Common Lisp Symbols

Symbols have several properties

- Package
  - The package where the symbol is "interned"

- Name
  - The print name of the symbol

- Function
  - The function value

- Value
  - The "ordinary" value

- plist
  - The property list

The reader looks up names in the current package

- If found returns it
- If not found creates a new symbol and interns it in the current package and returns it
Common Lisp Special Forms and Functions

Evaluation and Compilation

lambda
A lambda expression represents a function without a name
The form is
(lambda (args*) body)
A call on this function will bind the args to the caller's parameters and
then evaluate the body with these bindings

eval
eval form => result*
calls the interpreter on the argument form and returns the result

quote
quote object => object
returns its argument unevaluated

defmacro
defmacro name lambda-list form* => name
evaluates the form and the result is substituted for the original call so it
is reevaluated
Macros are the way to add new syntax to the language

nil
The value of the symbol nil is nil
This is the only "false" value, all other values are true
This represents the empty list

t
The value of the symbol t is t
This can be used to represent the true value
Data and Control Flow

**apply**

apply function &rest args+ => result*

Apply the function to the list of arguments extended by the last argument as a list and return the result

The last argument must be a list

apply function args => result*

This is the common form where args is a list of the arguments to be passed in to the function

**defun**

defun function-name lambda-list form*

This defines function-name as a top-level function with the given arguments and body

**flet, labels, macrolet**

flet ((function-name lambda-list local-form*)*) form* => result*

labels ((function-name lambda-list local-form*)*) form* => result*

macrolet ((name lambda-list local-form*)*) form* => result*

These define local function or macros that are effective only for the form*

flet defines functions but all bodies cannot see other flet definitions - recursive calls won't work

labels defines functions that can see their own and each other's definitions - recursive and mutually recursive functions work

macrolet defines local macro definitions
funcall
funcall function &rest args => result*
  call the function with the given arguments
  useful to call a function that is the value (not function value) of a
  variable

function
function name => function
  get the function value of name
  can also be written #'name

defconstant
defconstant name initial-value => name
  defines a top-level constant
  WARNING - this makes the variable name special

defparameter, defvar
defparameter name initial-value => name
defvar name [initial-value] => name
  Defines a top-level variable
  defparameter always sets the value
  defvar sets the value only if not already set
  You can use setq to change this value later
  WARNING - this makes the variable name special
let, let*

let* ({var | (var [init-form])}*) form* => result*

let* ({var | (var [init-form])}*) form* => result*

Creates new variable bindings and evaluates form* in this environment

let makes the bindings simultaneously so no binding can depend on another

let* makes them one at a time so previous bindings can be seen

(let ((x 2) (y 3)) z)

is equivalent to

((lambda (x y) z) 2 3)

setq

setq var form => result

rebinds var to the value of form and returns this new value

This is the lisp equivalent of assignment

not

not x => boolean

returns t if x is nil otherwise returns nil

eq

eq x y => generalized-boolean

true if x and y are identical objects

eql

eql x y => generalized-boolean

true if x and y are eq or if x and y are both numbers of the same type and the same value or if they are both characters that represent the same character. Otherwise the value of eql is false.
equal

equal x y => generalized-boolean

Returns true if x and y are structurally similar (isomorphic) objects.
For conses, equal is defined recursively as the two cars being equal and the two cdrs being equal.
Strings are equal if they have the same characters

equalp

equalp x y => generalized-boolean

In addition to equal compares arrays element by element and same-type structures slot by slot.
Also compares entries in hash tables

and

and form* => result*

True if all form* are true. Stops evaluating on first false
Returns last form if all previous forms are true

cond

cond {clause}* => result*
clause::= (test-form form*)

Successively evaluates the test-forms of clause until one evaluates non-nil
Then evaluates form* in this clause and returns the last form evaluated (which may be the value of the test-form)
If no test-form evaluates non-nil then returns nil

if

if test-form then-form [else-form] => result*

if the test-form evaluates non-nil then returns the result of evaluating the then-form. Otherwise the result is the result of evaluating the else-form or nil if there is no else-form.
or
or form* => results*
    false if all forms are false
    Returns value of first non-nil form and stops evaluating forms
when, unless
when test-form form* => result*
unless test-form form* => result*
    when evaluates form* when test-form is true
    unless evaluates form* when test-form is false
    Result is last form evaluated or nil if the test fails and no forms are evaluated
**case, ccase, ecase**

case keyform {normal-clause}* [otherwise-clause] => result*
ccase keyplace {normal-clause}* => result*
ecase keyform {normal-clause}* => result*

normal-clause::= (keys form*)
otherwise-clause::= ({otherwise | t} form*)
clause::= normal-clause | otherwise-clause

def case k ((1 2) 'clause1)
(3 'clause2)
(nil 'no-keys-so-never-seen)
((nil) 'nilslot)
((:four #\v) 'clause4)
((t) 'tslot)
(otherwise 'others)))

Evaluates the keyform and looks for a normal-clause that has this key
If so then evaluates corresponding form* and returns last value

Otherwise clause always matches if there is one
case returns nil if no match and no otherwise clause
ecase causes an error if no match
ccase causes a continuable error if no match

**prog, prog***

prog* ({var | (var [init-form])}*) declaration* {tag | statement}* => result*
prog* ({var | (var [init-form])}*) declaration* {tag | statement}* => result*

Sets up a section of code with variable bindings where gotos and returns are valid
prog1, prog2, progn

prog1 first-form form* => result-1
prog2 first-form second-form form* => result-2
progn form* => result*

Evaluate the forms in order but
  prog1 return the value of the first form
  prog2 return the value of the second form
  progn return the value of the last form

setf

setf place newvalue => result

Finds the "place" of the place expression and sets it to newvalue
  and returns this result
(setf (car x) 3)
Iteration

\[ \text{do, do}^* \]
\[ \text{do} \left( \{\text{var} \mid (\text{var [init-form [step-form]]})\}^* \right) \]
\[ (\text{end-test-form result-form}^*) \]
\[ \text{statement}^* \]
\[ \Rightarrow \text{result}^* \]

\[ \text{do}^* \left( \{\text{var} \mid (\text{var [init-form [step-form]]})\}^* \right) \]
\[ (\text{end-test-form result-form}^*) \]
\[ \text{statement}^* \]
\[ \Rightarrow \text{result}^* \]

Bind all the var to the init-form then loop
while the end-test-form evaluates false
evaluate the statement*
rebind the variables to the step-form
evaluate the result-form* and return the last value

loop

This is an extensive looping facility

Here are some examples

\( \text{(loop for i from 1 to 10 collect i)} \)
\[ \Rightarrow (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10) \]
\( \text{(loop for i in '(1 2 3) sum i)} \)
\[ \Rightarrow 6 \]
Conditions

assert

error
datum &rest arguments => [[no result]]

datum is a format string that interprets the arguments to print an error message before entering the debugger
Symbols

gensym
gensym &optional x => new-symbol

returns a new symbol based on the string or number x
The new symbol is not interned in any package

symbol-function
symbol-function symbol => contents

Accesses the symbol's function cell

symbol-name
symbol-name symbol => name

Accesses the symbol's name

symbol-package
symbol-package symbol => contents

Accesses the symbol's package

symbol-plist
symbol-plist symbol => plist

Accesses the symbol's property list

symbol-value
symbol-value symbol => value

Accesses the symbol's value
Packages

in-package
in-package name => package

Causes name to be the current package

intern
intern string &optional package => symbol, status

Makes a symbol in a given package

*package*

Special variable that holds the current package
Numbers

There are all sorts of numbers

- integers
  - "infinite" precision
- rationals
  - fractions
- reals
  - floating-point numbers
- complexes
  - complex numbers
- bytes of various flavors

mod

=, /=, <, >, <=, >=
  - does comparisons
  - may take more than 2 arguments

max, min
  - may take more than 2 arguments

minusp, plusp
  - test for positive or negative

zerop
  - test for zero

floor, ffloor, ceiling, fceiling, truncate, ftruncate, round, fround
  - various mod and remainder functions

sin, cos, tan
asin, acos, atan

pi
  - the value of pi
sinh, cosh, tanh, asinh, acosh, atanh

*, +, -, /

1+, 1-
    add and subtract 1

abs
evenp, oddp
    test for even or odd

exp, expt
gcd
incf, decf
    similar to += and -=
lcm
log
mod, rem
sqrt, isqrt
random
logand, logandc1, logandc2, logeqv, logior, lognand, lognor, lognot, logorc1, logorc2, logxor
    bitwise operations on integers
Characters

char=, char/=, char<, char>, char<=, char>=, char-equal,
char-not-equal, char-lessp, char-greaterp, char-not-greaterp,
char-not-lessp

  character comparison operations

char-code

  convert character to integer

code-char

  convert integer to character
Conses

null
null object => boolean
true if the nil symbol

cons
cons object-1 object-2 => cons
make a cons cell with the specified car and cdr

consp
consp object => generalized-boolean
return true if object id a cons cell

atom
atom object => generalized-boolean
true if object is not a cons cell

rplaca, rplacd
rplaca cons object => cons
rplacd cons object => cons
replace the car or cdr of the cons with object and return the cons cell

car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar, caddr, cdaar, cdadr, cdddr, caaaar, caaadr, caadar, caaddr, cadaar, cadaadr, caddar, cadddr, cdbaar, cdaadr, cdaaddr, cddadar, cddadr, cdddar, cddddd

cadaadr is same as (car (cdr (car (cdr x)))) for up to 4 a's and d's

list, list*
list &rest objects => list
list* &rest objects+ => result
return a list of the arguments
for list* the last argument is the cdr of the last cons cell
list-length
list-length list => length
   length of list or nil if a circular list

listp
listp object => generalized-boolean
   return true if a cons cell or nil

push
push item place => new-place-value
   replaces the value at place with (cons item place) and returns this
   new cons cell

pop
pop place => element
   returns car of place and puts (cdr place) in place

append
append &rest lists => result
   returns a list that is the concatenation of all of the constituent
   lists

mapc, mapcar, mapcan, mapl, maplist, mapcon
assoc, assoc-if, assoc-if-not
Arrays

make-array
make-array dimensions &key element-type initial-element
initial-contents adjustable fill-pointer displaced-to
displaced-index-offset
=> new-array

makes a new array with the specified dimensions and keyed
arguments

aref
aref array &rest subscripts => element

Referencing an array element
(aref a 1 2)

Setting an array element
(setf (aref a 1 2) 6)
Strings

string
string x => string

returns a string representation of x
   If x is a string, it is returned.
   If x is a symbol, its name is returned.
   If x is a character, then a string containing that one character is returned.

string might perform additional, implementation-defined conversions
string=, string/=, string<, string>, string<=, string>=, string-equal, string-not-equal, string-lessp, string-greaterp, string-not-greaterp, string-not-lessp

string= string1 string2 &key start1 end1 start2 end2 => generalized-boolean
string/= string1 string2 &key start1 end1 start2 end2 => mismatch-index
string< string1 string2 &key start1 end1 start2 end2 => mismatch-index
string> string1 string2 &key start1 end1 start2 end2 => mismatch-index
string<= string1 string2 &key start1 end1 start2 end2 => mismatch-index
string>= string1 string2 &key start1 end1 start2 end2 => mismatch-index
string-equal string1 string2 &key start1 end1 start2 end2 => generalized-boolean
string-not-equal string1 string2 &key start1 end1 start2 end2 => mismatch-index
string-lessp string1 string2 &key start1 end1 start2 end2 => mismatch-index
string-greaterp string1 string2 &key start1 end1 start2 end2 => mismatch-index
string-not-greaterp string1 string2 &key start1 end1 start2 end2 => mismatch-index
string-not-lessp string1 string2 &key start1 end1 start2 end2 => mismatch-index

returns index of where comparison decided relationship
string-equal, string-lessp, string-greaterp, string-not-greaterp, string-not-lessp are not case sensitive
Sequences

A sequence is an ordered collection of elements, implemented as either a vector or a list. There are several operations that operate on sequences:

- length
- reverse, nreverse
- sort, stable-sort
- find, find-if, find-if-not
- position, position-if, position-if-not
- search
- substitute, substitute-if, substitute-if-not, nsubstitute, nsubstitute-if, nsubstitute-if-not
- concatenate
- merge
- remove, remove-if, remove-if-not, delete, delete-if, delete-if-not
- remove-duplicates, delete-duplicates
Hash Tables

make-hash-table
make-hash-table &key test size rehash-size rehash-threshold => hash-table

makes a new hash table with test as the equality test
  test must be one of the functions eq, eql, equal, or equalp. The
default is eql.

gethash
gethash key hash-table &optional default => value, present-p

  returns value corresponding to key in hash-table or default if not
  present (or nil if default not supplied)

remhash
remhash key hash-table => generalized-boolean

  removes hash-table entry having key and returns true if there was
  such an entry or false otherwise
Filenames

There are several functions dealing with path-names which are general file names
Streams

Files
There are several functions dealing with files

**read-byte**
read-byte stream &optional eof-error-p eof-value => byte
read the next byte from the stream

**write-byte**
write-byte byte stream => byte
write the byte to the stream

**read-char**
read-char &optional input-stream eof-error-p eof-value recursive-p => char

**terpri, fresh-line**
terpri &optional output-stream => nil
fresh-line &optional output-stream => generalized-boolean
writes a newline to the stream
fresh-line only writes a newline if not at the beginning of a line

**write-char**
write-char character &optional output-stream => character
writes an output character to stream

**read-line**
read-line &optional input-stream eof-error-p eof-value recursive-p
returns a string containing the next line of the stream
open
open filespec &key direction element-type if-exists if-does-not-exist external-format

returns an open file stream

with-open-file

with-open-file (stream filespec options*) declaration* form*

opens a file stream with open and binds it to stream
then executes form*
then closes the file

close
close stream &key abort => result

closes the stream
*debug-io*, *error-output*, *query-io*, *standard-input*,
*standard-output*, *trace-output*, *terminal-io*

These are special variables bound to various streams