If the <skip-flag> expression is NIL, then the next character will be looked-at, without advancing the position within the file. If the <skip-flag> expression is non-NIL, then the next non-white-space character will be looked-at. This skipping does advance the position within the file. White-space characters include blank, tab and new-line characters. If <skip-flag> is not used, no skipping will occur. The <source> may be a file pointer or a stream. If there is no <source>, *STANDARD-INPUT* is the default. If an end-of-file is encountered in the <source>, then NIL will be returned as the result.

EXAMPLES

```
(setq fp (open "f" :direction :output)) ; create file "f"
(print 12 fp)                           ;
(princ "  34" fp)  (terpri fp)          ;
(close fp)                      ;
                                ;
(setq fp (open "f" :direction :input))  ; open "f" for reading
(peek-char NIL fp)                      ; returns #\1
(peek-char NIL fp)                      ; returns #\1 - didn't advance
(read-char fp)                          ; returns #\1 - force advance
(peek-char NIL fp)                      ; returns #\2
(read-char fp)                          ; returns #\2 - force advance
(peek-char NIL fp)                      ; returns #\Newline
(peek-char T fp)                ; returns #\3 - skipped blanks
(read-line fp)                          ; returns "34"
(close fp)                      ;
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP PEEK-CHAR functions are compatable for simple cases. They both allow for the optional <skip-flag> and <source>. However, in Common LISP, there are addition parameters which occur right after <source> that support various end-of-file operations and recursive calls. So, when porting from Common LISP to XLISP, remember there are

additional arguments in Common LISP's PEEK-CHAR that are not supported in XLISP.

plusp

---

type: predicate function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(plusp <expr>)
     <expr>   -   the numeric expression to check

DESCRIPTION

The PLUSP predicate checks to see if the number <expr> is positive. T is returned if the number is positive (greater than 0), NIL is returned otherwise. A bad argument type error is generated if the <expr> is not a numeric expression.

EXAMPLES

```
     (plusp 0)                  ; returns NIL
     (plusp 1)                  ; returns T
     (plusp -1)                 ; returns NIL
     (plusp #xFFFFFFFF)              ; returns NIL
     (plusp #x0FFFFFFF)              ; returns T

     (plusp 'a)                 ; error: bad argument type
     (setq a 4)                 ; set value of A to 4
     (plusp a)                  ; returns T
```

poke

---

type: function (subr)
location: built-in
source file: xlsys.c
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(poke  <address>  <expr> )
      <address>   -     an integer expression
      <expr>            -     an integer expression
```

DESCRIPTION

The POKE function writes the <expr> at the internal memory value at the specified <address>. The returned value is <expr>. Be very careful with this function.

EXAMPLES

```
      (setq var 0)                        ; set up VAR with 0
      (address-of var)              ; returns 123224
      (address-of 'var)             ; returns 161922
      (peek (address-of var))                   ; returns 83951616
      (peek (1+ (address-of var)))      ; returns 16777216
      (peek (+ 2 (address-of var)))             ; returns 0  <-- value of
VAR
      (setq var 14)                       ; change the value to 14
      (peek (+ 2 (address-of var)))             ; returns 14
      (poke (+ 2 (address-of var)) 1023); POKE the value to 1023
      (print var)                   ; prints  1023
```

CAUTION:
Be careful when modifying the internal state of XLISP. If you have modified it, it would be a good idea to exit XLISP and re-enter before doing any work you really want to retain.

ADDITIONAL CAUTION:
It is possible to PEEK and POKE not just XLISP's memory but other parts of your computer's memory. Be very careful when doing this. Also, in some computers, just looking at a memory location can cause things to happen - I/O locations fall in this category.

pp

---

type: function (subr)
location: extension
source file: pp.lsp
Common LISP compatible: no
supported on: all machines

SYNTAX

```
(pp  <expr> [ <destination>  [ <break-size> ] ] )
     <expr>           -     an expression to be pretty printed
     <destination>    -     an optional destination - must be a file
pointer
                      or stream, the default is *standard-output*
     <break-size>     -     an integer expression - default is 45
```

DESCRIPTION

The PP function produces a pretty looking version of the <expr> and prints it to the specified <destination>. If <expr> is an atom (like a string, a symbol, a number, etc.) PP will print it like PRINT. If <expr> is a list, it will perform indenting. T is always returned as the result of PP. The <destination> may be a file pointer or a stream. If there is no <destination> (or it is NIL), *STANDARD-OUTPUT* is the default. The <break-size> paramter is used to determine when an expression should be broken into multiple lines. The default <break-size> is 45.

EXAMPLES

```
     (pp 'a)                          ; prints  A    returns T
     (pp "abcd")                 ; prints  "abcd"    returns T
     (pp '(a (b c)))                  ; prints  (A (B C)) returns T
```

COMMON LISP COMPATABILITY:
In Common LISP, the PP functionality is provided with the PPRINT function. The PP function was available in previous XLISP releases and is still accessible. It does allow you to specify the length through <break-size>, which is not available in PPRINT.

NOTE:
PP is an extension to the XLISP system provided in the PP.LSP file. The default system and INIT.LSP start-up code do not automatically load PP.LSP. You need to specifically LOAD it yourself during your XLISP session. Another alternative is to put the code or a (LOAD "pp") in the INIT.LSP file. The PP.LSP should have come with your XLISP system. If it doesn't, refer to the EXAMPLE PROGRAMS chapter.

pprint

---

type: function (subr)
location: built-in
source file: xlpp.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(pp <expr> [<destination>])
<expr> - an expression to be pretty printed
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The PPRINT function produces a pretty looking version of the <expr> and prints it to the specified <destination>. If <expr> is an atom (like a string, a symbol, a number, etc.) PPRINT will print it like PRINT. If <expr> is a list, it will perform indenting, as necessary. NIL is always returned as the result of PPRINT. The <destination> may be a file pointer or a stream. If there is no <destination> (or it is NIL), *STANDARD-OUTPUT* is the default.

EXAMPLES

```
(pprint 'a)      ; prints  A    returns T
(pprint "abcd")        ; prints  "abcd"    returns T

(pprint '(a-very-long-name (first list) (second list)))
                 ;
                 ; prints (A-VERY-LONG-NAME (FIRST LIST)
                 ;                    (SECOND LIST))
```

COMMON LISP COMPATABILITY:
Common LISP specifies that PPRINT with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

prin1

---

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(prin1 <expr> [<destination>])
<expr> - an expression
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The PRIN1 function prints the <expr> to the specified <destination>. The <expr> is printed without a new-line. If <expr> is a string, it will be printed with quotes around the string. The <expr> is returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default. The TERPRI function is used to terminate the print lines produced.

EXAMPLES

```
(prin1 'a)                    ; prints  A without <NL>
(prin1 '(a b))                     ; prints  (A B)  without <NL>
(prin1 2.5)                   ; prints  2.5    without <NL>
(prin1 "hi")                       ; prints  "hi"   without <NL>

(setq f (open "f" :direction :output))  ; create file
(prin1 "hi" f)                     ; returns "hi"
(prin1 1234 f)                     ; returns 1234
(prin1 "he" f)                     ; returns "he"
(close f)                     ; file contains <"hi"1234"he">
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

princ

---

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(princ <expr> [<destination>])
<expr> - an expression
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The PRINC function prints the <expr> to the specified <destination>. The <expr> is printed without a new-line. If <expr> is a string, it will not be printed with quotes around the string. The <expr> is returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default. The TERPRI function is used to terminate the print lines produced.

EXAMPLES

```
(princ 'a)                    ; prints  A without <NL>
(princ '(a b))                     ; prints  (A B)  without <NL>
(princ 99)                    ; prints  99     without <NL>
(princ "hi")                       ; prints  hi     without <NL>

(setq f (open "f" :direction :output))  ; create file
(princ "hi" f)                     ; returns "hi"
(princ 727 f)                      ; returns 727
(princ "ho" f)                     ; returns "ho"
(close f)                     ; file contains <hi727ho>
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

print

---

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(print <expr> [<destination>])
<expr> - an expression
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The PRINT function prints the <expr> to the specified <destination>. The <expr> is printed followed by a new-line. If <expr> is a string, it will be printed with quotes around the string. The <expr> is returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default.

EXAMPLES

```
(print 'a)                    ; prints  A with <NL>
(print '(a b))                      ; prints  (A B)  with <NL>
(print 99)                    ; prints  99     with <NL>
(print "hi")                       ; prints  "hi"   with <NL>

(setq f (open "f" :direction :output))  ; create file
(print "hi" f)                       ; returns "hi"
(print 727 f)                       ; returns 727
(print "ho" f)                       ; returns "ho"
(close f)                     ; file contains "hi"<NL>
                              ;               727<NL>
                              ;          "ho"<NL>
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

print-case

---

type: system variable
location: built-in
source file: xlprin.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

print-case

DESCRIPTION

PRINT-CASE is a system variable that allows a user to specify how symbols are to be printed by XLISP. If *PRINT-CASE* is set to :DOWNCASE, all symbols will be printed in lower case characters. If *PRINT-CASE* is set to :UPCASE, all symbols will be printed in upper case characters. If *PRINT-CASE* is set to anything other than :UPCASE or :DOWNCASE, all symbols will be printed in upper case characters. The default value for *PRINT-CASE* is the keyword :UPCASE.

EXAMPLES

```
(setq *print-case* :downcase)          ; returns :downcase
(setq a 'b)                  ; returns b

(setq *print-case* 'foo)          ; returns FOO
(setq a 'b)                  ; returns B

(setq *print-case* :upcase)       ; returns :UPCASE
(setq a 'b)                  ; returns B
```

COMMON LISP COMPATABILITY:
Common LISP supports a third keyword :CAPITALIZE. XLISP does not support this, but this should not be a major problem. If set to :CAPITALIZE, XLISP will print all symbols in upper-case characters.

prog

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(prog  ( [ <binding> ... ]  )  [ <expr> ... ]  )
     <binding>  -     a variable binding which is can take one of
                      <symbol>    or     ( <symbol> <init-expr> )
     <symbol>   -     a symbol
     <init-expr> -    an initialization expression for <symbol>
     <expr>           -     expressions comprising the body of the loop
                      which may contain RETURNs, GOs or tags for GO
```

DESCRIPTION

The PROG special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains symbols (with optional initializations) and a block of code (expressions) to evaluate. The PROG form evaluates its initializations in no specified order (as opposed to PROG* which does it sequential order). The first form after the PROG is the <binding> form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. There is no specification as to the order of execution of the bindings or the step expressions - except that they happen all together. If a RETURN form is evaluated, its value will be returned. Otherwise, NIL is returned. When the PROG is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
     (prog () (print "hello"))          ; prints  "hello"  returns NIL
     (prog (i j)                   ; PROG with vars I and J
           (print i) (print j))         ; prints  NIL NIL  returns NIL
     (prog ((i 1) (j 2))                ; PROG with vars I and J
           (print i) (print j)          ;
           (return (+ i j)))                 ; prints  1 2      returns 3
```

prog*

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(prog*  ( [ <binding> ... ]  )  [ <expr> ... ]  )
     <binding>  -     a variable binding which is can take one of
                      <symbol>     or    ( <symbol> <init-expr> )
     <symbol>   -     a symbol
     <init-expr> -    an initialization expression for <symbol>
     <expr>           -     expressions comprising the body of the loop
                      which may contain RETURNs, GOs or tags for GO
```

DESCRIPTION

The PROG* special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains symbols (with optional initializations) and a block of code (expressions) to evaluate. The PROG* form evaluates its initializations in sequential order (as opposed to PROG which does it in no specified order). The first form after the PROG* is the 'binding' form. It contains a series of <symbol>'s or <binding>'s. The <binding> is a <symbol> followed by an initialization expression <init-expr>. If there is no <init-expr>, the <symbol> will be initialized to NIL. The order of execution of the bindings is sequential. If a RETURN form is evaluated, its value will be returned. Otherwise, NIL is returned. When the PROG* is finished execution, the <symbol>'s that were defined will no longer exist or retain their values.

EXAMPLES

```
     (prog* (i j)                        ; PROG* with vars I and J
           (print i) (print j))          ; prints  NIL NIL  returns NIL
     (prog* ((i 1) (j 2))                ; PROG* with vars I and J
           (print i) (print j)           ;
           (return (+ i j)))                   ; prints  1 2      returns 3
     (prog* () (print "hello"))          ; prints  "hello"  returns NIL

     (prog ((i 1) (j (+ i 1)))           ; PROG won't work due to order
           (print (+ i j)) )                   ; error: unbound variable -
I
     (prog* ((i 1) (j (+ i 1)))          ; PROG* will work due to order
            (print (+ i j)) )            ; prints  3        returns NIL
```

prog1

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(prog1  [ <expr1>  <expr2> ... ]  )
     <exprN>          -     expressions comprising the body of the loop
```

DESCRIPTION

The PROG1 special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains a block of code (expressions) to evaluate. <expr1>'s value will be returned as the result of PROG1. If there is no <expr1>, NIL is returned.

EXAMPLES

```
     (prog1 (print "hi") (print "ho")) ; prints "hi" "ho" returns "hi"
     (prog1)                           ; returns NIL
     (prog1 'a)                    ; returns A
     (prog1 "hey" (print "ho"))        ; prints "ho"      returns "hey"
```

NOTE:
PROG1, PROG2, PROGN and PROGV do not allow the use of RETURN or GO or tags for GO.

prog2

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(prog2 [<expr1> <expr2> ...])
<exprN> - expressions comprising the body of the loop

DESCRIPTION

The PROG2 special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains a block of code (expressions) to evaluate. <expr2>'s value will be returned as the result of PROG2. If there is no <expr2>, <expr1> is returned. If there is no <expr1>, NIL is returned.

EXAMPLES

```
(prog2 (print "hi") (print "ho")) ; prints "hi" "ho" returns "ho"
(prog2)                           ; returns NIL
(prog2 (print "hi"))              ; prints "hi"      returns "hi"
(prog2 (print "ho") "hey")        ; prints "ho"      returns "hey"
(prog2 'a 'b 'c)            ; returns B
```

NOTE:
PROG1, PROG2, PROGN and PROGV do not allow the use of RETURN or GO or tags for GO.

progn

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(progn [<expr1> <expr2> ...])
<exprN> - expressions comprising the body of the loop

DESCRIPTION

The PROGN special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains a block of code (expressions) to evaluate. The last <expr>'s value will be returned as the result of PROGN. If there are no <expr>s, NIL is returned.

EXAMPLES

```
(progn (print "hi") (print "ho")) ; prints "hi" "ho" returns "ho"
(progn)                           ; returns NIL
(progn "hey" (print "ho"))        ; prints "ho"      returns "ho"
(progn 'a)                   ; returns A
(progn 'a 'b 'c)             ; returns C
```

NOTE:
PROG1, PROG2, PROGN and PROGV do not allow the use of RETURN or GO or tags for GO.

progv

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(progv  <symbols> <values> [ <expr1>  <expr2> ... ]  )
     <symbols>   -     a list comprising symbols to be bound
     <values>    -     a list comprising values to be bound to symbols
     <exprN>           -     expressions comprising the body of the loop
```

DESCRIPTION

The PROGV special form is basically a 'block' construct (like a PASCAL BEGIN / END) that contains a block of code (expressions) to evaluate. PROGV is different from PROG1, PROG2 and PROGN in that it contains a pair of lists - <symbols> and <values>. Before evaluating the <exprN> expressions, PROGV will dynamically bind the <values> to the corresponding <symbols>. If there are too many <symbols> for the <values>, the <symbols> with no corresponding <values> will be bound to NIL. The variables will be unbound after the execution of PROGV. The last <expr>'s value will be returned as the result of PROGV. If there are no <expr>s, NIL is returned.

EXAMPLES

```
     (progv '(var) '(2)                ;
          (print var) (print "two"))   ; prints 2 "two" returns "two"

     (setq a "beginning")              ; initialize A
     (progv '(a) '(during) (print a))  ; prints DURING
     (print a)                    ; prints "beginning"

     (progv '(no-way) '(no-how) )      ; returns NIL
     (progv)                           ; error: too few arguments
```

NOTE:
PROGV is different from PROG (which allows symbols and initialization forms) in that PROGV allows its <symbols> and <values> to be evaluated. This allows you to pass in forms that generate the <symbols> and their <values>.

NOTE:
PROG1, PROG2, PROGN and PROGV do not allow the use of RETURN or GO or tags for GO.

psetq

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(psetq [ <symbol> <expr> ] ... )
     <symbol>    -     un-evaluated symbol
     <expr>      -     value for <symbol>
```

DESCRIPTION

PSETQ sets <expr> as the value of <symbol>. There can be several pairs of assignment. PSETQ performs these assignments in parallel - the <symbol>'s are not assigned new values until all the <expr>'s have been evaluated. PSETQ returns the value from the last <expr> as it's result.

EXAMPLES

```
     (psetq a 1)                  ; symbol A gets value 1
     (psetq b '(a b c))                ; symbol B gets value (A B C)
     (psetq mynum (+ 3 4))             ; symbol MYNUM gets value 7

     (setq goo 'ber)                   ; returns BER
     (setq num 1)                      ; returns 1
     (psetq goo num num goo)                ; returns BER
     (print goo)                  ; returns 1
     (print num)                  ; returns BER
```

putprop

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(putprop <symbol> <value> <property>)
     <symbol>   -   the symbol with a property list
     <value>   -   the value to be assigned to the property
     <property> -   the property name being changed/added

DESCRIPTION

PUTPROP sets the value of the <property> in the <symbol>. If the <property> does not exist, the <property> is added to the property list. The <symbol> must be an existing symbol. The <value> may be a single value or a list.

Property lists are lists attached to any user defined variables. The lists are in the form of (name1 val1 name2 val2 ....). Any number of properties may be attached to a single variable.

EXAMPLES

```
     (setq person 'bobby)              ; create a var with a value
     (putprop person 'boogie 'last-name)     ; add a LAST-NAME property
     (putprop person 'disc-jockey 'job); add a JOB property
     (get person 'last-name)                 ; retrieve LAST-NAME -
boogie
     (get person 'job)           ; retrieve JOB - disc-jockey
     (get person 'height)              ; non-existant - returns NIL
     (putprop person '(10 20 30) 'stats)     ; add STATS - a list
     (get person 'stats)               ;
```

NOTE:
You can set a property to the value NIL. However, this NIL value is indistinguishable from the NIL returned when a property does not exist.

COMMON LISP COMPATIBILITY:
Common LISP does not have a PUTPROP function. It uses a SETF to achieve this functionality. Porting from Common LISP to XLISP will work fine since XLISP supports the SETF modifications of property lists and GET. Porting from XLISP to Common LISP will require translating PUTPROP into SETF forms.

LISP DIALECTS:
The order of PUTPROP arguments is <symbol>, <value>, <property>. This is different from many other LISPs which normally use <symbol>, <property>, <value>. Be careful when porting existing LISP code.

quote

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(quote <expr>)
      <expr>   -   an expression

DESCRIPTION

QUOTE returns the the <expr> un-evaluated.

EXAMPLES

```
      my-var                              ; error: unbound variable
      (quote my-var)                      ; returns MY-VAR
      my-var                              ; still error: unbound variable
      (set (quote my-var) 111)            ; give MY-VAR a value -
                                    ;   make it exist
      my-var                              ; returns 111
      (quote my-var)                      ; returns MY-VAR

                                    ; SAME AS ABOVE BUT USING THE
                                    ; READ MACRO FOR QUOTE - '
      new-var                             ; error: unbound variable
      'new-var                      ; returns NEW-VAR
      new-var                             ; still error: unbound variable
      (setq new-var 222)                  ; give NEW-VAR a value -
                                    ;   make it exist
      new-var                             ; returns 222
      'new-var                      ; returns NEW-VAR
```

READ MACRO:
XLISP supports the normal read macro of a single quote as a short-hand method of writing the QUOTE function.

random

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(random <expr>)
     <expr>   -   integer number/expression

DESCRIPTION

The RANDOM function generates and returns a random number between 0 and <expr> - 1. If <expr> is negative, the number range is forced to be positive.

EXAMPLES

```
     (random 100)                    ; returns 7
     (random 100)                    ; returns 49
     (random 100)                    ; returns 73
     (random -100)                   ; returns 58
     (random 100.01)                 ; error: bad flt.pt. operation
```

COMMON LISP COMPATABILITY:
Common LISP allows an optional state parameter, which is not supported in XLISP. Also, Common LISP allows floating point numbers, which XLISP does not support.

NOTE:
This function is an extension of the XLISP system. It is provided in the MSSTUFF.C source code file. If your XLISP system is built for an IBM PC and compatibles, this function will work. If your system is built on UNIX or some other operating system, it will need the code in the corresponding STUFF.C file.

read

---

type: function (subr)
location: built-in
source file: xlfio.c and xlread.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(read   [ <source>  [ <eof-result> [ <recursive-flag> ] ] ]  )
     <source>    -     an optional source - must be a file pointer
                       or stream, the default is *standard-input*
     <eof-result>      -     an optional expression (default is NIL)
     <recursive-flag>-an optional expression ( NIL or non-NIL )
```

DESCRIPTION

The READ function reads an expression from the specified <source>. The expression read is a 'normal' XLISP expression - not a line. This means that white space is removed - blanks, empty lines and comment lines. Read-macro expansions will occur (like QUOTE instead of '). The expression needs to be an atom (numeric, string or symbol) or a valid list. It can span several lines. The expression read is returned as the result. The <source> may be a file pointer or a stream. If there is no <source>, *STANDARD-INPUT* is the default. If an end-of-file is encountered in the <source>, then the <eof-result> value will be returned as the result.

If you wish to read just lines or characters, refer to READ-LINE or READ-CHAR.

The <recursive-flag> is intended for use with embedded calls to READ. This is useful in read-macro and read-table uses. If <recursive-flag> is non-NIL, the READ does not expect itself to be at a 'top-level', but recursively executing within another READ that is in progress.

EXAMPLES

```
     (setq fp (open "f" :direction :output)) ; set up file
     (print "hello" fp)                ;
     (print 12.34 fp)            ;
     (princ "'(a b" fp)     (terpri fp);     fill with stuff
     (princ "; comment" fp) (terpri fp);
     (princ " c d)" fp )                ;
     (close fp)                  ;
                                 ;
     (setq fp (open "f" :direction :input))  ; now read the file
     (read fp "done")            ; returns "hello"
     (read fp "done")            ; returns 12.34
     (read fp "done")            ; returns (QUOTE (A B C D))
                                 ; note macro expansion of QUOTE
                                 ; note that comment is gone
```

```
(read fp "done")            ; returns "done"
(close fp)
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP READ functions are similar. They both allow for <source>, <eof-result> and <recursive-flag>. However, in Common LISP, there is an addition end-of-file error parameter. This parameter occurs right after <source> and specifies whether or not to flag an error on end-of-file. So, when porting, remember there is one additional argument in Common LISP's READ. You need to be concerned about this if you use more than just a <source> argument - going either from XLISP to Common LISP or vice versa.

COMMON LISP COMPATABILITY:
Common LISP specifies that read operations with a <source> of NIL, will come from *STANDARD-INPUT*. XLISP does not read the input from *STANDARD-INPUT* with a <source> of NIL. Common LISP also specifies that a <source> of T will read from *TERMINAL-IO*. XLISP does not allow T as a valid argument for <source>.

read-byte

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(read-byte [<source>])
<source> - an optional source - must be a file pointer or stream, the default is *standard-input*

DESCRIPTION

The READ-BYTE function reads a single character from the specified <source>. The character read is returned as an integer value for the result. The <source> may be a file pointer or a stream. If there is no <source>, *STANDARD-INPUT* is the default. If an end-of-file is encountered in the <source>, then NIL will be returned as the result.

EXAMPLES

```
(setq fp (open "f" :direction :output)) ; set up file
(print 12.34 fp)            ;
(close fp)                  ;
                            ;
(setq fp (open "f" :direction :input))  ; now read the file
(read-byte fp)                  ; returns 49          "1"
(read-byte fp)                  ; returns 50          "2"
(read-byte fp)                  ; returns 46          "."
(read-byte fp)                  ; returns 51          "3"
(read-byte fp)                  ; returns 52          "4"
(read-byte fp)                  ; returns 10          "\n"
(read-byte fp)                  ; returns NIL         empty
(close fp)                  ;
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP READ-BYTE functions are compatable for simple cases. They both allow for the optional <source>. However, in Common LISP, there are addition parameters which occur right after <source>. So, when porting from Common LISP to XLISP, remember there are additional arguments in Common LISP's READ-BYTE.

COMMON LISP COMPATABILITY:
Common LISP specifies that read operations with a <source> of NIL, will come from *STANDARD-INPUT*. XLISP does not read the input from *STANDARD-INPUT* with a <source> of NIL. Common LISP also specifies that a <source> of T will read from *TERMINAL-IO*. XLISP does not allow T as a valid argument for <source>.

read-char

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(read-char [<source>])
<source> - an optional source - must be a file pointer or stream, the default is *standard-input*

DESCRIPTION

The READ-CHAR function reads a single character from the specified <source>. The character read is returned as a single character value for the result. The <source> may be a file pointer or a stream. If there is no <source>, *STANDARD-INPUT* is the default. If an end-of-file is encountered in the <source>, then NIL will be returned as the result.

EXAMPLES

```
(setq fp (open "f" :direction :output)) ; set up file
(print 12.34 fp)            ;
(close fp)                  ;
                            ;
(setq fp (open "f" :direction :input))  ; now read the file
(read-char fp)                    ; returns #\1
(read-char fp)                    ; returns #\2
(read-char fp)                    ; returns #\.
(read-char fp)                    ; returns #\3
(read-char fp)                    ; returns #\4
(read-char fp)                    ; returns #\Newline
(read-char fp)                    ; returns NIL          empty
(close fp)                  ;
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP READ-CHAR functions are compatable for simple cases. They both allow for the optional <source>. However, in Common LISP, there are addition parameters which occur right after <source>. So, when porting from Common LISP to XLISP, remember there are additional arguments in Common LISP's READ-CHAR.

COMMON LISP COMPATABILITY:
Common LISP specifies that read operations with a <source> of NIL, will come from *STANDARD-INPUT*. XLISP does not read the input from *STANDARD-INPUT* with a <source> of NIL. Common LISP also specifies that a <source> of T will read from *TERMINAL-IO*. XLISP does not allow T as a valid argument for <source>.

read-line

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(read-line [<source>])
<source> - an optional source - must be a file pointer or stream, the default is *standard-input*

DESCRIPTION

The READ-LINE function reads a single line from the specified <source>. The line read is returned as a string value for the result. The <source> may be a file pointer or a stream. If there is no <source>, *STANDARD-INPUT* is the default. If an end-of-file is encountered in the <source>, then NIL will be returned as the result.

EXAMPLES

```
(setq fp (open "f" :direction :output)) ; set up file
(print "fe fi" fp)                ;
(print 12.34 fp)            ;
(close fp)                  ;
                            ;
(setq fp (open "f" :direction :input))  ; now read the file
(read-line fp)                  ; returns ""fe fi""
(read-line fp)                  ; returns "12.34"
(read-line fp)                  ; returns NIL
(close fp)                  ;
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP READ-LINE functions are compatable for simple cases. They both allow for the optional <source>. However, in Common LISP, there are addition parameters which occur right after <source>. So, when porting from Common LISP to XLISP, remember there are additional arguments in Common LISP's READ-LINE.

COMMON LISP COMPATABILITY:
Common LISP specifies that read operations with a <source> of NIL, will come from *STANDARD-INPUT*. XLISP does not read the input from *STANDARD-INPUT* with a <source> of NIL. Common LISP also specifies that a <source> of T will read from *TERMINAL-IO*. XLISP does not allow T as a valid argument for <source>.

readtable

---

type: system variable
location: built-in
source file: xlread.c
Common LISP compatible: related
supported on: all machines

SYNTAX

readtable

DESCRIPTION

The *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The table is 128 entries (0..127) for each of the 7-bit ASCII characters that XLISP can read. Each entry in the *READTABLE* array must be one of NIL, :CONSTITUENT, :WHITE-SPACE, :SESCAPE, :MESCAPE, a :TMACRO dotted pair or a :NMACRO dotted pair.

Table entry	Meaning
NIL	Invalid character
:CONSTITUENT	The character is valid, as is.
:WHITE-SPACE	The character may be skipped over.
:SESCAPE	The single escape character ('\');
:MESCAPE	The multiple escape character ('|');
(:TMACRO . <f>)	A terminating read-macro
(:NMACRO . <f>)	A non-terminating read-macro

In the case of :NMACRO and :TMACRO, the form of the *READTABLE* entry is a list like (:TMACRO . <function>) or (:NMACRO . <function>). The <function> can be a built-in read-macro function or a user defined lambda expression. The <function> takes two parameters, an input stream specification, and an integer that is the character value. The <function> should return NIL if the character is 'white-space' or a value CONSed with NIL to return the value.

EXAMPLES

```
*readtable*                  ; returns the current table

(defun look-at (table)             ; define a function to
 (dotimes (ch 127)                 ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
    (case entry                    ;   entries with a function
      (NIL     NIL)                ;
      (:CONSTITUENT NIL)           ;
      (:WHITE-SPACE NIL)           ;
      (:SESCAPE     NIL)           ;
      (:MESCAPE     NIL)           ;
```

```
            (T      (princ (int-char ch)));
            )))                     ;
       (terpri))                    ;
                                    ;
      (look-at *readtable*)               ;  prints  "#'(),;`
```

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

rem

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(rem <expr1> ...)
     <exprN>   -   integer number/expression

DESCRIPTION

The REM function takes the first pair of expressions and determines what is the remainder from dividing the first by the second expression. If there are no other arguments, this value is returned. If there are additional arguments, the remainder of the first pair is applied to the next and then the next and so on. In other words, (REM A B C D) is equivalent to (REM (REM (REM A B) C) D).

EXAMPLES

```
     (rem 1)                          ; returns 1
     (rem 1 2)                   ; returns 1
     (rem 13 8)                  ; returns 5
     (rem 13 8 3)                     ; returns 2
     (rem 13.5 8)                     ; error: bad flt.pt. operation
```

COMMON LISP COMPATABILITY:
Common LISP only allows two arguments. XLISP supports an arbitrary number of arguments. Also, Common LISP allows for floating point expressions where XLISP does not support this.

remove

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(remove <expr> <list-expr> [ { :test | :test-not } <test> ] )
     <expr>           -    the expression to remove - an atom or list
     <list-expr> -    the list to remove from
     <test>           -    optional test function (default is EQL)
```

DESCRIPTION

REMOVE searches through <list-expr> for <expr>. If <expr> is found, REMOVE returns the list with the <expr> deleted. All occurances of <expr> are deleted. If <expr> is not found, then the <list-expr> is returned unaltered. You may specify your own test with the :TEST and :TEST-NOT keywords followed by the test you which to perform. Note that this operation is non-destructive, it does not modify or affect <list-expr> directly - it creates a modified copy.

EXAMPLES

```
     (setq mylist '(a b c d it e f))         ; set up a list
     (remove 'it mylist)                       ; returns (A B C D E F)
     (print mylist)                            ; prints (A B C D IT E F)
                                  ;   note that MYLIST is not
                                  ;   affected
     (setq mylist '(a b c b d b))              ; change list to include
                                  ;   duplicates
     (remove 'b mylist)                        ; returns (A C D)

     (setq alist '( (a) (b) (it) (c))) ; set up another list
     (remove '(it) alist)                      ; returns ((A) (B) (IT) (C))
                                  ;   the EQ test doesn't work
                                  ;   for lists
     (remove '(it) alist :test 'equal) ; returns ((A) (B) (C))

     (setq slist '( "a" "b" "it" "c")) ; set up yet another list
     (remove "it" slist)                       ; returns ("a" "b" "c")
     (remove "it" slist :test-not 'equal)    ; returns ("it") - REMOVE
                                  ;   takes away everything but IT
```

NOTE:
The REMOVE function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

remove-if

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(remove-if <test> <list-expr> )
      <test>           -     the test function to be performed
      <list-expr> -     the list to remove from
```

DESCRIPTION

REMOVE-IF searches through <list-expr> and removes any elements that pass the <test>. Note that this operation is non-destructive, it does not modify or affect <list-expr> directly - it creates a modified copy.

EXAMPLES

```
      (setq mylist '(1 2 3 4 5 6 7 8))  ; set up a list
      (remove-if 'oddp mylist)          ; returns (2 4 6 8)
      (remove-if 'evenp mylist)         ; returns (1 3 5 7)
      (print mylist)                    ; prints (1 2 3 4 5 6 7 8)
                                  ;   note that MYLIST is not
                                  ;   affected

      (setq mylist '(a nil b nil c))          ; set up a list
      (remove-if 'null mylist)          ; returns (A B C)
```

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

remove-if-not

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(remove-if-not <test> <list-expr> )
      <test>           -     the test function to be performed
      <list-expr> -     the list to remove from
```

DESCRIPTION

REMOVE-IF-NOT searches through <list-expr> and removes any elements that fail the <test>. Note that this operation is non-destructive, it does not modify or affect <list-expr> directly - it creates a modified copy.

EXAMPLES

```
      (setq mylist '(1 2 3 4 5 6 7 8))  ; set up a list
      (remove-if-not 'oddp mylist)      ; returns (1 3 5 7)
      (remove-if-not 'evenp mylist)           ; returns (2 4 6 8)
      (print mylist)                    ; prints (1 2 3 4 5 6 7 8)
                                  ;   note that MYLIST is not
                                  ;   affected

      (setq mylist '(a nil b nil c))          ; set up a list
      (remove-if-not 'null mylist)      ; returns (NIL NIL)
```

COMMON LISP COMPATABILITY:
XLISP does not support the :FROM-END, :START, :END, :COUNT and :KEY keywords which Common LISP does.

remprop

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(remprop <symbol> <property>)
      <symbol>   -   the symbol with a property list
      <property> -   the property name being removed

DESCRIPTION

REMPROP removes the <property> from the <symbol>. The function returns a NIL. If the <property> does not exist, there is no error generated. The <symbol> must be an existing symbol. Property lists are lists attached to any user defined variables. The lists are in the form of (name1 val1 name2 val2 ....). Any number of properties may be attached to a single variable.

EXAMPLES

```
      (setq person 'bobby)              ; create a var with a value
      (putprop person 'boogie 'last-name)     ; add a LAST-NAME property
      (putprop person 'disc-jockey 'job); add a JOB property
      (get person 'last-name)                 ; retrieve LAST-NAME -
boogie
      (get person 'job)           ; retrieve JOB - disc-jockey
      (get person 'height)              ; non-existant - returns NIL
      (remprop person 'job)             ; remove JOB
      (remprop person 'height)          ; remove non-existant
```

COMMON LISP COMPATIBILITY:
Common LISP does not have a REMPROP function. It uses a SETF to achieve this functionality. Porting from Common LISP to XLISP will work fine since XLISP supports the SETF modifications of property lists and GET. Porting from XLISP to Common LISP will require translating REMPROP into SETF forms.

rest

---

type: function (subr)
location: built-in
source file: xlinit.lsp
Common LISP compatible: yes
supported on: all machines

SYNTAX

(rest <expr>)
     <expr> - a list or list expression

DESCRIPTION

REST returns the remainder of a list or list expression after first element of the list is removed. If the list is NIL, NIL is returned.

EXAMPLES

```
     (rest '(a b c))                    ; returns (B C)
     (rest '((a b) c d))                ; returns (C D)
     (rest NIL)                    ; returns NIL
     (rest 'a)                     ; error: bad argument type
     (rest '(a))                   ; returns NIL

     (setq sisters '(virginia vicki cindy))  ; set up variable SISTERS
     (first sisters)                    ; returns VIRGINIA
     (rest sisters)                     ; returns (VICKI CINDY)
```

NOTE:
The REST function is set to the same code as CDR.

&rest

---

type: keyword
location: built-in
source file: xleval.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
&rest [ <rest-arg> ]
      <rest-arg> -     rest argument symbol
```

DESCRIPTION

In XLISP, there are several times that you define a formal argument list for a body of code (like DEFUN, DEFMACRO, :ANSWER and LAMBDA). All of the formal arguments that are defined are required to appear in the invocation of the defined function or operation. If there are any &OPTIONAL arguments defined, they will be filled in order. If there is a &REST argument defined, and all the required formal arguments and &OPTIONAL arguments are filled, any and all further parameters will be passed into the function via the <rarg> argument. There can be only one <rest-arg> argument for &REST. If there are insufficient parameters for any of the &OPTIONAL or &REST arguments, they will contain NIL. At the end of the function or operation execution, these local symbols and their values are are removed.

EXAMPLES

```
      (defun foo                    ; define function FOO
        (a b &optional c d &rest e)           ;   with some of each
argument
        (print a) (print b)             ;
        (print c) (print d)             ;   print out each
        (print e))                      ;
      (foo)                         ; error: too few arguments
      (foo 1)                           ; error: too few arguments
      (foo 1 2)                     ; prints 1 2 NIL NIL NIL
      (foo 1 2 3)                   ; prints 1 2 3 NIL NIL
      (foo 1 2 3 4)                     ; prints 1 2 3 4 NIL
      (foo 1 2 3 4 5)                   ; prints 1 2 3 4 (5)
      (foo 1 2 3 4 5 6 7 8 9)                ; prints 1 2 3 4 (5 6 7 8 9)


      (defun my-add                     ; define function MY-ADD
        (num1 &rest num-list &aux sum)  ;   with 1 arg, rest, 1 aux var
        (setq sum num1)             ;   clear SUM
        (dotimes (i (length num-list) ) ;   loop through rest list
           (setq sum (+ sum (car num-list)))  ;      add the number to sum
           (setq num-list (cdr num-list)))    ;      and remove num from
list
        sum)                            ;   return sum when finished
```

```
(my-add 1 2 3 4)            ; returns 10
(my-add 5 5 5 5 5)                ; returns 25
```

restore

---

type: function (subr)
location: built-in
source file: xldmem.c xlimage.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(restore <file>)
     <file>   -   a string or symbol for the name of the file

DESCRIPTION

The RESTORE function restores the previously saved XLISP workspace (system state) from the specified file. The <file> may be a string or a symbol. If the <file> does not include a '.wks' suffix, it will be extended to be called <file>.wks. If successful, RESTORE will print a message saying

     [returning to the top level]

and will not return any value. If RESTORE fails, it will return NIL. There can be several saved workspaces. These workspaces can be restored as often as desired.

EXAMPLES

```
     (defun myfoo (fee fi)             ; create a function
          (+ fee fi))
     (setq myvar 5)                    ; set MYVAR to value 5
     myvar                        ; returns 5
     (save 'farp)                      ; save workspace in FARP.wks

     (setq myvar "garp")               ; change MYVAR to "garp"
     myvar                        ; returns "garp"

     (restore 'farp)                   ; restore workspace
     myvar                        ; returns 5
```

FILE NAMES:
In the PC and DOS world, all file names and extensions ("FOO.BAT") are automatically made uppercase. In using XLISP, this means you don't have to worry about whether the name is "foo.bat", "FOO.BAT" or even "FoO.bAt" - they will all work. However, in other file systems (UNIX in particular), uppercase and lowercase do make a difference. So, in UNIX if you do a (open 'foo-file :direction :output), this will create a file named "FOO-FILE" because XLISP uppercases its symbols. If you do a (open "foo-file" :direction :output), this will create a file named "foo-file" because UNIX doesn't uppercase its file names. Another case is if you do (save 'world), this will create the file "WORLD.wks". So, if you are having trouble with opening and accessing files, check to make sure the file name is in the proper case.

return

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(return [<expr>])
<expr> - an expression

DESCRIPTION

The RETURN special form allows the return of an arbitrary value at arbitrary times within 'block' constructs (DO, DO*, DOLIST, DOTIMES, LOOP, PROG and PROG*). The <expr> will be returned by the outer 'block' construct. A NIL will be returned by the outer 'block' construct if there is no <expr> specified. If RETURN is used without being within a valid 'block' construct, an error is generated: "error: no target for RETURN".

EXAMPLES

```
(prog (i)                       ; PROG form
  (print i) (RETURN "foo") (print j))   ; prints  NIL     returns
"foo"

(dotimes (i 10)                 ;
  (if (eql i 5) (RETURN 20)         ;
            (princ i)))         ; prints  01234    returns 20

(prog1 (print "hi") (RETURN "foo"))     ; prints  "hi"
                                ; error: no target for RETURN

(return 9)                      ; error: no target for RETURN
```

return-from

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(return-from  <name> [ <expr> ] )
      <name>           -     an unevaluated symbol for the block name
      <expr>           -     an expression
```

DESCRIPTION

The RETURN-FROM special form allows the return of an arbitrary value at arbitrary times within a 'named-block' construct (BLOCK) of the specified <name>. The <expr> will be returned by the BLOCK construct. A NIL will be returned by the BLOCK construct if there is no <expr> specified. If RETURN-FROM is used without being within a valid BLOCK construct, an error is generated: "error: no target for RETURN".

EXAMPLES

```
      (block out                    ; outer BLOCK
         (print "outer")                 ;
         (block in                       ; inner BLOCK
            (print "inner")              ;
            (return-from out "all done") ;
            (print "won't get here")     ;
         )                          ;
      )                             ; prints "outer"
                                    ; prints "inner"
                                    ; returns "all done"

      (return-from nobody 9)              ; error: no target for RETURN
```

reverse

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(reverse <list-expr>)
      <list-expr> -   a list or list expression

DESCRIPTION

The REVERSE function reverses the <list-expr>. The reversed list is the returned value. The reversal process only occurs on the 'top-level' of the <list-expr>. If there are nested sub-lists, these are left intact.

EXAMPLES

```
      (reverse NIL)                     ; returns NIL
      (reverse 'a)                      ; error: bad argument type
      (reverse '(a))                    ; returns (A)
      (reverse '(a b c))                ; returns (C B A)
      (reverse '((a b) (c d) (e f)))          ; returns ((E F) (C D) (A
B))
      (reverse (list (+ 1 2) (+ 3 4)))  ; returns (7 3)
```

room

---

type: function (subr)
location: built-in
source file: xldmem.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(room [<info>])
     <info>   -   an optional, unused expression

DESCRIPTION

The ROOM function prints the current memory statistics to *STANDARD-OUTPUT*. NIL is always returned. The message shows the statistics for total NODES, current FREE NODES, current number of allocated memory SEGMENTS, node size of the ALLOCATEd memory segments, TOTAL memory in bytes and total number of garbage COLLECTIONS that have occured since this session of XLISP started.

EXAMPLES

```
     (room)                          ; prints  Nodes:       4000
                                     ;       Free nodes:  1723
                                     ;       Segments:    4
                                     ;         Allocate:    1000
                                     ;       Total:       52566
                                     ;       Collections: 8
                                     ; returns NIL
```

COMMON LISP COMPATABILITY:
In Common LISP, the <info> argument controls the amount of information that is printed.

COMMON LISP COMPATABILITY:
The form of and information provided by the ROOM output is implementation dependent. For portability, you should not count on this information or form.

rplaca

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(rplaca <list> <expr>)

<list>	-	the list to DESTRUCTIVELY modify
<expr>	-	the expression to replace CAR of <list>

DESCRIPTION

RPLACA destructively modifies the CAR of <list> and replaces it with the <expr>. The destructive aspect of this operation means that the actual symbol value is used in the list-modifying operations - not a copy. <list> must evaluate to a valid list. An atom or NIL for <list> will result in an error.

EXAMPLES

```
(setq a '(1 2 3))           ; make A with value (1 2 3)
(setq b '(1 2 3))           ; make B with value (1 2 3)
(setq c a)                  ; make C point to A's value
(rplaca a 'new)                  ; returns (NEW 2 3)
(print a)                   ; prints (NEW 2 3)
                            ; NOTE THAT A IS MODIFIED!
(print b)                   ; prints (1 2 3)
                            ; note that B is not modified
(print c)                   ; prints (NEW 2 3)
                            ; NOTE THAT C IS MODIFIED TOO!

(setq a '(1 2 3))           ; reset A to value (1 2 3)
(rplaca a '(the sub list))       ; returns ((THE SUB LIST) 2 3)
(rplaca '(1 2 3) 'more)               ; returns (MORE 2 3)

(rplaca 'a 'b)                   ; error: bad argument type
(rplaca NIL 'b)                  ; error: bad argument type
```

rplacd

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(rplacd <list> <expr>)
     <list>   -   the list to DESTRUCTIVELY modify
     <expr>   -   the expression to replace the CDR of <list>

DESCRIPTION

RPLACD destructively modifies the CDR of <list> and replaces it with the <expr>. The destructive aspect of this operation means that the actual symbol value is used in the list-modifying operations - not a copy. <list> must evaluate to a valid list. An atom or NIL for <list> will result in an error.

EXAMPLES

```
     (setq a '(1 2 3))           ; set up A with (1 2 3)
     (rplacd a 'new)                  ; returns (1 . NEW)
     (print a)                   ; prints (1 . NEW)
                                 ; NOTE THAT A IS MODIFIED!
                                 ;
     (rplacd a '(a new list))         ; returns (1 A NEW LIST)
     (rplacd '(1 2 3) '(more))        ; returns (1 MORE)
     (rplacd 'a 'b)                   ; error: bad argument type
     (rplacd NIL 'b)                  ; error: bad argument type
```

save

---

type: function (subr)
location: built-in
source file: xldmem.c xlimage.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(save <file>)
     <file>   -   a string or symbol for the name of the file

DESCRIPTION

The SAVE function saves the current XLISP workspace (system state) to the specified file. The <file> may be a string or a symbol. If the <file> does not include a '.wks' suffix, it will be extended to be called <file>.wks. The function returns T if the workspace was properly created and saved, NIL is returned otherwise. There can be several saved workspaces. These workspaces can be restored as often as desired.

EXAMPLES

```
     (defun myfoo (fee fi)             ; create a function
          (+ fee fi))
     (setq myvar 5)                    ; set MYVAR to value 5
     myvar                        ; returns 5
     (save 'farp)                      ; save workspace in FARP.wks

     (setq myvar "garp")               ; change MYVAR to "garp"
     myvar                        ; returns "garp"

     (restore 'farp)                   ; restore workspace
     myvar                        ; returns 5
```

BUG:
The SAVE function generates a system error if the <file> being created already exists. This <file> will be modified and will not be restorable after restarting XLISP.

NOTE:
The saved workspace size is implementation dependent, but can be fairly large.

FILE NAMES:
In the PC and DOS world, all file names and extensions ("FOO.BAT") are automatically made uppercase. In using XLISP, this means you don't have to worry about whether the name is "foo.bat", "FOO.BAT" or even "FoO.bAt" - they will all work. However, in other file systems (UNIX in particular), uppercase and lowercase do make a difference. So, in UNIX if you do a (open 'foo-file :direction :output), this will create a file named "FOO-FILE" because XLISP uppercases its symbols. If you do a (open "foo-file" :direction :output), this will create a file named

"foo-file" because UNIX doesn't uppercase its file names. Another case is if you do (save 'world), this will create the file "WORLD.wks". So, if you are having trouble with opening and accessing files, check to make sure the file name is in the proper case.

COMMON LISP COMPATABILITY:
The SAVE function is similar in use to the SAVE-WORLD function in Common LISP. The primarily difference is that SAVE-WORLD allows you to restart everything since it creates an executable file. The SAVE function requires you to start XLISP up first and then do a RESTORE. Depending on the operating system that you are using, it is possible to write a SAVE-WORLD equivalent using SAVE, RESTORE and SYSTEM functions.

savefun

---

type: defined macro (closure)
location: extension
source file: init.lsp
Common LISP compatible: no
supported on: all machines

SYNTAX

(savefun <function>)
     <function> -   the name of the function or macro to be saved

DESCRIPTION

The SAVEFUN macro saves the specified function or macro to a file. The file will be called <function>.lsp. The macro returns the file name that was created. An error will occur if the <function> parameter is not a function or macro.

EXAMPLES

```
     (defun myfoo (fee fi)             ; create a function
          (+ fee fi))
     (savefun myfoo)                   ; saves MYFOO to "MYFOO.lsp"
     (savefun savefun)           ; saves SAVEFUN to "SAVEFUN.lsp"
     (savefun 'a)                      ; error: bad argument type
```

NOTE:
The SAVEFUN macro is defined in the INIT.LSP file. If SAVEFUN does not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP. Another thing to try is to put a PRINT message in the INIT.LSP file and make sure that it is printed out when XLISP starts running.

second

type: function (subr)
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(second <expr> )
     <expr>           -     a list or list expression
```

DESCRIPTION

SECOND returns the second element of a list or list expression. If the list is NIL, NIL is returned.

EXAMPLES

```
     (second '(1 2 3))           ; returns 2
     (second NIL)                      ; returns NIL

     (setq carol '(a b c))             ; set up variable CAROL
     (first carol)                     ; returns A
     (second carol)                    ; returns B
     (rest carol)                      ; returns (B C)

     (setq children '(amanda ben))          ; set up variable CHILDREN
     (second children)           ; returns BEN
```

NOTE:
This function is set to the same code as CADR.

self

---

type: symbol
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

self

DESCRIPTION

SELF evaluates to the current object when used within a message context.

EXAMPLES

```
      (setq my-class                    ; create MY-CLASS with STATE
            (send class :new '(state)))  ;
      (send my-class :answer :isnew '()  ; set up initialization
            '((setq state nil) SELF))    ;     returning SELF
      (send my-class :answer :set-it '(value) ; create :SET-IT message
            '((setq state value)))       ;
      (setq my-obj (send my-class :new)); create MY-OBJ of MY-CLASS
      (send my-obj :set-it 5)                 ; STATE is set to 5
```

CONTEXT:
SELF does not exist except within the context of a method and it's execution.

NOTE:
In the previous example, there is a SELF in the line that creates the :SET-IT message. What this does is to return the object as the last operation when you do an :ISNEW.

send

---

type: function (subr)
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(send <object> <message> [<args>])
      <object>   -   an object
      <message>   -   message selector for object
      <arg>   -   parameter sent to object method

DESCRIPTION

The SEND function is the mechanism used to send a <message> to an <object>. The <message> is the message selector symbol that is used to select a particular action (method) from the object.

EXAMPLES

```
      (setq myclass (send class :new '(var))) ; create MYCLASS with VAR
      (send myclass :answer :isnew '()  ; set up initialization
            '((setq var nil) self))
      (send myclass :answer :set-it '(value)  ; create :SET-IT message
            '((setq var value)))
      (setq my-obj (send myclass :new)) ; create MY-OBJ of MYCLASS
      (send my-obj :set-it 5)                 ; VAR is set to 5
```

BUILT-IN METHODS:
The built in methods in XLISP include:

<message>	operation
:ANSWER	Add a method to an object.
:CLASS	Return the object's class.
:ISNEW	Run initialization code on object.
:NEW	Create a new object (instance or class).
:SHOW	Show the internal state of the object.

MESSAGE STRUCTURE:
The normal XLISP convention for a <message> is to have a valid symbol preceeded by a colon like :ISNEW or :MY-MESSAGE. However, it is possible to define a <message> that is a symbol without a colon, but this makes the code less readable.

send-super

---

type: function (subr)
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(send-super <message> [<args>])
      <message> -   the message selector
      <args>   -   the optional message arguments

DESCRIPTION

The SEND-SUPER function sends the specified arguments <args> to the <message> specified method of the superclass. It is necessary for SEND-SUPER to be executed from within a method being performed on an object. It will return the result of sending the message. If SEND-SUPER is performed outside of a method an error "error: not in a method" will result.

EXAMPLES

```
      (setq a-class (send class :new '()))    ; create A-CLASS
      (send a-class :answer :show '()         ; set up special SHOW method
            '((print "nobody here") self))    ;
      (setq an-obj (send a-class :new)) ; create AN-OBJ of A-CLASS
      (send an-obj :show)               ; prints "nobody here"
      (send a-class :answer :myshow '() ; set up MYSHOW method which
            '((send-super :show )))           ;     calls :SHOW in
superclass
      (send an-obj :myshow)             ; prints Object is ............
```

:sescape

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

:sescape

DESCRIPTION

:SESCAPE is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :SESCAPE keyword means that the specified character is to be used as a single escape character. The system defines that the the vertical bar character \ is the only defined :SESCAPE character.

EXAMPLES

```
(defun look-at (table)            ; define a function to
 (dotimes (ch 127)                ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
    (case entry             ;   entries with a function
      (:SESCAPE             ;
          (princ (int-char ch)))  ;
      (T       NIL))))      ;
 (terpri))                  ;
(look-at *readtable*)             ;  prints  \
```

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

set

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(set <symbol> <expr>)
      <symbol>   -   expression that evaluates to a symbol name
                        (if expression is quoted, no evaluation occurs)
      <expr>   -   an expression - which will be the new value

DESCRIPTION

SET evaluates <symbol> and sets <expr> as it's value. If the <symbol> value is quoted (via the QUOTE special form or read-macro expansion), the <symbol> is not evaluated. SET returns the value from <expr> as it's result.

EXAMPLES

```
      (set 'a 2)                  ; sets symbol A to value 2
      (set 'value a)                    ; sets symbol VALUE to value 2
      (print value)                     ; show the value - prints 2
      (set 'name 'myvar)                ; set symbol NAME to value MYVAR
      (set name 12345)            ; set symbol which is the value
                                  ;     of NAME (MYVAR) to 12345
      (print name)                      ; prints MYVAR
      (print myvar)                     ; prints 12345

      (set notsymbol 1)           ; error: unbound variable
      (set name notvalue)               ; error: unbound variable
```

setf

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(setf [ <place1> <expr1> ... ] )
     <placeN>    -     a field specifier which may be one of:
                       <symbol>          (car <expr> )
                       (cdr <expr> )           (nth <n> <expr> )
                       (aref <expr> <n> )      (get <symb> <property> )
                       (symbol-value <symb> ) (symbol-plist <symb> )
     <exprN>           -     an expression - which will be the new value
```

DESCRIPTION

SETF evaluates the field <placeN> and sets <exprN> as it's value. This is a generalized tool that allows you to set the value of the various data types of the system. SETF returns the value from <exprN> as it's result. The specific action of SETF depends on the <placeN> field.

EXAMPLES

```
                                ; SETF SYMBOL
     (setf a 123)                     ; set a symbol A to value 123

                                ; SETF SYMBOL-VALUE
     (setq x 'y)                ; make symbol X with value Y
     (setf (symbol-value x) 'z)       ; set symbol that X contains (Y)
                                ;   to value Z

                                ; SETF LIST ELEMENTS
     (setq mylist '(a b c d))         ; MYLIST with value (A B C D)
     (setf (car mylist) 'x)           ; change CAR of MYLIST to X
                                ;   MYLIST now is (X B C D)
     (setf (cdr mylist) '(y z da-end)) ; change CDR of MYLIST to
                                ;   (Y Z DA-END) so that
                                ;   MYLIST now is (X Y Z DA-END)
     (setf (nth 3 mylist) 'here-i-am)  ; change 3rd of MYLIST to
                                ;   HERE-I-AM so that MYLIST
                                ;   now is (X Y Z HERE-I-AM)

                                ; SETF AREF
     (setq myarray (make-array 5))          ; make MYARRAY
     (aref myarray 2)           ; get value of element 2 = NIL
     (setf (aref myarray 2) 'new-value); set value of element 2 to
                                ;   value NEW-VALUE
     (print myarray)                  ; prints
                                ;   #(NIL NIL NEW-VALUE NIL NIL)
```

```
                                ; SETF PROPERTIES
      (setq person 'joe-bob)            ; make PERSON with value JOE-BOB
      (putprop person 'critic 'profession)    ; set PROFESSION property to
                                ;   value CRITIC
      (setf (get person 'profession)          ; change PROFESSION to value
            'texas-critic)              ;   TEXAS-CRITIC
      (setf (get person 'home) 'texas)  ; add property HOME with
                                ;   value TEXAS
      (symbol-plist person)             ; returns property list:
                                ;   (HOME TEXAS
                                ;    PROFESSION TEXAS-CRITIC)
      (setf (symbol-plist person)       ; change the property list
           '(home on-the-range          ;
             profession movie-critic))  ;
      (get person 'profession)          ; now returns MOVIE-CRITIC
      (get person 'home)                ; now returns ON-THE-RANGE
```

OPERATIONS:

<placeN>	SETF action
<symbol>	Sets the value of <symbol> to the value of <exprN>. This is equivalent to a (SETQ <symbol> <exprN>).
(car <expr>)	Sets the first element of the <expr> list to <exprN>. <expr> must be a list. This is equivalent to a (RPLACA <expr> <exprN>) except that SETF will return <exprN> as the value. (cdr <expr>) Sets the tail of the <expr> list to <exprN>. <expr> must be a list. This is equivalent to a (RPLACD <expr> <exprN>) except that SETF will return <exprN> as the value.
(nth <n> <expr>)	Sets the <n>th element of the <expr> list to <exprN>. <expr> must be a list. This allows you to set an arbitrary element of a list to an arbitrary value. Note that the list is numbered from the 0th element (0, 1, 2, 3, ...).
(aref <expr> <n>)	Sets the <n>th element of the <expr> array to <exprN>. <expr> must be an array. This allows you to set an arbitrary element of an array to an arbitrary value. Note that the list is numbered from the 0th element (0, 1, 2, 3, ...). Note also that this is the intended way to set the value of an array element.
(get <sym> <prop>)	Sets the <prop> of <sym> to the value <exprN>. If <sym> does not have the

<prop>, one will be created. This is equivalent to (PUTPROP <sym> <exprN> <prop>).

(symbol-value <symbol>) Sets the symbol's value to contain <exprN>. <symbol> is an expression that must evaluate to a valid symbol - it doesn't have to exist before the SETF, it just has to be a valid symbol. This is equivalent to (SET <symbol> <exprN>).

(symbol-plist <symbol>) Sets the property list of <symbol> to <exprN>. This allows you to change (or destroy) the entire property list of a <symbol> at one time.

set-macro-character

---

type: defined function (closure)
location: extension
source file: init.lsp
Common LISP compatible: related
supported on: all machines

SYNTAX

```
(set-macro-character <char-num> <function>  [ <termflag> ] )
      <char-num>  -     an integer expression
      <function>  -     a function definition
      <termflag>  -     an expression - NIL or non-NIL
```

DESCRIPTION

The SET-MACRO-CHARACTER function installs the code that will be executed when the specified character <char-num> is encountered by the XLISP reader. The <function> is placed in the *READTABLE* system variable which contains the reader table array. The table is 128 entries (0..127) for each of the 7-bit ASCII characters that XLISP can read. Each entry in the table must be one of NIL, :CONSTITUENT, :SESCAPE, :MESCAPE, :WHITE-SPACE, a :TMACRO dotted pair or a :NMACRO dotted pair. The SET-MACRO-CHARACTER function only allows you to put in a terminating read-macro function (:TMACRO) or a non-terminating read-macro-function (:NMACRO). If the <termflag> is present and non-NIL, then the <function> will be put in *READTABLE* as a :TMACRO entry. If <termflag> is not present or NIL, then <function> will be put in *READTABLE* as a :NMACRO entry. The <function> can be a built-in read-macro function or a user defined defun symbol or a lambda expression. The <function> takes two parameters, an input stream specification, and an integer that is the character value. The <function> should return NIL if the character is 'white-space' or a value CONSed with NIL to return the value. The function SET-MACRO-CHARACTER always returns T.

EXAMPLES

```
      (print "hi") % comment            ; prints  "hi"  and gives
                                  ; error: unbound variable - %
                                  ; because percent is viewed
                                  ; as a variable
                                  ;
      (setq semi (get-macro-character #\;))   ; get semi-colon code
                                  ;
      (SET-MACRO-CHARACTER #\% semi T)  ; set % to work as a comment
                                  ;
      (print "hi") % comment            ; prints  "hi" and no error
                                  ; because % is now a comment
                                  ; character in *READTABLE*
```

NOTE:
In the normal XLISP system the following characters have code associated

with them in the *READTABLE*:

" # ' ( ) , ; `

NOTE:
The functions GET-MACRO-CHARACTER and SET-MACRO-CHARACTER are created in the INIT.LSP file. If they do not exist in your XLISP system, you might be having a problem with INIT.LSP. Before you start XLISP, look in the directory you are currently in, and check to see if there is an INIT.LSP.

COMMON LISP COMPATABILITY:
The SET-MACRO-CHARACTER function is somewhat related to the Common LISP SET-DISPATCH-MACRO-CHARACTER function.

setq

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(setq [ <symbol1> <expr1> ] ... )
      <symbolN>   -     un-evaluated symbol
      <exprN>           -     value for <symbolN>
```

DESCRIPTION

SETQ sets <expr> as the value of <symbol>. SETQ returns the value from <expr> as it's result.

EXAMPLES

```
      (setq a 1)                    ; symbol A gets value 1
      (setq b '(a b c))             ; symbol B gets value (A B C)
      (setq mynum (+ 3 4))                ; symbol MYNUM gets value 7
```

:show

---

type: message selector
location: built-in
source file: xlobj.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(send <object> :show)
      <object>   -   an existing object

DESCRIPTION

The :SHOW message selector attempts to find the 'show' method in the specified <object>'s class. Since the :SHOW message selector is built-in in the root class (CLASS), this is always a valid message selector. The object must already exist.

EXAMPLES

```
      (setq my-class                    ; create MY-CLASS with STATE
            (send class :new '(state)))  ;
      (send my-class :answer :isnew '() ; set up initialization
            '((setq state nil) self))
      (send my-class :answer :set-it '(value) ; create :SET-IT message
            '((setq state value)))
      (setq my-obj (send my-class :new)); create MY-OBJ of MY-CLASS
      (send my-obj :show)               ; returns object state including
                                  ;   STATE = NIL
      (send my-obj :set-it 5)                ; STATE is set to 5
      (send new-obj :show)              ; error: unbound variable
```

sin

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(sin <expr>)
     <expr>   -   floating point number/expression

DESCRIPTION

The SIN function returns the sine of the <expr>. The <expr> is in radians.

EXAMPLES

```
     (sin 0.0)                  ; returns 0
     (sin .5)                   ; returns 0.479426
     (sin 1.0)                  ; returns 0.841471
     (sin (/ 3.14159 2))             ; returns 1
     (sin 3.14159)                   ; returns 2.65359e-06
     (sin 0)                         ; error: bad integer operation
     (sin 1.)                   ; error: bad integer operation
```

COMMON LISP COMPATABILITY:
Common LISP allows for integer numbers, which XLISP does not support for SIN.

sort

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(sort <list> <test>)
     <list>   -   a list containing elements to be sorted
     <test>   -   the test to use for the sort

DESCRIPTION

The SORT function sorts the <list> using the <test> to order the list. The SORT function is destructive and modifies the <list>.

EXAMPLES

```
     (setq a '(3 1 4 1 5 9 6 7))       ; returns (3 1 4 1 5 9 6 7)
     (sort a '<)                  ; returns (1 1 3 4 5 6 7 9)
     (print a)                    ; returns (1 1 3 4 5 6 7 9)
                                  ; notice that A is modified
     (sort a '> )                      ; returns (9 7 6 5 4 3 1 1)

     (sort '("a" "bar" "foo") 'string> )     ; returns ("foo" "bar" "a")
```

BUG:
XLISP returns the proper value, but improperly modifies the symbol or actual <list>.

COMMON LISP COMPATABILITY:
Common LISP allows for a :KEY keyword (which allows a specified function to be run before the ordering takes place), which XLISP does not support.

sqrt

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(sqrt <expr>)
     <expr>   -   floating point number/expression

DESCRIPTION

The SQRT function calculates the square root of <expr> and returns this result.

EXAMPLES

```
     (sqrt 1.0)                ; returns 1
     (sqrt 2.0)                ; returns 1.41421
     (sqrt 3.0)                ; returns 1.73205
     (sqrt 4.0)                ; returns 2
     (sqrt 5.0)                ; returns 2.23607
     (sqrt -1.0)               ; error: sqrt of a neg. number
     (sqrt 2)                  ; error: bad integer operation
```

COMMON LISP COMPATABILITY:
Common LISP allows for integer numbers, which XLISP does not support for SQRT.

standard-input

---

type: system variable
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

standard-input

DESCRIPTION

STANDARD-INPUT is a system variable that contains a file pointer that points to the file where all normal input from the programmer or user comes from. The default file for *STANDARD-INPUT* is the system standard input device - normally the system keyboard.

EXAMPLES

```
      *standard-input*            ; returns #<File-Stream: #2442e>
```

NOTE:
Be careful when modifying the *STANDARD-INPUT*. If you do not save the old file pointer, you will not be able to return to normal operation and will need to exit XLISP. If the file or source that you have set *STANDARD-INPUT* to does not reset *STANDARD-INPUT* to its previous value, you will never get control back to the keyboard.

standard-output

---

type: system variable
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

standard-output

DESCRIPTION

STANDARD-OUTPUT is a system variable that contains a file pointer that points to the file where all normal printing and messages from XLISP will go. The default file for *STANDARD-OUTPUT* is the system standard output device - normally the screen display/crt.

EXAMPLES

```
*standard-output*            ; returns #<File-Stream: #24406>
(setq old-so *standard-output*)         ; save the file pointer
(setq fp (open "f" :direction :output)) ; open a new output file
(setq *standard-output* fp)        ; change where output goes
                                ;
(+ 2 2)                              ; you won't see any messages
                                ; just the echo of input line
                                ;
(setq *standard-output* old-so)         ; restore standard output
(close fp)                   ; close file
```

NOTE:
Be careful when modifying the *STANDARD-OUTPUT*, you will not be able to see what you are doing. If you do not save the old file pointer, you will not be able to return to normal operation and will need to exit XLISP.

strcat

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(strcat [<string1> ...])
     <stringN> -   a string expression

DESCRIPTION

The STRCAT function returns the concatenation of a sequence of string expressions. If there are no strings, an empty string is returned.

EXAMPLES

```
(strcat)                    ; returns ""
(strcat "a")                     ; returns "a"
(strcat "a" "b")            ; returns "ab"
(strcat "ab" "cd" "ef")                ; returns "abcdef"
(strcat "f" "ire tr" "uck")       ; returns "fire truck"
(strcat 1 2)                     ; error: bad argument type
```

streamp

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(streamp <expr>)
     <expr> - the expression to check

DESCRIPTION

The STREAMP predicate checks if an <expr> is a stream. T is returned if <expr> is a stream, NIL is returned otherwise.

EXAMPLES

```
(streamp *standard-input*)        ; returns T - stream
(streamp *debug-io*)              ; returns T - stream
(streamp (make-string-output-stream))   ; returns T - stream
(setq a *standard-output*)        ;
(streamp a)                 ; returns T - evaluates to stream

(streamp "a")                     ; returns NIL - string
(streamp #\a)                     ; returns NIL - character
(streamp '(a b c))                ; returns NIL - list
(streamp 1)                 ; returns NIL - integer
(streamp 1.2)                     ; returns NIL - float
(streamp '*debug-io*)             ; returns NIL - symbol
(streamp 'a)                      ; returns NIL - symbol
(streamp #(0 1 2))                ; returns NIL - array
(streamp NIL)                     ; returns NIL - NIL
```

string

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(string <expr>)
     <expr> - a string, symbol or character expression

DESCRIPTION

The STRING function forces the <expr> to be a string. If the <expr> is a string, it is returned, as is. If the <expr> is a character, a one-character string is returned. If the <expr> is a symbol, the symbol is turned into a string.

EXAMPLES

```
(string 'foo)                    ; returns "FOO"
(string 'x)                  ; returns "X"
(string "abcdef")            ; returns "abcdef"
(string #\a)                     ; returns "a"
(string #\A)                     ; returns "A"
(string #\Newline)               ; returns "\n"
```

string/=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string/= <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING/= (string-NOT-EQUAL) function takes two string arguments. A non-NIL value is returned if <string1> is not equal to <string2>, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR/= the corresponding character of <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string/= "a" "b")                ; returns 0
     (string/= "a" "a")                ; returns NIL
     (string/= "a" "A")                ; returns 0
     (string/= "A" "a")                ; returns 0
     (string/= "abc" "abc ")                ; returns 3

     (string/= "J Smith" "K Smith"          ; strip off the first chars
               :start1 1 :start2 1)         ; returns NIL
     (string/= "abc" "123456789"       ; leave just the first 3 chars
               :end2 3 :end1 3)        ; returns 0
```

NOTE:
Be sure that the STRING/= function is properly typed in. The '/' is a forward slash. It is possible to mistakenly type a '\' (backslash). This is especially easy because the character mechanism is '#\a'. If you do use the backslash, no error will be reported because backslash is the single escape character and the LISP reader will evaluate 'STRING\='

as 'STRING='. No error will be reported, but the sense of the test is reversed.

string<

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string< <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING< (string-LESS-THAN) function takes two string arguments. A non-NIL value is returned if <string1> is less than <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR< the corresponding character of <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string< "a" "b")           ; returns 0
     (string< "a" "a")           ; returns NIL
     (string< "a" "A")           ; returns NIL
     (string< "A" "a")           ; returns 0
     (string< "abc" "abc ")            ; returns 3
     (string< "1234567" "1234qrst")          ; returns 4

     (string< "J Smith" "K Smith"            ; strip off the first chars
            :start1 1 :start2 1)      ; returns NIL
```

string<=

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string<= <string1> <string2> [ <key> <offset> ] ... )
      <string1>   -     a string expression
      <string2>   -     a string expression
      <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
      <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING<= (string-LESS-THAN-OR-EQUAL) function takes two string arguments. A non-NIL value is returned if <string1> is less than or equal to <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR<= the corresponding character of <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
      (string<= "a" "b")                ; returns 0
      (string<= "a" "a")                ; returns 1
      (string<= "a" "A")                ; returns NIL
      (string<= "A" "a")                ; returns 0
      (string<= "abc" "abc ")                ; returns 3
      (string<= "1234567" "1234qrst")        ; returns 4

      (string<= "J Smith" "K Smith"          ; strip off the first chars
             :start1 1 :start2 1)       ; returns 7
```

string=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(string= <string1> <string2> [<key> <offset>] ...)
     <string1> -   a string expression
     <string2> -   a string expression
     <key>   -   a keyword (one of :START1 :START2 :END1 :END2)
     <offset>   -   an optional integer expression (for a keyword)

DESCRIPTION

The STRING= (string-EQUALITY) function takes two string arguments. It checks to see if the string arguments have the same values. T is returned if <string1> is equal to <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string= "a" "b")           ; returns NIL
     (string= "a" "a")           ; returns T
     (string= "a" "A")           ; returns NIL
     (string= "A" "a")           ; returns NIL
     (string= "abc" "abc ")           ; returns NIL

     (string= "J Smith" "K Smith"           ; strip off the first chars
              :start1 1 :start2 1)          ; returns T
     (string= "abc" "123456789"        ; leave just the first 3 chars
              :end2 3 :end1 3)         ; returns NIL
```

string>

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string> <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING> (string-GREATER-THAN) function takes two string arguments. A non-NIL value is returned if <string1> is greater than <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR> the corresponding character of <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string> "a" "b")           ; returns NIL
     (string> "a" "a")           ; returns NIL
     (string> "a" "A")           ; returns 0
     (string> "A" "a")           ; returns NIL
     (string> "abc" "abc ")            ; returns NIL
     (string> "1234qrst" "12345678")         ; returns 4

     (string> "J Smith" "K Jones"            ; strip off the first chars
            :start1 1 :start2 1)      ; returns 2
```

string>=

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string>= <string1> <string2> [ <key> <offset> ] ... )
      <string1>   -     a string expression
      <string2>   -     a string expression
      <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
      <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING>= (string-GREATER-THAN-OR-EQUAL) function takes two string arguments. A non-NIL value is returned if <string1> is greater than or equal to <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR>= the corresponding character of <string2>. This test is case sensitive - the character #\a is different (and of greater ASCII value) than #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
      (string>= "a" "b")                ; returns NIL
      (string>= "a" "a")                ; returns 1
      (string>= "a" "A")                ; returns 0
      (string>= "A" "a")                ; returns NIL
      (string>= "abc" "abc ")                ; returns NIL
      (string>= "1234qrst" "12345678")  ; returns 4

      (string>= "J Smith" "K Jones"           ; strip off the first chars
             :start1 1 :start2 1)       ; returns 2
```

stringp

---

type: predicate function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(stringp <expr>)
     <expr>   -   the expression to check

DESCRIPTION

The STRINGP predicate checks if an <expr> is a string. T is returned if <expr> is a string, NIL is returned otherwise.

EXAMPLES

```
     (stringp "a")                    ; returns T - string
     (setq a "hi there"               ;
     (stringp a)                 ; returns T - evaluates to string

     (stringp #\a)                    ; returns NIL - character
     (stringp '(a b c))               ; returns NIL - list
     (stringp 1)                 ; returns NIL - integer
     (stringp 1.2)                    ; returns NIL - float
     (stringp 'a)                     ; returns NIL - symbol
     (stringp #(0 1 2))               ; returns NIL - array
     (stringp NIL)                    ; returns NIL - NIL
```

string-downcase

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-downcase <string> [ { :start | :end } <offset> ] ... )
     <string>    -     a string expression
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-DOWNCASE function takes a string argument and returns a new string that has been made lower case.

The keyword arguments allow for accessing substrings within <string>. The keyword arguments require a keyword (:START or :END) first and a single integer expression second. The :START keyword specifies the starting offset for the STRING-DOWNCASE operation on <string>. A value of 0 starts the string at the beginning (no offset). The :END keyword specifies the end offset for the operation on <string>.

EXAMPLES

```
     (string-downcase "ABcd+-12&[")          ; returns "abcd+-&["
     (string-downcase "ABCDEFGH"       ;
                 :start 2 :end 4); returns "ABcdEFGH"

     (setq mystr "ABcdEFgh")                 ; set up variable
     (string-downcase mystr)                 ; returns "abcdefgh"
     (print mystr)                     ; prints  "ABcdEFgh"
```

string-equal

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(string-equal <string1> <string2> [<key> <offset>] ...)
     <string1> -   a string expression
     <string2> -   a string expression
     <key>   -   a keyword (one of :START1 :START2 :END1 :END2)
     <offset>   -   an optional integer expression (for a keyword)

DESCRIPTION

The STRING-EQUAL function takes two string arguments. It checks to see if the string arguments have the same values. T is returned if <string1> is equal to <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-equal "a" "b")            ; returns NIL
     (string-equal "a" "a")            ; returns T
     (string-equal "a" "A")            ; returns T
     (string-equal "A" "a")            ; returns T
     (string-equal "abc" "abc ")       ; returns NIL

     (string-equal "J Smith" "K Smith" ; strip off the first chars
              :start1 1 :start2 1)          ; returns T
     (string-equal "abc" "123456789"   ; leave just the first 3 chars
              :end2 3 :end1 3)         ; returns NIL
```

NOTE:
The STRING-EQUAL function is listed in the documentation that comes with XLISP as STRING-EQUALP. It functions properly in the XLISP code as STRING-EQUAL.

string-greaterp

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-greaterp <string1> <string2> [ <key> <offset> ] ... )
     <string1>  -    a string expression
     <string2>  -    a string expression
     <key>      -    a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>   -    an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-GREATERP function takes two string arguments. A non-NIL value is returned if <string1> is greater than <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR-GREATERP the corresponding character of <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-greaterp "a" "b")         ; returns NIL
     (string-greaterp "a" "a")         ; returns NIL
     (string-greaterp "a" "A")         ; returns NIL
     (string-greaterp "A" "a")         ; returns NIL
     (string-greaterp "abc" "abc ")          ; returns NIL
     (string-greaterp "1234qrst" "12345678") ; returns 4

     (string-greaterp "J Smith" "K Jones"    ; strip off the first chars
            :start1 1 :start2 1)       ; returns 2
```

string-left-trim

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(string-left-trim <trim-stuff> <string>)
     <trim-stuff>   -   a string expression
     <string>   -   a string expression

DESCRIPTION

The STRING-LEFT-TRIM function takes the <trim-stuff> characters and removes them from the left end of the <string>. The <trim-stuff> characters are an un-ordered set of characters to be removed - so any character that occurs in <trim-stuff> is removed if it appears in left portion of <string>. A new string is created and returned as the result of this function.

EXAMPLES

```
     (string-left-trim "." "....foo....")    ; returns "foo...."
     (string-left-trim "<>" "<<<<bar>>>>")   ; returns "bar>>>>"
     (string-left-trim "(.)" "..(12.34)..")  ; returns "12.34).."
```

COMMON LISP COMPATABILITY:
Common LISP also supports a list of characters as a valid <trim-stuff> argument. An example of this is: (STRING-TRIM '(#\Tab #\Newline) mystring). XLISP does not support this non-string parameter. Porting from XLISP will be no problem, but modifications will be necessary if porting from Common LISP code which uses a list of characters.

string-lessp

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-lessp <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-LESSP function takes two string arguments. A non-NIL value is returned if <string1> is less than <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR-LESSP the corresponding character of <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-lessp "a" "b")            ; returns 0
     (string-lessp "a" "a")            ; returns NIL
     (string-lessp "a" "A")            ; returns NIL
     (string-lessp "A" "a")            ; returns NIL
     (string-lessp "abc" "abc ")       ; returns 3
     (string-lessp "1234567" "1234qrst")     ; returns 4

     (string-lessp "J Smith" "K Smith" ; strip off the first chars
            :start1 1 :start2 1)       ; returns NIL
```

string-not-equal

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(string-not-equal <string1> <string2> [<key> <offset>] ...)
     <string1> -   a string expression
     <string2> -   a string expression
     <key>   -   a keyword (one of :START1 :START2 :END1 :END2)
     <offset>   -   an optional integer expression (for a keyword)

DESCRIPTION

The STRING-NOT-EQUAL function takes two string arguments. A non-NIL value is returned if <string1> is not equal to <string2>, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR-NOT-EQUAL the corresponding character of <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-not-equal "a" "b")        ; returns 0
     (string-not-equal "a" "a")        ; returns NIL
     (string-not-equal "a" "A")        ; returns NIL
     (string-not-equal "A" "a")        ; returns NIL
     (string-not-equal "abc" "abc ")         ; returns 3

     (string-not-equal "J Smith" "K Smith"   ; strip off the first chars
              :start1 1 :start2 1)           ; returns NIL
     (string-not-equal "abc" "123456789"     ; leave just the first 3
chars
              :end2 3 :end1 3)          ; returns 0
```

NOTE:
The STRING-NOT-EQUAL function is listed in the documentation that comes with XLISP as STRING-NOT-EQUALP. It functions properly in the XLISP code as STRING-NOT-EQUAL.

string-not-greaterp

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-not-greaterp <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-NOT-GREATERP function takes two string arguments. A non-NIL value is returned if <string1> is less than or equal to <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR-NOT-GREATERP the corresponding character of <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-not-greaterp "a" "b")           ; returns 0
     (string-not-greaterp "b" "a")           ; returns NIL
     (string-not-greaterp "a" "a")           ; returns 1
     (string-not-greaterp "a" "A")           ; returns 1
     (string-not-greaterp "A" "a")           ; returns 1
     (string-not-greaterp "abc" "abc "); returns 3
     (string-not-greaterp "12345" "1234qr")  ; returns 4

     (string-not-greaterp "J Smith" "K Smith"; strip off the first chars
            :start1 1 :start2 1)      ; returns 7
```

string-not-lessp

---

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-not-lessp <string1> <string2> [ <key> <offset> ] ... )
     <string1>   -     a string expression
     <string2>   -     a string expression
     <key>       -     a keyword (one of :START1 :START2 :END1 :END2 )
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-NOT-LESSP function takes two string arguments. A non-NIL value is returned if <string1> is greater than or equal to <string2> in an ASCII ordering, otherwise NIL is returned. The non-NIL value returned is the integer index of the first character of <string1> which is CHAR-NOT-LESSP the corresponding character of <string2>. This test is not case sensitive - the character #\a is considered to be the same as #\A.

The keyword arguments allow for accessing substrings within <string1> and <string2>. The keyword arguments each require the keyword (:START1 :END1 :START2 :END2) and a single integer expression as a pair with the keyword first and the integer second. The pairs may be in any order. The start keywords specify the starting offset of the substring. A value of 0 starts the string at the beginning (no offset). The end keywords specify the ending offset of the substring. A value of 3 ends the string after the 3rd character (an offset of 3 characters).

EXAMPLES

```
     (string-not-lessp "a" "b")        ; returns NIL
     (string-not-lessp "a" "a")        ; returns 1
     (string-not-lessp "a" "A")        ; returns 1
     (string-not-lessp "A" "a")        ; returns 1
     (string-not-lessp "abc" "abc ")         ; returns NIL
     (string-not-lessp "1234qr" "123456")    ; returns 4

     (string-not-lessp "J Smith" "K Jones"   ; strip off the first chars
            :start1 1 :start2 1)       ; returns 2
```

string-right-trim

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(string-right-trim <trim-stuff> <string> )
     <trim-stuff>     -     a string expression
     <string>    -     a string expression
```

DESCRIPTION

The STRING-RIGHT-TRIM function takes the <trim-stuff> characters and removes them from the right end of the <string>. The <trim-stuff> characters are an un-ordered set of characters to be removed - so any character that occurs in <trim-stuff> is removed if it appears in right portion of <string>. A new string is created and returned as the result of this function.

EXAMPLES

```
     (string-right-trim "." "....foo....")   ; returns "....foo"
     (string-right-trim "<>" "<<<<bar>>>>")  ; returns "<<<<bar"
     (string-right-trim "(.)" "..(12.34)..") ; returns "..(12.34"
```

COMMON LISP COMPATABILITY:
Common LISP also supports a list of characters as a valid <trim-stuff> argument. An example of this is: (STRING-TRIM '(#\Tab #\Newline) mystring). XLISP does not support this non-string parameter. Porting from XLISP will be no problem, but modifications will be necessary if porting from Common LISP code which uses a list of characters.

string-trim

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(string-trim <trim-stuff> <string>)
     <trim-stuff>   -   a string expression
     <string>   -   a string expression

DESCRIPTION

The STRING-TRIM function takes the <trim-stuff> characters and removes them from both ends of the <string>. The <trim-stuff> characters are an un-ordered set of characters to be removed - so any character that occurs in <trim-stuff> is removed if it appears in <string>. A new string is created and returned as the result of this function.

EXAMPLES

```
     (string-trim "." "....foo....")         ; returns "foo"
     (string-trim "<>" "<<<<bar>>>>")  ; returns "bar"
     (string-trim "(.)" "..(12.34)..") ; returns "12.34"
```

COMMON LISP COMPATABILITY:
Common LISP also supports a list of characters as a valid <trim-stuff> argument. An example of this is: (STRING-TRIM '(#\Tab #\Newline) mystring). XLISP does not support this non-string parameter. Porting from XLISP will be no problem, but modifications will be necessary if porting from Common LISP code which uses a list of characters.

string-upcase

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(string-upcase <string> [ { :start | :end } <offset> ] ... )
     <string>    -     a string expression
     <offset>    -     an optional integer expression (for a keyword)
```

DESCRIPTION

The STRING-UPCASE function takes a string argument and returns a new string that has been made upper case.

The keyword arguments allow for accessing substrings within <string>. The keyword arguments require a keyword (:START or :END) first and a single integer expression second. The :START keyword specifies the starting offset for the STRING-UPCASE operation on <string>. A value of 0 starts the string at the beginning (no offset). The :END keyword specifies the end offset for the operation on <string>.

EXAMPLES

```
     (string-upcase "ABcd+-12&[")      ; returns "ABCD+-&["
     (string-upcase "abcdefgh"         ;
                 :start 2 :end 4); returns "abCDefgh"

     (setq mystr "ABcdEFgh")                ; set up variable
     (string-upcase mystr)             ; returns "ABCDEFGH"
     (print mystr)                     ; prints  "ABcdEFgh"
```

sublis

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(sublis <a-list> <expr> [ { :test | :test-not } <test> ] )
     <expr>          -    the expression to substitute within - an atom
                     or list
     <a-list>   -    the association list to search
     <test>          -    optional test function (default is EQL)
```

DESCRIPTION

SUBLIS searches through an <expr> and replaces each of the elements in the <expr> that match the CAR of the elements of the association list <a-list> with the CDR of elements of the <a-list>. The <expr> with the substitutions (if any) is returned. You may specify your own test with the :TEST and :TEST-NOT keywords followed by the test you which to perform. The SUBLIS function is normally used with a dotted pair (A . B) association list. It is possible to use a normal list pair (A B) or a list of the form (A (B C)).

EXAMPLES

```
     (sublis '( (a . b))   '(a a))           ; returns (B B)
     (sublis '( (a b))     '(a a))           ; returns ((B) (B))
     (sublis '( (a (b c))) '(a a))           ; returns (((B C)) ((B C)))

     (setq newlist '( (a . 1)          ; set up an association list
                 (b . 2)         ;
                 (c . 3) ))      ;
     (sublis newlist '(a b c d e f b a c))   ; returns (1 2 3 D E F 2 1
3)
     (sublis newlist 'a)               ; returns 1

     (setq mylist '((a my-a) (b his-b) ; set up a non-dotted pair
                 (c her-c) (d end)))   ;   assoc list
     (sublis mylist '(a b c d e f g))  ; returns ((MY-A) (HIS-B)
                                ;          (HER-C) (END) E F G)
     (sublis mylist 'a)                ; returns (MY-A)

     (setq numlist '((1 . a) (2 . b)) ); set up a new assoc list
     (defun mytest (x y) (princ ": ")  ; set up my own test function
                    (princ x)          ;   with 2 parameters
                    (princ " ")  ;   to see what SUBLIS does
                    (princ y) (terpri) ;
                    T)                 ;   always return TRUE
     (sublis numlist '(3 1) :test mytest)    ; prints    : (3 1) 1
```

```
                                    ;   returns A
                                    ;   because the entire list
                                    ;   succeeds with the test
                                    ;   and so (1 . A) produces
                                    ;   the returned value
(sublis numlist '(1) :test-not mytest)  ; prints    : (1) 1
                                    ;           : (1) 2
                                    ;           : 1 1
                                    ;           : 1 2
                                    ;           : NIL 1
                                    ;           : NIL 2
                                    ;   returns (1)
                                    ;   because SUBLIS tried to
                                    ;   match every list/sublist
                                    ;   against each entry in the
                                    ;   assoc. list and failed
                                    ;   because of the :TEST-NOT
                                    ;   and so returned the
                                    ;   original list unaltered
```

NOTE:
The SUBLIS function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
Common LISP supports the use of the :KEY keyword which specifies a function that is applied to each element of <a-list> before it is tested. XLISP does not support this.

subseq

---

type: function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(subseq <string> <start> [ <end> ] )
      <string>    -     a string expression
      <start>           -     an integer expression
      <end>       -     an integer expression
```

DESCRIPTION

The SUBSEQ function extracts a substring from <string> starting with the <start> offset and ending with the <end> offset. The <start> offset has a origin or 0. The substring is returned.

EXAMPLES

```
      (subseq "12345678" 0)             ; returns "12345678"
      (subseq "12345678" 2)             ; returns "345678"
      (subseq "12345678" 2 4)                 ; returns "34"
      (subseq "1234" 3)           ; returns "4"

      (subseq "1234" 4)           ; returns ""
      (subseq "1234" 4 2)               ; returns ""
      (subseq "1234" 5)           ; error: string index out of
                                  ;      bounds - 5
```

COMMON LISP COMPATABILITY:
The SUBSEQ in Common LISP is intended to return a portion of a sequence - a SUBSEQuence. This function operates on lists and vectors (one-dimensional arrays of data) - basically ordered data. Strings are just one of the valid types operated on by SUBSEQ in Common LISP. The XLISP SUBSEQ function only operates on strings.

subst

---

type: function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

```
(subst <new-expr> <old-expr> <expr> [ { :test | :test-not } <test> ] )
     <old-expr> -     the expression to search for
     <new-expr> -     the expression to replace <old-expr> with
     <expr>           -     the expression to substitute within -
atom/list
     <test>           -     optional test function (default is EQL)
```

DESCRIPTION

SUBST searches through an <expr> and replaces each of the <old-expr> elements with the <new-expr>. The <expr> with the substitutions (if any) is returned. You may specify your own test with the :TEST and :TEST-NOT keywords followed by the test you which to perform.

EXAMPLES

```
     (subst 'new 'old '(old mid dif))  ; returns (NEW MID DIF)
     (subst '(a) 'old '(old mid dif))  ; returns ((A) MID DIF)
     (subst "a" 'old '(old mid dif))         ; returns ("a" MID DIF)

     (defun mytest (x y) (princ x)(princ " "); define a test function
                    (princ y)(terpri)  ;   that prints the arguments
                    T )                ;   and always returns TRUE
     (subst 'a 'b '(a b c d) :test 'mytest)  ; prints (A B C D) B
returns A
     (subst 'a 'b '(a b) :test-not 'mytest)  ; prints (A B) B
                                 ;      A B
                                 ;      (B) B
                                 ;      B B
                                 ;      NIL B     returns (A B)
```

NOTE:
The SUBST function can work with a list or string as the <expr>. However, the default EQL test does not work with lists or strings, only symbols and numbers. To make this work, you need to use the :TEST keyword along with EQUAL for <test>.

COMMON LISP COMPATABILITY:
Common LISP supports the use of the :KEY keyword which specifies a function that is applied to each element of <expr> before it is tested. XLISP does not support this.

symbol-name

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(symbol-name <symbol>)
     <symbol>   -   an expression that evaluates to a symbol name

DESCRIPTION

The SYMBOL-NAME function takes the <symbol> expression and returns the printable string of the <symbol>. If the <symbol> had not existed, then it will be created and INTERNed into the system symbol table *OBARRAY* - but with it's value unbound and an empty property list.

EXAMPLES

```
     (symbol-name 'foo)                ; returns "FOO"
     (symbol-name 'gleep)              ; returns "GLEEP"

     (setq my-symbol 'flop)            ; define MY-SYMBOL
     (symbol-name my-symbol)                ; returns "FLOP"
```

symbol-plist

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(symbol-plist <symbol>)
     <symbol>   -   the symbol name with a property list

DESCRIPTION

SYMBOL-PLIST returns the actual property list from the <symbol>. The <symbol> must be an existing, bound variable, but it does not need to have anything in it's property list.

Property lists are lists attached to any user defined variables. The lists are in the form of (name1 val1 name2 val2 ....). Any number of properties may be attached to a single variable.

EXAMPLES

```
     (setq person 'bobby)              ; create a var with a value
     (putprop person 'boogie 'last-name)     ; add a LAST-NAME property
     (putprop person 'disc-jockey 'job); add a JOB property
     (putprop person '(10 20 30) 'stats)     ; add a STATS list
     (symbol-plist person)             ; returns the property list:
                                ; (STATS (10 20 30)
                                ;  JOB DISC-JOCKEY
                                ;  LAST-NAME BOOGIE)
```

symbol-value

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(symbol-value <symbol>)
     <symbol>   -   an expression that evaluates to a symbol name

DESCRIPTION

The SYMBOL-VALUE function takes the <symbol> expression and returns the current value of the <symbol>.

If the <symbol> had not existed, then it will be created and INTERNed into the system symbol table *OBARRAY* - but with it's value unbound and an empty property list. In this case of a previously non-existant <symbol>, since it has no bound value, the SYMBOL-VALUE will still report an error due to an unbound variable.

EXAMPLES

```
     (setq myvar 55)                   ; set MYVAR to value 55
     (symbol-value 'myvar)             ; returns 55

     (symbol-value 'floop)             ; error: unbound variable

     (setq my-symbol 'a)               ; set MY-SYMBOL to A
     (setq a '(contents of symbol a))  ; set A to value -
                                  ;   (CONTENTS OF SYMBOL A)
     (symbol-value my-symbol)          ; returns (CONTENTS OF SYMBOL A)
```

symbolp

type: predicate function (subr)
location: built-in
source file: xllist.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(symbolp <expr> )
     <expr>           -     the expression to check
```

DESCRIPTION

The SYMBOLP predicate checks if an <expr> is a valid symbol. T is returned if <expr> is a symbol, NIL is returned otherwise. An <expr> that evaluates to an integer, function (subr or otherwise), and so on is not a symbol. However, the quoted (un-evaluated) name of these objects (like 'MYARRAY) is a valid symbol.

EXAMPLES

```
     (symbolp (make-symbol "a"))       ; returns T - symbol
     (symbolp 'a)                      ; returns T - symbol

     (symbolp #(1 2 3))                ; returns NIL - array
     (symbolp (lambda (x) (print x)))  ; returns NIL - closure
     (symbolp *standard-output*)       ; returns NIL - stream
     (symbolp 1.2)                     ; returns NIL - float
     (symbolp 2)                  ; returns NIL - integer
     (symbolp object)             ; returns NIL - object
     (symbolp "hi")                    ; returns NIL - string

     (symbolp #'car)                   ; returns NIL - subr
     (symbolp 'car)                    ; returns T - it is a symbol now
     (symbolp '2)                      ; returns NIL - not a symbol
```

system

---

type: function (subr)
location: system extenstion
source file: msstuff.c and osdefs.h and osptrs.h
Common LISP compatible: no
supported on: MS-DOS compatibles

SYNTAX

(system <command>)
     <command> -   the OS command string to be executed

DESCRIPTION

The SYSTEM function will send the <command> string to the underlying operating system for execution. After execution of the <command>, the SYSTEM function will return a T result if the <command> was successful. If the <command> was not successful, the numeric error code will be returned. Any output from the <command> execution will not be put in the transcript file.

EXAMPLES

```
(system "dir")                    ; do a PC directory listing
(system "mycmd")             ; execute a special command
```

NOTE:
This function is an extension of the XLISP system. It is provided in the MSSTUFF.C source code file. If your XLISP system is built for an IBM PC and compatibles or generic MS-DOS, this function will work. If your system is built on UNIX or some other operating system, it is unlikely that these functions will work unless you extend the appropriate STUFF.C file (which may be called something different like UNIXSTUFF.C). The source that could be put in the appropriate STUFF.C file for this extension to work on a UNIX style system is:

```
/* xsystem - execute a system command */
LVAL xsystem()
{
    char *cmd="COMMAND";
    if (moreargs())
      cmd = (char *)getstring(xlgastring());
    xllastarg();
    return (system(cmd) == 0 ? true : cvfixnum((FIXTYPE)errno));
}
```

The source that gets added to the OSDEFS.H file is:

```
extern LVAL xsystem();
```

The source that gets added to the OSPTRS.H file is:

```
{     "SYSTEM",   S,    xsystem     },
```

t

---

type: system constant
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

t

DESCRIPTION

The T constant is built into XLISP. It represents True - as oppossed to false (NIL).

EXAMPLES

```
      (setq myvar T)                    ; set MYVAR to True
      (setq myvar 'T)                   ; T and 'T both evaluate to T
      (if t (print "this will print")         ; if/then/else
            (print "this won't print"))
```

NOTE:
Be careful with the T value. It is possible to do a SETQ on T and set it to other values (like NIL). Some operations will still return proper T or NIL values, but the system will be in a bad state.

tagbody

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(tagbody  [ <expr> ... ]  )
     <expr>           -     expressions comprising the body of the block
                      which may contain GOs or tags for GO
```

DESCRIPTION

The TAGBODY special form is basically a 'block' construct that contains a block of code (expressions) to evaluate. After the execution of the TAGBODY <expr>'s, NIL is returned. The TAGBODY special form allows 'go-to' style branching within the 'block' construct via the GO special form. To allow this, each <expr> may be a tag or a form. The tag-symbol is the 'label' and must exist somewhere within the 'block' that the GO occurs within.

EXAMPLES

```
     (tagbody                     ; build the 'block'
        start (print "begin")          ; tag - start
            (GO end)             ;
            (print "hello")      ; won't ever be reached
        end   (print "done"))          ; tag - END
                                 ; prints  "begin" "done"
                                 ;   returns NIL
```

tan

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(tan <expr>)
     <expr>   -   floating point number/expression

DESCRIPTION

The TAN function calculates the tangent of the <expr> and returns the result. The <expr> is in radians.

EXAMPLES

```
     (tan 0.0)                  ; returns 0
     (tan 1.0)                  ; returns 1.55741
     (tan (/ 3.14159 2))             ; returns 753696
     (tan 2.0)                  ; returns -2.18504
     (tan 3.0)                  ; returns -0.142547
     (tan 3.14159)                   ; returns -2.65359e-06
     (tan 4.5)                  ; returns 4.63733
```

COMMON LISP COMPATABILITY:
Common LISP allows for integer numbers, which XLISP does not support for TAN.

terpri

---

type: function (subr)
location: built-in
source file: xlfio.c and xlprin.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(terpri [<destination>])
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The TERPRI function prints a new-line to the specified <destination>. This will terminate the current print line for <destination>. NIL is always returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default.

EXAMPLES

```
(terpri)                    ; prints  <NL>

(setq f (open "pr" :direction :output )); create a file
(princ "hi" f)                  ; returns "hi"
(princ 727 f)                   ; returns 727
(princ "ho" f)                  ; returns "ho"
(terpri f)                  ; returns NIL
(close f)                   ; file contains hi727ho\n
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

third

type: function (subr)
location: built-in
source file: xlinit.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(third <expr> )
     <expr>           -     a list or list expression
```

DESCRIPTION

THIRD returns the third element of a list or list expression. If the list is NIL, NIL is returned.

EXAMPLES

```
     (third '(1 2 3 4))                ; returns 3
     (third NIL)                  ; returns NIL

     (setq kids '(junie vickie cindy chris)) ; set up variable KIDS
     (first kids)                       ; returns JUNIE
     (second kids)                      ; returns VICKIE
     (third kids)                       ; returns CINDY
     (fourth kids)                      ; returns CHRIS
     (rest kids)                  ; returns (VICKIE CINDY CHRIS)
```

NOTE:
This function is set to the same code as CADDR.

throw

---

type: special form (fsubr)
location: built-in
source file: xlcont.c and xljump.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(throw <tag-symbol> [<expr>])
<tag-symbol> - an expression that evaluates to a symbol
<expr> - an optional expression to be returned

DESCRIPTION

The CATCH and THROW special forms allow for non-local exits and traps without going through the intermediate evaluations and function returns. The <expr> in THROW specifies what value is to be returned by the corresponding CATCH. If there is no <expr>, a NIL is returned to the corresponding CATCH. If a THROW is evaluated with no corresponding CATCH, an error is generated - "error: no target for THROW". If, in the calling process, more than one CATCH is set up for the same <tag-symbol>, the most recently evaluated <tag-symbol> will be the one that does the actual catching.

EXAMPLES

```
(catch 'mytag)                      ; returns NIL    - no THROW
(catch 'mytag (+ 1 (+ 2 3)))       ; returns 6      - no THROW
(catch 'mytag (+ 1 (throw 'mytag)))     ; returns NIL    - caught it
(catch 'mytag (+ 1 (throw 'mytag 55)))  ; returns 55     - caught it
(catch 'mytag (throw 'foo))        ; error: no target for THROW

(defun in (x)                       ; define IN
   (if (numberp x) (+ x x)          ;   if number THEN double
               (throw 'math 42)))   ;              ELSE throw 42
(defun out (x)                      ; define OUT
  (princ "<") (princ  (* (in x) 2))     ;   double via multiply
  (princ ">"))                      ;
(defun main (x)                     ; define MAIN
  (catch 'math (out x)))            ;   with CATCH
(in 5)                              ; returns 10
(out 5)                             ; prints  <20>   returns ">"
(main 5)                    ; prints  <20>   returns ">"
(main 'a)                   ; prints  < returns 42
```

NOTE:
Although CATCH and THROW will accept a <tag-symbol> that is not a symbol, it will not find this improper <tag-symbol>. An error will be generated - "error: no target for THROW".

:tmacro

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(:tmacro . <function>)
      <function> -   a function

DESCRIPTION

:TMACRO is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :TMACRO keyword means that the specified character is a terminal read macro. For :TMACRO, the form of the *READTABLE* entry is a dotted pair like (:TMACRO . <function>). The <function> can be a built-in read-macro function or a user defined lambda expression. The <function> takes two parameters, an input stream specification, and an integer that is the character value. The <function> should return NIL if the character is 'white-space' or a value CONSed with NIL to return the value. The <function> will probably read additional characters from the input stream.

EXAMPLES

```
      (defun look-at (table)            ; define a function to
       (dotimes (ch 127)                ;   look in a table
        (prog ( (entry (aref table ch)) )     ;   and print out any
              (if (and (consp entry)          ;   :TMACRO entries
                       (equal (car entry)     ;
                         ':TMACRO))     ;
                (princ (int-char ch)))));
       (terpri))                    ;
                                    ;
      (look-at *readtable*)             ;  prints "'(),;`
```

NOTE:
The system defines that the following are :TMACRO characters:

      \ " ` , ( ) ;

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

top-level

---

type: function (subr)
location: built-in
source file: xlbfun.c and xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

(top-level)

DESCRIPTION

The TOP-LEVEL function aborts to the top level of XLISP. This may be from within several levels of the break loop. This is valid for BREAKs, ERRORs and CERRORs (continuable errors). If TOP-LEVEL is evaluated while not in a break loop, a message is printed - "[back to the top level]". This message does not cause XLISP to go into a break loop. TOP-LEVEL never actually returns a value.

EXAMPLES

```
(top-level)                 ; [ back to the top level ]

(break "out")                     ; break: out          (1st)
(break "twice")                   ; break: twice        (2nd)
(top-level)                 ; to exit out of break loop
```

KEYSTROKE EQUIVALENT:
In the IBM PC and MS-DOS versions of XLISP, a CTRL-c key sequence has the same effect as doing a (TOP-LEVEL). On a Macintosh, this can be accomplished by a pull-down menu or a COMMAND-t.

trace

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(trace <function> ...)
<function> - an unquoted function

DESCRIPTION

The TRACE special form allows the tracing of user or system functions. TRACE returns a list containing the current set of functions that are being traced. The <function> does not have to be currently defined, it can be created as part of the execution. The trace output consists of entry and exit information. At entry and exit of a traced <function>, lines will be printed of the form:

```
Entering: <function>, Argument list: <arg-list>
      .
      .
      .
Exiting: <function>, Value: <ret-value>
```

EXAMPLES

```
(defun foo (x) (print (car x)))         ; define FOO
(trace 'foo)                        ; returns (FOO)
(trace 'car)                        ; returns (CAR FOO)
(foo '(a))                  ; Entering: FOO, Argument list: ((A))
                            ;  Entering: CAR, Argument list: ((A))
                            ;  Exiting: CAR, Value: A
                            ; A
                            ; Exiting: FOO, Value: A
                            ; returns A
```

COMMON LISP COMPATABILITY:
The XLISP TRACE function does not support any keyword options, which Common LISP allows.

tracelimit

---

type: system variable
location: built-in
source file: xlinit.c and xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

tracelimit

DESCRIPTION

TRACELIMIT is a system variable that controls the number of forms printed on entry to the break loop. If *TRACELIMIT* is an integer, then the integer is the maximum number of forms that will be printed. If *TRACELIMIT* is NIL or a non-integer, then all of the forms will be printed. Note that *TRACENABLE* needs to be set to a non-NIL value to enable the printing of back-trace information on entry to the break loop.

EXAMPLES

```
(defun foo (x) (fee x))                ; define FOO
(defun fee (y) (break))                ; define FEE
(setq *tracenable* T)              ; enable the back trace
(setq *tracelimit* NIL)                ; show all the entries
(foo 5)                            ; break: **BREAK**
                                ; prints  Function:.....BREAK..
                                ;       Function:.....FEE....
                                ;       Arguments:
                                ;         5
                                ;       Function:.....FOO....
                                ;       Arguments:
                                ;         5
(clean-up)                      ; from break loop
(setq *tracelimit* 2)               ; show only 2 entries
(foo 5)                             ; break: **BREAK**
                                ; prints  Function:.....BREAK..
                                ;       Function:.....FEE....
                                ;       Arguments:
                                ;         5
(clean-up)                      ; from break loop
```

NOTE:
TRACENABLE and *TRACELIMIT* have to do with back trace information at entry to a break loop and have nothing to do with TRACE and UNTRACE.

tracelist

---

type: system variable
location: built-in
source file: xlinit.c and xleval.c
Common LISP compatible: no
supported on: all machines

SYNTAX

tracelist

DESCRIPTION

TRACELIST is a system variable that contains a list of the current functions being traced.

EXAMPLES

```
(defun foo (x) (print (car x)))         ; define FOO
(trace foo)                  ; returns (FOO)
(trace car)                  ; returns (CAR FOO)
(print *tracelist*)               ; prints  (CAR FOO)
(untrace foo)                     ; returns (CAR)
(untrace car)                     ; returns NIL
(print *tracelist*)               ; prints  NIL
```

tracenable

type: system variable
location: built-in
source file: xlinit.c and xldbug.c
Common LISP compatible: no
supported on: all machines

SYNTAX

tracenable

DESCRIPTION

TRACENABLE is a system variable that controls whether or not the break loop prints any back trace information on entry to the break loop. If *TRACENABLE* is NIL, then there will be no information printed on entry to the break loop. If *TRACENABLE* is non-NIL, then information will be printed. The INIT.LSP initialization file sets *TRACENABLE* to NIL, which suppresses the printing.

EXAMPLES

```
(defun foo (x) (fee x))                ; define FOO
(defun fee (y) (break))                ; define FEE
(setq *tracenable* T)             ; enable the back trace
(setq *tracelimit* NIL)                ; show all the entries
(foo 5)                           ; break: **BREAK**
                                ; prints  Function:.....BREAK..
                                ;       Function:.....FEE....
                                ;       Arguments:
                                ;         5
                                ;       Function:.....FOO....
                                ;       Arguments:
                                ;         5
(clean-up)                      ; from break loop
(setq *tracelimit* 2)                ; show only 2 entries
(foo 5)                           ; break: **BREAK**
                                ; prints  Function:.....BREAK..
                                ;       Function:.....FEE....
                                ;       Arguments:
                                ;         5
(clean-up)                      ; from break loop
```

NOTE:
TRACENABLE and *TRACELIMIT* have to do with back trace information at entry to a break loop and have nothing to do with TRACE and UNTRACE.

trace-output

---

type: system variable
location: built-in
source file: xlinit.c xlio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

trace-output

DESCRIPTION

TRACE-OUTPUT is a system variable that contains a file pointer that points to the file where all trace output goes to. The default file for *TRACE-OUTPUT* is the system standard error device - normally the screen.

EXAMPLES

```
      *trace-output*                    ; returns #<File-Stream: #243de>
```

NOTE:
TRACE-OUTPUT, *DEBUG-IO* and *ERROR-OUTPUT* are normally all set to the same file stream - STDERR.

truncate

---

type: function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(truncate <expr>)
     <expr>   -   integer or floating point number/expression

DESCRIPTION

The TRUNCATE function takes the <expr> and truncates it to an integer value and returns this result.

EXAMPLES

```
     (truncate 123.456)              ; returns 123
     (truncate -1.49)           ; returns -1
     (truncate -1.59)           ; returns -1
     (truncate 123)                  ; returns 123
     (truncate 123.999)              ; returns 123
```

COMMON LISP COMPATABILITY:
Common LISP allows an optional division parameter, which XLISP does not support.

type-of

---

type: function (subr)
location: built-in
source file: xlsys.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(type-of <expr>)
     <expr>   -   an expression to check

DESCRIPTION

The TYPE-OF function returns the type of the expression.

EXAMPLES

```
     (type-of NIL)                     ; returns NIL
     (type-of '#(1 2 3))               ; returns ARRAY
     (type-of (lambda (x) (print x)))  ; returns CLOSURE
     (type-of '(a b))             ; returns CONS
     (type-of #'savefun)               ; returns CLOSURE
     (type-of '(a . b))                ; returns CONS
     (type-of *standard-output*)       ; returns FILE-STREAM
     (type-of 1.2)                     ; returns FLONUM
     (type-of #'do)                    ; returns FSUBR
     (type-of 1)                  ; returns FIXNUM
     (type-of object)             ; returns OBJECT
     (type-of "str")                   ; returns STRING
     (type-of #'car)                   ; returns SUBR
     (type-of 'a)                      ; returns SYMBOL
     (type-of #\a)                     ; returns CHARACTER
     (type-of (make-string-input-stream "a")); returns UNNAMED-STREAM
```

COMMON LISP COMPATABILITY:
The XLISP and Common LISP TYPE-OF functions are basically the same. Differences between the two can occur in what the types are called (like CHARACTER in XLISP and STANDARD-CHAR in Common LISP). Also, Common LISP can give additional information - for strings, it returns a list of the form (SIMPLE-STRING 32) where the number 32 is the string size.

unbound

---

type: system constant
location: built-in
source file: xlinit.c and xlsym.c
Common LISP compatible: no
supported on: all machines

SYNTAX

unbound

DESCRIPTION

UNBOUND is a system constant that is used to indicate when a symbol has no value. *UNBOUND* is set to the value *UNBOUND*. This means that the system thinks the symbol *UNBOUND* has no value.

EXAMPLES

```
      *unbound*                    ; error: unbound variable
      (setq a 5)                   ; returns 5
      a                            ; returns 5
      (setq a '*unbound*)               ; returns *UNBOUND*
      a                            ; error: unbound variable
```

unless

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(unless <test> [ <expr> ... ] )
     <test>           -     an expression - NIL or non-NIL
     <expr>           -     expressions comprising a body of code
```

DESCRIPTION

The UNLESS macro executes the <expr> forms if <test> evaluates to a NIL value. If <test> is NIL, the value of the last <expr> is returned as the result. If <test> is non-NIL, NIL is returned with none of <expr> evaluated.

EXAMPLES

```
     (unless NIL)                        ; returns NIL
     (unless T)                    ; returns NIL
     (unless NIL (print "hi") 'foo)          ; prints  "hi"   returns FOO
     (unless (listp "a")                ;
           (print "not a list"))        ; prints  "not a list"
                                   ; returns "not a list"
```

untrace

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: similar
supported on: all machines

SYNTAX

(untrace <function> ...)
<function> - a function name

DESCRIPTION

The UNTRACE special form removes <function> from the current list of traced functions. UNTRACE returns a list containing the current set of functions that are being traced. If the <function> does currently exist or is currently be traced, there will be no error reported. If there are no functions being traced, a NIL is returned.

EXAMPLES

```
(defun foo (x) (print (car x)))         ; define FOO
(trace 'foo)                        ; returns (FOO)
(foo '(a))                  ; Entering: FOO, Argument list: ((A))
                            ; A
                            ; Exiting: FOO, Value: A
                            ; returns A
(untrace 'foo)                      ; returns NIL
(untrace 'glip)                     ; returns NIL
(foo '(a))                  ; prints  A   and returns  A
```

COMMON LISP COMPATABILITY:
The XLISP UNTRACE function does not support any options, which Common LISP allows.

upper-case-p

type: predicate function (subr)
location: built-in
source file: xlstr.c
Common LISP compatible: yes
versions: all machines

SYNTAX

(upper-case-p <char>)
     <char> - a character expression

DESCRIPTION

The UPPER-CASE-P predicate checks if the <char> expression is an upper case character. If <char> is upper case a T is returned, otherwise a NIL is returned. Upper case characters are 'A' (ASCII decimal value 65) through 'Z' (ASCII decimal value 90).

EXAMPLES

```
(upper-case-p #\A)                ; returns T
(upper-case-p #\a)                ; returns NIL
(upper-case-p #\1)                ; returns NIL
(upper-case-p #\[)                ; returns NIL
```

unwind-protect

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(unwind-protect <protect-form> <clean-up-form> ...)
<protect-form> - a form that is to be protected
<clean-up-form> - a sequence forms to execute after <protect-form>

DESCRIPTION

The UNWIND-PROTECT special form allows the protecting (trapping) of all forms of exit from the <protect-form>. The exits that are trapped include errors, THROW, RETURN and GO. The <clean-up-form> will be executed in all cases - when there is an exit from <protect-form> and when the form does not have exit. UNWIND-PROTECT will return the result from the <protect-form>, not from the <clean-up-form>s. Errors or exits that occur in the <clean-up-form> are not protected. It is possible to trap these with another UNWIND-PROTECT.

EXAMPLES

```
(unwind-protect              ;
      (+ 2 2)                ; protected form
      (print "an exit"))           ; clean up form
                             ; prints "an exit"
                             ; returns 4

(nodebug)                    ; to turn off break loop traps
(unwind-protect              ;
      (+ 1 "2")              ; protected form
      (print "something happened"))     ; clean up form
                             ; error: bad argument type - "2"
                             ; prints "something happened"

(catch 'mytag                      ;
  (unwind-protect                  ;
      (throw 'mytag)               ; protected form
      (print "an exit") ) )        ; clean up form
                             ; prints "an exit"

(nodebug)                    ; to turn off break loop traps
(unwind-protect              ;
      (throw 'notag)               ; protected form
      (print "an exit"))           ; clean up form
                             ; error: no target for THROW
                             ; prints "an exit"
```

```
(prog ()                      ;
      (print "start")               ;
      (unwind-protect         ;
            (go end)          ; protected form
            (print "an exit"))      ; clean-up form
  end       (print "end") )               ; prints "start"
                              ; prints "an exit"
                              ; prints "end"

(prog ()                      ;
      (print "start")               ;
      (unwind-protect         ;
            (return "I'm done")     ; protected form
            (print "but first"))    ; clean-up form
      (print "won't get here") )    ; prints "start"
                              ; prints "but first"
                              ; returns "I'm done"
```

vector

---

type: function (subr)
location: built-in
source file: xlbfun.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(vector [<expr> ...])
     <expr>   -   an expression

DESCRIPTION

The VECTOR function creates an initialized vector and returns it as the result. VECTOR is essentially a fast method to do a one-dimensional MAKE-ARRAY with initial data in the vector.

EXAMPLES

```
(vector 'a 'b 'c)           ; returns #(A B C)
(vector '(a b) '(c d))           ; returns #((A B) (C D))
(vector)                    ; returns #()
(vector NIL)                     ; returns #(NIL)
(vector 'a () 4 "s")             ; returns #(A NIL 4 "s")
```

---

type: special form (fsubr)
location: built-in
source file: xlcont.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

```
(when  <test> [ <expr> ... ] )
      <test>           -     an expression - NIL or non-NIL
      <expr>           -     expressions comprising a body of code
```

DESCRIPTION

The WHEN macro executes the <expr> forms if <test> evaluates to a non-NIL value. If <test> is non-NIL, the value of the last <expr> is returned as the result. If <test> is NIL, NIL is returned with none of <expr> evaluated.

EXAMPLES

```
      (when NIL)                    ; returns NIL
      (when T)                      ; returns T
      (when T (print "hi") 'foo)          ; prints  "hi"   returns FOO
      (when (listp '(a))                  ;
            (print "a list"))                   ; prints  "a list"
                                    ; returns "a list"
```

:white-space

---

type: keyword
location: built-in
source file: xlread.c
Common LISP compatible: no
supported on: all machines

SYNTAX

:white-space

DESCRIPTION

:WHITE-SPACE is an entry that is used in the *READTABLE*. *READTABLE* is a system variable that contains XLISP's data structures relating to the processing of characters from the user (or files) and read-macro expansions. The existance of the :WHITE-SPACE keyword means that the specified character may be skipped over. The system defines that tab, space, return and line-feed are :WHITE-SPACE characters.

EXAMPLES

```
(defun look-at (table)            ; define a function to
 (dotimes (ch 127)                ;   look in a table
  (prog ( (entry (aref table ch)) )     ;   and print out any
    (case entry             ;   entries with a function
      (NIL     NIL)         ;
      (:CONSTITUENT NIL)           ;
      (:WHITE-SPACE (print ch))         ;
      (T       NIL))))      ;
 (terpri))                   ;
(look-at *readtable*)              ;  prints  9          tab
                            ;          10          newline
                            ;          12          formfeed
                            ;          13          return
                            ;          32          space
```

CAUTION:
If you experiment with *READTABLE*, it is useful to save the old value in a variable, so that you can restore the system state.

write-byte

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(write-byte <expr> [<destination>])
<expr> - an integer expression
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The WRITE-BYTE function writes the <expr> as a single byte to the specified <destination>. Only the <expr> byte is written. The <expr> must be an integer expression. The <expr> is returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default.

EXAMPLES

```
(write-byte 67)                    ; prints  C returns 67

(setq fp (open "t" :direction :output)) ; create file
(write-byte 65 fp)                 ; returns 65
(write-byte 66 fp)                 ; returns 66
(write-byte 10 fp)                 ; returns 10
(close fp)                    ; returns NIL
(read (open "t" :direction :input))     ; returns AB
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

write-char

---

type: function (subr)
location: built-in
source file: xlfio.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(write-char <char-expr> [<destination>])
<char-expr> - a character expression
<destination> - an optional destination - must be a file pointer
or stream, the default is *standard-output*

DESCRIPTION

The WRITE-CHAR function writes the <char-expr> to the specified <destination>. Only the <char-expr> is written. The <char-expr> must be a character expression. The <char-expr> is returned as the result. The <destination> may be a file pointer or a stream. If there is no <destination>, *STANDARD-OUTPUT* is the default.

EXAMPLES

```
(write-char #\C)            ; prints  C

(setq fp (open "t" :direction :output)) ; create file
(write-char #\A fp)               ; returns #\A
(write-char #\B fp)               ; returns #\B
(write-char #\Newline fp)         ; returns #\Newline
(close fp)                  ; returns NIL
(read (open "t" :direction :input))     ; returns AB
```

COMMON LISP COMPATABILITY:
Common LISP specifies that print operations with a <destination> of NIL, will go to *STANDARD-OUTPUT*. XLISP does not send the output to *STANDARD-OUTPUT* with a <destination> of NIL. Common LISP also specifies that a <destination> of T will be sent to *TERMINAL-IO*. XLISP does not allow T as a valid argument for <destination>.

zerop

type: predicate function (subr)
location: built-in
source file: xlmath.c
Common LISP compatible: yes
supported on: all machines

SYNTAX

(zerop <expr>)
<expr> - the numeric expression to check

DESCRIPTION

The ZEROP predicate checks to see if the number <expr> is zero. T is returned if the number is zero, NIL is returned otherwise. A bad argument type error is generated if the <expr> is not a numeric expression.

EXAMPLES

```
(zerop 0)                    ; returns T
(zerop 0.0)                  ; returns T
(zerop 99999.9)                   ; returns NIL
(zerop -0.000000000002)                ; returns NIL

(zerop 'a)                   ; error: bad argument type
(setq a 0)                   ; set value of A to 0
(zerop a)                    ; returns T
```