1 Common LISP Hints[1]

Note: This tutorial introduction to Common Lisp was written for the CMU environment, so some of the details of running lisp toward the end may differ from site to site.

Further Information: The best LISP textbook I know of is:

Guy L. Steele Jr., *Common LISP: the Language*. Digital Press. 1984.

The first edition is easier to read; the second describes a more recent standard. (The differences between the two standards shouldn't affect casual programmers.)

A book by Dave Touretsky has also been recommended to me, although I haven't read it, so I can't say anything about it.

1.1 Symbols

A symbol is just a string of characters. There are restrictions on what you can include in a symbol and what the first character can be, but as long as you stick to letters, digits, and hyphens, you'll be safe. (Except that if you use only digits and possibly an initial hyphen, LISP will think you typed an integer rather than a symbol.) Some examples of symbols:

```
a
b
c1
foo
bar
baaz-quux-garply
```

Some things you can do with symbols follow. (Things after a "`>`" prompt are what you type to the LISP interpreter, while other things are what the LISP interpreter prints back to you. The "`;`" is LISP's comment character: everything from a "`;`" to the end of line is ignored.)

```
> (setq a 5)       ;store a number as the value of a symbol
5
> a                ;take the value of a symbol
5
> (let ((a 6)) a)  ;bind the value of a symbol temporarily to 6
6
> a            ;the value returns to 5 once the let is finished
5
> (+ a 6) ;use the value of a symbol as an argument to a function
11
> b         ;try to take the value of a symbol which has no value
Error: Attempt to take the value of the unbound symbol B
```

There are two special symbols, `t` and `nil`. The value of `t` is defined always to be `t`, and the value of `nil` is defined always to be `nil`. LISP uses `t` and `nil` to represent true and false. An example of this use is in the if statement, described more fully later:

```
> (if t 5 6)
5
> (if nil 5 6)
6
```

[1] This tutorial was written by Geoffrey J. Gordon, February 5, 1993 and modified by Bruno Haible

```
> (if 4 5 6)
5
```

The last example is odd but correct: `nil` means false, and anything else means true. (Unless we have a reason to do otherwise, we use `t` to mean true, just for the sake of clarity.)

Symbols like `t` and `nil` are called self-evaluating symbols, because they evaluate to themselves. There is a whole class of self-evaluating symbols called keywords; any symbol whose name starts with a colon is a keyword. (See below for some uses for keywords.) Some examples:

```
> :this-is-a-keyword
:THIS-IS-A-KEYWORD
> :so-is-this
:SO-IS-THIS
> :me-too
:ME-TOO
```

1.2 Numbers

An integer is a string of digits optionally preceded by `+` or `-`. A real number looks like an integer, except that it has a decimal point and optionally can be written in scientific notation. A rational looks like two integers with a `/` between them. LISP supports complex numbers, which are written `#c(`r i`)` (where r is the real part and i is the imaginary part). A number is any of the above. Here are some numbers:

```
5
17
-34
+6
3.1415
1.722e-15
#c(1.722e-15 0.75)
```

The standard arithmetic functions are all available: `+`, `-`, `*`, `/`, `floor`, `ceiling`, `mod`, `sin`, `cos`, `tan`, `sqrt`, `exp`, `expt`, and so forth. All of them accept any kind of number as an argument. `+`, `-`, `*`, and `/` return a number according to type contagion: an integer plus a rational is a rational, a rational plus a real is a real, and a real plus a complex is a complex. Here are some examples:

```
> (+ 3 3/4)             ;type contagion
15/4
> (exp 1)               ;e
2.7182817
> (exp 3)               ;e*e*e
20.085537
> (expt 3 4.2)          ;exponent with a base other than e
100.90418
> (+ 5 6 7 (* 8 9 10))  ;the fns +-*/ accept multiple arguments
```

There is no limit to the absolute value of an integer except the memory size of your computer. Be warned that computations with bignums (as large integers are called) can be slow. (So can computations with rationals, especially compared to the corresponding computations with small integers or floats.)

1.3 Conses

A cons is just a two-field record. The fields are called *car* and *cdr*, for historical reasons. (On the first machine where LISP was implemented, there were two instructions `car` and `cdr` which stood for *contents of address register* and *contents of decrement register*. Conses were implemented using these two registers.)

Conses are easy to use:

```
> (cons 4 5) ;Allocate a cons. Set the car to 4 and the cdr to 5.
(4 . 5)
> (cons (cons 4 5) 6)
((4 . 5) . 6)
> (car (cons 4 5))
4
> (cdr (cons 4 5))
5
```

1.4 Lists

You can build many structures out of conses. Perhaps the simplest is a linked list: the car of each cons points to one of the elements of the list, and the cdr points either to another cons or to `nil`. You can create such a linked list with the `list` fuction:

```
> (list 4 5 6)
(4 5 6)
```

Notice that LISP prints linked lists a special way: it omits some of the periods and parentheses. The rule is: if the cdr of a cons is `nil`, LISP doesn't bother to print the period or the `nil`; and if the cdr of consA is cons B, then LISP doesn't bother to print the period for cons A or the parentheses for cons B. So:

```
> (cons 4 nil)
(4)
> (cons 4 (cons 5 6))
(4 5 . 6)
> (cons 4 (cons 5 (cons 6 nil)))
(4 5 6)
```

The last example is exactly equivalent to the call `(list 4 5 6)`. Note that `nil` now means the list with no elements: the cdr of `(a b)`, a list with 2 elements, is `(b)`, a list with 1 element; and the cdr of `(b)`, a list with 1 element, is `nil`, which therefore must be a list with no elements.

The car and cdr of `nil` are defined to be `nil`.

If you store your list in a variable, you can make it act like a stack:

```
> (setq a nil)
NIL
> (push 4 a)
(4)
> (push 5 a)
(5 4)
> (pop a)
5
> a
(4)
> (pop a)
```

```
4
> (pop a)
NIL
> a
NIL
```

1.5 Functions

You saw one example of a function above. Here are some more:

```
> (+ 3 4 5 6)      ;this function takes any number of arguments
18
> (+ (+ 3 4) (+ (+ 4 5) 6))     ;isn't prefix notation fun?
22
> (defun foo (x y) (+ x y 5))   ;defining a function
FOO
> (foo 5 0)                     ;calling a function
10
> (defun fact (x)               ;a recursive function
    (if (> x 0)
      (* x (fact (- x 1)))
      1
  ) )
FACT
> (fact 5)
120
> (defun a (x) (if (= x 0) t (b (- x))) ;mutually recursive func.
A
> (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
B
> (a 5)
T
> (defun bar (x)          ;a function with multiple statements in
    (setq x (* x 3))      ;its body -- it will return the value
    (setq x (/ x 2))      ;returned by its final statement
    (+ x 4)
  )
BAR
> (bar 6)
13
```

When we defined `foo`, we gave it two arguments, `x` and `y`. Now when we call `foo`, we are required to provide exactly two arguments: the first will become the value of `x` for the duration of the call to `foo`, and the second will become the value of `y` for the duration of the call. In LISP, most variables are lexically scoped; that is, if `foo` calls `bar` and `bar` tries to reference `x`, `bar` will not get `foo`'s value for `x`.

The process of assigning a symbol a value for the duration of some lexical scope is called binding.

You can specify optional arguments for your functions. Any argument after the symbol `&optional` is optional:

```
> (defun bar (x &optional y) (if y x 0))
BAR
> (defun baaz (&optional (x 3) (z 10)) (+ x z))
```

```
BAAZ
> (bar 5)
0
> (bar 5 t)
5
> (baaz 5)
15
> (baaz 5 6)
11
> (baaz)
13
```

It is legal to call the function `bar` with either one or two arguments. If it is called with one argument, `x` will be bound to the value of that argument and `y` will be bound to `nil`; if it is called with two arguments, `x` and `y` will be bound to the values of the first and second argument, respectively.

The function `baaz` has two optional arguments. It specifies a default value for each of them: if the caller specifies only one argument, `z` will be bound to 10 instead of to nil, and if the caller specifies no arguments, `x` will be bound to 3 and `z` to 10.

You can make your function accept any number of arguments by ending its argument list with an `&rest` parameter. LISP will collect all arguments not otherwise accounted for into a list and bind the `&rest` parameter to that list. So:

```
> (defun foo (x &rest y) y)
FOO
> (foo 3)
NIL
> (foo 4 5 6)
(5 6)
```

Finally, you can give your function another kind of optional argument called a keyword argument. The caller can give these arguments in any order, because they're labelled with keywords.

```
> (defun foo (&key x y) (cons x y))
FOO
> (foo :x 5 :y 3)
(5 . 3)
> (foo :y 3 :x 5)
(5 . 3)
> (foo :y 3)
(NIL . 3)
> (foo)
(NIL)
```

An `&key` parameter can have a default value too:

```
> (defun foo (&key (x 5)) x)
FOO
> (foo :x 7)
7
> (foo)
5
```

1.6 Printing

Some functions can cause output. The simplest one is print, which prints its argument and then returns it.

```
> (print 3)
3
3
```

The first 3 above was printed, the second was returned.

If you want more complicated output, you will need to use format. Here's an example:

```
> (format t "An atom: ~S~%and a list: ~S~%and an integer: ~D~%"
          nil (list 5) 6)
An atom: NIL
and a list: (5)
and an integer: 6
```

The first argument to format is either `t`, `nil`, or a stream. `t` specifies output to the terminal. `nil` means not to print anything but to return a string containing the output instead. Streams are general places for output to go: they can specify a file, or the terminal, or another program. This handout will not describe streams in any further detail.

The second argument is a formatting template, which is a string optionally containing formatting directives.

All remaining arguments may be referred to by the formatting directives. LISP will replace the directives with some appropriate characters based on the arguments to which they refer and then print the resulting string.

Format always returns `nil` unless its first argument is `nil`, in which case it prints nothing and returns a string.

There are three different directives in the above example: `~S`, `~D`, and `~%`. The first one accepts any LISP object and is replaced by a printed representation of that object (the same representation which is produced by print). The second one accepts only integers. The third one doesn't refer to an argument; it is always replaced by a carriage return.

Another useful directive is `~~`, which is replaced by a single `~`.

Refer to a LISP manual for (many, many) additional formatting directives.

1.7 Forms and the Top-Level Loop

The things which you type to the LISP interpreter are called forms; the LISP interpreter repeatedly reads a form, evaluates it, and prints the result. This procedure is called the read-eval-print loop.

Some forms will cause errors. After an error, LISP will put you into the debugger so you can try to figure out what caused the error. LISP debuggers are all different; but most will respond to the command `help` or `:help` by giving some form of help.

In general, a form is either an atom (for example, a symbol, an integer, or a string) or a list. If the form is an atom, LISP evaluates it immediately. Symbols evaluate to their value; integers and strings evaluate to themselves. If the form is a list, LISP treats its first element as the name of a function; it evaluates the remaining elements recursively, and then calls the function with the values of the remaining elements as arguments.

For example, if LISP sees the form `(+ 3 4)`, it treats `+` as the name of a function. It then evaluates 3 to get 3 and 4 to get 4; finally it calls `+` with 3 and 4 as the arguments. The `+` function returns 7, which LISP prints.

The top-level loop provides some other conveniences; one particularly convenient convenience is the ability to talk about the results of previously typed forms. LISP always saves its most recent three results; it stores them as the values of the symbols `*`, `**`, and `***`. For example:

```
> 3
3
> 4
4
> 5
5
> ***
3
> ***
4
> ***
5
> **
4
> *
4
```

1.8 Special forms

There are a number of special forms which look like function calls but aren't. These include control constructs such as `if` statements and `do` loops; assignments like `setq`, `setf`,`push`, and `pop`; definitions such as `defun` and `defstruct`; and binding constructs such as `let`. (Not all of these special forms have been mentioned yet. See below.)

One useful special form is the `quote` form: `quote` prevents its argument from being evaluated. For example:

```
> (setq a 3)
3
> a
3
> (quote a)
A
> 'a                    ;'a is an abbreviation for (quote a)
A
```

Another similar special form is the `function` form: `function` causes its argument to be interpreted as a function rather than being evaluated. For example:

```
> (setq + 3)
3
> +
3
> '+
+
> (function +)
#<Function + @ #x-fbef9de>
> #'+                   ;#'+ is an abbreviation for (function +)
#<Function + @ #x-fbef9de>
```

The `function` special form is useful when you want to pass a function as an argument to another function. See below for some examples of functions which take functions as arguments.

1.9 Binding

Binding is lexically scoped assignment. It happens to the variables in a function's parameter list whenever the function is called: the formal parameters are bound to the actual parameters for the duration of the function call. You can bind variables anywhere in a program with the `let` special form, which looks like this:

```
(let ((var1 val1)
      (var2 val2)
      ...
     )
  body
)
```

`let` binds `var1` to `val1`, `var2` to `val2`, and so forth; then it executes the statements in its body. The body of a `let` follows exactly the same rules that a function body does. Some examples:

```
> (let ((a 3)) (+ a 1))
4
> (let ((a 2)
        (b 3)
        (c 0))
    (setq c (+ a b))
    c
  )
5
> (setq c 4)
4
> (let ((c 5)) c)
5
> c
4
```

Instead of `(let ((a nil) (b nil)) ...)`, you can write `(let (a b) ...)`.

The `val1`, `val2`, etc. inside a `let` cannot reference the variables `var1`, `var2`, etc. that the `let` is binding. For example,

```
> (let ((x 1)
        (y (+ x 1)))
    y
  )
Error: Attempt to take the value of the unbound symbol X
```

If the symbol `x` already has a global value, stranger happenings will result:

```
> (setq x 7)
7
> (let ((x 1)
        (y (+ x 1)))
    y
  )
8
```

The `let*` special form is just like `let` except that it allows values to reference variables defined earlier in the `let*`. For example,

```
> (setq x 7)
7
> (let* ((x 1)
         (y (+ x 1)))
    y
  )
2
```

The form:

```
(let* ((x a)
       (y b))
  ...
)
```

is equivalent to:

```
(let ((x a))
  (let ((y b))
    ...
) )
```

1.10 Dynamic Scoping

The `let` and `let*` forms provide lexical scoping, which is what you expect if you're used to programming in C or Pascal. Dynamic scoping is what you get in BASIC: if you assign a value to a dynamically scoped variable, every mention of that variable returns that value until you assign another value to the same variable.

In LISP, dynamically scoped variables are called special variables. You can declare a special variable with the `defvar` special form. Here are some examples of lexically and dynamically scoped variables.

In this example, the function `check-regular` references a regular (ie, lexically scoped) variable. Since `check-regular` is lexically outside of the `let` which binds `regular`, `check-regular` returns the variable's global value.

```
> (setq regular 5)
5
> (defun check-regular () regular)
CHECK-REGULAR
> (check-regular)
5
> (let ((regular 6)) (check-regular))
5
```

In this example, the function `check-special` references a special (ie, dynamically scoped) variable. Since the call to `check-special` is temporally inside of the `let` which binds `*special*`, `check-special` returns the variable's local value.

```
> (defvar *special* 5)
*SPECIAL*
> (defun check-special () *special*)
CHECK-SPECIAL
```

```
> (check-special)
5
> (let ((*special* 6)) (check-special))
6
```

By convention, the name of a special variable begins and ends with a `*`. Special variables are chiefly used as global variables, since programmers usually expect lexical scoping for local variables and dynamic scoping for global variables.

For more information on the difference between lexical and dynamic scoping, see *Common LISP: the Language*.

1.11 Arrays

The function `make-array` makes an array. The `aref` function accesses its elements. All elements of an array are initially set to `nil`. For example:

```
> (make-array '(3 3))
#2a((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))
> (aref * 1 1)
NIL
> (make-array 4)        ;1D arrays don't need the extra parens
#(NIL NIL NIL NIL)
```

Array indices always start at 0.

See below for how to set the elements of an array.

1.12 Strings

A string is a sequence of characters between double quotes. LISP represents a string as a variable-length array of characters. You can write a string which contains a double quote by preceding the quote with a backslash; a double backslash stands for a single backslash. For example:

```
"abcd" has 4 characters
"\"" has 1 character
"\\" has 1 character
```

Here are some functions for dealing with strings:

```
> (concatenate 'string "abcd" "efg")
"abcdefg"
> (char "abc" 1)
#\b                 ;LISP writes characters preceded by #\
> (aref "abc" 1)
#\b                 ;remember, strings are really arrays
```

The `concatenate` function can actually work with any type of sequence:

```
> (concatenate 'string '(#\a #\b) '(#\c))
"abc"
> (concatenate 'list "abc" "de")
(#\a #\b #\c #\d #\e)
> (concatenate 'vector '#(3 3 3) '#(3 3 3))
#(3 3 3 3 3 3)
```

1.13 Structures

LISP structures are analogous to C structs or Pascal records. Here is an example:

```
> (defstruct foo
    bar
    baaz
    quux
  )
FOO
```

This example defines a data type called `foo` which is a structure containing 3 fields. It also defines 4 functions which operate on this data type: `make-foo`, `foo-bar`, `foo-baaz`, and `foo-quux`. The first one makes a new object of type `foo`; the others access the fields of an object of type `foo`. Here is how to use these functions:

```
> (make-foo)
#s(FOO :BAR NIL :BAAZ NIL :QUUX NIL)
> (make-foo :baaz 3)
#s(FOO :BAR NIL :BAAZ 3 :QUUX NIL)
> (foo-bar *)
NIL
> (foo-baaz **)
3
```

The `make-foo` function can take a keyword argument for each of the fields a structure of type `foo` can have. The field access functions each take one argument, a structure of type `foo`, and return the appropriate field.

See below for how to set the fields of a structure.

1.14 Setf

Certain forms in LISP naturally define a memory location. For example, if the value of `x` is a structure of type`foo`, then `(foo-bar x)` defines the `bar` field of the value of `x`. Or, if the value of `y` is a one-dimensional array, `(aref y 2)` defines the third element of `y`.

The `setf` special form uses its first argument to define a place in memory, evaluates its second argument, and stores the resulting value in the resulting memory location. For example,

```
> (setq a (make-array 3))
#(NIL NIL NIL)
> (aref a 1)
NIL
> (setf (aref a 1) 3)
3
> a
#(NIL 3 NIL)
> (aref a 1)
3
> (defstruct foo bar)
FOO
> (setq a (make-foo))
#s(FOO :BAR NIL)
> (foo-bar a)
```

```
NIL
> (setf (foo-bar a) 3)
3
> a
#s(FOO :BAR 3)
> (foo-bar a)
3
```

`setf` is the only way to set the fields of a structure or the elements of an array.

Here are some more examples of `setf` and related functions.

```
> (setf a (make-array 1)) ;setf on a variable is equivalent
#(NIL)                    ;to setq
> (push 5 (aref a 1))     ;push can act like setf
(5)
> (pop (aref a 1))        ;so can pop
5
> (setf (aref a 1) 5)
5
> (incf (aref a 1))       ;incf reads from a place, increments,
6                         ;and writes back
> (aref a 1)
6
```

1.15 Booleans and Conditionals

LISP uses the self-evaluating symbol `nil` to mean false. Anything other than `nil` means true. Unless we have a reason not to, we usually use the self-evaluating symbol `t` to stand for true.

LISP provides a standard set of logical functions, for example `and`, `or`, and `not`. The `and` and `or` connectives are short-circuiting: `and` will not evaluate any arguments to the right of the first one which evaluates to `nil`, while `or` will not evaluate any arguments to the right of the first one which evaluates to `t`.

LISP also provides several special forms for conditional execution. The simplest of these is `if`. The first argument of `if` determines whether the second or third argument will be executed:

```
> (if t 5 6)
5
> (if nil 5 6)
6
> (if 4 5 6)
5
```

If you need to put more than one statement in the then or else clause of an `if` statement, you can use the `progn` special form. `progn` executes each statement in its body, then returns the value of the final one.

```
> (setq a 7)
7
> (setq b 0)
0
> (setq c 5)
5
> (if (> a 5)
    (progn
```

```
      (setq a (+ b 7))
      (setq b (+ c 8)))
    (setq b 4)
  )
13
```

An `if` statement which lacks either a then or an else clause can be written using the `when` or `unless` special form:

```
> (when t 3)
3
> (when nil 3)
NIL
> (unless t 3)
NIL
> (unless nil 3)
3
```

`when` and `unless`, unlike `if`, allow any number of statements in their bodies. (Eg, `(when  x a b c)` is equivalent to `(if x (progn a b c))`.)

```
> (when t
    (setq a 5)
    (+ a 6)
  )
11
```

More complicated conditionals can be defined using the `cond` special form, which is equivalent to an if ... else if ... fi construction.

A `cond` consists of the symbol `cond` followed by a number of cond clauses, each of which is a list. The first element of a cond clause is the condition; the remaining elements (if any) are the action. The `cond` form finds the first clause whose condition evaluates to true (ie, doesn't evaluate to `nil`); it then executes the corresponding action and returns the resulting value. None of the remaining conditions are evaluated; nor are any actions except the one corresponding to the selected condition. For example:

```
> (setq a 3)
3
> (cond
   ((evenp a) a)        ;if a is even return a
   ((> a 7) (/ a 2))    ;else if a is bigger than 7 return a/2
   ((< a 5) (- a 1))    ;else if a is smaller than 5 return a-1
   (t 17)               ;else return 17
  )
2
```

If the action in the selected cond clause is missing, `cond` returns what the condition evaluated to:

```
> (cond ((+ 3 4)))
7
```

Here's a clever little recursive function which uses `cond`. You might be interested in trying to prove that it terminates for all integers `x` at least 1. (If you succeed, please publish the result.)

```
> (defun hotpo (x steps)      ;hotpo stands for Half Or Triple
    (cond                     ;Plus One
     ((= x 1) steps)
     ((oddp x) (hotpo (+ 1 (* x 3)) (+ 1 steps)))
     (t (hotpo (/ x 2) (+ 1 steps)))
  ) )
A
> (hotpo 7 0)
16
```

The LISP `case` statement is like a C switch statement:

```
> (setq x 'b)
B
> (case x
   (a 5)
   ((d e) 7)
   ((b f) 3)
   (otherwise 9)
  )
3
```

The `otherwise` clause at the end means that if `x` is not `a`, `b`, `d`, `e`, or `f`, the `case` statement will return 9.

1.16 Iteration

The simplest iteration construct in LISP is `loop`: a `loop` construct repeatedly executes its body until it hits a `return` special form. For example,

```
> (setq a 4)
4
> (loop
   (setq a (+ a 1))
   (when (> a 7) (return a))
  )
8
> (loop
   (setq a (- a 1))
   (when (< a 3) (return))
  )
NIL
```

The next simplest is `dolist`: `dolist` binds a variable to the elements of a list in order and stops when it hits the end of the list.

```
> (dolist (x '(a b c)) (print x))
A
B
C
NIL
```

`dolist` always returns `nil`. Note that the value of `x` in the above example was never `nil`: the `NIL` below the `C` was the value that `dolist` returned, printed by the read-eval-print loop.

The most complicated iteration primitive is called `do`. A `do` statement looks like this:

```
> (do ((x 1 (+ x 1))
       (y 1 (* y 2)))
      ((> x 5) y)
    (print y)
    (print 'working)
  )
1
WORKING
2
WORKING
4
WORKING
8
WORKING
16
WORKING
32
```

The first part of a `do` specifies what variables to bind, what their initial values are, and how to update them. The second part specifies a termination condition and a return value. The last part is the body. A `do` form binds its variables to their initial values like a `let`, then checks the termination condition. As long as the condition is false, it executes the body repeatedly; when the condition becomes true, it returns the value of the return-value form.

The `do*` form is to `do` as `let*` is to `let`.

1.17 Non-local Exits

The `return` special form mentioned in the section on iteration is an example of a nonlocal return. Another example is the `return-from` form, which returns a value from the surrounding function:

```
> (defun foo (x)
    (return-from foo 3)
    x
  )
FOO
> (foo 17)
3
```

Actually, the `return-from` form can return from any named block —it's just that functions are the only blocks which are named by default. You can create a named block with the `block` special form:

```
> (block foo
    (return-from foo 7)
    3
  )
7
```

The `return` special form can return from any block named `nil`. Loops are by default labelled `nil`, but you can make your own `nil`-labelled blocks:

```
> (block nil
    (return 7)
```

```
    3
  )
7
```

Another form which causes a nonlocal exit is the `error` form:

```
> (error "This is an error")
Error: This is an error
```

The `error` form applies `format` to its arguments, then places you in the debugger.

1.18 Funcall, Apply, and Mapcar

Earlier I promised to give some functions which take functions as arguments. Here they are:

```
> (funcall #'+ 3 4)
7
> (apply #'+ 3 4 '(3 4))
14
> (mapcar #'not '(t nil t nil t nil))
(NIL T NIL T NIL T)
```

`funcall` calls its first argument on its remaining arguments.

`apply` is just like `funcall`, except that its final argument should be a list; the elements of that list are treated as if they were additional arguments to a `funcall`.

The first argument to `mapcar` must be a function of one argument; `mapcar` applies this function to each element of a list and collects the results in another list.

`funcall` and `apply` are chiefly useful when their first argument is a variable. For instance, a search engine could take a heuristic function as a parameter and use `funcall` or `apply` to call that function on a state description. The sorting functions described later use `funcall` to call their comparison functions.

`mapcar`, along with nameless functions (see below), can replace many loops.

1.19 Lambda

If you just want to create a temporary function and don't want to bother giving it a name, `lambda` is what you need.

```
> #'(lambda (x) (+ x 3))
(LAMBDA (X) (+ X 3))
> (funcall * 5)
8
```

The combination of `lambda` and `mapcar` can replace many loops. For example, the following two forms are equivalent:

```
> (do ((x '(1 2 3 4 5) (cdr x))
       (y nil))
      ((null x) (reverse y))
    (push (+ (car x) 2) y)
  )
(3 4 5 6 7)
> (mapcar #'(lambda (x) (+ x 2)) '(1 2 3 4 5))
(3 4 5 6 7)
```

1.20 Sorting

LISP provides two primitives for sorting: `sort` and `stable-sort`.

```
> (sort '(2 1 5 4 6) #'<)
(1 2 4 5 6)
> (sort '(2 1 5 4 6) #'>)
(6 5 4 2 1)
```

The first argument to `sort` is a list; the second is a comparison function. The `sort` function does not guarantee stability: if there are two elements `a` and `b` such that `(and (not (< a b)) (not (< b a)))`, `sort` may arrange them in either order. The `stable-sort` function is exactly like `sort`, except that it guarantees that two equivalent elements appear in the sorted list in the same order that they appeared in the original list.

Be careful: `sort` is allowed to destroy its argument, so if the original sequence is important to you, make a copy with the `copy-list` or `copy-seq` function.

1.21 Equality

LISP has many different ideas of equality. Numerical equality is denoted by `=`. Two symbols are `eq` if and only if they are identical. Two copies of the same list are not `eq`, but they are `equal`.

```
> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (= 3 4)
NIL
> (eq '(a b c) '(a b c))
NIL
> (equal '(a b c) '(a b c))
T
> (eql 'a 'a)
T
> (eql 3 3)
T
```

The `eql` predicate is equivalent to `eq` for symbols and to `=` for numbers.

The `equal` predicate is equivalent to `eql` for symbols and numbers. It is true for two conses if and only if their `car`s are equal and their `cdr`s are equal. It is true for two structures if and only if the structures are the same type and their corresponding fields are equal.

1.22 Some Useful List Functions

These functions all manipulate lists.

```
> (append '(1 2 3) '(4 5 6))    ;concatenate lists
(1 2 3 4 5 6)
> (reverse '(1 2 3))            ;reverse the elements of a list
(3 2 1)
> (member 'a '(b d a c))  ;set membership - returns the first
(A C)                     ;tail whose car is the desired element
> (find 'a '(b d a c))          ;another way to do set membership
```

```
A
> (find '(a b) '((a d) (a d e) (a b d e) ()) :test #'subsetp)
(A B D E)                       ;find is more flexible though
> (subsetp '(a b) '(a d e))     ;set containment
NIL
> (intersection '(a b c) '(b))  ;set intersection
(B)
> (union '(a) '(b))             ;set union
(A B)
> (set-difference '(a b) '(a))  ;set difference
(B)
```

`subsetp`, `intersection`, `union`, and `set-difference` all assume that each argument contains no duplicate elements —`(subsetp '(a a)  '(a b b))` is allowed to fail, for example.

`find`, `subsetp`, `intersection`, `union`, and `set-difference` can all take a `:test` keyword argument; by default, they all use `eql`.

1.23 Getting Started with Emacs

You can use Emacs to edit LISP code: most Emacses are set up to enter LISP mode automatically when they find a file which ends in `.lisp`, but if yours isn't, you can type `M-x lisp-mode`.

You can run LISP under Emacs, too: make sure that there is a command in your path called `lisp` which runs your favorite LISP. For example, you could type

```
ln -s /usr/local/bin/clisp ~/bin/lisp
```

Then in Emacs type `M-x run-lisp`. You can send LISP code to the LISP you just started, and do all sorts of other cool things; for more information, type `C-h m` from any buffer which is in LISP mode.

Actually, you don't even need to make a link. Emacs has a variable called `inferior-lisp-program`; so if you add the line

```
(setq inferior-lisp-program "/usr/local/bin/clisp")
```

to your `.emacs` file, Emacs will know where to find CLISP when you type `M-x run-lisp`.