Practical Common Lisp - Distilled

An aide-mémoire for readers of “Practical Common Lisp” by Peter Seibel.

http://gigamonkeys.com/book/

This document attempts to summarise the core facilities of Common Lisp as presented by the technical chapters of “Practical Common Lisp”. I hope the reader will find these notes useful while working through the practical chapters and perhaps beyond.

Distilled from “Practical Common Lisp” by Andrew Walrond
Comments and suggestions welcomed: andrew@walrond.org
This version released on the 8th October 2009

SYNTAX and SEMANTICS

REPL - Read-Evaluate-Print Loop

Reader (R of REPL):

The Reader converts strings of characters into s-expressions.
The basic elements of s-expressions are lists and atoms.
Lists are whitespace separated s-expressions, delimited by parentheses.
Atoms are everything else.
Comments start with a semicolon and extend to the end of the line.

Atoms:

```
123             ; An integer.
+43             ; Another integer.
-42             ; And another.
3/7             ; The ratio 3 sevenths.
-2/8            ; Negative one quarter.
1.0             ; A default precision floating point number.
1.0e-4          ; Another default precision floating point number.
1.0d0           ; A double precision floating point number.
"foo"           ; A String literal.
"\"foo\\bar\""  ; Another string. Only \ and " need to be escaped.
foo             ; A name.
*db*            ; Another name.
NIL             ; Canonical false.
()              ; A synonym for NIL. NIL is the only object that's both an atom and a list.
```

Ratios are converted and stored in their 'simplified' form: 2/8 = 1/4 and 284/2 = 142.
Names cannot contain any of ()"`',:;\| but can contain digits as long as the whole cannot be interpreted as a number.
All unescaped characters in a name are converted to upper-case.
Names are represented by symbol objects; the reader has no idea how a name is going to be used but creates a symbol object for each new name it encounters and enters it into a table. This is referred to as the 'interning' of symbols. There is a separate symbol table for each PACKAGE.

Certain naming conventions exists in lisp:

Names are hyphenated:	hello-world
Global (dynamic) variables are delimited by asterisks:	*database*
Constants are delimited by plus signs:	+a-constant+

S-expressions are arbitrary trees of s-expressions:

```
123
("foo" 1 2)
(foo 1 2)
(foo 1 (+ 2 3))
(defun hello-world () (format t "hello, world"))
```

Evaluator (E of REPL):

The Evaluator defines the syntax of allowed lisp forms, evaluates the form and returns a 'value'.
Valid lisp forms are:

- Any atom (any non-list or the empty list)
- Any list which has a symbol as its first element

A symbol evaluated as a form is considered the name of a variable and evaluates to the current value of the variable.
All other atoms; numbers,strings etc. are self-evaluating objects. They evaluate to themselves. It's also possible for symbols to be self-evaluating in the sense that the variable they name can be assigned the value of the symbol itself. Two important self-evaluating variables are these constants

```
T       - canonical true
NIL     - canonical false
```

Another class of self-evaluating symbols are keyword symbols, which start with a colon. When the reader interns a keyword, it defines a constant variable with the name and with the symbol as the value. Examples:

```
:min-words
:max-words
```

All legal list forms start with a symbol, but there are three different types of evaluation depending on whether the symbol names a function, a macro or a special operator. If the symbol hasn't been defined yet, it is assumed to be a function name.

Function call evaluation

Evaluation rule: Evaluate the remaining elements of the list as Lisp forms and pass the resulting values to the named function. I.e.

(function-name argument*)

Special operators evaluation

Not all operations can be defined as function calls. For example:

(if x (format t "yes") (format t "no"))

should only evaluate one of the last two forms, which breaks the function call evaluation rule above. Lisp defines 25 special operators with unique evaluation rules to do things that function calls cannot. Examples:

IF - see above
QUOTE - takes an s-expression as an argument and returns it unevaluated
(quote (+ 1 2)) => (+ 1 2)
'(+ 1 2) - syntactic sugar for (quote (+ 1 2))
LET - create a new variable binding

Macro evaluation

The set of special operators is fixed by the language standard, but macros give users a way to extend lisps syntax.

Evaluation rule: 1st) the remaining elements of the macro form are passed unevaluated to the macro function.
2nd) the form returned by the macro function, the 'expansion', is evaluated according to the normal rules.

The first part is done during compilation, the second part at runtime.
Since the elements of the macro form are not evaluated before being passed to the macro form, they do not need to be valid lisp forms.

Truth and Equality

The symbol NIL (the Reader also interprets the empty list () as the symbol NIL) is the only *false* value.
NIL is a self evaluating symbol; it is the name of a constant variable with the symbol NIL as its value, so

NIL === 'NIL === () === '()

Everything else is true, although the self-evaluating symbol T is defined to be the canonical true value.

The four "generic" equality functions, that can be passed any two lisp objects and will return T if they are equivalent and NIL otherwise, are, in order of discrimination:

EQ
EQL
EQUAL
EQUALP

EQ - Tests for "object identity". Two objects are EQ if they are identical, unless they are numbers or characters. Use with caution:

(eq 3 3) => indeterminate
(eq x x) => indeterminate

EQL - same as EQ except that two objects of the same class representing the same numeric or character value are guaranteed to be equivalent.

(eql 3 3) => true
(eql 1 1.0) => false (the integer 1 and the float 1.0 are different instances of different classes)
(eql "hello" "hello") => false

EQUAL - Loosens the discrimination of EQL to consider lists equivalent if they have the same structure and contents, recursively, according to EQUAL. Same for bit vectors and pathnames. For all other cases, uses EQL.

(equal "hello" "hello") => true
(equal "HELLO" "hello") => false

EQUALP - Like EQUAL but also compares strings and characters case-insensitively. Numbers are also equal if they represent the same mathematical value, so

(equalp 1 1.0) => true
(equal "hello" "hello") => true
(equal "HELLO" "hello") => true

FUNCTIONS

Functions are defined using the DEFUN macro:

```
(defun name (parameter*) "optional documentation string" body-form*)
```

The parameter list names the variables that will be used to hold arguments passed to the function.
The documentation string can be retrieved by the DOCUMENTATION function.
The body of a defun consists of any number of lisp forms. They are evaluated in order and the value of the last expression evaluated is returned as the value of the function.

Example:

```
(defun verbose-sum (x y)
  "sum any two numbers after printing a message"
  (format t "Summing ~d and ~d.~%" x y)
  (+ x y))
```

When a parameter list is a simple list of variable names, these are 'required parameters'. One parameter must be supplied for each required parameter.
Optional parameters can also be specified (default values are NIL if not specified):

```
(defun foo (a b &optional c d) (list a b c d))
```

Optional parameters with default values:

```
(defun foo (a &optional (b 10)) (list a b))
```

The default-value expression can refer to parameters that occur earlier in the expression:

```
(defun make-rect (width &optional (height (* width 2)) ...)
```

Occasionally it's useful to know whether the value of an optional argument was supplied or is the default value:

```
(defun foo (a b &optional (c 3 c-supplied-p)) ...)
```

The *-supplied-p (conventional name) is bound to true if the argument was supplied or NIL otherwise.
You can also except a variable number of arguments:

```
(defun + (&rest numbers) ...)
(defun format (stream &rest values) ...)
```

Keyword parameters allow the caller to specify which values go with which parameters:

```
(defun foo (&key a b c) (list a b c))
```

then

```
(foo :c 3 :a 1) => (1 NIL 3)
```

Keyword parameters can also supply a default value form (referring to parameters that appear earlier in the list) and the name of a -supplied-p variable:

```
(defun foo (&key (a 0) (b 0 b-supplied-p) (c (+ a b))) (list a b c b-supplied-p)
```

Combinations are possible, but they must be defined in required, &optional, &rest, &key order.
Combining &optional and &key parameters should be avoided:

```
(defun foo ( x &optional y &key z) (list x y z))
```

then

```
(foo 1 2 :z 3)          => (1 2 3)  OK
(foo 1)                 => (1 nil nil)          OK
(foo 1 :z 3)            => ERROR - keyword :z is taken to fill optional y leaving an unsatisfied :z parameter
```

Combining &rest and &key is safe but assignments overlap:

```
(defun foo (&rest rest &key a b c) (list rest a b c))
```

then

```
(foo :a 1 :b 2 :c 3) => ((:A 1 :B 2 :C 3) 1 2 3)
```

You can use the RETURN-FROM special operator to immediately return any value from the function, but you must provide the name of the function as the first parameter. (RETURN-FROM is actually associated with the BLOCK special operator used by the DEFUN macro to wrap the function body in a block of the same name):

```
(defun foo (n)
  (dotimes (i 10)
            (dotimes (j 10)
              (when (> (*i j) n)
                return-from foo (list i j)))))
```

Higher Order Functions (aka Functions as Data)

Functions are just another kind of object.
DEFUN creates a new function object and gives it a name.
LAMBDA expressions create a function without giving it a name.
The FUNCTION special operator provides the mechanism for getting at the function object:

```
(defun foo (x) (* 2 x))          => FOO
(function foo)                   => #<Interpreted Function FOO>
#'foo                            => #<Interpreted Function FOO>  (#' is syntactic sugar for FUNCTION)
```

There are two functions for invoking a function through a function object; FUNCALL and APPLY.
Use FUNCALL if you know the number of parameters you're going to pass to the function when you write the code:

```
(funcall #'foo 1 2 3)  is equivalent to  (foo 1 2 3)
```

Use APPLY when the argument list is only known at runtime. It takes a list instead of individual arguments:

```
(apply #'foo (list 1 2 3))  is equivalent to  (foo 1 2 3)
```

APPLY can also accept loose arguments as long as the last argument is a list:

```
(apply #'foo 1 (list 2 3))
```

APPLY can deal with &optional, &rest and &key arguments; the combination of loose arguments and the final list just has to be a valid argument list for the function object.

You can create "anonymous" functions using LAMBDA expressions:

```
(lambda (parameter*) body-form*)
```

Think of a lambda expression as a special kind of function name which describes what the function does. Then you can understand why we can do:

```
((lambda (x y) (+ x y)) 2 3)                 => 5
(funcall #'(lambda (x y) (+ x y)) 2 3)       => 5
```

Anonymous functions are useful when you want to pass a small function 'inline' to another function:

```
(plot #'(lambda (x) (* 2 x)) 0 10 1)
```

The other important use of lambda expressions is in making closures.

VARIABLES

Common lisp supports two types of variables; Lexical and Dynamic.
All variables are references to objects, aka 'bindings'.
Assigning a new value to a variable simply references a new object and has no effect on the original object. [That said, if the variable holds a reference to an object which is a mutable instance of a class, the reference can be used to modify the object and the modification will be seen by all other references to the same object]

A function declaration defines variables which be used to hold its arguments. Each time it is called, new bindings are created to hold the arguments. The scope of any binding is delimited by the form that introduced it, known as the 'binding form'. A function declaration is a binding form:

```
(defun foo (x y z) (+ x y z))
```

Another binding form that introduces new variables is the LET form:

```
(let (variable*) body-form*)
```

where each variable is an initialization form; either a variable name or a list containing the variable name and an initial value form.

```
(let ((x 10) (y 20) z) ...)
```

A variant of LET is LET* which allows the initial value forms for each variable to refer to variables introduced earlier in the list:

```
(let* ((x 10) (y (+ x 10))) ...)
```

In nested binding forms that introduce variables of the same name, the innermost variable shadows the outer bindings during their scope:

```
(defun foo (x)
  ...                       x = argument
  (let ((x 2))
     ...                    x = 2
     (let ((x 3))
          ...)              x = 3
     ...)                   x = 2
  ...)                      x = argument
```

Closures

Created when an anonymous function references a lexically variable from an enclosing scope. The anonymous function 'closes over' the binding created by let, which will hang around for as long as someone holds a reference to the anonymous function. Since it is the binding that is captured, new value assignments will persist between calls to the closure. For example, we can capture this closure in a global variable like this:

```
(defparameter *fn* (let ((count 0)) #'(lambda () (setf count (1+ count)))))
```

Now each invocation will increment count:

```
(funcall *fn*)      => 1
(funcall *fn*)      => 2
(funcall *fn*)      => 3
```

A single closure can close over many bindings. Or multiple closures can capture the same binding. Example:

```
(let ((count 0))
  (list
   #'(lambda () (incf count))
   #'(lambda () (decf count))
   #'(lambda () count)))
```

Dynamic (aka Special, Global) Variables

Dynamic variable names are conventionally (and for good reason) delimited by asterisks, and can be referenced from anywhere. There are two ways to create dynamic variables, DEFPARAMETER and DEFVAR.

DEFPARAMETER must be provided with an initial value form and it always assigns the initial value to the named variable:

```
(defparameter *tolerance* 0.001)
```

DEFVAR on the other hand takes an optional initial value form and only assigns the initial value if the variable is undefined. Where the variable is undefined and no initial value is supplied, the variable will be defined but without a value and is known as 'unbound'.

```
(defvar *database*)
(defvar *backupdb* NIL)
```

DEFVAR should be used when you will want to preserve the existing contents of a variable even if you make a change to and re-evaluate the source code that uses it.

When you bind a dynamic variable, for example with a function parameter or LET, the binding replaces the global binding for the duration of the binding form. Unlike lexical bindings which can only be referenced within the lexical scope of the binding form, a dynamic binding can be referenced by any code that's invoked during the execution of the binding form. For example:

```
(let ((*standard-output* *log-file*)) (stuff))
```

Any code that runs as a result of the call to stuff will write to the *log-file* stream, but the original *standard-output* is restored on exit from the LET binding form.

Constants

All constants are global and defined with DEFCONSTANT. Constant names are conventionally delimited by plus signs:

```
(defconstant name initial-value-form [documentation-string])
```

Although you can redefine a constant by re-evaluating a DEFCONSTANT form with a different value, what happens next isn't well defined. Most implementations will have inlined the constant so you will probably have to re-evaluate any code that refers to it. Only define constants that really are constant and won't need to be changed.

Assignment

A new value can be assigned to a binding with the SETF macro, which returns the last assigned value

```
(setf place value [place value ...])
(setf x 10)                     => 10
(setf x 10 y 20)                => 20
(setf x (setf y (random 10)))   x=y
```

Assigning a new value with setf has no effect on any other bindings of the variable or the original value that was stored prior to assignment. So

```
(defun foo(x) (setf x 10))
```

has no effect on any value outside of FOO.

SETF is also used to assign values to places inside composite or user defined data structures. Examples:

```
(setf (aref a 0) 10)            Array
(setf (gethash 'key hash) 10)   Hash table
(setf (field o) 10)             Slot named 'field' in user defined object o
```

While SETF can be used for any assignment, some common types of assignment are characterised by 'modify macros':

```
(incf x)        === (setf x (+ x 1))
(decf x)        === (setf x (- x 1))
(incf x 10)     === (setf x (+ x 10))
(rotatef a b)   Swaps the values of the two variables and returns NIL.
(shiftf a b 10) Shifts values to the left. The original value of the first argument is returned.
```

All modify macros are defined such that the place expression is evaluated only once.

Standard Control Construct Macros

PROGN - executes any number of forms in order and returns the value of the last form.
Since, for example, the IF special operator

```
(if condition then-form else-form)
```

takes only a single then and else form, PROGN is very useful:

```
(if (spam-p current-message)
   (progn
         (file-in-spam-folder current-message)
         (update-spam-database current-message)))
```

Common lisp defines WHEN and UNLESS macros to streamline common IF/PROGN expressions:

```
(when condition form*)
(unless condition form*)
```

COND -streamlines multi-branch IF expressions:

```
(cond
      (test-1 form*)
      ...
      (test-N Form*))
```

The boolean logic operators AND, OR and NOT are useful when composing IF, WHEN, UNLESS and COND expressions:

(not form)	Inverts the truth value, returning T or NIL
(and form*)	Evaluates sub-forms, left to right, until a NIL value is found. Returns NIL or the last form value
(or form*)	Evaluates sub-forms, left to right, until a non-NIL value if found. Returns the value or NIL.

DOLIST - Loops across the items of a list, executing the body with a variable holding the list entry

```
(dolist (var list-form) body-form*)
(dolist (x '(1 2 3)) (print x))
```

You can break out of a DOLIST loop before the end of the list with RETURN:

```
(dolist (x '(1 2 3)) (print x) (if (evenp x) (return)))
```

DOTIMES - High level construct for counting loops

```
(dotimes (var count-form) body-form*)
```

The count form must evaluate to an integer. The loop is evaluated from 0 to one less than the count value:

```
(dotimes (i 4) (print i))          Prints 0,1,2,3. Returns NIL.
```

As with DOLIST, you can break out early with return.

DO - More powerful Looping construct

```
(do (variable-definition*)
   (end-test-form result-form*)
  statement*)
```

Each variable-definition looks like this:

```
(var [init-form [step-form]])
```

or without init or step forms can be shortened to

```
var
```

The init-form, if present, is evaluated at the beginning of the loop and its value assigned to the variable. Otherwise the variable is bound to NIL. The optional step-form is evaluated before subsequent iterations of the loop and its value is assigned to the variable. At each step of the iteration, the step-forms for all the variables are evaluated before any of them are assigned (so they can all use current values of the other variables).

```
(do ((n 0 (1+ n)) (cur 0 next) (next 1 (+ cur next)))
   ((= 10 n) cur)
      (print cur))
```

LOOP - Powerful macro for looping and accumulating over lists, vectors, hash tables and packages.
The simple version is an infinite loop that doesn't bind any variables:

```
(loop body-form*)
```

which will iterate forever unless you break out with return.
The extended version has its own language for expressing looping idioms. Examples:

```
(loop for i from 1 to 10 collecting i)                          => (1 2 3 4 5 6 7 8 9 10)
(loop for x from 1 to 10 summing (expt x 2))                    => 385
(loop for x across "the quick brown fox" counting (find x "aeiou"))  => 5
```

MACROS

The most important thing to remember about macros is that you are writing programs that will be used by the compiler to generate the code that will be executed later at runtime. Macro expansion time is distinct from runtime. Macro arguments are Lisp lists representing the source code, not variable bindings which do not exist until runtime. Nonetheless, macros can use the full power of Lisp to generate their expansions.

Special Backquote syntax

A particularly concise way of writing code that generates lists, the special backquote syntax is a mechanism provided by the Reader and invaluable when writing macros. The backquote inhibits evaluation just like ' or (QUOTE), but any sub-expression preceded by a comma is evaluated. Furthermore, any list expression preceded by ,@ will be 'spliced' into the enclosing list. Examples:

Source	**Reader translates to this**	**Result**
`(a (+ 1 2) c)	(list 'a '(+ 1 2) 'c)	(a (+ 1 2) c)
`(a ,(+ 1 2) c)	(list 'a (+ 1 2) 'c)	(a 3 c)
`(a (list 1 2) c)	(list 'a '(list 1 2) 'c)	(a (list 1 2) c)
`(a ,(list 1 2) c)	(list 'a (list 1 2) 'c)	(a (1 2) c)
`(a ,@(list 1 2) c)	(append (list 'a) (list 1 2) (list 'c))	(a 1 2 c)

Writing Macros

Consider the function FOO

```
(defun foo (x) (when (> x 10) (print 'big)))
```

using the standard WHEN macro which might be defined as

```
(defmacro when (condition &rest body)
  `(if ,condition (progn ,@body)))
```

which will be expanded as

```
(if (> x 10) (progn (print 'big)))
```

The generic DEFMACRO form looks like this

```
(defmacro name (parameter*) "Optional Documentation String" body-form*)
```

Special features of a macro parameter list:

- It is a 'destructuring parameter list'. We can use nested parameter lists to destructure list arguments
- &body can be used as a synonym for &rest in macro parameter lists. (It helps SLIME to display the macro syntax)
- &whole can be used to bind a variable to the whole argument list form. If used, it must be the first parameter in the list. Other parameter types can follow and are unaffected by the &whole parameter

Steps for writing a macro:

1) Write a sample call to the macro, and the code it should expand into
2) Write the code that generates the handwritten expansion from the arguments in the sample call
3) Make sure the abstraction doesn't "leak"

Example: DO-PRIMES will provide a looping construct that iterates over successive prime numbers.
We need a couple of utility functions first:

```
(defun test-prime (number)
  (when (> number 1)
    (loop for fac from 2 to (isqrt number) never (zerop (mod number fac)))))

(defun next-prime (number)
  (loop for n from number when (primep n) return n))
```

Now we can write an example call to the macro:

```
(do-primes (p 0 19) (format t "~d " p))
```

And the expansion we require:

```
(do ((p (next-prime 0) (next-prime (1+ p))))
    ((> p 19))
  (format t "~d " p))
```

Now we can write the macro:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var))))
```

```
  ,@body))
```

Notice the destructuring nested parameter list (var start end) and the use of the special backquote syntax; ` , and ,@. We can check the expansion of our macro with the MACROEXPAND-1 function:

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19) (format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P 19))
          (FORMAT T "~d " P))
T
```

(Or we could use Slimes short-cut). However, we can see our macro works.

Finally, we must check for "leaks". Imagine we use (random 100) instead of 19 for the end form. It will be evaluated twice by the expansion which is not what we want and may well catch out the user. We can fix this leak by modifying the do loop a little:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((ending-value ,end)
              (,var (next-prime ,start) (next-prime (1+ ,var))))
             ((> ,var ending-value))
    ,@body))
```

but this introduces two new leaks. The first is that the end form is now evaluated before the start form, which might again cause unforeseen side effects. We can fix this by swapping them around:

```
(defmacro do-primes ((var start end) &body body)
  `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
       (ending-value ,end))
             ((> ,var ending-value))
    ,@body))
```

The last leak is introduced by our variable ending-value, which might shadow an existing variable using in the code passed to the macro or in the context in which the macro is called. What we need is a symbol that will never be used outside the code generated by the macro. The function GENSYM returns a unique symbol which is not interned, so we can use it to generate a new symbol every time do-primes is expanded.

```
(defmacro do-primes ((var start end) &body body)
  (let ((ending-value-name (gensym)))
    `(do ((,var (next-prime ,start) (next-prime (1+ ,var)))
         (,ending-value-name ,end))
               ((> ,var ending-value))
      ,@body)))
```

Rules of thumb:

1) Try and arrange for sub-forms in the expansion to be evaluated in the same order as in the macro call
2) If possible, only evaluate sub-forms once
3) Use GENSYM to create unique variable names in the expansion

Numbers

Exact integers are arbitrarily large in lisp, rather than being limited by the size of a machine word.
Dividing two integers results in an exact ratio of arbitrary precision.
All ratios are stored in a canonicalised format (6/8 => 3/4, 20/2 => 10, etc.) and printed in reduced form.

Numeric literals:	123 +123 -123 123. 2/3 -2/3 4/6 6/3
Binary:	#b10101 #B11101
Octal:	#o777 #O767
Hexadecimal:	#xff7e #Xe7fc3
Base 2-36:	#36rABC23H

Floating-point subtypes:	short	(s|S)	1s3
	single	(f|F)	1.4f3
	double	(d|D)	0.17d3
	long	(l|L)	12.37L+6
	default	(e|E)	1.0 6.0e-3

Complex numbers are stored internally as a pair of same-type numbers. If specified with distinct types of real and imaginary parts, the lesser is promoted to the larger type.
Examples: #c(2 1) #c(2/3 3/4) #c(1.0 3.0d0) #c(3 -2.0)

Basic math:

(+ 1)=1 (+ 1 2)=3 (+ 1 2 3)=>(+ (+ 1 2) 3)=6 (+ 10 3.0)=13.0 (+ #c(1 2) 6.0)=#c(7.0 2)
(- 3)=-3 (- 5 4)=1
(* 1)=1 (* 2 3 4)=24
(/ 4)=1/4 (/ 10 5)=2 (/ 10 5 2)=1

Converting real (rational or float) numbers to integers:

FLOOR	truncates towards -infinity (returns the largest integer <= the argument)
CEILING	truncates towards +infinity (returns the smallest integer >= the argument)
TRUNCATE	truncates towards zero (FLOOR for +ve arguments, CEILING for -ve)
ROUND	rounds to the nearest integer

Misc numeric functions:

(MOD x y)	=> the modulus of x/y
(REM x y)	=> the remainder of x/y
(1+ x)	=> (+ x 1)
(1- y)	=> (- x 1)

Comparisons:

= compares numbers by mathematical value, ignoring differences in type. EQUALP uses = for numbers.
(= 1 1) => T
(= 1 2/2 1.0 #c(1.0 0.0) #c(1 0) 1.0d0) => T

/= returns true *** only if all its arguments are different values ***
(/= 1 1) => NIL
(/= 1 1.0 2) => NIL *** !!! ***
(/= 1 2 3) => T

These comparisons compare each argument with the one its right
(< 2 3) => T
(< 2 3 2) => NIL
(<= 1 2 2 3) => T
(>= 3 3 2) => T

(max 10 11) => 11
(min -1 2 -3) => -3

ZEROP, MINUSP and PLUSP test whether a single argument is =0, <0 or >0 respectively
EVENP, ODDP test if a single argument is even or odd.

Note: The P suffix is a standard naming convention for predicate functions that test some condition and return a boolean.

Higher math: LOG, EXP, EXPT, SIN, COS, TAN, ASIN, ACOS, ATAN, SINH, COSH, TANH, ASINH, ACOSH, ATANH etc.

Characters

The character type is a distinct object from numbers and implementation specific, but usually unicode.
The read syntax for characters is #\ followed by the desired character. Whitespace and special characters are written by name. Examples:

```
#\x  #\X  #\\  #\Space  #\Tab  #\Newline  #\Linefeed  #\Return  #\Backspace
```

There are two sets of comparators, case-sensitive and case-insensitive. Like their numeric counterparts, they can take one or more arguments:

case-sensitive	case-insensitive
CHAR=	CHAR-EQUAL
CHAR\=	CHAR-NOT-EQUAL
CHAR<	CHAR-LESSP
CHAR>	CHAR-GREATERP
CHAR<=	CHAR-NOT-GREATERP
CHAR>=	CHAR-NOT-LESSP

Strings

Strings are a composite data type, a sequence, a one dimensional array of characters.
Literal strings are written enclosed with double quotes. Backslash will escape any character, although it is only necessary for the double quote and itself.

There are case-sensitive and insensitive comparison functions:

case-sensitive	case-insensitive
STRING=	STRING-EQUAL
STRING\=	STRING-NOT-EQUAL
STRING<	STRING-LESSP
STRING>	STRING-GREATERP
STRING<=	STRING-NOT-GREATERP
STRING>=	STRING-NOT-LESSP

Unlike the numeric and character equivalents, they can only compare two strings because they accept keyword arguments that allow you to restrict the comparison to a sub-string of either or both strings. Example:

```
(string= "foobarbaz" "quuxbarfoo" :start1 3 :end1 6 :start2 4 :end2 7)          => true
```

The comparators that return true when there arguments differ actually return the index in the first string where the mismatch was detected:

```
(string/= "lisp" "lissome")       => 3
(string< "lisp" "lisper")         => 4
```

All the sequence functions described next can be used with strings.

Collections

Vectors

Lisps basic integer indexed collection, vectors can be fixed-size or resizeable
Example fixed size vectors:

```
(vector)                          => #()
(vector 1)                        => #(1)
(vector 1 "abc"                   => #(1 "abc")
(make-array 5)                    => #(NIL NIL NIL NIL NIL)
(make-array 3 :initial-element 1) => #(1 1 1)
```

#(...) is the literal notation syntax used by the lisp Reader and Printer. While it is possible to use the literal notation in your code, the effects of modifying literal objects isn't defined (similar to constants) so always use VECTOR or MAKE-ARRAY for vectors that might change.

MAKE-ARRAY is more general than VECTOR and can be used to create multi-dimensional arrays as well as resizeable vectors.
Vectors maintain a 'fill-pointer' which stores the next position to be filled. To create a fixed size empty vector:

```
(make-array 5 :fill-pointer 0)
```

VECTOR-PUSH can be used to add an element at the current value of the fill-pointer, incrementing the fill pointer but returning the position of the added element, or NIL if the vector was already full.

```
(defvar *v* (make-array 3 : fill-pointer 0))
(vector-push 'a *v*)              => 0
(vector-push 'b *v*)              => 1
(vector-push 'c *v*)              => 2
(vector-push 'd *v*)              => NIL
*v*                               => #(A B C)
```

VECTOR-POP returns the last element, decrementing the fill-pointer. It will raise a condition if the vector is empty.

```
(vector-pop *v*)                  => C
(vector-pop *v*)                  => B
(vector-pop *v*)                  => A
(vector-pop *v*)                  => Error!
```

To make an arbitrarily resizeable vector, we need to pass MAKE-ARRAY another argument:

```
(make-array 5 :fill-pointer 0 :adjustable t)
```

To add elements to a resizeable array, we use VECTOR-PUSH-EXTEND which works just like VECTOR-push except it will automatically expand the array if it is full.

It is also possible to create specialised vectors that are restricted to holding certain types of elements. Specialised vectors may be more compact and efficient than general arrays.
Strings are vectors specialised to hold characters:

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character)
```

and are important enough to have "literal string" reader support so we can type "Hello" rather than #(#\H #\e #\l #\l #\o)

Bit vectors are another specialisation with reader support; Bit vectors can be written like this:

```
#*01010011
```

and come with a substantial library of bitwise functions.

Sequences

Both vectors and lists are concrete examples of the abstract type 'sequence'. All of the following are applicable to both.

```
LENGTH - returns the length of a sequence
ELT    - short for element. References the element at the given integer index
       - elt will signal an error if the index is out of bounds
       - elt is a SETFable place
```

```
(defparameter *x* (vector 1 2 3))
(length *x*)                      => 3
(elt *x* 0)                       => 1
(elt *x* 1)                       => 2
(elt *x* 2)                       => 3
(elt *x* 3)                       => Error!
(setf (elt *x* 0 10)
```

```
*x*                                     => #(10 2 3)
```

Some useful sequence iterating functions are COUNT, FIND, POSITION, REMOVE and SUBSTITUTE

```
(defparameter *x* #(1 2 1 2 3 1 2 3 4))   => *X*
(count 1 *x*)                             => 3
(remove 1 *x*)                            => #(2 2 3 2 3 4)
(substitute 10 1 *x* )                    => #(10 2 10 2 3 10 2 3 4)
(find 3 *x*)                              => 3
(position 4 *x*)                          => 8
*x*                                       => #(1 2 1 2 3 1 2 3 4)
```

These sequence functions take various extra keyword arguments; :test, :key, :start, :end :from-end - see the hyperspec for more info.
For each of these functions, Lisp also provides two higher-order function variants with -IF and -NOT-IF appended. For example:

```
(count-if #'evenp #(1 2 3 4 5))           => 2
(count-if-not #'evenp #(1 2 3 4 5))       => 3
```

There are loads more sequence functions; See the Hyperspec.

Hash Tables

With no arguments, MAKE-HASH-TABLE makes a hash table that considers two keys equivalent if they are EQL. This is a good default unless you want to use strings as keys in which case you will want to use EQUAL or EQUALP (case insensitive). MAKE-HASH-TABLE's :test argument is actually limited to the four equality predicates EQ, EQL, EQUAL and EQUALP since it needs to have a hashing algorithm for each predicate. The GETHASH function provides access to the elements of the hash table, returning the value or NIL if unavailable:

```
(defparameter *h* (make-hash-table))      => *H*
(gethash 'foo *h*)                        => NIL, NIL
(setf (gethash 'foo *h*) 'quux)           => QUUX
(gethash 'foo *h*)                        => QUUX, T
```

Since GETHASH returns NIL if the key isn't present in the hash table, it would seem that there is no way to tell the difference between a missing key and a key with a value of NIL. But GETHASH actually returns two values, the second being a boolean indicating whether the key is present in the hash table.

```
(setf (gethash 'bar *h*) NIL)             => QUUX
(gethash 'bar *h*)                        => NIL, T
```

We use the MULTIPLE-VALUE-BIND macro to get access to GET-HASHs extra return value:

```
(multiple-bind-value (value present) (gethash key hashtable) ...)
```

See also REMHASH and CLRHASH and the iteration function MAPHASH in the Hyperspec.

Cons Cells and Lists

A cons cell is a pair of values named after the CONS function used to create them. If the second value is NIL or another cons cell, it becomes a list:

```
(cons 1 2)                  => (1 . 2)
(cons 1 NIL)                => (1)
(cons 1 (cons 2 NIL))       => (1 2)
(cons 1 (cons 2 3))         => (1 2 . 3)
```

The first value is known as the 'car' and the second 'cdr' after the CAR and CDR functions used to access them. (These names were apparently mnemonic on an IBM 704 way back when). CAR and CDR are SETFable places.

```
(defparameter *cons* (cons 1 2))    => *CONS*
*cons*                              => (1 . 2)
(setf (car *cons*) 10)              => 10
*cons*                              => (10 . 2)
(setf (cdr *cons*) 20)              => 20
*cons*                              => (10 . 20)
```

When thinking in terms of lists, FIRST and REST are synonyms for CAR and CDR:

```
(defparameter *list* (list 1 2 3 4))    => *LIST*
(first *list*)                          => 1
(rest *list*)                           => (2 3 4)
(first (rest *list*))                   => 2
```

Be aware of the two types of sequence functions:

- functional programming type functions which will not alter their arguments, but whose result may share cons cells with them. E.g. APPEND, REVERSE.
- Non-functional programming functions which may alter their arguments in forming their result. E.g. NCONS, NREVERSE.

Since the result of even the non-destructive functional programming type functions may share cons cells, later modification of either the arguments or the result may corrupt the other.

Trees

Lists can have other lists as elements so can represent trees of arbitrary depth and complexity. Lisp code itself is a good example. There are standard functions that deal with cons trees. See for example the COPY-LIST and COPY-TREE functions, SUBST/NSUBST et al.

Sets

Similarly lists can be considered as sets and there are several functions for performing set operations on lists. See ADJOIN, PUSHNEW, MEMBER, MEMBER-IF, MEMBER-IF-NOT etc.

Association Lists (alist)

An alist maps keys to values and also supports reverse lookups. It is constructed as a list where the car values are cons cells containing the key/value pairs. Values can also be temporarily shadowed. Set ASSOC etc.

Property Lists (plist)

A plist is just a regular list with the keys and values as alternating values. See GETF, REMF, GET-PROPERTIES etc.

All lisp symbols have an associated plist that can be used to store information about that symbol. The plist can be obtained with SYMBOL-PLIST. The function GET is a useful shorthand when you are only interested in a single property:

```
(get 'symbol 'key) === (getf (symbol-plist 'symbol) 'key)
```

GET is SETFable. See also REMPROP.

DESTRUCTURING-BIND

This macro provides a way to destructure arbitrary lists, similar to the way macro parameter lists can take apart their argument list.

```
(destructuring-bind (parameter*) list body-form*)
```

The parameter list can contain any of the parameter types accepted by macros; &whole, &optional, &rest, &key etc. and also accepts nested destructuring parameter lists, again just like macros.

Files and File I/O

See OPEN, CLOSE.

By default, OPEN returns a character stream which translates the underlying bytes according to the character encoding scheme. Character stream functions:

READ-CHAR	
READ-LINE	
READ	Unique to Lisp. The 'R' in REPL
WRITE-CHAR	
WRITE-LINE	
WRITE-STRING	
TERPRI	
FRESH-LINE	
PRINT	
PRIN1	
PPRINT	
PRINC	
FORMAT	

To read raw bytes, you must pass an extra keyword argument to open:

:element-type '(unsigned-byte 8)

Then you can use

READ-BYTE
WRITE-BYTE

Bulk reads

The sequence functions work with both character and binary streams and are typical much more efficient since they will use the underlying operating systems block i/o functions.

READ-SEQUENCE
WRITE-SEQUENCE

Closing files

Lisp provides the macro WITH-OPEN-FILE, built on top of UNWIND-PROTECT, to automatically ensure that files are closed when they go out of scope.

(with-open-file (stream-var open-argument*) body-form*)

File System Functions

PROBE-FILE	Test whether a file exists
ENSURE-DIRECTORIES-EXIST	Creates missing subdirectories as necessary
DIRECTORY	List a directory
DELETE-FILE	
RENAME-FILE	
FILE-WRITE-DATE	
FILE-AUTHOR	
FILE-LENGTH	
FILE-POSITION	

See also

STRING-STREAM, MAKE-STRING-INPUT-STREAM, MAKE-STRING-OUTPUT-STREAM, GET-OUTPUT-STREAM-STRING
WITH-INPUT-FROM-STRING, WITH-OUTPUT-TO-STRING
BROADCAST-STREAM, MAKE-BRADCAST-STREAM
CONCATENATED-STREAM, MAKE-CONCATENATED-STREAM
TWO-WAY-STREAM, MAKE-TWO-WAY-STREAM
ECHO-STREAM, MAKE-ECHO-STREAM

Broadcast stream forms a data black hole when called with no arguments.

CLOS - Generic Functions

In Common Lisp, all objects are instances of a class.
Classes are arranged in a hierarchy with a single root, T.
Common Lisp supports multiple-inheritance.

A generic function defines the name and parameter list of an abstract operation without an implementation:

```
(defgeneric withdraw (account amount) (:documentation "Withdraw an amount from the account"))
```

The actual implementation of a generic function is provided by methods. Each method provides an implementation of the generic function for particular classes of arguments. The methods don't belong to a particular class; they belong to the generic function which is responsible for determining what method or methods to run in response to a particular invocation.

Methods indicate what kinds of arguments they can handle by specialising the required parameters defined by the generic function. An individual parameter can be specialised in two ways:

- By specifying a class for an argument, the method can be applicable for any instance of that class or its subclasses.
- By using a so called EQL specialiser to specify a particular _object_ to which the method applies.

If we have a bank-account class we could define:

```
(defmethod withdraw ((account bank-account) amount)
  (when (< (balance account) amount)
    (error "Account overdrawn"))
  (decf (balance account) amount))
```

The first parameter is specialised by replacing it with a two element list containing the name and the specialiser. The specialiser is either the name of a class or an EQL specialiser (see below). In this example, the method is applicable as long as the first parameter is an instance or subclass of bank-account. The second parameter, amount, is implicitly specialised on T so will match any type of argument.

Now suppose we have sub-classes checking-account and savings-account and a method get-savings account which, when passed a checking account, will return the associated savings account. Then,

```
(defmethod withdraw ((account checking-account) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
          (withdraw (get-savings-account account) overdraft)
          (incf (balance account) overdraft)))
  (call-next-method))
```

The CALL-NEXT-METHOD function is part of the generic function machinery of 'Method Combination' and in this case would call the first, less specific but still applicable, withdraw method. We are not obliged to call CALL-NEXT-METHOD but if we don't then we are responsible for completely implementing the behaviour of the generic function. When called with no arguments, the next method is invoked with same arguments as were passed to this function, but we can substitute our own arguments if required.

As an example of EQL specialisation, imagine we have a corrupt bank president who has set-up a special withdraw method applicable only to his personal account:

```
(defmethod withdraw ((account (eql *account-of-bank-president*)) amount)
  (let ((overdraft (- amount (balance account))))
    (when (plusp overdraft)
          (incf (balance account) (embezzle *bank* overdraft))))
  (call-next-method))
```

Cautionary Note: the method is specialised on the value of *account-of-bank-president* once at the time the method is defined. Changing the variable later will NOT change the method.

Method Combination

CALL-NEXT-METHOD is given meaning inside a method by the generic functions method combination machinery each time the generic function is invoked. The effective method is built in three steps:

1) The generic function builds a list of applicable methods based on the arguments passed.
2) The list of applicable methods is sorted according to the specificity of their parameter specialisers.
3) Methods are taken in order from the sorted list and their code is combined to produce the effective method.

To order two applicable methods, their parameter specialisers are compared from left to right and the first specialiser that's different between the two methods determines their ordering, with the more specific specialiser coming first. (Multiple inheritance complicates this a little; see later). The EQL specialiser is always more specific than a class specialiser.

The Standard Method Combination

By default, generic functions use 'standard method combination' to produce the effective method from the sorted list of applicable methods. The most specific method runs first, and each method can pass control to the next most specific method via CALL-NEXT-METHOD.
However, as well as the 'primary' methods we have already seen, the standard method combination also supports three kinds of auxiliary methods:

```
:before
:after
:around
```

Auxiliary methods are written with DEFMETHOD just like the primary methods but with a method qualifier. E.g.

```
(defmethod withdraw :before ((account bank-account) amount) ...)
```

Each type of auxiliary method is combined into the effective method in a different way.

- All the :before methods run before the most specific primary method, in most-specific-first order, without using CALL-NEXT-METHOD.
- All the :after methods run after the primary methods in most-specific-LAST order (i.e. reverse order), without using CALL-NEXT-METHOD.
- The :around methods are combined much like primary methods except they are run "around" all the other methods. The most specific :around method is run before anything else, and CALL-NEXT-METHOD will lead to the next most specific :around method or, in the case of the least specific :around method, to the complex of :before, primary and after methods.

Typically :around methods are used to establish some dynamic context or to establish an error handler to be used by the rest of the methods.

The Simple Method Combinations

There are nine other built in method combinations and they all follow the same pattern; instead of invoking the most specific primary method and letting it invoke less specific primary methods via CALL-NEXT-METHOD, the simple method combinations produce an effective method that contains the code of ALL the primary methods, one after another, all wrapped in a call to the function, macro or special operator that gives the method combination its name:

+, AND, OR, LIST, APPEND, NCONC, MIN, MAX, PROGN.

The simple combinations do not support :before or :after auxiliary methods, but do support :around methods which are combined just as in the standard method combination.

To define a generic function that uses a particular method combination, include a :method-combination option in the form:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run")
  (:method-combination +))
```

By default all these method combinations combine the primary methods in most-specific-first order. However, you can reverse this order by including the keyword :most-specific-last after the name of the method combination:

```
(defgeneric priority (job)
  (:documentation "Return the priority at which the job should be run")
  (:method-combination + :most-specific-last))
```

The :most-specific-lst options doesn't affect the order of :around methods.

The primary methods of a generic function that uses one of these combinations must be qualified with the name of the method combination:

```
(defmethod priority + ((job express-job)) 10)
```

User defined Method Combinations

Lookup DEFINE-METHOD-COMBINATION in the Hyperspec.

CLOS - Classes

```
(defclass name (direct-superclass-name*) (slot-specifier*))
```

Class names live in a separate namespace from both functions and variables.
The direct-superclass-names specify the classes of which the new class is a subclass. If no superclasses are specified, the new class will directly subclass STANDARD-OBJECT. Superclasses must also be user-defined classes and therefore ultimately descended from STANDARD-OBJECT. You cannot subclass the built-in classes, although you can extend their functionality with new methods.

```
(defclass bank-account () ...)
(defclass checking-account (bank-account) ...)
```

The bulk of the defclass form is used to specify the slots used to hold values. A class also inherits slots from its superclasses. At the minimum, a slot specifier can be just a name:

```
(defclass bank-account ()
  (customer-name
   balance))
```

With this definition, we can create an instance of the class with MAKE-INSTANCE:

```
(make-instance 'bank-account)
```

Using this definition of the bank-account class, new objects will be created with their slots unbound and attempts to access them will signal an error, so we must set them first:

```
(defparameter *account* (make-instance 'bank-account))
(setf (slot-value *account* 'customer-name) "John Doe")
(setf (slot-value *account* 'balance) 1000)
(slot-value *account* 'balance)                                    => 1000
```

It is possible to create objects with their slots already initialised in three ways:

1) Use the :initarg option in the slot specifier to name a keyword parameter that can be passed to MAKE-INSTANCE and whose value will be stored in the slot.
2) Use the :initform option in the slot specifier to specify a Lisp expression that will be used to compute the slot value if no :initarg argument is passed to MAKE-INSTANCE.
3) For complete control over initialisation, define a method on the generic function INITIALIZE-INSTANCE which is called by MAKE-INSTANCE.

```
(defclass bank-account ()
  ((customer-name
          :initarg :customer-name
          :initform (error "You must supply a customer name"))
         (balance
          :initarg :balance
          :initform 0)
         (account-number
          :init-form (incf *account-numbers*))
         account-type))

(defparameter *account* (make-instance 'bank-account :customer-name "John Doe"))
(slot-value *account* 'customer-name)                              => "John Doe"
(slot-value *account* 'balance)                                    => 0
```

An initform has no access to the object being initialised, so it can't initialise a slot based on the value of another. For that we need to define a method on the generic function INITIALIZE-INSTANCE:

The primary method on INITIALIZE-INSTANCE specialised on STANDARD-OBJECT takes care of initialising slots based on their :initarg and :initform options. Since you don't want to disturb that, the most common way to add custom initialisation code is to define and :after method specialised on your class.

```
(defmethod initialize-instance :after ((account bank-account) &key)
  (let ((balance (slot-value account 'balance)))
          (setf (slot-value account 'account-type)
                  (cond
                    ((>= balance 100000) :gold)
                    ((>= balance 50000) :silver)
                    (t : bronze)))))
```

The &key in the parameter list is required to match the generic function which includes it to allow individual methods to define their own keyword parameters, but doesn't require any particular ones.

```
(defmethod initialize-instance :after ((account bank-account) &key opening-bonus) ...)
```

Accessor Functions

It's considered good practice to use accessor functions rather than directly accessing members with SLOT-VALUE.

You can manually define a 'reader' function like this:

```
(defgeneric balance (account))
(defmethod balance ((account bank-account))
  (slot-value account 'balance))
```

The cleanest way to manually define a 'writer' function is as a so called SETF function, extending SETF with a new kind of settable place.

```
(defgeneric (setf customer-name) (value account))
(defmethod (setf customer-name) (value (account bank-account))
  (setf (slot-value account 'customer-name) value))

(setf (customer-name *account*) "Sally Sue") => "Sally Sue"
```

A SETF function can actually take any number of arguments, but the first is always the value to be assigned to the place.

DEFCLASS supports three slot options that allow you to create reader and writer functions automatically:

```
(customer-name
 :initarg :customer-name
 :initform (error "Must supply a customer-name")
 :reader customer-name                    ; Creates a reader method
 :writer (setf customer-name)             ; Creates a SETF writer method
 :accessor customer-name                  ; Creates both reader and SETF writer methods
 :documentation "Customers full name")    ; Useful slot option
```

If you are going to be accessing particular slots frequently within a function, WITH-ACCESSORS can streamline your code somewhat. For example

```
(setf (customer-name *account*) "Sally Sue")
```

can become

```
(with-accessors (customer-name) *account* (setf customer-name "Sally Sue"))
```

or

```
(with-accessors ((custname customer-name)) *account* (setf custname "Sally Sue"))
```

Similarly when implementing low level methods you can reduce multiple verbose slot-value forms with the WITH-SLOTS macro. For example:

```
(setf (slot-value *account* customer-name) "Undefined")
```

can become

```
(with-slots (customer-name) *account* (setf customer-name "Undefined"))
```

or

```
(with-slots ((custname customer-name)) *account* (setf custname "Undefined"))
```

The basic forms are

```
(with-accessors (accessor*) instance-form body-form*)
(with-slots (slot*) instance-form body-form*)
```

Class-Allocated Slots

The :allocation slot option defaults to :instance, but it can also be set to :class in which case the slot has a single value shared by all instances of the class.

Slots and Inheritance

- Subclasses inherit all the slots of their superclasses.
- Any given object can only have one slot with a particular name.
- All the slots sharing a name are merged into a single slot.
- Where multiple :initform options exist, the one from the most specific class is used.
- All unique :initarg keyword parameters can be used. If you pass more than one to MAKE-INSTANCE then the leftmost argument is used.
- :reader, :writer and :accessor methods from superclasses are automatically applicable to a subclass but it can also create its own.
- :allocation is determined by the most specific class that specifies the slot. :class slots at different levels in the class hierarchy will refer to distinct variables.
- The Package system can be used to prevent unwanted slot name collisions and slot merging

Multiple Inheritance

Where a subclass has two or more superclasses, their order in DEFCLASS's superclass list form is also used to decide class specificity both when building the effective methods and when merging inherited slot specifiers. Superclasses appearing earlier in the list are considered more specific than those that follow.

FORMAT Recipes

(format destination control-string values*) => NIL or a string (see destination=NIL)

The destination can be:

T	- shorthand for *STANDARD-OUTPUT*
NIL	- format returns its output as a string
A stream	- format outputs to the stream
A string with a fill pointer	- format outputs to the string.

The control string can contain literal text and FORMAT directives.
All directives start with a tilde (~) and end with a single case-insensitive character that identifies the directive.
Directives can take prefix parameters between the tilde and the directive character, separated by commas.
Prefix parameters can be:

decimal numbers
characters preceded by a single quote
v - which causes FORMAT to consume one format argument and use its value as the prefix parameter
- which FORMAT evaluates as the number of remaining format arguments
: or @ - sometimes used to modify the directives behaviour in a small way.

Prefix arguments are positional so you must include a comma for each missing parameter before the one you wish to specify.

Directives

~a - Consumes one format argument of any type and outputs it aesthetically

(format t "~a ~a ~a ~a" 10 "foo\"bar" (list 1 2 3) NIL) >> 10 foo"bar (1 2 3) ()

~s - Consumes one format argument of any type and outputs it in READable format, if possible.

(format t "~s ~s ~s" 10 "foo\"bar" (list 1 2 3) NIL) >> 10 "foo\"bar" (1 2 3) ()

~% - Emits a newline.
- Takes a single numeric parameter specifying the number of newlines to emit

~& - Emits a newline unless already at the beginning of a line.
- Takes a single numeric parameter specifying the number of newlines to emit, which may be reduced by one if already at the beginning of a line.

~~ - Emits a literal tilde, or more than one if parametrized with a number.

(format NIL "~~ ~3~") => "~ ~~~"

~c - Emits a character. Same as ~a but only for characters.
- :-modifier - Outputs non-printing characters by name
- @-modifier - Outputs a character in lisp literal syntax

(format t " ~c ~:c ~@c" #\Q #\newline) >> Q Newline #\z

~d - Outputs integers in base 10
- :-modifier - adds commas
- @-modifier - always prepends the sign
- The first prefix parameter can specify a minimum width for the output
- The second prefix parameter can specify a padding character (default is space)

(format t "~d ~:d ~@d ~:@d" 11002 12003 13004 14005) >> 11002 12,003 +13004 +14,005
(format nil "~12d" 1000000) => " 1000000"
(format nil "~12,'0d" 1000000) => "000001000000"

~x - Same as ~d except in hexadecimal
~o - Same as ~d except in octal
~b - Same as ~d except in binary
~r - Same as ~d except the first prefix parameter specifies a base between 2 and 36 inclusive.
- with no prefix and no modifiers, outputs the number in words as a cardinal number.
- with no prefix and a :-modifier, outputs the number in words as an ordinal.
- with no prefix and a @-modifier, outputs the number in Roman numerals.
- with no prefix and both modifiers, outputs the number in old-style Roman numerals (4 = IIII not IV and 9 = VIIII not IX)

(format t "~x ~o ~b ~36r" 456 456 456 456) >> 1C8 710 111001000 CO
(format t "~r" 123456) >> one hundred and twenty-three thousand, four hundred and fifty-six
(format t "~:r" 123456) >> one hundred and twenty-three thousand, four hundred fifty-sixth
(format t "~@r ~:@r" 2009 2009) >> MMIX MMVIIII

~f - Outputs a floating point number, possibly in scientific notation if large/small enough
- @-modifier - Always output the sign

- The second prefix parameter controls the number of decimal points

~e - Same as ~f except always outputs scientific notation

```
(format t "~f ~,4f" pi pi) >> 3.1415926535897932385 3.1416
(format t "~e ~,ef" pi pi) >> 3.1415926535897932385L+0 3.1416L+0
```

~$ - Outputs a number in monetary format.
- @-modifier - Always output the sign
- The first prefix parameter specifies the number of digits printed after the decimal point.
- The second prefix parameter specifies the minimum number of digits before the decimal point.

```
(format t "~$ ~@$ ~2,4$" 6.47 6.47 6.47) >> 6.47 +6.47 0006.47
```

~p - Emits an 's' unless the corresponding argument is 1
- :-modifier - Use the previous argument rather than the current one
- @-modifier - emits 'y' or 'ies' rather than 's'

```
(format t "file~p file~p file~p file~:p famil~:@p" 0 1 3) >> files file files files families
```

~(~) - All output generated by the control string between these directives is converted to lowercase
- @-modifier - Capitalised the first word
- :-modifier - Capitalises every word
- Both modifiers - Converts all text to uppercase

```
(format t "~(~a~)" "tHe QuIck bRoWn fOX")   >> the quick brown fox
(format t "~@(~a~)" "tHe QuIck bRoWn fOX")  >> The quick brown fox
(format t "~:(~a~)" "tHe QuIck bRoWn fOX")  >> The Quick Brown Fox
(format t "~@:(~a~)" "tHe QuIck bRoWn fOX") >> THE QUICK BROWN FOX
```

~[~] - These directives delimit one or more clauses separated by ~; of which one is chosen.
- No modifier - The argument is a numeric index. An out of bounds index produces nothing.
- If the last clause separator is ~:; instead of ~; then the last clause is the default.
- You can also select the clause with a prefix parameter. See the description of '#' above
- :-modifier - Only two clauses allowed. Argument NIL -> 1st clause, otherwise 2nd clause.
- @-modifier - Only one clause. If the argument is true, it is reinstated to be consumed again.

```
(format t "~[zero~;one~;two~]" 2)                                  >> two
(format t "~[zero~;one~;two~]" 3)                                  >>
(format t "~[zero~;one~;two~:;lots~]" 3)                  >> lots
(format t "~#[zero~;one~;two~;three~] arguments left" 1 2) >> two arguments left
(format t "~:[false~;true~]" nil)                         >> false
(format t "~:[false~;true~]" t)                           >> true
(format t "~@[true~] ~a" NIL "ignore")                    >>  ignore
(format t "~@[true~] ~a" 42 "ignore")                     >> true 42
```

~{ ~} - The iteration directive. The text between the markers is processed as a control string.
- No modifiers - Consumes one argument which must be a list. FORMAT repeatedly processes the control string, consuming arguments from the original list argument until it is empty.
- @-modifier - the remaining FORMAT arguments are processed as a list.
- :-modifier - each element of the list must itself be a list and is passed as such to the control string.
- Both modifiers - Each remaining FORMAT argument must be a list and is passed as such to the control string.
- Within the body of the ~{~} directives, the ~^ directive will abort the iteration once the argument list is empty.
- Within the body of the ~{~} directives, # refers to the number of arguments left to process.
- A :-modifier on the closing ~} directive forces the iteration to be run at least once.
- The first prefix parameter specifies a maximum number of iterations

```
(format t "~{~a, ~}" (list 1 2 3))          >> 1, 2, 3,
(format t "~{~a~^, ~}" (list 1 2 3))        >> 1, 2, 3
(format t "~@{~a~^, ~}" 1 2 3)                      >> 1, 2, 3
```

~* - Skip the next argument (consuming it without emitting anything).
- :-modifier - back up one argument, allowing it to be consumed again.
- with ~{~} this directive skips or backs up over the items of the list.
- The first prefix parameter specifies the number of arguments to skip/back-up.
- @-modifier - Jumps to the absolute argument 0 or the index given by the prefix parameter.

~? - Can get snippets of control strings from the FORMAT arguments.

~/ - Allows you to call an arbitrary function to handle the next format argument.

Conditions and Restarts

A condition is an object whose:

- class indicates the general nature of the condition
- instance carries the details of this particular condition.

Condition classes are defined using the DEFINE-CONDITION macro which is essentially the same as DEFCLASS except that the default superclass is CONDITION rather than STANDARD-OBJECT. Slots are defined in the same way and condition classes can singly and multiply inherit from other classes that descend from CONDITION. There are some differences though:

- You can't use SLOT-VALUE which is only defined for STANDARD-OBJECT derived classes. Use accessors instead.
- Instances are create with MAKE-CONDITION instead of MAKE-INSTANCE.
- MAKE-CONDITION initialises the slots based on :initarg but lacks INITIALIZE-INSTANCE.

When using the condition system for error handling, you should subclass the ERROR class (a subclass of CONDITION):

```
(define-condition malformed-log-entry-error (error)
  ((text :initarg :text :reader text)))
```

The ERROR function is used to signal errors and can be called in two ways:

- With an already instantiated CONDITION derived object.
- With the name of a CONDITION derived class and any initargs needed to construct it.

```
(error 'malformed-log-entry-error :text "its broken")
```

ERROR calls the lower level SIGNAL function and drops into the debugger unless the condition is handled.

Condition handlers

The HANDLER-CASE macro can set-up one or more condition handlers for a given expression:

```
(handler-case expression error-clause*)
```

Error clauses:

```
(condition-type ([var]) handler-code)
```

The HANDLER-CASE expression is a single expression so you will need to use PROGN around multiple expressions.
The error clause declares a condition class that it can handle and, optionally, a variable which will be bound to the condition object.
HANDLER-CASE will return either the result of it's expression, or the result of an error clause if it is invoked.

```
(define-condition my-error (error)
  ((text :initarg :text :reader text)))

(defun low-level (signal-error)
  (format t "  low level - entering~%")
  (when signal-error
    (format t "  low level - signalling error~%")
    (error 'my-error :text "oops"))
  (format t "  low level - leaving~%"))

(defun high-level (signal-error)
  (handler-case
        (progn
          (format t "high level - entering~%")
          (low-level signal-error)
          (format t "high level - leaving~%"))
    (my-error (me)
        (format t "high level - handled an error: ~a~%" (text me)))))
```

```
CL-USER> (high-level nil)
  high level - entering
    low level - entering
    low level - leaving
  high level - leaving
  NIL
CL-USER> (high-level t)
  high level - entering
    low level - entering
    low level - signalling error
  high level - handled an error: oops
  NIL
```

Restarts

HANDLER-CASE destructively unwinds the stack when a condition is signalled, so it can't be used for restarts.
Instead, we use the lower level HANDLER-BIND macro which preserves the stack so that the associated INVOKE-RESTART macro can restore the

relevant frame and run the most recently bound restart.

```
(handler-bind (binding*) form*)
```

where each binding is a list of condition types and a handler function:

```
((condition*) function-object)
```

If a condition handler returns normally (i.e. doesn't INVOKE-RESTART) then the condition will be passed to other condition handlers higher on the stack to handle. To 'handle' the condition, the handler function must invoke a restart with INVOKE-RESTART. (INVOKE-RESTART will signal a CONTROL-ERROR if the requested restart isn't established, so FIND-RESTART can be used to location and conditionally invoke a restart).

Restarts themselves are establish by the RESTART-CASE form:

```
(restart-case expression restart-form*)
```

where the restart forms are:

```
(name (parameter-form*) expression*)
```

Example:

```
(define-condition my-condition (condition)
  ((text :initarg :text :reader text)))

(defun low-level (signal-condition)
  (format t "    low level - entered~%")
  (when signal-condition
    (format t "    low level - signalling condition~%")
    (error 'my-condition :text "oops"))
  (format t "    low level - leaving~%"))

(defun medium-level (signal-condition)
  (restart-case
      (progn
            (format t "  medium level - entered~%")
            (low-level signal-condition))
    (skip-it (why) (format t "  medium level - skipping error because: ~a~%" why)))
  (format t "  medium level - leaving~%"))

(defun high-level (signal-condition)
  (handler-bind (
                        (my-condition #'(lambda (me)
                                                (format t "high level caught the condition: ~a~%" (text me))
                                                (let ((restart (find-restart 'skip-it)))
                                                  (when restart (invoke-restart restart "high level says so"))))))
    (format t "high level - entered~%")
    (medium-level signal-condition)
    (format t "high level - leaving~%")))


CL-USER> (high-level nil)
high level - entered
  medium level - entered
    low level - entered
    low level - leaving
  medium level - leaving
high level - leaving
NIL
CL-USER> (high-level t)
high level - entered
  medium level - entered
    low level - entered
    low level - signalling condition
high level caught the condition: oops
  medium level - skipping error because: high level says so
  medium level - leaving
high level - leaving
NIL
```

Signalling functions

The primitive SIGNAL function implements the mechanism of searching for and calling condition handlers. When a handler declines to handle the condition by returning normally, control passes back to SIGNAL which then searches back up the tree for the next condition handler. When all condition handlers have been tried, SIGNAL also returns normally.

ERROR calls SIGNAL. If the condition is handled by HANDLER-CASE or by invoking a restart, then the call to SIGNAL never returns. If the condition isn't handled and SIGNAL returns normally, ERROR invokes the debugger by calling the function stored in *DEBUGGER-HOOK*. Thus

ERROR can never return normally; the condition must be handled by a condition handler or in the debugger.

WARN also calls SIGNAL. But if SIGNAL returns, WARN doesn't invoke the debugger, it prints the condition to the *ERROR-OUTPUT* stream and returns NIL instead, allowing its caller to proceed. WARN also establishes a MUFFLE-WARNING restart around the call to SIGNAL that can be used by a condition handler to prevent WARN from printing anything. The MUFFLE-WARNING _function_ calls its same-name restart, signalling a CONTROL-ERROR if the restart is unavailable.

CERROR, like ERROR, will drop into the debugger if the condition isn't handled. But it also establishes a CONTINUE restart around SIGNAL which causes CERROR to return normally.

Of course, you can also build your own signalling functions based on SIGNAL.

SPECIAL OPERATORS

QUOTE	- Prevents evaluation; to get s-expressions as data
IF	- The fundamental basis of all conditional execution constructs
PROGN	- Provides the ability to sequence atoms
LET	- Creates new variable bindings, in parallel
LET*	- Creates new variable bindings, sequentially
SETQ	- Assigns values to variables
FLET	- Defines non-recursive local functions
LABELS	- Defines recursive local functions
MACROLET	- Defines local macros
SYMBOL-MACROLET	- Defines symbol macros
BLOCK	- Block and return-from together provide a means of returning immediately
RETURN-FROM	/ from a section of code
TAGBODY	- Tagbody and go provide a low-level 'goto' construct that is the basis of all
GO	/ high level looping constructs
CATCH	- The dynamic counterparts of BLOCK and RETURN-FROM.
THROW	/
UNWIND-PROTECT	- Makes sure specified code is run regardless of how control leaves the scope
MULTIPLE-VALUE-CALL	- The basis of the MULTIPLE-VALUE-BIND macro, providing multiple value returns.
MULTIPLE-VALUE-PROG1	/
EVAL-WHEN	- Allows control over when specific bits of code are evaluated.
LOCALLY	- Part of Common Lisps declaration system
THE	/
LOAD-TIME-VALUE	- Create a value that's determined at load time.
PROGV	- Creates new dynamic bindings for variables whose names are determined at runtime.

Packages and Symbols

A name containing either a single or double colon is a package qualified name.
A name containing a single colon must refer to an 'exported' symbol of the specified package.
A double colon can be used to force access to unexported symbols in a package. Not usually a good idea.
Names starting with a colon are 'keyword' symbols, interned in the package named KEYWORD and automatically exported.
When the reader interns a keyword, it also defines a constant variable with the symbol as both its name and value. This allows keywords to be used unquoted in function calls etc. since:

```
(eql ':foo :foo) => T
```

A keyword symbol name doesn't include the colon:

```
(symbol-name :foo) => "FOO"
```

Uninterned symbols start with #: and are not interned in any package. Each time the reader reads a #: name, it creates a new symbol, so:

```
(eql '#:foo '#:foo) => NIL
```

The current package is stored in the global variable *package*.
When you start lisp, the packages is typically COMMON-LISP-USER, also known as CL-USER.
CL-USER 'uses' the package COMMON-LISP, which contains and exports all the names defined by the language standard.
COMMON-LISP also has the nickname CL.

DEFPACKAGE is used to define new packages. Example

```
(defpackage :org.heresymail.core (:use :common-lisp))
```

Note the use of a symbol for the package names. They could be given as strings, but then they would have to be written in uppercase:

```
(defpackage "ORG.HERESYMAIL.CORE" (:use "COMMON-LISP"))
```

You could also use non-keyword symbols, but the reader would then intern them in the current package while reading the DEFPACKAGE form, which might cause problems.

To read code into this package, you must make it the 'current' package with the IN-PACKAGE macro:

```
(in-package :org.heresymail.core)
```

This changes the value of *package*, affecting how the REPL reads subsequent expressions. Remember to change back to CL-USER.

To export names, making them available to other packages, use the :export keyword:

```
(defpackage :org.heresymail.core (:use :common-lisp) (:export :load :save))
```

The :use clause imports all exported symbols from the named packages into the current package.
It's also possible to import individual symbols rather than all exported symbols:

```
(defpackage :org.heresymail.app (:use :common-lisp) (:import-from :org.heresymail.core :load))
```

Each :import-from clause can import any number of symbols and multiple import-from clauses can be used to import symbols from multiple packages.

You can also import all exported symbols with :use but exclude some names, perhaps because they clash with your own names, by using a shadow clause:

```
(defpackage :org.heresymail.app (:use :common-lisp :org.heresymail.core) (:shadow :save))
```

When using multiple packages and getting name clashes, you can use a :shadowing-import-from to specify which package to import certain names from.