Lisp Quickstart

Lisp is a deep language with many unusual and powerful features. The goal of this tutorial is **not** to teach you many of those powerful features: rather it's to teach you just enough of Lisp that you can get up and coding quickly if you have a previous background in a procedural language such as C or Java.

Notably this tutorial does **not** teach macros, CLOS, the condition system, much about packages and symbols, or very much I/O.

Table of Contents

Legend

The table cell to the right shows what you type, and the output, for this tutorial. Text shown in blue you are responsible for typing, with a Return at the end of the line. Text shown in black indicates stuff that is printed back to you. Text shown in red are remarks -- do not type them.
 	
If the cell is divided by a line, as is shown at right, then this indicates two different examples.
	`This text is being printed out.`
	`You would type this text  [This is a remark]`
	--- 	
	`Here is another example.`

Running, Breaking, and Quitting Lisp

On your laptop you have several options for running Lisp. **SBCL** is a very popular one, and you start it (typically) by typing **sbcl** on the command line.
 	
On osf1 or mason2, you start lisp by typing **lisp** at the command line. This fires up an implementation of lisp called LispWorks.
 	
On zeus, you start lisp by typing **clisp** at the command line. This fires up an implementation of lisp called
	`[SBCL running on your laptop...]`
	`laptop> sbcl`
	`This is SBCL 1.2.12, an implementation of ANSI Common Lisp.`
	`More information about SBCL is available at .`
	
	`SBCL is free software, provided as is, with absolutely no warranty.`
	`It is mostly in the public domain; some portions are provided under`
	`BSD-style licenses.  See the CREDITS and COPYING files in the`
	`distribution for more information.`
	
	`*`
	
	--- 	
	
	`[LispWorks running on mason / osf1...]`

CLISP.

```
> lisp
LispWorks(R): The Common Lisp Programming Environment
Copyright (C) 1987-2011 LispWorks Ltd.  All rights reserved.
Version 6.1.0
Saved by root as lispworks, at 25 Jun 2012 13:14
User sean on mason
; Loading text file /usr/local/lispworks_6.1/lib/6-1-0-0/config/siteinit.lisp
;  Loading text file /usr/local/lispworks_6.1/lib/6-1-0-0/private-patches/load.lisp

CL-USER 4 >
```

```
[CLISP on zeus...]
zeus> clisp

i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
I I I I I I I      8     8   8           8     8     o  8    8
I  \ `+' /  I      8         8           8     8        8    8
 \  `-+-'  /       8         8           8      ooooo   8oooo
  `-__|__-'        8         8           8           8  8
      |            8     o   8           8     o     8  8
------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]>
```

In the previous examples, the very last line is the **command line**. Lisp has a command line where you type in things to execute. Here are the command lines in Lispworks and in clisp.

```
[SBCL running on your laptop...]
*
```

```
[LispWorks running on mason / osf1...]
CL-USER 4 >
```

```
[CLISP on zeus...]
[1]>
```

Think of the Lisp command line like the command line in a Unix shell or at a DOS prompt. Pressing Control-C in a Unix shell or at a DOS prompt halts the current running process and returns you to the command line. Similarly, pressing Control-C in Lisp halts whatever is presently running and returns you to the command line.

After you press Control-C, the command line changes to a "subsidiary" command line to reflect that you are in a **break** or **error condition**. Kinda like pressing Control-C in a debugger.

These conditions can be stacked: if you keep working while in a condition, and then get in *another condition* and so on, you're piling up

```
[SBCL running on your laptop...]
* (loop)                          [Press Return at this point,
                                  and you go into an infinite loop]

                                 [Now press Control-C, and you get...]
debugger invoked on a SB-SYS:INTERACTIVE-INTERRUPT in thread
#<THREAD "main thread" RUNNING {1002D8E743}>:
Interactive interrupt at #x10049770C0.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):
0: [CONTINUE] Return from SB-UNIX:SIGINT.
1: [ABORT   ] Exit debugger, returning to top level.

((FLET SB-UNIX::RUN-HANDLER :IN SB-SYS:ENABLE-INTERRUPT) 13
#.(SB-SYS:INT-SAP #X051FF658) #.(SB-SYS:INT-SAP #X051FF6C0))

0] :top

*
```

```
[LispWorks running on mason / osf1...]
```

conditions on a stack.

Just like in a debugger, at any break or error condition, you have a bunch of options (like examining the stack, changing what the return value should be, etc.) You can even continue the infinite loop we just broke out of. But you probably just want to escape. The easiest option is to escape out of all of your error conditions, right back up to the top.

- In SBCL on your laptop, this is done by typing **:top** or ***:n*** (where *n* is the largest number presented to you -- here it's 1).
- In LispWorks on OSF1/MASON, it's similar: type **:top** or **:c *n*** (where *n* is the largest number presented to you -- here it's 3).
- In clisp on zeus, you'd type **:R1**

```
CL-USER 4 > (loop)                [Press Return at this point,
                           and you go into an infinite loop]

                          [Now press Control-C, and you get...]
Keyboard break to NIL.
1 (continue) Return from break.
2 (abort) Return to level 0.
3 Restart top-level loop.

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :? for other options.

CL-USER 5 : 1 > :top

CL-USER 6 >
```

```
[CLISP on zeus...]
[1]> (loop)                       [Press Return at this point,
                           and you go into an infinite loop]

                          [Now press Control-C, and you get...]
** - Continuable Error
EVAL: User break
If you continue (by typing 'continue'): Continue execution
The following restarts are also available:
ABORT          :R1      Abort main
Break 1 [2]> :R1
[3]>
```

You can quit your lisp session by getting to the command line (possibly through pressing Control-C), **then** typing **(quit)** and pressing return. Here are examples using Lispworks or using clisp.

```
[SBCL running on your laptop...]
* (quit)
laptop>
```

```
[LispWorks running on mason / osf1...]
CL-USER 4 > (quit)
>
```

```
[CLISP on zeus...]
[1]> (quit)
Bye.
zeus>
```

Evaluating Simple Expressions

From now on, we will only use examples in **clisp**. But it works basicallly the same on all Lisp systems.

An **expression** is something that is **evaluated**, by which we mean, submitted to Lisp and executed. All things that are evaluated will return a **value**. If you type in an expression at the command line, Lisp will print its value before it returns the command line to you.

Numbers are expressions. **The value of a number is itself.** Lisp can represent a great many kind of numbers: integers, floating-point expressions, fractions, giant numbers, complex numbers, etc.

A fraction is straightforwardly represented with the form **x/y** Note that here the / does *not* mean "divide".

```
[3]> -3
-3
[4]> 2.43
2.43
[5]> 12334230392341232341132323401292349234123123023492341023923441231233423039234123234113232340129234923412312302349234102392344123
[6]> #C(3.2 2)              [the complex number 3.2 + 2i ]
#C(3.2 2)
[7]> 2/3                    [the fraction 2/3.  NOT "divide 2 by 3"]
2/3
[8]> -3.2e25                [the number -3.2 x 10^25]
-3.2E25
[9]>
```

A complex number *a*+*b*i takes the form **#C(*a b*)**

Individual **characters** are expressions. Like a number, the value of a character is itself. A character is represented using the #\ syntax. For example, **#\A** is the character 'A'. **#\%** is the character '%'.

Control characters have very straightforward formats:

#\tab
#\newline
#\space
#\backspace
#\escape

(etc.)

```
[3]> #\g
#\g
[4]> #\{
#\{
[5]> #\space
#\Space
[6]> #\newline
#\Newline
[7]> #\\       [The character '\']
#\\
[8]>
```

Strings are expressions. Just like a numbers and characters, the value of a string is itself.

A Lisp string is a sequence of characters. Lisp strings begin and end with double-quotes. Unlike in C++ (but like Java) a Lisp string does **not** terminate with a **\0**.

Like C++ and Java, Lisp strings have an escape sequence to put special characters in the string. The escape sequence begins with the backslash \ . To put a double-quote in the middle of a string, the sequence is **\"** To put a backslash in the middle of a string, the sequence is \\ Lisp tries to return values in a format that could be typed right back in again. Thus, it will also print return values with the escape sequences shown.

Unlike C++, you do not normally add returns and tabs to strings using an escape sequence. Instead, you just type the tab or the return right in the string itself.

```
[14]> "Hello, World!"
"Hello, World!"
[15]> "It's a glorious day."
"It's a glorious day."
[16]> "He said \"No Way!\" and then he left."
"He said \"No Way!\" and then he left."
[17]> "I think I need a backslash here:  \\  Ah, that was better."
"I think I need a backslash here:  \\  Ah, that was better."
[18]> "Look, here are      tabs      and some

returns!

Cool, huh?"
"Look, here are      tabs      and some

returns!

Cool, huh?"
[19]>
```

In Lisp, the special constant **nil** (case insensitive) all by itself represents "false". **nil** evaluates to itself.

Every other expression but nil is considered to be "true". However, Lisp also provides an "official" constant which represents "true", for your convenience. This is **t** (also case-insensitive). **t** also evaluates to itself.

```
[3]> t
T
[4]> nil
NIL
[5]>
```

Evaluating Lists as Functions

Lisp program code takes the form of **lists**. A list begins with a parenthesis, then immediately contains a **symbol**, then zero or more expressions separated with whitespace, then a closing parenthesis.

We'll discuss the format of symbols further down. In the examples at right, + and * are symbols, and denote the addition and multiplication functions respectively.

```
[14]> (+ 3 2 7 9)       [add 3+2+7+9 and return the result]
21
[15]> (* 4 2.3)     [multiply 4 by 2.3 and return the result]
9.2
```

Like everything else in Lisp, **lists are expressions**.

This means that lists return a value when evaluated.

An **atom** is **every expression that is not a list**. Among other things, strings and numbers and boolean values are atoms.

When Lisp evaluates a list, it first examines (but **does not evaluate**) the symbol at the beginning of the list. Usually this symbol is associated with a **function**. Lisp looks up this function.

Then each expression in the list (except the beginning symbol) is evaluated **exactly once, *usually* (but not necessarily) left-to-right**.

The values of these expressions are then passed in as parameters to the function, and the function is called. The list's return value is then the value returned by the function.

```
[14]> (+ 3 2)     [Look up the + function, evaluate
          3 and 2 (numbers evaluate to themselves),
          then pass their values (3 and 2) into the
          + function, which returns 5,
          which is then returned].
5
[15]> (subseq "Hello, World" 2 9)     [Look up the subseq function,
                                 evaluate "Hello, World", 2,
                                 and 9 (they evaluate to
                                 themselves), then pass their
                                 values in as arguments.  The
                                 subseq function will return
                                 the substring in "Hello, World"
                                 starting at character #2 and
                                 ending just before character #9.
"llo, Wo"
```

A **symbol** is a series of characters which typically do not contain whitespace, parentheses ( ), pound (#), quote ('), double-quote ("), period (.), or backquote (`), among a few others. Symbols generally don't take the form of numbers. It's very common for symbols to have hyphens (-) or asterisks (*) in them -- that's perfectly fine. **Symbols are case-INSENSITIVE**. Here are some interesting symbols:

\+ * 1+ / string-upcase
reverse length sqrt

Guess what function is associated with each of these symbols.

In C++ and in Java, there are operators like `+`, `<<`, `&&`, etc. But in Lisp, there are no operators: instead, there are only functions. For example, + is a function.

```
[23]> (+ 27/32 32/57)
2563/1824
[24]> (* 2.342 3.2e4)
74944.0
[25]> (* 2.342 9.212 -9.23 3/4)      [You can mix number types]
-149.34949
[26]> (/ 3 5)
3/5                  [The return type stays as general as possible]
[27]> (/ 3.0 5)
0.6                  [Here Lisp had no choice: convert to a float]
[28]> (1+ 3)
4
[29]> (string-upcase "How about that!")
"HOW ABOUT THAT!"
[30]> (reverse "Four score and seven years ago")
"oga sraey neves dna erocs ruoF"
[31]> (length "Four score and seven years ago")
30
[32]> (sqrt 2)
1.4142135
[33]> (sqrt -1.0)
#C(0 1.0)
[34]> (SqRt -1.0)     [Lisp symbols are case-insensitive]
#C(0 1.0)
```

While some functions require a fixed number of arguments, other ones (like + or *) can have any number of arguments.

```
[23]> (+ 100 231 201 921 221 231 -23 12 -34 134)
1994
```

Other functions have a fixed number of arguments, plus an *optional argument* at the end.

For example, **subseq** takes a string followed by one *or* two numbers. If only one number *i* is provided, then **subseq** returns the substring starting a position *i* in the string and ending at the end of the string.

If two numbers *i* and *j* are provided, then **subseq** returns the substring starting a position *i* in the string and ending at position *j*.

```
[23]> (subseq "Four score and seven years ago" 9)
"e and seven years ago"
[24]> (subseq "Four score and seven years ago" 9 23)
"e and seven ye"
```

Lisp has a special name for functions which return "true" (usually **t**) or "false" (**nil**). These functions are called **predicates**. Traditionally, many Lisp predicate names end with a **p**. Here are some predicates.

```
[5]> (= 4 3)               [is 4 == 3 ? ]
NIL
[6]> (< 3 9)               [is 3 < 9 ?]
T
[7]> (numberp "hello")     [is "foo" a number?]
NIL
[8]> (oddp 9)              [is 9 an odd number?]
T
[9]>
```

When an expression is evaluated which generates an error, Lisp breaks and returns to the command prompt with a break sequence, just like what happens when you press Control-C.

```
[26]> (/ 1 0)

*** - division by zero
1. Break [27]> :a       [clisp's way of exiting a break sequence]
[28]>
```

Errors can also occur if there is no function associated with a given symbol in a list.

```
[26]> (blah-blah-blah 1 0 "foo")

*** - EVAL: the function BLAH-BLAH-BLAH is undefined
1. Break [27]> :a
[28]>
```

When a list contains another list among its expressions, **the evaluation procedure is recursive**. The example at left thus does the following things:

1. The + function is looked up.
2. 33 is evaluated (its value is 33).
3. `(* 2.3 4)` is evaluated:
    1. The * function is looked up.
    2. 2.3 is evaluated (its value is 2.3)
    3. 4 is evaluated (its value is 4)
    4. 2.3 and 4 are passed to the * function.
    5. The * function returns 9.2. This is the value of `(* 2.3 4)`.
4. 9 is evaluated (its value is 9).
5. 33, 9.2, and 9 are passed to the + function.
6. The + function returns 51.2. This is the value of `(+ 33 (* 2.3 4) 9)`.
7. The Lisp system returns 51.2.

```
[44]> (+ 33 (* 2.3 4) 9)
51.2
[45]>
```

Here are some more examples.

Now you see how easy it is to get lost in the parentheses!

```
[44]> (+ (length "Hello World") 44)
55
[45]> (* (+ 3 2.3) (/ 3 (- 9 4)))    [in C++: (3+2.3) * (3 / (9-4)) ]
3.1800003
[46]> (log (log (log 234231232234234123)))
1.3052895
[47]> (+ (* (sin 0.3)
            (sin 0.3))          [expressions may use multiple lines]
         (* (cos 0.3)
            (cos 0.3)))         [ sin(0.3)^2 + cos(0.3)^2 ]
1.0000001                               [ = 1.  Rounding inaccuracy]
[48]> (and (< 3 (* 2 5))
           (not (>= 2 6)))      [ (3 < 2 * 5) && !(2 >= 6) ]
T
[49]>
```

One particularly useful function is **print**, which takes the form **(print *expression-to-print*)**. This function evaluates its argument, then prints it, then **returns the argument.**

As can be seen at right, if you just use **print** all by itself, the screen will appear to print the element twice. Why is that? It's because **print** printed its argument, then returned it, and Lisp always prints [again] the final return value of the expression.

One nice use of **print** is to stick it in the middle of an expression, where it will print elements without effecting the final return value of the whole expression.

```
[41]> (print (+ 2 3 4 1))
10
10
[42]> (print "hello")
"hello"
"hello"
[43]> (+ (* 2 3) (/ 3 2) 9)
33/2
[44]> (+ (print (* 2 3)) (print (/ 3 2)) 9)
6
3/2
33/2
[45]>
```

Control Structures and Variables

There are some evaluatable lists which are **not functions** because they do not obey the function rule ("evaluate each argument exactly one time each"). These lists are known as **macros** or **special forms**. For now we will not distinguish between these two terms, though there is a massive difference underneath.

Macros and special forms are mostly used as control structures. For example, the control structure **if** is a special form. **if** takes the form:

**(if *test-expression*
then-expression
optional-else-expression)**

if evaluates *test-expression*. If this returns true, then **if** evaluates and returns *then-expression*, else it evaluates and returns *optional-else-expression* (or if *optional-else-expression* is missing, returns nil).

Because **if** is an expression, unlike most languages it's quite common to see it embedded inside other expressions (like the last expression at right). This is roughly equivalent to C's ***i?j:k*** expression form.

Why can't **if** be a function? Because it may not necessarily evaluate the *then-expression*, or if it does, it will not evaluate the *optional-else-expression*. Thus it violates the function rule.

```
[44]> (if (<= 3 2) (* 3 9) (+ 4 2 3))   [if 3<=2 then return 3*9
                                          else return 4+2+3]
9
[45]> (if (> 2 3) 9)    [if 2>3 then return 9 else return nil]
NIL
[46]> (if (= 2 2) (if (> 3 2) 4 6) 9)      [if 2==2,
                                        then if 3>2,
                                               then return 4
                                               else return 6
                                        else return 9]
4
[47]> (+ 4 (if (= 2 2) (* 9 2) 7))    [NOTE: the 'if' evaluates to 18!]
22
```

if only allows one test-expression, one then-expression, and one optional-else-expression. What if you want to do three things in the then-expression? You need to make a **block** (a group of expressions executed one-by-one). Blocks are made with the special form **progn**, which takes the form:

(progn *expr1 expr2 expr3 ...*)

progn can take any number of expressions, and evaluates each of its expressions in order. **progn** then returns the value of the last expression.

```
[44]> (if (> 3 2)
   (progn (print "hello") (print "yo")
          (print "whassup?") 9)
   (+ 4 2 3))
"hello"
"yo"
"whassup?"
9
```

Except when they're at the head of a list, **symbols are also expressions**. When it's not the head of a list, a symbol **represents a variable**. When evaluated, a symbol will return the value of a variable.

The value of a symbol's variable **has nothing to do** with the function, special form, or macro associated with the symbol. You can thus have variables called **print**, **if**, etc.

Variables are set with the macro **setf**. For now, as far as you're concerned, this macro looks

```
[27]> (setf x (* 3 2))
6
[28]> x
6
[29]> (setf y (+ x 3))
9
[30]> (* x y)
54
[31]> (setf sin 9)     [ you really can do this! ]
9
[32]> (sin sin)        [ huh! ]
0.4121185
[33]> z                [ z not set yet ]
```

like this:

(setf *variable-symbol expression*)

setf is a macro and not a function because it does not evaluate *variable-symbol*. Instead, it just evaluates *expression*, and stores its value in the variable associated with *variable-symbol*. Then it returns the value of *expression*.

If a symbol is evaluated before anything has been stored in its variable, it will generate an error.

Be careful with **setf**. Lisp doesn't need to declare variables before they are used. Therefore, unless variables are declared to be local (discussed later), **setf** will make **global variables**. And **setf** is the first operation we've seen with side effects -- so the order of operations will matter! See the example at right.

```
*** - EVAL: variable Z has no value
1. Break [34]> :a

[Keep in mind that + is a function, so in most lisp systems it evaluates its
arguments left-to-right.  So x is evaluated -- returning 6; then
(setf x 3) is evaluated, which sets x to 3 and returns 3; then
x is evaluated -- and now it returns 3.  So + will return 6+3+3]
[35]> (+ x (setf x 3) x)
12
[Just like in C++/Java:   x + (x = 3) + x ]
```

Because special forms and macros don't obey the function rule, they can take whatever syntax they like. Here is **let**, a special form which declares local variables:

(let (*declaration1 declaration2* ...)
expr1
expr2
...)

let declares local variables with each *declaration*. Then it evaluates the expressions in order (as a block). These expressions are evaluated in the context of these local variables (the expressions can see them). **let** then gets rid of the local variables and returns the value of the **last expression**. Thus the local variables are only declared within the **scope** of the **let expression**.

A declaration takes one of two forms:

var	A symbol representing the variable. It is initialized to **nil**.
(*var expr*)	A list consisting of the variable symbol followed by an expression. The expression is evaluated and the variable is initialized to that value.

You can use **setf** to change the value of a local variable inside a **let** statement. You can also nest **let** statements within other **let** statements. Locally declared variables may shadow outer local and global variables with the same name, just as is the case in C++ and in Java.

```
[1]> (setf x 4)             [x set globally]
4
[2]> (let ((x 3))           [x declared local]
(print x)
(setf x 9)          [the local x is set]
(print x)
(print "hello"))    [Why does "hello" print twice? Think.]
3
9
"hello"
"hello"
[3]> x               [outside the let, we're back to global again]
4
[4]> (let ((x 3) (y (+ 4 9)))    [declare x and y locally]
(* x y))
39
[5]> (let ((x 3))              [declare x locally]
(print x)
(let (x)               [declare x locally again (nested)]
   (print x)
   (let ((x "hello"))  [declare x locally again! (nested)]
      (print x))
   (print x))
(print x)
(print "yo"))          [Why does "yo" print twice?]
3
NIL
"hello"
NIL
3
"yo"
"yo"
```

Another reason a list might be a special form or macro is because it **repeatedly evaluates** its arguments. One example is **dotimes**. This macro is an iterator (a looping control structure). Like most iterators in Lisp, **dotimes** requires a variable. Here's the format:

```
[26]> (setf x 3)
3
[27]> (dotimes (x 4 "yo") (print "hello"))
"hello"
"hello"
"hello"
"hello"
"yo"
```

```
(dotimes (var high-val
         optional-return-val)
 expr1
 expr2
 ...)
```

Here, dotimes first evaluates the expression *high-val*, which should return a positive integer. Then it sets the variable *var* (which is a symbol, and is **not** evaluated) to 0. Then it evaluates the zero or more expressions one by one. Then it increments *var* by 1 and reevaluates the expressions one by one. It does this until *var* reaches *high-val*. At this time, *optional-return-val* is evaluated and returned, or nil is returned if *optional-return-val* is missing.

You don't need to declare the dotimes variable in an enclosing let -- dotimes declares the variable locally for you. The dotimes variable is local only to the dotimes scope -- when dotimes exits, the variable's value resumes its previous setting (or none at all).

```
[28]> x
3                              [x was local in dotimes]
[29]> (setf bag 2)
2
[30]> (dotimes (x 6) (setf bag (* bag bag)))
NIL                            [No return expression was given]
[31]> bag
18446744073709551616           [ Understand why? ]
```

Writing Functions

In Lisp, functions are created by calling a **function-making macro**. This macro is called **defun**.

A simple version of **defun** takes the following general form:

```
(defun function-name-symbol
    (param1 param2 param3 ...)
expr1
expr2
expr3
... )
```

defun builds a function of zero or more arguments of the local-variable names given by the parameter symbols, then evaluates the expressions one by one, then returns the value of the last expression. The name of the function is the function-name-symbol. **defun** defines the function, sets it to this symbol, then returns the symbol -- you rarely use the return value of **defun**.

At right is a really simple example: a function of no arguments which simply returns the string "Hello, World!".

```
[44]> (defun do-hello-world ( )
  "Hello, World!")  ["Hello, World!" is last expression]
DO-HELLO-WORLD
[45]> (do-hello-world)      [ No arguments ]
"Hello, World!"
```

Here are some examples with one, two, and three arguments but just one expression.

```
[44]> (defun add-four (x)
  (+ x 4))
ADD-FOUR
[45]> (add-four 7)
11
[46] (defun hypoteneuse (length width)
  (sqrt (+ (* length length)
           (* width width))))
HYPOTENEUSE
[47]> (hypoteneuse 7 9)
11.401754
[48]> (defun first-n-chars (string n reverse-first)
  (if reverse-first      [if reverse-first is "true" ]
```

```
    (subseq (reverse string) 0 n)
    (subseq string 0 n)))

FIRST-N-CHARS
[49]> (first-n-chars "hello world" 5 nil)
"hello"
[50]> (first-n-chars "hello world" 5 t)
"dlrow"
[51]> (first-n-chars "hello world" 5 18)       [ 18 is "true"! ]
"dlrow"
```

Here are some examples with several expressions in the function Remember, the function returns the value of the last expression.

```
[44]> (defun print-string-stuff (string-1)
  (print string-1)
  (print (reverse string-1))
  (print (length string-1))
  string-1)  [ string-1 is returned ]
PRINT-STRING-STUFF
[45]> (print-string-stuff "Hello, World!")
"Hello, World!"
"!dlroW ,olleH"
13
"Hello, World!"
[46] (setf my-global-counter 0)
0
[47] (defun increment-global-and-multiply (by-me)
  (setf my-global-counter (1+ my-global-counter))
  (* my-global-counter by-me))
INCREMENT-GLOBAL-AND-MULTIPLY
[48]> (increment-global-and-multiply 3)
3
[49]> (increment-global-and-multiply 5)
10
[50]> (increment-global-and-multiply 4)
12
[51]> (increment-global-and-multiply 7)
28
```

Lisp functions can have local variables, control structures, whatnot. Try to use local variables rather than global variables! Declare local variables with **let**.

```
[ In C++: long factorial (long n) {
      long sum = 1;
      for (int x=0;x<n;x++)
          sum = sum * (1 + x);
      return sum; }              ]
[44]> (defun factorial (n)
  (let ((sum 1))
    (dotimes (x n)
      (setf sum (* sum (1+ x))))
    sum))
FACTORIAL
 [ ... but try doing *this* with C++ :-)  ]
[45]> (factorial 1000)
4023872600770937735437024339230039857193748642107146325437999
1042993851239862902059204420848696940480047998861019719605863
1666872994808558901323829669944590997424504087073759918823627
7271887325197795059509952761208749754624970436014182780946464
9629105639388743788648733711918104582578364784997701247663288
9835955735432513185323958463075557409114262417474349347553428
6465766116677973966688202912073791438537195882498081268678383
7455973174613608537953452422158659320192809087829730843139284
4403281231558611036976801357304216168747609675871348312025478
5893207671691324484262361314125087802080002616831510273418279
7770478463586817016436502415369139828126481021309276124489635
9928705114964975419909342221566832572080821333186116811553615
8365469840467089756029009505376164758477284218896796462449451
6076535340819890138544248798495995331910172335555660213945039
9736280750137837615307127761926849034352625200015888535147331
6117021039681759215109077880193931781141945452572238655414610
6289218796022383897147608850627686296714667469756291123408243
9208160153780889893964518263243671616762179168909779911903754
0312746222899880051954444142820121873617459926429565817466283
0295557029902432415318161721046583203678690611726015878352075
1516284225540265170483304226143974286933061690897968482590125
4583271682264580665267699586526822728070757813918581788896522
0816434834482599326604336760176999612831860788386150279465955
131156552036093988180612138558600301435694527224206344631797
```

```
460594682573103790084024432438465657245014402821885252470935 1
906209290231364932734975655139587205596542287497740114133469 6
271542284586237738753823048386568897646192738381490014076731 0
446640259899490222221765904339901886018566526485061799702356 1
938970178600408118897299183110211712298459016419210688843871 2
185564612496079872290851929681937238864261483965738229112312 5
024186649353143970137428531926649875337218940694281434118520 1
580141233448280150513996942901534830776445690990731524332782 8
826986460278986432113908350621709500259738986355427719674282 2
248757586765752344220207573630569498825087968928162753848863 3
969099598262809561214509948717012445164612603790293091208890 8
694202851064018215439945715680594187274899809425474217358240 1
063677404595741785160829230135358081840096996372524230560855 9
037006242712434169090041536901059339838357779394109700277534 7
200000000000000000000000000000000000000000000000000000000000 0
000000000000000000000000000000000000000000000000000000000000 0
000000000000000000000000000000000000000000000000000000000000 0
000000000000000000000000000000000000000000000000000000000000 0
000000
```

Actually, it is surprisingly rare in Lisp to have more than one expression in a function. Instead, expressions tend to get nested together. Lisp functions tend to take on **functional** form rather than **declarative** form. In C++ or Java, usually you set local variables a lot. In Lisp you don't -- you nest functions.

```
[ a declarative style -- yuck ]
[44]> (defun my-equation (n)
  (let (x y z)
    (setf x (sin n))
    (setf y (cos n))
    (setf z (* x y))
    (+ n z)))
MY-EQUATION
 [ ... a functional style  ]
[45]> (defun my-equation (n)
  (+ n (* (sin n) (cos n))))
MY-EQUATION
```

Like Java, Lisp is pass-by-value. The parameters of a function are considered to be local variables to that function, and can be set with **setf**. This does not change the values of things passed in.

```
[44]> (defun weird-function (n)
    (setf n 4)
    n)
WEIRD-FUNCTION
[45]> (setf abc 17)
17
[46]> (weird-function abc)
4
[47]> abc
17
```

You can also make recursive functions. Lisp style often makes heavy use of recursion.

You'll find that functional style and recursion together result in a need for very few local variables.

Here's the factorial function again, only done recursively.

```
[44]> (defun factorial (n)
  (if (<= n 0)
    1
    (* n (factorial (- n 1)))))

FACTORIAL
```

You can make functions with an optional argument using the special term **&optional**, followed by the optional parameter name, at the end of your parameter list.

If the optional parameter isn't provided when the function is called, then the parameter is set to **nil**.

Alternatively you can provide the default value to set the parameter to when it's not provided when the function is called. You can do this by following **&optional** not by a parameter name but by a list of the form **(*param-name default-value*)**

You can have only one optional parameter.

```
[48]> (defun first-n-chars (string n &optional reverse-first)
  (if reverse-first
    (subseq (reverse string) 0 n)
    (subseq string 0 n)))

REVERSE-FIRST
[49]> (first-n-chars "hello world" 5 nil)
"hello"
[50]> (first-n-chars "hello world" 5)  [ nil is default]
"hello"
[51]> (first-n-chars "hello world" 5 t)
"dlrow"
[52]> (defun multiply-then-maybe-add (x y &optional (z 0) )
  (+ (* x y) z))
MULTIPLY-THEN-MAYBE-ADD
[53]> (multiply-then-maybe-add 9 2)
18
[54]> (multiply-then-maybe-add 9 2 7)
25
```

Lisp can also have **keyword parameters**. These are parameters which can appear or not appear, or be in any order, because they're given names. Keyword parameters are very much like the <foo arg1=val arg2=val ... > arguments in the "foo" html tag.

Keyword parameters appear at the end of a parameter list, after the term **&key**. Similarly to optional arguments, each keyword parameter is either a parameter name (whose value defaults to **nil** if not passed in when the function is called) or is a list of the form ***(param-name default-value)***

Keyword parameters may appear only at the end of the parameter list.

You pass a keyword parameter whose name is **foo** into a function by using the term **:foo** followed by the value to set **foo** to. Keyword parameters can be passed in in any order, but must appear at the end of the parameter list.

Though it's possible to have both keyword parameters and optional parameters in the same function, don't do it. Gets confusing.

Many built-in Lisp functions use lots of keyword parameters to "extend" them!

```
[48]> (defun first-n-chars (string n
         &key reverse-first  nil by default
              (capitalize-first t) )   t by default
  (let ((val (if capitalize-first
                    (string-upcase string)
                    string)))
    (if reverse-first
      (subseq (reverse val) 0 n)
      (subseq val 0 n))))
 [ take a while to understand the LET before going on... ]

FIRST-N-CHARS
[49]> (first-n-chars "hello world" 5 :reverse-first t)
"DLROW"
[50]> (first-n-chars "hello world" 5
    :reverse-first t :capitalize-first nil)
"dlrow"
[51]> (first-n-chars "hello world" 5
    :capitalize-first nil :reverse-first t )
"dlrow"
[52]> (first-n-chars "hello world" 5)
"HELLO"
[53]> (first-n-chars "hello world" 5 :capitalize-first nil)
"hello"
```

Lists and Symbols as Data

Lists are normally evaluated as function or macro calls. Symbols are normally evaluated as variable references. But they don't have to be. Lists and symbols are data as well!

The special form **quote** can be used to *bypass the evaluation of its argument*. **quote** takes a single argument, and instead of evaluating that argument, it simply returns the argument as you had typed it ... as data!

```
[48]> (quote (hello world 1 2 3))

(HELLO WORLD 1 2 3)
[49]> (quote (what is (going on) here?))
(WHAT IS (GOING ON) HERE?)
[50]> (quote my-symbol)
MY-SYMBOL
[51]> (quote (+ 4 (* 3 2 9)))
(+ 4 (* 3 2 9))
```

What is a symbol when used in data form? It's just itself. The symbol **foo** is just a thing that looks like **foo** (case insensitive of course). It is a data type like any other. You can set variables to it.

What is a list when used in data form? A list is a **singly-linked list**. It is a data type like any other. You can set variables to it. There are a great many functions which operate on lists as well.

first returns the first item in a list. The old name of **first** is **car**.

rest returns a list consisting of everything *but* the first item. It does not damage the original list. The old name of **rest** is **cdr**.

append hooks multiple lists together.

cons takes an item and a list, and returns a new list consisting of the old list with the item tacked on the front.

```
[48]> (setf my-variable (quote hello))

HELLO
[49]> my-variable    [ stores the symbol HELLO ]
HELLO
[50]> (setf my-variable (quote (hey yo yo)))
(HEY YO YO)
[51]> my-variable
(HEY YO YO)
[52]> (setf var2 (first my-variable))
HEY
[53]> (setf var3 (rest my-variable))
(YO YO)
[54]> (cons 4 (rest my-variable))
(4 YO YO)
[55]> (append my-variable (quote (a b c)) my-variable)
(HEY YO YO A B C HEY YO YO)
[56]> my-variable
(HEY YO YO)     [ See?  No damage ]
[57]> (quote "hello")
"hello"     [ makes no difference ]
[58]> (quote 4.3)
4.3     [ makes no difference ]
```

quote is so common that there is a special abbreviation for it - a single

quote is so common that there is a special abbreviation for it...a single quote at the beginning of the item:

'hello-there

...is the same as...

(quote hello-there)

Here's how it's done for lists:

'(a b c d e)

...is the same as...

(quote (a b c d e))

Here's a repeat of some the previous code, but with the abbreviation.

```
[48]> (setf my-variable 'hello)
HELLO
[50]> (setf my-variable '(hey yo yo))
(HEY YO YO)
[55]> (append my-variable '(a b c) my-variable)
(HEY YO YO A B C HEY YO YO)
[57]> '"hello"
"hello"     [ makes no difference ]
[58]> '4.3
4.3     [ makes no difference ]
```

Lists as data can of course contain sublists.

In data form, the first item of a list can be anything -- it's not restricted to be just a symbol.

```
[48]> '(123.32 "hello" (how are (you there)) a)
(123.32 "hello" (HOW ARE (YOU THERE)) A)
[49]> '(((wow)) a list consisting of a list of a list!)
(((WOW)) A LIST CONSISTING OF A LIST OF A LIST!)
```

nil isn't just "false". It's also the empty list, **'()**

```
[48]> '()
NIL
[49]> (rest '(list-of-one-thing))
NIL
[50]> (append '(list-of-one-thing) nil)
(LIST-OF-ONE-THING)
[51]> '(a b c () g h i)
(A B C NIL G H I)
```

Lists have a common control structure, **dolist**, which iterates over a list. The format of **dolist** is very similar to **dotimes**:

**(dolist (*var list-to-iterate-over*
optional-return-val)**
expr1
expr2
...)

dolist evaluates the list-to-iterate-over, then one by one sets *var* to each element in the list, and evaluates the expressions. **dolist** then returns the optional return value, else **nil** if none is provided.

```
[48]> (dolist (x '(a b c d e))
  (print x))
A
B
C
D
E
NIL
[49]> (defun my-reverse (list)
  (let (new-list)   [initially nil, or empty list]
    (dolist (x list)
      (setf new-list (cons x new-list)))
    new-list))
MY-REVERSE
[50]> (my-reverse '(a b c d e f g))
(G F E D C B A)
```

Lists and strings share a common supertype, **sequences**.

There are a great many sequence functions. All sequence functions work on any kind of sequence (including strings and lists). Here are two sequence functions we've seen so far.

```
[48]> (reverse '(a b c d e))
(E D C B A)
[49]> (reverse "abcde")
"edcba"
[50]> (subseq "Hello World" 2 9)
"llo Wor"
[51]> (subseq '(yo hello there how are you) 2 4)
(THERE HOW)
```

Loading and Compiling Lisp

Lisp is both an **interpreter** and a **compiler**.

```
[48]> (defun slow-function (a)
```

If you type in code at the command line, it is (on most Lisp systems) **interpreted**.

You can compile a function by passing its symbol name (quoted!) to the **compile** function.

You can time the speed of any expression, and its garbage collection, with the **time** function.

```
(dotimes (x 100000)
  (setf a (+ a 1)))
a)

SLOW-FUNCTION
[49]> (time (slow-function 0))

Real time: 1.197806 sec.
Run time: 1.15 sec.
Space: 0 Bytes
100000

[50]> (compile 'slow-function)
SLOW-FUNCTION ;
NIL ;
NIL

[51]> (time (slow-function 0))

Real time: 0.066849 sec.
Run time: 0.07 sec.
Space: 0 Bytes
100000
```

You don't have to type all your code in on the command line. Instead, put it in a file named **"myfile.lisp"** (or whatever, so long as it ends in ".lisp"). Then load it with the **load** command.

load works exactly as if you had typed in the code directly at the command line.

By default, **load** is fairly silent -- it doesn't print out all the return values to the screen like you'd get if you typed the code in at the command line. If you'd like to see these return values printed out, you can add the **:print t** keyword parameter.

You can load and reload files to your heart's content.

```
[ Make a file called "myfile.lisp", containing this: ]
(setf foo 3)
(defun my-func ()
(print 'hello))
foo
(sin foo)
(my-func)
```

```
[ At the command line, you type: ]

[49]> (load "myfile.lisp")
;; Loading file myfile.lisp ...
HELLO      [ because we called (my-func), which printed ]
;; Loading of file myfile.lisp is finished.
T          [ load returns t ]

[ To get the return values for each item entered in:  ]

[50]> (load "myfile.lisp" :print t)
;; Loading file myfile.lisp ...
3
MYFUNC
3
0.14112
HELLO
HELLO
;; Loading of file myfile.lisp is finished.
T
```

You can also compile a whole file with the **compile-file** function.

When a file is compiled, the object file created has a **.fas** or **.fsl** or **.fasl** or **.afasl** extension. Depends on the Lisp compiler.

You load object files with the **load** function as well.

You can omit the extension (".lisp", ".afasl", etc.) from the filename, but what happens as a result is implementation-dependent. Some systems load the most recent version (either the source or the .afasl file); others may load the .afasl file always but warn you if there's a more recent .lisp file, etc. **In general, to be safe, always load the full name of the file including the extension.**

When the compiler compiles the file, one common thing it will complain of is **special variables**. For all intents and purposes, a special variable is a **global variable**. With very few exceptions, you should never use global variables when you can use local variables instead.

```
[18]> (compile-file "myfile.lisp")

Compiling file myfile.lisp ...
WARNING in function #:TOP-LEVEL-FORM-1 in line 1 :
FOO is neither declared nor bound,
it will be treated as if it were declared SPECIAL.
WARNING in function #:TOP-LEVEL-FORM-3 in lines 4..5 :
FOO is neither declared nor bound,
it will be treated as if it were declared SPECIAL.
WARNING in function #:TOP-LEVEL-FORM-4 in line 5 :
FOO is neither declared nor bound,
it will be treated as if it were declared SPECIAL.

Compilation of file myfile.lisp is finished.
The following special variables were not defined:
FOO
0 errors, 3 warnings
#P"myfile.fas" ;
3 ;
3

[19]> (load "myfile.fas")
;; Loading file myfile.fas ...
```

In our file we had declared a global variable (**foo**). Look at the warnings when we compile!	HELLO ;; Loading of file myfile.fas is finished. T

Lisp Style

Arrays and Vectors

As you can see, Lisp can get quite confusing because of the parentheses. How tedious it is reading code based on parentheses! That's why Lisp programmers don't do it. Lisp programmers don't rely much on the parentheses when reading code. Instead, they rely heavily on breaking expressions into multiple lines and indenting them in a very peculiar way. There is a "cannonical" indent format and style for Lisp. Code which adheres to the standard format can be read very rapidly by Lisp programmers who have developed a "batting eye" for this format. Important formatting rules: • Put a single space between each item in a list. • Do NOT put space between the opening parenthesis and the first item in a list. Similarly, do NOT put space between the closing parenthesis and the last item. • Never put parentheses all by themselves on lines like a C++/Java brace. Do not be afraid to pile up parentheses at the end of a line. Do NOT use simplistic editors like pico or Windows Notepad. You will regret it. Deeply. **Use an editor designed for Lisp.** Integrated Lisp systems (the big three are Franz Allegro Common Lisp, Xanalys Harlequin Common Lisp, and Macintosh Common Lisp) with graphical interfaces have built-in editors which will automatically indent text for you in the official style, will colorize your text, will tell you whether your syntax is right or not, and will match parentheses for you. Another good choice, indeed the classic option in Lisp systems, is the editor **emacs**. It is written in its own version of Lisp, and is very good at editing Lisp code and working with Lisp systems, especially with an add-on Lisp-editing plug-in called **slime**. emacs is the program whose auto-indent facilities established the "cannonical" style of Lisp formatting. If you can't find an editor which can do the cannonical style, there are still plenty of choices which do a reasonable job. Any professional-grade code editor will do in a pinch. Without a good editor, writing large Lisp programs is **painful**. GET A CODE EDITOR. You are an adult now, and will soon be a professional. Use real tools to get your job done.	[BAD Lisp Style Formatting Examples] (if(< (* 3 4)5)(sin(+ 3 x)) (- x y)) (If ((* 3 4) 5) (SIN (+ 3 x)) (- x y)) [A reasonably GOOD Lisp Style Formatting Example] (if (< (* 3 4) 5) (sin (+ 3 x)) (- x y)) [A more canonical Lisp Indent Format] (if (< (* 3 4) 5) (sin (+ 3 x)) (- x y))
Comments in Lisp are of three forms.	

Winged comments (the equivalent of /* and */ in C++ or Java) begin with a #| and end with a |# They are not commonly used in Lisp except to temporarily eliminate chunks of code, because it's hard to tell they exist by examining your code.

Inline comments (the equivalent of // in C++ or Java) begin with a semicolon **;** and end with a return.

Many Lisp structures have built-in documentation comments. For example, if the first expression in a **defun** statement is a string, that string is not part of the code but instead is considered to be the "documentation" for the function. You can access the documentation for an object with the **documentation** function.

It is common in Lisp to pile up several semicolons **;;** or **;;;** to make the comment more visible.

Here is a common approach:

- Use one semicolon for inline code.
- Use two semicolons to comment the head of a function.
- Use three semicolons to comment the head of a file or other big region.
- Use winged comments only to comment-out a region temporarily.

```
[A well-commented file]
;;; pi-estimation package
;;; Sean Luke
;;; Wednesday, 8/21/2002



;; ESTIMATE-PI will compute the value of pi to
;; the degree given, maintaining the value as a giant
;; fraction.  It uses the Leibniz (1674)
;; formula of pi = 4 * ( 1/1 - 1/3 + 1/5 - 1/7 + ...  )
;; degree must be an integer > 0.

(defun estimate-pi (degree)
  "Estimates pi using Leibniz's formula.
degree must be an integer greater than 0."
  (let ((sum 0) (inc 1))                ; inc goes 1, 5, 7, ...
    (dotimes (x degree (* 4 sum))       ; we return 4*sum
         #| (setf sum (+ sum (/ 1 inc))
                  (- 0 (/ 1 (+ inc 2)))) |#     ; yucky
      (setf sum (+ sum (/ 1 inc) (/ -1 (+ inc 2))))
      (setf inc (+ 4 inc)))))
```

```
[...after estimate-pi has been entered into Lisp...]

[13]> (documentation 'estimate-pi 'function)
"Estimates pi using Leibniz's formula.
degree must be an integer greater than 0."
[14]> (describe 'estimate-pi)
[Get ready for more information than you really need!]

ESTIMATE-PI is the symbol ESTIMATE-PI, lies in
#<PACKAGE COMMON-LISP-USER>, is accessible in
the package COMMON-LISP-USER, names a function,
has the properties SYSTEM::DOCUMENTATION-STRINGS,
SYSTEM::DEFINITION.

Documentation as a FUNCTION:
Estimates pi using Leibniz's formula.
degree must be an integer greater than 0.
For more information, evaluate (SYMBOL-PLIST 'ESTIMATE-PI).

#<PACKAGE COMMON-LISP-USER> is the package
named COMMON-LISP-USER. It has the nicknames CL-USER,
USER.It imports the external symbols of the packages
COMMON-LISP, EXT and exports no symbols, but no
package uses these exports.

#<CLOSURE ESTIMATE-PI (DEGREE) (DECLARE #)
(BLOCK ESTIMATE-PI #)> is an interpreted function.
argument list: (DEGREE)
```

Lisp has important style rules about symbols, used for both variables and function names.

- Although Lisp symbols are case-insensitive, **ALWAYS use lower-case**. There is a good reason for this. Keep in mind that Lisp is an interactive system: both you and the system are producing text on the screen. Lisp systems spit out symbols in UPPER-CASE. By sticking with lower-case yourself, you can distinguish between the text you typed and the text the Lisp system generated.
- Do NOT use underscores in symbols. **Use hyphens.**
- Although the previous examples above didn't do it to avoid confusing you, you should always **denote global variables by wrapping them with *asterisks*.** Global variable names should also be self-explanatory.
- Variable names should be nouns.

```
[BAD Lisp Style Symbols:]
my_symbol_name
mySymbolName
MySymbolName
MY_SYMBOL_NAME


[A GOOD Lisp Style Symbol]
my-symbol-name



[A BAD Global Variable Name]
aprintf


[A GOOD Global Variable Name]
*alpha-print-format*
```

- Function names should be verbs.
- Though you can always name variables the same names as functions, it's more readable not to do so.

Lisp is a functional language. Learn to use functional style. One way you can tell you're using functional style is if you have *very* few (or even no) local variables, and rarely if ever use a global variable.

As Paul Graham says, "treat **setf** as if there were a tax on its use."

```
[HORRIBLE Lisp Style]

(defun do-the-math (x y z)
(setf w (+ x y))
(setf n (* z w))
(+ x n))


[MERELY BAD Lisp Style -- no global variables]

(defun do-the-math (x y z)
(let (w n)
        (setf w (+ x y))
        (setf n (* z w))
        (+ x n)))


[BETTER Lisp Style -- functional style]

(defun do-the-math (x y z)
(+ x (* z (+ x y))))
```

Declare your global variables once with **defparameter** before you start using them in **setf** statements.

(defparameter *var-symbol initial-value optional-documentation-string*)

Declare global constants with **defconstant**.

(defconstant *var-symbol value optional-documentation-string*)

The documentation strings can be accessed via **documentation**, and of course, **describe**.

```
[13]> (defparameter *tuning-value* 4.0
"The tuning value of the amplitude dial")
*TUNING-VALUE*
[14]> (defconstant *low-quality-pi* 3.14159
"Pi to only six digits")
*LOW-QUALITY-PI*
[15]> (documentation '*tuning-value* 'variable)
"The tuning value of the amplitude dial"
[16]> (describe '*low-quality-pi*)

*LOW-QUALITY-PI* is the symbol *LOW-QUALITY-PI*,
lies in #<PACKAGE COMMON-LISP-USER>, is accessible
in the package COMMON-LISP-USER, a constant,
value: 3.14159, has the property SYSTEM::DOCUMENTATION-STRINGS.
Documentation as a VARIABLE:
Pi to only six digits
For more information, evaluate (SYMBOL-PLIST '*LOW-QUALITY-PI*).

#<PACKAGE COMMON-LISP-USER> is the package named
COMMON-LISP-USER. It has the nicknames CL-USER, USER.
It imports the external symbols of the packages
COMMON-LISP, EXT and exports no symbols, but no
package uses these exports.

3.14159 is a float with 24 bits of mantissa (single-float).
```

Lisp has many kinds of arrays: multidimensional arrays, variable-length arrays, fixed-length simple arrays, arrays guaranteed to have certain types in them, arrays which can hold anything, etc.

Lisp arrays are created with the function **make-array**. The simplest form of this function is:

(make-array *length*)

This form makes a one-dimensional fixed-length array *length* elements long. The elements are each initialized to **nil**.

An array of this form is called a *simple-vector*. You don't just

```
[1]> (make-array 4)
#(NIL NIL NIL NIL)
[2]> #(a b c)
#(A B C)
[3]>
```

have to use **make-array** to build a simple-vector. Just as you can make a list of the symbols *a b c* by typing **'(a b c)**, you can make a simple vector of the symbols *a b c* by typing **#(a b c)**

A multidimensional array is created as follows:

(make-array *dimension-list*)

This form makes an N-dimensional fixed-length array of the dimensions given by elements in the list. The elements are each initialized to **nil**.

You can specify the initial value of the elements with the keyword **:initial-element**.

The general function for extracting the element of any array is **aref**. It takes the form:

(aref *array index1 index2 ...*)

Simple vectors have a special version, **svref**, which is slightly faster than **aref** (in fact, **aref** just calls **svref** for simple vectors):

(svref *simple-vector index*)

Lisp arrays are zero-indexed. This is just like saying (in C++/Java): **array[index1][index2]...**

Multidimensional arrays can also be specified with **#*n*A(...)**, where *n* is the number of dimensions. See the example at right.

```
[1]> (make-array '(4 3 8))  [it has to be quoted]
#3A(((NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL))
((NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL))
((NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL))
((NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)
(NIL NIL NIL NIL NIL NIL NIL NIL)))
[2]> (make-array '(2 2) :initial-element 0)
#2A((0 0) (0 0))
[3]> (setf *j* #2A((1 2 3) (4 5 6)))
#2A((1 2 3) (4 5 6))
[4]> (aref *j* 1 1)
5
[5]> (aref #(a b c d e) 3)
D
[6]> (svref #(a b c d e) 3)  [ faster ]
D
```

Vectors are one-dimensional arrays. You've already seen fixed-length vectors (known in Lisp as *simple-vectors*). Lisp also has variable-length vectors.

Variable-length vectors are created with the keywords **:adjustable** and **:fill-pointer** in the following fashion:

(make-array *length* :fill-pointer t :adjustable t)

You can have a zero-length vector. It's very common to start a variable-length array at length 0.

You can tack new stuff onto the end of a variable-length vector with the command **vector-push-extend**. You can "pop" elements off the end of the variable-length vector with **vector-pop**.

To use these functions, the vector *must* be variable-length. You cannot push and pop to a simple vector.

Multidimensional arrays can also have their sizes adjusted. We'll just leave it at that -- look it up if you're interested.

```
[1]> (setf *j* (make-array 0 :fill-pointer t :adjustable t))
#()
[2]> (vector-push-extend 10 *j*)
0
[3]> (vector-push-extend 'hello *j*)
1
[3]> *j*
#(10 HELLO)
[3]> (aref *j* 1)
HELLO
[4]> (vector-pop *j*)
HELLO
[5]> *j*
#(10)
```

A string is, more or less, a vector of characters. You can access elements with **aref**. But because a string is *not* a simple vector (oddly enough), you *cannot* use **svref**. I have no idea why.

Although string elements can be accessed via **aref**, strings have their own special function which does the same thing: **char**, which takes the form:

(char *string index*)

In most systems, the two functions are about the same speed.

```
[1]> (aref "hello world" 3)
#\l
[2]> (char "hello world" 6)
#\w
```

Setf and Friends

setf doesn't just set variables. In general, **(setf *foo bar*)** "sees to it" that *foo* will evaluate to *bar*. **setf** can "see to" an amazing number of things.

To set the value of an element in an array (I bet you were wondering about that!) you say

(setf (aref *array indices...) val*)

You can do the same trick with **svref** and **char**.

You can also use **setf** to modify lists. However, this is dangerous if you don't know what you're doing. For now, don't do it. Stick with modifying arrays and strings.

```
[1]> (setf *j* #(a b c d e))
#(A B C D E)
[2]> (setf (svref *j* 3) 'hello)
HELLO
[3]> *j*
#(A B C HELLO E)
[4]> (setf *k* (make-array '(3 3 3) :initial-element 4))
#3A(((4 4 4) (4 4 4) (4 4 4))
((4 4 4) (4 4 4) (4 4 4))
((4 4 4) (4 4 4) (4 4 4)))[4]> (vector-pop *j*)
HELLO
[5]> (setf (aref *k* 2 1 1) 'yo)
YO
[6]> *k*
#3A(((4 4 4) (4 4 4) (4 4 4))
((4 4 4) (4 4 4) (4 4 4))
((4 4 4) (4 YO 4) (4 4 4)))
[7]> (setf *l* "hello world")
"hello world"
[8]> (setf (char *l* 4) #\B)
#\B
[9]> *l*
"hellB world"
```

A variant of **setf** called **incf** does more or less the same thing as the ++ operator in C++ or Java, except that it works on all sorts of things (array slots, etc.) in addition to just variables. The form:

(incf *expression* 4)

...will see to it that *expression* evaluates to 4 more than it used to (by adding 4 to it). If you just say:

(incf *expression*)

...this by default sees to it that *expression* evaluates to 1 more than it used to.

The macro **decf** does the opposite.

```
[1]> (setf *j* #(1 2 3 4 5))
#(1 2 3 4 5)
[2]> (incf (svref *j* 3) 4)
8
[3]> *j*
#(1 2 3 8 5)
[4]> (setf *k* 4)
4
[5]> (incf *k*)
5
[6]> *k*
5
[7]> (decf *k* 100)
-95
```

Another variant of **setf** called **push** can be used to "see to it" that an expression (which must evaluate to a list) now evaluates to a list with an element tacked onto the front of it. If you say:

(push *val expression*)

...this is roughly the same as saying

(setf *expression* (cons *val expression*))

You can also "see to it" that a list has an element removed from the front of it with **pop**:

(pop *expression*)

```
[1]> (setf *j* #((a b) (c d) (e f)))   [ a simple-vector of lists ]
#((A B) (C D) (E F))
[2]> (push 'hello (svref *j* 1))
(HELLO C D)
[3]> *j*
#((A B) (HELLO C D) (E F))
[4]> (setf *k* '(yo yo ma))
(YO YO MA)
[5]> (pop *k*)
YO
[6]> *k*
(YO MA)
```

Another useful variant, **rotatef**, can be used to swap several elements.

(rotatef *expresion1 expression2 ... expressionN*)

...this is roughly the same as saying

```
(setf tempvar expression1)
(setf expression1 expression2)
...
(setf expressionN-1 expressionN)
```

```
[1]> (setf *j* #(gracias senor))
#(GRACIAS SENOR)
[2]> (setf *k* 'hello)
HELLO
[3]> (rotatef (elt *j* 0) (elt *j* 1) *k*)
NIL
[4]> *j*
#(SENOR HELLO)
[5]> *k*
GRACIAS
[6]> (setf *z* #(1 2 3 4 5))
#(1 2 3 4 5)
```

(setf *expressionN tempvar*)

A simple use of this is simply **(rotatef *expression1 expression2*)** which sees to it that the values of *expression1* and *expression2* are swapped.

```
[7]> (rotatef (elt *z* 1) (elt *z* 4))   [ swap 'em ]
NIL
[8]> *z*
#(1 5 3 4 2)
```

Function, Funcall, and Apply

In Lisp, pointers to functions are first-class data objects. They can be stored in variables, passed into arguments, and returned by other functions.

The special form **function** will return a pointer to a function. It takes the form **(function *function-symbol*)**. Notice that just like **quote**, **function** doesn't evaluate its argument -- instead it just looks up the function by that name and returns a pointer to it.

Also like **quote**, **function** is so common that there is a shorthand for it: a pound sign followed by a quote at the beginning of the function name:

#'print

...is the same as...

(function print)

Keep in mind that you can *only* get pointers to *functions*, not macros or special forms.

```
[1]> (function print)
#<SYSTEM-FUNCTION PRINT>
[2]> (function if)  ["if" isn't a function -- it's a macro]

*** - FUNCTION: undefined function IF
1. Break [3]> :a

[4]> (setf *temp* (function *))  [the "*" function]
#<SYSTEM-FUNCTION *>
[5]> *temp*
#<SYSTEM-FUNCTION *>
[6]> #'print
#<SYSTEM-FUNCTION PRINT>
[7]> (setf *temp* #'*)
#<SYSTEM-FUNCTION *>
[8]> *temp*
#<SYSTEM-FUNCTION *>
```

A common mistake among Lisp newbies is to think that variables with function pointers stored in them can be used to make a traditional function call by sticking the variable at the beginning of a list.

Remember that the first item in an evaluated list must be a *symbol* which is *not evaluated*. If a variable could be put as the first item, it would have to be evaluated first (to extract the function pointer).

Thus, Common Lisp can associate a *function* with a symbol (by using **defun**) and it can *also* associate a *value* with the same symbol as a variable (by using **setf**). A Lisp which can associate two or more different kinds of things at the same time with a symbol is called a **Lisp 2**. Common Lisp is a Lisp 2. Emacs Lisp is also a Lisp 2.

Scheme, another popular Lisp dialect, evaluates the first item in the list as a *variable*, looking up its function-pointer value. Scheme associates only one thing with a symbol: the item stored in its variable. Thus Scheme is a **Lisp 1**.

Lisp 1's are simpler and more intuitive than Lisp 2's. But it is more difficult to do certain kinds of powerful things with them, like macros. We'll get to that later on.

```
[6]> (setf *new-print* (function print))
#<SYSTEM-FUNCTION PRINT>
[7]> (*new-print* "hello world")>

*** - EVAL: the function *NEW-PRINT* is undefined
1. Break [8]> :a

[9]>
```

If you can't just call a function pointer by sticking it in the first spot in a list, how *do* you call it?

There are a great many functions and macros which use function pointers. One basic one is **funcall**. This function takes the form

(funcall *function-pointer arg1 arg2 ...)

```
[6]> (setf *new-print* (function print))
#<SYSTEM-FUNCTION PRINT>
[7]> (funcall *new-print* "hello world")>

"hello world"
"hello world"
[8]> (funcall #'+ 1 2 3 4 5 6 7)
28
[9]> (funcall #'funcall #'+ 1 2 3 4 5 6 7) [hee hee!]
```

funcall is a function which evaluates *function-pointer*, which returns a pointer to a function, then it evaluates each of the arguments, then passes the argument values into the function. **funcall** returns the value of the function.

```
28
[10]>
```

Another useful function which takes function pointers is **apply**. The simple version of this function takes the form

(apply *function-pointer list-arg*)

apply takes a function pointer, plus one more argument which **must evaluate to a list**. It then takes each element in this list and passes them as arguments to the function pointed to by *function-pointer*. **apply** then returns the value the that the function returned.

It so happens that **apply** can do one additional trick. Alternatively, **apply** can look like this: **(apply *function-pointer arg1 arg2 ... list-arg*)**

The last argument **must evaluate to a list**. Here, **apply** builds a list before passing it to the function. This list is built by taking each of the *arg1, arg2,* arguments and concatenating their values to the front of the list returned by *list-arg*. For example, in **(apply #'+ 1 2 3 '(4 5 6))**, the concatenation results in the list **'(1 2 3 4 5 6)**. Thus

(apply #'+ 1 2 3 '(4 5 6)) is the same thing as

(apply #'+ '(1 2 3 4 5 6)) which is the same thing as

(apply #'+ 1 2 3 4 5 6 '()) which of course is the same thing as

(apply #'+ 1 2 3 4 5 6 nil)

```
[6]> (apply #'+ '(1 2 3 4 5 6))
21
[7]> (apply #'+ 1 2 3 4 5 6 nil)
21
[8]> (apply #'+ 1 2 3 '(4 5 6))
21
[9]> (apply #'apply #'+ '(1 2 3 (4 5 6)))    [woo hoo!]
21
[10]> (apply #'funcall #'+ '(1 2 3 4 5 6))   [yee haw!]
21
[11]> (funcall #'apply #'+ '(1 2 3 4 5 6))   [hmmmm...]
21
```

Mapping

Lisp uses pointers to functions *everywhere*. It's what makes Lisp's built-in functions so powerful: they take optional functions which let you customize the built-in ones in special ways.

One very common use of pointers to functions is *mapping*. Mapping applies a function repeatedly over one or more lists, resulting in a new list. The most common mapping function is **mapcar**, which in a basic form looks like this:

(mapcar *function-pointer list*)

Since we're providing just one list, *function-pointer* must be a pointer to a function which can take just one argument, for example, **sqrt**.

In this form, **mapcar** repeatedly applies the function to each element in the list. The return values are then put into a list and returned.

```
[1]> (mapcar #'sqrt '(3 4 5 6 7))
(1.7320508 2 2.236068 2.4494898 2.6457512)
[2]> (mapcar (function print) '(hello there how are you))

HELLO
THERE
HOW
ARE
YOU
(HELLO THERE HOW ARE YOU)
[3]>
```

mapcar more generally looks like this:

(mapcar *function-pointer list1 list2 ...*)

If *function-pointer* points to a function which takes *N* arguments, then we must provide *N* lists.

```
[1]> (mapcar #'/ '(1 2 3 4 5) '(7 8 9 10 11))
(1/7 1/4 1/3 2/5 5/11)
[2]> (mapcar #'* '(1 2 3 4)
     '(5 6) '(7 8 9))  [one list is only 2 long]
(35 96)
[3]>
```

mapcar takes the first element out of each list and passes them as arguments to the function. **mapcar** then takes the second element out of each list and passes them as arguments to the function. And so on. **mapcar** then returns a list of the return values of the function.

If any list is shorter than the others, **mapcar** operates only up to the shortest list and then stops.

Lisp provides a number of other useful mapping functions: **map, mapc, mapcan, mapcon ...**

A related feature is *reduction*: composing a function in on itself. The basic **reduce** function looks similar to **mapcar**:

(reduce *function-pointer list*)

function-pointer must point to a function which takes exactly *two arguments*. If the elements in *list* are *a b c d*, and the function *func* is stored in the function pointer, this is the same thing as doing:

(func (func (func a b) c) d)

You can also change the order of operations with the **:from-end t** keyword argument, resulting in the ordering:

(func a (func b (func c d)))

reduce has other gizmos available. Check 'em out.

```
[Note: (expt a b) computes a ^ b  (a to the power of b) ]
[1]> (reduce #'expt '(2 3 4 5))  [((2^3)^4)^5)]
1152921504606846976
[2]> (reduce #'expt '(2 3 4) :from-end t)    [2^(3^4)]
241785163922925834941235 2
[3]>
```

Lambda and Closures

A **lambda expression** is one of the more powerful concepts in Lisp. A lambda expression is an *anonymous function*, that is one that doesn't have a name -- just a pointer to it.

Lambda expressions are created using the form:

(function (lambda (*args...*) *body...*))

Note how similar this is to **defun**:

(defun *function-name* (*args...*) *body...*)

A lambda expression builds a function just like **defun** would, except that there's no name associated with it. Instead, the lambda expression returns a pointer to the function.

Remember that **function** has a shorthand of **#'** so the lambda expression is usually written like this:

#'(lambda (*args...*) *body...*)

To make things even more confusing, Common Lisp has provided for you an actual *macro* called **lamda**, which does exactly the same thing. Thus if you really want to (but it's not good style) you can write it as just:

(lambda (*args...*) *body...*)

```
[1]> (mapcar #'(lambda (x) (print (* x 2)))
      '(1 2 3 4 5 6))

2
4
6
8
10
12
(2 4 6 8 10 12)
[2]> (reduce #'(lambda (a b) (/ a (* b b)))
       '(2 3 4))    [(2 / 3^2) / 4^2]
1/72
[3]> (funcall #'(lambda (a b) (/ a (* b b))) 9 7)
9/49    [Not too useful this one, just an example...]
```

Lambda expressions are useful when you need to pass in a quick, short, temporary function. But there is another very powerful use of lambda expressions: making **closures**.

```
[1]> (defun build-a-function (x)
#'(lambda (y) (+ x y)))
BUILD-A-FUNCTION
```

A closure is a function bundled together with its own *lexical scope*. Usually you can think of this as a closure being a function plus its own personal, private global variables.

When a function is built from a lambda expression, it is usually created in the context of some outer local variables. After the function is built, these variables are "trapped" with the lambda expression if anything in the lambda expression referred to them. Since the lambda expression is hanging on to these variables, they're not garbage collected when the local scope is exited. Instead they become private variables that only the function can see.

We can use this concept to make **function-building functions**. Consider:

```
(defun build-a-function (x)
#'(lambda (y) (+ x y)))
```

...examine this function carefully. **build-a-function** takes a value *x* and then returns a function which adds that amount *x* to things!

```
[2]> (setf *+3* (build-a-function 3))
#<CLOSURE :LAMBDA (Y) (+ X Y)>
[3]> (funcall *+3* 9)
12
[4]> (funcall *+3* 2)
5
[5]> (setf *-6* (build-a-function -6))
#<CLOSURE :LAMBDA (Y) (+ X Y)>
[6]> (funcall *-6* 21)
15
[7] (funcall *-6* (funcall *+3* 38))
35
```

Closures are also common when we need to make a quick custom function based on information the user provided. Consider:

```
(defun add-to-list (val list-of-numbers)
(mapcar #'(lambda (num) (+ val num))
  list-of-numbers))
```

...examine this function carefully as well. **add-to-list** takes a number *val* and a list of numbers. It then maps a custom function on the list of numbers. This custom function adds *val* to each one. The new list is then returned.

Notice that the lambda expression is converted into a function even though it refers to *val* **inside the lambda expression**.

Closures are examples of powerful things which **C++ simply cannot do.** Java gets there part-way. Java can do lambda expressions in the form of "anonymous classes". But it too cannot do real closures, though there are nasty hacks to work around the issue.

```
[1]> (defun add-to-list (val list-of-numbers)
(mapcar #'(lambda (num) (+ val num))
       list-of-numbers))
ADD-TO-LIST
[2]> (add-to-list 4 '(1 2 3 4 5))
(5 6 7 8 9)
[ Here's the more C++ way to do it... ]
[3]> (defun icky-add-to-list (val list-of-numbers)
(let (bag)
 (dolist (x list-of-numbers)
   (push (+ val x) bag))
 (reverse bag)))
ICKY-ADD-TO-LIST
[4]> (icky-add-to-list 4 '(1 2 3 4 5))
(5 6 7 8 9)
```

Closures also occur with **defun**. Imagine if **defun** were called *inside* a **let** statement:

```
(let ((seed 1234))
(defun rand ()
(setf seed (mod (* seed 16807) 2147483647))))
```

Here we defined a local variable called **seed**. Inside this local environment, we defined a function called **rand** which uses **seed**. When we leave the **let**, what happens to **seed**? Normally it would get garbage collected. But it can't here -- because **rand** is holding on to it. **seed** becomes a *private global variable* of the function **rand**. No one else can see it but **rand**.

You can use this for other interesting purposes. Imagine that you want to make a private bank account:

```
(let ((account 0))
(defun deposit ($$$)
(setf account (+ account $$$)))
(defun withdraw ($$$)
(setf account (- account $$$)))
(defun amount ()
account))
```

The functions **deposit**, **withdraw**, and **amount** share a common private variable called **account** that no one else can see.

This isn't much different from a Java or C++ object with a private instance variable and three methods. Where did you think object-oriented programming came from? You got it.

```
[1]> (let ((seed 1234))
(defun rand ()
(setf seed (mod (* seed 16807) 2147483647))))
RAND
[2]> (rand)
20739838
[3]> (rand)
682106452
[4]> (rand)
895431078
[5]> seed

*** - EVAL: variable SEED has no value
1. Break [6]> :a
[7]> (let ((account 0))
(defun deposit ($$$)
(setf account (+ account $$$)))
(defun withdraw ($$$)
(setf account (- account $$$)))
(defun amount ()
account))
AMOUNT
[8]> (deposit 42)
42
[9]> (withdraw 5)
37
[10]> (amount)
37
[11]> account

*** - EVAL: variable ACCOUNT has no value
```

In fact, Lisp can be easily modified to do rather OOP built on top of closures. It comes with an OOP system, CLOS, as part of the language (though I think CLOS is too mammoth, so I usually make my own little OOP language in Lisp instead).

Sequence Functions

Vectors (both simple and variable-length), lists, and strings are all **sequences**. Multidimensional arrays are *not* sequences.

A function which works with any kind of sequence is a **sequence function** (duh). We've seen some examples of sequence functions before: **length**, **reverse**, **subseq**.

Another common sequence function is **elt**, of the form:

(elt *sequence index*)

elt returns element #*index* in the sequence. You can use **elt** inside **setf** to set the element (again, don't change elements in lists unless you know what you're doing. Strings and vectors are fine).

elt is an example of a general function: it works with a variety of data types, but as a result is slower than custom-made functions for each data type. For example, if you know your sequence is a string, **aref** is probably faster. If you know your sequence is a simple-vector, **svref** is much faster. Lists also have a faster function: **nth**.

```
[1]> (elt "hello world" 4)
#\o
[2]> (elt '(yo yo yo whats up?) 4)
UP?
[3]> (elt #(yo yo yo whats up?) 4)
UP?
```

copy-seq makes a duplicate copy of a sequence. It does not copy the elements (both sequences will point to the same elements).

concatenate concatenates copies of sequences together, producing a new sequence of a given type. The original sequences can be different types. **concatenate** looks like this:

(concatenate *new-sequence-type sequences...*)

new-sequence-type is a quoted symbol representing the *type* of the new sequence. For example, simple vectors use **'simple-vector** and lists use **'list** and strings use **'string**.

make-sequence builds a sequence of a given type and length. Like **elt**, it is a general function (it calls faster, more type-specific functions underneath). It looks like this:

(make-sequence *sequence-type length*)

make-sequence has a keyword argument **:initial-element** which can be used to set the initial element of the sequence.

concatenate and **make-sequence** show the first examples of **type symbols**. We'll talk about types more later.

```
[1]> (copy-seq "hello world")
"hello world"   [ the copied string ]
[2]> (concatenate 'string '(#\y #\o) #(#\space) "what's up?")
"yo what's up?"
[3]> (make-sequence 'string 4 :initial-element #\e)
"eeee"
```

A host of sequence-manipulative functions have very similar forms.

First off, most sequence-manipulative functions are either

```
[1]> (count #\l "hello world")
3
[ count the number of vowels in "hello world" ]
[2]> (count-if #'(lambda (i) (find i "aeiou")) "hello world")
```

destructive or **non-destructive**. That is, either they modify or destroy the original sequence to achieve their goals (faster), or they make a copy of the sequence first. We'll show the non-destructive versions first.

Second, a great many sequence functions have three versions, the function, the **-if** version, and the **-if-not** version. For example, the **count** function also has **count-if** and **count-if-not**. The forms look like this:

(count *object sequence keywords...*) counts the number of times *object* appears in *sequence*.

(count-if *test-predicate sequence keywords...*) counts the number of times in which *test-predicate* (a function pointer) returns true for elements in *sequence*.

(count-if-not *test-predicate sequence keywords...*) counts the number of times in which *test-predicate* (a function pointer) returns false (**nil**) for elements in *sequence*.

Third, many such functions take a *lot* of optional keyword arguments. Before testing to see if an element is the one we're looking for, these functions give you a chance to "extract" the relevant item out of the element with an optional function passed in with the keyword argument **:key**. You can tell the system to scan backwards with **:from-end t**. You can tell the system to only scan from a certain location to another location in the sequence with the keywords **:start** and **:end**. There are other keywords as well.

Other functions which follow this pattern include: **find** (returns the first element matching the pattern) else **nil**, **position** (returns the index of the first element matching the pattern) else **nil**, **remove** (removes all the elements matching the pattern from a *copy* of the sequence), and **substitute** (replaces all the elements matching the pattern with some other element). **substitute** has an additional argument indicating the item to replace stuff with, thus its three versions **substitute**, **substitute-if**, and **substitute-if-not** start like this:

(substitute[-if[-not]] *thing-to-replace-with rest-of-arguments-as-before...*)

```
3
[ count the number of non-alpha-chars in "hello world4" ]
[3]> (count-if-not #'alpha-char-p "hello world4")
2
[ remove the alpha chars from "hello world4" ]
[4]> (remove-if #'alpha-char-p "hello world4")
" 4"
[ find the first element < 4 in #(4 9 7 2 1 0 3) ]
[5]> (find-if #'(lambda (x) (< x 4)) #(4 9 7 2 1 0 3))
2
[ give the index of the first element < 4 in #(4 9 7 2 1 0 3) ]
[6]> (position-if #'(lambda (x) (< x 4)) #(4 9 7 2 1 0 3))
3
[ replace with NUMBER all the numbers in the list ]
[7]> (substitute-if 'number #'numberp '(a b 3 d "yo" 4.2 e))
(A B NUMBER D "yo" NUMBER E)
[ replace PI with 3.14159 for elements #4 through #9 ]
[8]> (substitute 3.14159 'pi
    '(pi a b pi c d e pi f pi pi g h pi)
    :start 4 :end 10)
(PI A B PI C D E 3.14159 F 3.14159 PI G H PI)
```

Another useful function, **search**, searches for the first index where one subsequence appears in another sequence. It takes the form:

(search *subsequence sequence keywords...*)

Keywords include **:key, :test, :test-not, :from-end, :start1, :end1, :start2, :end2**. Try them out and see what they do.

```
[1]> (search "wor" "hello world")
6
```

Many sequence functions have *destructive* counterparts which are faster but **may** modify the original sequence rather than making a copy first and modifying the copy.

There are no promises with destructive functions: they may or may not modify the original. They may or may not modify the original into the form you're hoping for. The only guarantee they make is that the value they *return* will be what you're hoping for. Thus you should only use them on data that you don't care about any more.

The destructive form of **remove[-if[-not]]** is **delete[-if[-**

```
[1]> (setf *j* "hello world")
"hello world"
[2]> (substitute #\Q #\l *j*)
"heQQo worQd"
[3]> *j*
"hello world"
[4]> (nsubstitute #\Q #\l *j*)
"heQQo worQd"
[5]> *j*
"heQQo worQd"
[6]> (sort '(4 3 5 2 3 1 3) #'>)
(5 4 3 3 3 2 1)
```

not]].

The destructive form of **substitute[-if[-not]]** is **nsubstitute[-if[-not]]**.

The destructive form of **reverse** is **nreverse**.

sort is destructive. Its basic form looks like this:

(sort *sequence predicate*)

Functions With Variable Arguments

List Functions

Now that we have enough functions to extract the elements of a list, we can talk about how to make a function which takes a variable number of arguments. The special term **&rest**, followed by an parameter name, can appear at the end of a parameter list in **defun**, musch as **&key** and **&optional** can appear.

If a function call provides any extra arguments beyond those defined in the parameter list, the additional arguments are all placed in a *list*, which the **&rest** parameter is set to. Otherwise it is set to **nil**.

Though it's possible to have rest-parameters along with keyword parameters and optional parameters in the same function, don't do it. Ick.

```
[1]> (defun mean (first-num &rest others)
[ others will be the list with remaining variable arguments ]
"Returns the mean of a bunch of numbers.
There must be at least one number."
(let ((nums (cons first-num others)))  ; nums is list of the numbers
  (/ (apply #'+ nums)
     (length nums))))

MEAN
[2]> (mean 10 2 3 4)
19/4
[3]> (mean 10)
10
[4]> (mean)

*** - EVAL/APPLY: too few arguments given to MEAN
[5]>
```

Lisp's primary data type is the **list**.

A list is a linked list elements. The linking structures are called **cons cells**. A cons cell is a structure with two fields: **car** and **cdr**. The **car** field points to the element the cons cell is holding. The **cdr** field points to the next cons cell, or to **nil** if the cons cell is at the end of the list.

When you are storing a list, you are really storing a pointer to the first cons cell in the list. Thus **car** (**first**) returns the thing that cons cell is pointing to, and **cdr** (**rest**) returns the next cons cell (what the **cdr** is pointing to). Similarly, the **cons** function allocates a new cons cell, sets its **cdr** to the cons cell representing the original list, and sets its **car** to the element you're tacking onto the list. Thus the original list isn't modified at all! You've just made a new cons cell which reuses the list to "extend" it by one.

last returns (as a list) the last *n* elements in the list. Really it just marches down the list and finds the appropriate cons and returns that to you. **last**'s argument is optional -- if it's not there, it returns (as a list) the last element in the list.

butlast returns (as a list) a copy of everything *but* the last *n* elements in the list.

list takes some *n* arguments and makes a list out of them. It differs from just quoting a list because the arguments are evaluated first (it's a function).

```
[1]> (last '(a b c d e))
(E)
[2]> (last '(a b c d e) 3)
(C D E)
[3]> (butlast '(a b c d e))
(A B C D)
[4]> (butlast '(a b c d e) 3)
(A B)
[5]> (list 1 2 (+ 3 2) "hello")
(1 2 5 "hello")
[6]> '(1 2 (+ 3 2) "hello")
(1 2 (+ 3 2) "hello")
```

nth works on lists just like **elt**. You can use it in **setf**. For lists, it's a tiny bit

faster than (the more general function) **elt**. Note that **nth**'s arguments are not the same order as those for **elt**. This is historical.

listp is a predicate which returns true if its argument is a list.

atom is a predicate which returns true if its argument is an atom. Note that **nil** is both a list and an atom. It's the only thing which is both.

consp is a predicate which is true if its argument is a list but is *not* nil.

null is a predicate which returns true if its argument is **nil**.

If you think about it, the logical **not** function is identical to the **null** function.

There are a great many more list functions. See "conses" in the text.

```
[1]> (listp '(4 3 8))
T
[2]> (listp nil)
T
[3]> (consp '(4 3 8))
T
[4]> (consp nil)
NIL
[5]> (atom 'a)
T
[6]> (atom nil)
T
[7]> (null nil)
T
[8]> (not nil)
T
[9]> (nth 4 '(a b c d e))
E
```

There are a great many permuations of **cdr** and **car**.

caar is the same as **`(car (car ... ))`**

cdar is the same as **`(cdr (car ... ))`**

caadr is the same as **`(car (car (cdr ... )))`**

cddddr is the same as **`(cdr (cdr (cdr (cdr ... ))))`**

second through **tenth** return the second through the tenth elements of a list.

```
[1]> (caar '((a b c) ((d)) e f g h))
A
[2]> (cdar '((a b c) ((d)) e f g h))
(B C)
[3]> (caadr '((a b c) ((d)) e f g h))
(D)
[4]> (cddddr '((a b c) ((d)) e f g h))
(G H)
[5]> (fourth '((a b c) ((d)) e f g h))
F
```

The **cdr** field of a cons cell doesn't have to point to a list (or to **nil**). Technically, it can point to anything you like. For example, a single cons cell can point to 45 **car** and "hello" in its **cdr**. Such a cons cell is called a **dotted pair**, because its printed form is **`(45 . "hello")`**. Note the period.

You can construct long lists where the last item doesn't point to **nil** but instead is a dotted pair.

If you think about it, the dotted pair can only be at the *end* of a list.

Dotted pairs qualify as lists as far as **listp** is concerned.

```
[1]> (cons 45 "hello")
(45 . "hello")
[2]> (cons 'a (cons 'b (cons 45 "hello")))
(A B 45 . "hello")
[3]> (listp '(45 . "hello"))
T
```

Lists are constructed like beads of pearls. So far we've been careful not to damage them.

setf can be used to do evil things to a list.

You can use **setf** to modify a list. Since we construct lists which point to original lists, if we modify a list then it might modify other lists which pointed to it.

One particular danger is to use **setf** to modify cons cells to have circular references. For example, you could use **setf** to set the **cdr** of a cell point to the same cell.

Most Lisp systems are smart enough to detect circular references while printing a list to the screen. Some (like CLISP) are not! They'll go into an infinite loop trying to print such beasts. Try it out on osf1 for some fun as well.

```
[1]> (setf *x* '(a b c d e))
(A B C D E)
[2]> (setf *y* (rest *x*))
(B C D E)
[3]> (setf (rest *y*) '(1 2 3))
(1 2 3)
[4]> *y*
(B 1 2 3)
[5]> *x*
(A B 1 2 3)     [ hey, wait a minute! ]
[6]> (setf *x* '(a))
(A)
[7]> (setf (cdr *x*) *x*)
(A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
A A A A A A A A A A A A A A A A A A A A A A A A
...  [ ad naseum ]
```

There are a number of destructive versions of list functions. Just as in applying the destructive sequence functions to list, the same warning applies: destructive functions are faster and use much less memory, but because lists are strung together like beads of pearls, make sure you know what you're

```
[1]> (setf *x* '(a b c d e))
(A B C D E)
[2]> (setf *y* '(1 2 3 4 5))
(1 2 3 4 5)
```

doing if you choose to use them! Here are two common ones:

nconc is the destructive version of **append**.

nbutlast is the destructive version of **butlast**.

```
[3]> (nconc *x* *y*)
(A B C D E 1 2 3 4 5)
[4]> *x*
(A B C D E 1 2 3 4 5)   [ what the ... ]
[5]> (nbutlast 4 *x*)
(A B C D E 1)
[6]> *x*
(A B C D E 1)  [ it keeps modifying *x*! ]
```

Predicates and Types

Lisp has a number of predicates to compare equality. Here are some type-specific ones.

(= *num1 num2*) compares two numbers to see if they are equal. 2.0 and 2 are considered =. Also, -0 and 0 are =.

(char= *char1 char2*) compares two characters. (can you guess what **char>**, **char<=**, etc. do?)

(char-equal *char1 char2*) compares two characters in a case-insensitive way.

(string= *str1 str2*) compares two strings.

(string-equal *str1 str2*) compares two strings in a case-insensitive way.

There are also general equality predicates. These predicates vary in strength. Here are some loose descriptions.

(eq *obj1 obj2*) is true if *obj1* and *obj2* are *the exact same thing in memory*. Symbols and same-type numbers are the same thing: **(eq 'a 'a)** is true for example. But complex objects made separately aren't the same thing: **(eq '(1 2 3) '(1 2 3))** is false. Neither are integers and floats **eq** with one another: **(eq 0 0.0)** is false. **eq** is fast (it's a pointer comparison).

(eql *obj1 obj2*) is is like **eq** but also allows integers and floats to be the same (as in **(eq 0 0.0)** is true). **eql** is the default comparator for most stuff.

(equal *obj1 obj2*) says two objects are equal if they are **eql** or if they "look equal" and are lists, strings and pathnames, or bit-vectors.

(equalp *obj1 obj2*) says two objects are equal if they look equal. **equalp** compares nearly every kind of Lisp thing, including all sorts of numbers, symbols, characters, arrays, strings, lists, hash tables, structures, files, you name it. **equalp** is the slowest comparator predicate, but you will generally find it to be the most useful.

Some numbers *should* be = but may not be due to numeric precision.

```
[1]> (eql '(a b) '(a b))
NIL
[2]> (equalp '(a b) '(a b))
T
[3]> (= 0.25 1/4)
T
[4]> (eq (setf *q* '(a b)) *q*)  [ remember the function rule ]
T
[5]> (string-equal "hello" "HelLo")
T
[6]> (= 1/5 .2)
NIL        [ what the ... ? ]
```

Lisp also has predicates to determine the **type** of objects. You've already seen some such predicates: **atom**, **null**, **listp**.

```
[1]> (numberp 'a)
NIL
[2]> (stringp "hello")
T
```

(numberp *obj*) is true if *obj* is a number. There are a number of useful numerical predicates as well: **oddp** is true if the number is odd (see also **evenp**). **zerop** is true if the number is zero. **plusp** is true if the number is > 0. Etc.

characterp is true if *obj* is a character. There are a number of subpredicates, such as **alphanumericp** which is true if the character is a letter or a number.

symbolp is true if it's a symbol. **stringp** is true if it's a string. **arrayp** is true if it's an array. **vectorp** and **simple-vector-p** are...well you get the idea. There's a lot of this stuff.

Lisp has a general type-determination predicate called **typep**. It looks like this:

(typep *expression type*)

A type is (usually but not always) a symbol representing the type (you have to quote it -- it's evaluated). Example types include **number, list, simple-vector, string,** etc.

Types are organized into a hierarchy: thus types can have subtypes (simple-vector is a subtype of vector, which is a subtype of array, for example). The root type is **t**. The **typep** function returns true if the expression has the type that as its base type or as a supertype.

Numeric types in particular have quite a lot of subtypes, such as **fixnum** (small integers), **bignum** (massive integers), **float**, **double-float**, **rational**, **real**, **complex**, etc.

```
[1]> (typep 'a 'symbol)
T
[2]> (typep "hello" 'string)
T
[3]> (typep 234098123423412341341234342 34 'bignum)
T
[4]> (typep 234098123423412341341234342 34 'rational)
T
[5]> (typep 1/9 'rational)
T
[6]> (typep 1/9 'list)
NIL
[7]> (typep 1/9 'foo)

*** - TYPEP: invalid type specification FOO
```

You can get the type of any expression with **(type-of *expr*)**

```
[1]> (type-of 'float)
SYMBOL
[2]> (type-of 1/3)
RATIO
[3]> (type-of -2)
FIXNUM
[4]> (type-of "hello")
(SIMPLE-BASE-STRING 5) [ types can be lists starting with a symbol ]
[5]> (type-of (make-array '(3 3)))
(SIMPLE-ARRAY T (3 3))
[6]> (type-of nil)
NULL
```

Many objects may be *coerced* into another type, using the **coerce** function:

(coerce *expression type*)

Vectors and lists may be coerced into one another.

Strings may be coerced into other sequences, and lists or vectors of characters can be coerced into strings.

Integers may be coerced into floats. To convert a float or other rational into an integer, use one of the functions **floor**, **round**, **truncate** (round towards zero), or **ceiling**.

```
[1]> (coerce 4 'float)
4.0
[2]> (coerce "hello world" 'list)
(#\h #\e #\l #\l #\o #\Space #\w #\o #\r #\l #\d)
[3]> (coerce '(#\h #\e #\l #\l #\o #\Space #\w #\o #\r #\l #\d)
     'string)
"hello world"
[4]> (floor -4.3)
-5
[5]> (coerce '(a b c) 'simple-vector)
#(A B C)
[6]> (coerce '(a b c) 'string)

*** - SYSTEM::STORE: A does not fit into "", bad type
```

While we're on the subject of the four rounding functions (**floor**, **round**, **truncate**, **ceiling**), these are how you do integer division. Each function takes an

```
[1]> (floor 9 4)
2;    [ the primary return value ]
```

optional argument, and divides the first argument by the second, then returns the appropriate rounding as an integer.

If you're used to C++ or Java's integer division, probably the most obvious choice is **truncate**.

Lisp functions can actually return more than one item. For example, integer division functions return both the divided value and the remainder. Both are printed to the screen. The primary return value (in this case, the divided value) is returned as normal. To access the "alternate" return value (in this case, the remainder), you need to use a macro such as **multiple-value-bind** or **multiple-value-list** (among others).

```
1     [ the alternative return value ]
[2]> (floor -9 4)
-3 ;
3
[3]> (truncate -9 4)
-2 ;
-1
[4]> (* 4 (truncate -9 4))
-8    [ 4 mulplied against the primary return value ]
[5]> (multiple-value-list (truncate -9 4))
(-2 -1)
[6]> (multiple-value-bind (x y) (truncate -9 4)
  (* x y))
2
```

Hash Tables

Hash tables are created with **make-hash-table**. You can hash with anything as a key. Hash tables by default use **eql** as a comparison predicate. This is almost always the wrong predicate to use: you usually would want to use **equal** or **equalp**. To do this for example, you type:

`(make-hash-table :test #'equalp)`

Elements are accessed with **gethash**. If the element doesn't exist, **nil** is returned. An alternative return value indicates whether or not the element exists (returning T or NIL). If you stored **nil** as the value, then we have a problem! Instead of having to look up the alternate return value, you can supply an optional return value (instead of **nil**) to return if the slot really *is* empty.

`(gethash` *`key hashtable &optional`* `return-if-empty)`

Use **setf** to set hashed values.

`(setf (gethash` *`key hashtable`*`) `*`value`*`)`

Remove elements with **remhash**.

`(remhash` *`key hashtable`*`)`

Although it's not very efficient, you can map over a hashtable with **maphash**.

`(maphash` *`function hashtable`*`)`

function must take two arguments (the key and the value).

```
[1]> (setf *hash* (make-hash-table :test #'equalp))
#S(HASH-TABLE EQUALP)
[2]> (setf (gethash "hello" *hash*) '(a b c))
(A B C)
[3]> (setf (gethash 2 *hash*) 1/2)
1/2
[4]> (setf (gethash 2.0 *hash*) 9.2)  [ 2.0 is equalp to 2 ]
9.2
[5]> (gethash 2 *hash*)
9.2   [ because we're using equalp as a test ]
T     [ T because the slot exists in the hashtable ]
[6]> (setf (gethash #\a *hash*) nil)    [ store NIL as the value ]
NIL
[7]> (gethash #\b *hash*)
NIL;    [ No such key #\b in *hash* ]
NIL
[8]> (gethash #\a *hash*)
NIL;   [ uh... wait a minute... -- NIL is returned! ]
T
[9]> (gethash #\b *hash* 'my-empty-symbol)
MY-EMPTY-SYMBOL ;
NIL
[10]> (gethash #\a *hash* 'my-empty-symbol)
NIL;    [ that's better! ]
T
[11]> (maphash #'(lambda (key val) (print key)) *hash*)

2
"hello"
NIL
```

Printing and Reading

`(tepri)` prints a linefeed.

```
[1]> (progn (terpri) (terpri) (terpri) (print 'hello))
```

(print *obj*) of course prints a linefeed followed by *obj* (in a computer readable fashion). Unlike Java's **System.println("foo")** or C's **printf("foo\n")**, in Lisp it's traditional to print the newline *first*.

(prin1 *obj*) prints *obj* (in a computer readable fashion) -- no prior linefeed.

(princ *obj*) prints *obj* in a *human* readable fashion -- no prior linefeed. Strings are printed without "quotes", for example. Such printed elements aren't guaranteed to be readable back into the intepreter.

```
HELLO
HELLO
[2]> (progn (prin1 2) (prin1 '(a b c)) (prin1 "hello"))
2(A B C)"hello"
"hello"
[3]> (progn (princ 2) (princ '(a b c)) (princ "hello"))
2(A B C)hello
"hello"
```

(prin1-to-string *obj*) is like **prin1**, but the output is into a string.

(princ-to-string *obj*) is like **princ**, but the output is into a string.

```
[1]> (prin1-to-string 4.324)
"4.324"
[2]> (prin1-to-string "hello world")
"\"hello world\""
[3]> (princ-to-string "hello world")
"hello world"
[4]> (prin1-to-string '(a b "hello" c))
"(A B \"hello\" C)"
[5]> (princ-to-string '(a b "hello" c))
"(A B hello C)"
```

(read) reads in an expression from the command line.

read is a complete Lisp parser: it will read any expression.

(read-from-string *string*) reads in an expression from a string, and returns the expression plus an integer indicating at what point reading was completed.

(y-or-no-p) waits for the user to type in a yes or a no somehow, then returns it. The way the question is presented the user (graphical interface, printed on screen, etc.) is up to the Lisp system. **y-or-no-p** is a predicate.

```
[1]> (read)    [ Lisp waits for you to type an expression ]
'(a b c d)
(A B C D)
[2]> (read-from-string "'(a b c d)")
'(A B C D) ;   [ Or equivalently (QUOTE (A B C D)) ]
10     [ Reading the expression finished before the tenth character ]
[3]> (y-or-n-p)   [ clisp waits for you to type y or n ]
TRUE      [ I tried to type in TRUE ]

Please answer with y or n : y   [ oh, okay! ]
T
```

format is a much more sophisticated printing facility. It is somewhat similar to C's **printf** command plus formating string. But **format**'s formatting string is much more capable. Generally, **format** looks like:

(format *print-to-where format-string obj1 obj2 ...*)

print-to-where can be **t** (print to the screen) or **nil** (print to a string).

Formatting sequences begin with a tilde (~). The simplest sequences include: **~a** (princ an element); **~%** (print a linefeed); **~s** (prin1 an element). Much more complex formatting includes very complex numerical printing, adding spaces and buffers, printing through lists, even printing in roman numerals! **format** has its own little programming language. It's astounding what **format** can do.

```
[1]> (format t "~%My name is ~a and my ID is ~a" "Sean" 1231)

My name is Sean and my ID is 1231
NIL
[2]> (format nil "~%~%~%~a~a~s    ~a" '(a b c)
 #(1 2 3) "yo" 'whatever)
"


(A B C)#(1 2 3)\"yo\"    WHATEVER"
[3]> (format t "~% ~a ~R ~:R ~@R ~:@R ~$ ~E" 4 4 4 4 4 4 4)

4 four fourth IV IIII 4.00 4.0E+0    [ hee hee hee! ]
NIL
```

More Control Structures

(when *test expr1 expr2 ...*) evaluates the expressions (and returns the last) only if *test* is true, else it returns **nil**.

(unless *test expr1 expr2 ...*) evaluates the expressions (and returns the last) only if *test* is **nil**, else it returns **nil**.

(case *test-object case1 case2 ...*) goes through the cases one by one and returns the one which "matches" the *test-object*. A case looks like this:

(*obj expr1 expr2 ...*)

If *obj* (not evaluated, so you shouldn't quote it) is an object which is **eql** to *test-object*, or is a list in which *test-object* appears, then the case "matches" *test-object*. In this case, the expressions are evaluated left-to-right, and the last one is returned. *obj* can also be **t**, which matches anything. This is the "default" case.

If no case matches, then **case** returns nil.

case is a lot like the Java/C++ **switch** statement. There are other versions: **ecase**, **ccase**.

```
[1]> (unless (y-or-n-p) (print "you picked no!")
         (print "good for you!"))
n

"you picked no!"
"good for you!"
"good for you!"
[2]> (defun type-discriminator (obj)
"Prints out a guess at the type"
(let ((typ (type-of obj)))
(when (consp typ) (setf typ (first typ)))
(case typ
 ((fixnum rational ratio complex real bignum)
    (print "a number perhaps?"))
 ((simple-vector vector string list)
    (print "some kind of sequence?"))
 (hash-table (print "hey, a hash table..."))
 (nil (print "it's nil!"))
 (t (print "beats me what this thing is.  It says:")
    (print (type-of obj))))))

TYPE-DISCRIMINATOR
[3]> (type-discriminator 42)

"a number perhaps?"
"a number perhaps?"
[4]> (type-discriminator "hello")

"beats me what this thing is.  It says:"
(SIMPLE-BASE-STRING 5)
(SIMPLE-BASE-STRING 5)
```

cond is a powerful generalization of **case**. It takes the form:

(cond (*test1 expr expr ...*)
(*test2 expr expr ...*)
(*test3 expr expr ...*)
...)

cond works like this. First, *test1* is evaluated. If this is true, the following expressions are evaluated and the last one is returned. If not, then *test2* is evaluated. If this is true, its following expressions are evaluated and the last one is true. And so on. If no test evaluates to true, then **nil** is returned.

```
[ Previously, type-discriminator didn't work for string.
let's get it working right. ]
[2]> (defun type-discriminator (obj)
"Prints out a guess at the type"
(cond
((find-if #'(lambda (x) (typep obj x))
   '(fixnum rational ratio complex real bignum))
 (print "a number perhaps?"))
((find-if #'(lambda (x) (typep obj x))
   '(simple-vector vector string list))
 (print "some kind of sequence?"))
((typep obj 'hash-table) (print "hey, a hash table..."))
((typep obj null) (print "it's nil!"))
(t (print "beats me what this thing is.  It says:")
 (print (type-of obj)))))

TYPE-DISCRIMINATOR
[3]> (type-discriminator "hello")

"some kind of sequence?"
"some kind of sequence?"
```

do is a general iterator. It takes the form:

(do (*initial-variable-declarations*)
(*test res-expr1 res-expr2 ...*)
expr1
expr2
...)

do works like this. First, local variables are declared in a way somwhat similarly to **let** (we'll get to that). Then *test* is evaluated. If it is true, then the *res-expr*'s are evaluated and the last one is returned (if there are none, then **nil** is returned).

If *test* returned false, then *expr*'s in the body are evaluated. Then **do** iterates again, starting with trying *test* again. And so on.

A variable declaration is either a variable name (a symbol), just as in **let**, or it is a list of the form **(*var optional-init optional-update*)** The *optional-init* expression initializes the variable (else it's **nil**). The *optional-update* expression specifies the new value of *var*

```
[ generate some random numbers ]
[2]> (defun generate (num)
(do ((y 0 (1+ y))
    (x 234567 (mod (* x 16807) 2147483647)))
   ((>= y num) "the end!")
   (print x)))
GENERATE
[3]> (generate 20)

234567
1794883922
911287645
158079111
398347238
1315501367
1287329304
245868803
558435193
1116751361
233049547
2001047948
```

each iteration. *optional-update* is evaluated in the context of the variables of the previous iteration.

```
2018950016
103812665
1022739291
720153249
397821451
1068533846
1590093508
1415085688
"the end!"
```

loop is a very powerful, complex iteration macro which can do nearly anything. Literally. It has its own language built into it. **loop** is one of the few things in Lisp more complex than **format**.

loop has an idiosyncratic syntax that is very un-lisp-like. It is also so complex that few people understand it, and it is not recommended for use. We will not discuss **loop** except to mention that its very simplest form: **`(loop` *`expressions ...`* `)`** makes a very nice infinite loop.

```
[2]> (loop (print 'hello) (print 'yo))
HELLO
YO
HELLO
YO
HELLO
YO
HELLO
YO
 [ ... ad nauseum until you press Control-C ]
```

A **block** is a sequence of expressions. Blocks appear in lots of control structures, such as **let**, all iterators (**do**, **dotimes**, **dolist**, **loop**, etc.), many conditional statements (**cond**, **case**, **when**, etc.), **progn**, etc.

Blocks have **labels** (names). In control structures, the implicit blocks are all named **nil**.

Functions created with **defun** have an implicit block whose label is the same name as the function. Functions created with **lambda** have an implicit block whose label is **nil**.

`(return-from` *`label optional-value`*`)` will exit prematurely from a block whose label is *label* (not evaluated -- don't quote it). This is somewhat like Java/C++'s **break** statement. The return value of the block is *optional-value* (or **nil** if no value provided).

Because so many blocks are named **nil**, the simpler **`(return` *`optional-value`*`)`** is the same thing as **`(return-from nil` *`optional-value`*`)`**

Use **return** and **return-from** sparingly. They should be rare.

```
[1]> (dotimes (x 100)
 (print x)
 (if (> x 10) (return 'hello)))

0
1
2
3
4
5
6
7
8
9
10
11
HELLO
[2]> (defun differents (list &key (test #'eql))
"Returns the first different pair in list"
 (dolist (x list)
   (dolist (y list)
     (unless (funcall test x y)
       (return-from differents (list x y))))))
DIFFERENTS
[3]> (differents '(a a a b c d))
(A B)
[4]> (differents '(a a a a a a))
NIL
```

Another way to escape is with **catch** and **throw**. **catch** looks like this:

`(catch` *`catch-symbol expressions ...`* `)`

throw looks like:

`(throw` *`catch-symbol return-value`*`)`

Normally, **catch** works just like **progn**. But if there is a **throw** statement inside the **catch** whose *catch-symbol* matches the **catch**'s, then we prematurely drop out of the **catch** and the **catch** returns the return value of the **throw**.

This works even if the **throw** appears in a subfunction called inside the **catch**.

In C++ such a thing is done with **longjump**. In Java such a thing is done with an exception.

```
[ another way to do the differents function ]
[2]> (defun differents (list &key (test #'eql))
"Returns the first different pair in list"
 (catch 'my-return-value
   (dolist (x list)
     (dolist (y list)
       (unless (funcall test x y)
         (throw 'my-return-value (list x y)))))))
DIFFERENTS
[3]> (differents '(a a a b c d))
(A B)
[4]> (differents '(a a a a a a))
NIL
```

Writing Lisp in Lisp

Lisp has a built-in interpreter. It is called **eval**, and looks like this:

(eval *data*)

eval takes *data* and submits it to the Lisp interpreter to be executed.

The data submitted to the interpeter is not evaluated in the context of any current local variables.

eval is powerful. You can assemble lists and then have them executed as code. Thus **eval** allows you to make lisp programs which generate lisp code on-the-fly. C++ and Java can only do this in truly evil ways (like writing machine code to an array, then casting it into a function, yikes!).

```
[1]> (list '+ 4 7 9)
(+ 4 7 9)
[2]> (eval (list '+ 4 7 9))
20
```

Lisp has an interpeter **eval**, a full-featured printer **print**, and a full-featured parser **read**. Using these tools, we can create our own Lisp command line!

```
[1]> (loop (format t "~%my-lisp --> ")
   (print (eval (read))))

my-lisp --> (dotimes (x 10) (print 'hi))

HI
HI
HI
HI
HI
HI
HI
HI
HI
HI
NIL
my-lisp -->
```

Debugging

(break) signals an error, just as if the user pressed Control-C.

You can continue from a break.

```
[1]> (defun foo (x)
(print (+ x 3))
(break)
(print (+ x 4)))

FOO
[2]> (foo 7)

10
** - Continuable Error
Break
If you continue (by typing 'continue'): Return from BREAK loop
2. Break [4]> continue  [in clisp, anyway]

11
11
```

(trace *function-symbol*) turns on tracing of a function. *function-symbol* is not evaluated (don't quote it or sharp-quote it).

When a trace function is

```
[1]> (defun factorial (n)
  (if (<= n 0)
    1
    (* n (factorial (- n 1)))))
FACTORIAL
[2]> (trace factorial)
(FACTORIAL)
[3]> (factorial 15)
```

entered, the function and its arguments are printed to the screen. When the trace function exits, its return value is printed to the screen.

You can trace multiple functions at the same time.

You turn off tracing of a function with **(untrace *function-symbol*)**

```
1. Trace: (FACTORIAL '15)
2. Trace: (FACTORIAL '14)
3. Trace: (FACTORIAL '13)
4. Trace: (FACTORIAL '12)
5. Trace: (FACTORIAL '11)
6. Trace: (FACTORIAL '10)
7. Trace: (FACTORIAL '9)
8. Trace: (FACTORIAL '8)
9. Trace: (FACTORIAL '7)
10. Trace: (FACTORIAL '6)
11. Trace: (FACTORIAL '5)
12. Trace: (FACTORIAL '4)
13. Trace: (FACTORIAL '3)
14. Trace: (FACTORIAL '2)
15. Trace: (FACTORIAL '1)
16. Trace: (FACTORIAL '0)
16. Trace: FACTORIAL ==> 1
15. Trace: FACTORIAL ==> 1
14. Trace: FACTORIAL ==> 2
13. Trace: FACTORIAL ==> 6
12. Trace: FACTORIAL ==> 24
11. Trace: FACTORIAL ==> 120
10. Trace: FACTORIAL ==> 720
9. Trace: FACTORIAL ==> 5040
8. Trace: FACTORIAL ==> 40320
7. Trace: FACTORIAL ==> 362880
6. Trace: FACTORIAL ==> 3628800
5. Trace: FACTORIAL ==> 39916800
4. Trace: FACTORIAL ==> 479001600
3. Trace: FACTORIAL ==> 6227020800
2. Trace: FACTORIAL ==> 87178291200
1. Trace: FACTORIAL ==> 1307674368000
1307674368000
[4]> (untrace factorial)
(FACTORIAL)
[5]>  (factorial 15)
1307674368000
```

You can step through an expression's evaluation, just as in a debugger, using **(step *expression*)**. The features available within the **step** environment are implementation-dependent.

In clisp, the **step** function lets you interactively type, among other things, **:s** (to step into an expression), **:n** (to complete the evaluation of the expression and step out), and **:a** (to abort stepping)

```
[1]> (defun factorial (n)
  (if (<= n 0)
    1
    (* n (factorial (- n 1)))))
FACTORIAL
[2]> (step (factorial 4))
(step (factorial 4))
step 1 --> (FACTORIAL 4)
Step 1 [26]> :s
step 2 --> 4
Step 2 [27]> :s

step 2 ==> value: 4
step 2 --> (IF (<= N 0) 1 (* N (FACTORIAL #)))
Step 2 [28]> :s
step 3 --> (<= N 0)
Step 3 [29]> :n

step 3 ==> value: NIL
step 3 --> (* N (FACTORIAL (- N 1)))
Step 3 [30]> :s
step 4 --> N
Step 4 [31]> :s

step 4 ==> value: 4
step 4 --> (FACTORIAL (- N 1))
Step 4 [32]> :s
step 5 --> (- N 1)
Step 5 [33]> :n

step 5 ==> value: 3
step 5 --> (IF (<= N 0) 1 (* N (FACTORIAL #)))
Step 5 [34]> :n

step 5 ==> value: 6
step 4 ==> value: 6
step 3 ==> value: 24
step 2 ==> value: 24
step 1 ==> value: 24
24
```

The **apropos** function can be used to find all the defined symbols in the system which match a given string. Ordinarily, **apropos** will return *everything*, including private system symbols. That's not what you'd want. But the following will do the trick:

(apropos *matching-string* 'cl-user)

Notice that there are five items at right whose name has the word "random" in them:

- *RANDOM-STATE*, a global variable, the current state of the random number generator
- MAKE-RANDOM-STATE, a function which builds a new random number generator
- RANDOM, a function which gives a new random number
- RANDOM-STATE, a type class. Random number generators are of this type.
- RANDOM-STATE-P, a function which indicates if the thing passed to is in fact a random number generator

```
[ in clisp  ]
[1]> (apropos "random" 'cl-user)
*RANDOM-STATE*                          variable
MAKE-RANDOM-STATE                       function
RANDOM                                  function
RANDOM-STATE                            type   class
RANDOM-STATE-P                          function
```

Let's look a little more closely at the function RANDOM. We can ask for more details about the nature of RANDOM by typing:

(describe 'random)

Note the single quote. Also beware that on clisp on zeus, the DESCRIBE function attempts to connect with an HTTP server at MIT to provide

```
[1]> (setf custom:*browser* nil)
NIL
[2]> (describe 'random)

RANDOM is the symbol RANDOM, lies in #<PACKAGE COMMON-LISP>, is
accessible in 11 packages CLOS, COMMON-LISP, COMMON-LISP-USER, EXPORTING, EXT,
FFI, POSIX, READLINE, REGEXP, SCREEN, SYSTEM, names a function, has 1 property
SYSTEM::DOC.

ANSI-CL Documentation is at
"http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/Body/fun_random.html"
For more information, evaluate (SYMBOL-PLIST 'RANDOM).

 #<PACKAGE COMMON-LISP> is the package named COMMON-LISP. It has 2
 nicknames LISP, CL.  It imports the external symbols of 1 package CLOS and
 exports 978 symbols to 10 packages READLINE, REGEXP, POSIX, EXPORTING, FFI,
 SCREEN, CLOS, COMMON-LISP-USER, EXT, SYSTEM.
```

you with nice documentation: but it doesn't work right now. So at right we have first executed a command to prevent that from happening.

```
 #<SYSTEM-FUNCTION RANDOM> is a built-in system function.
 Argument list: (#:ARG0 &OPTIONAL #:ARG1)
 For more information, evaluate (DISASSEMBLE #'RANDOM).

Documentation:
CLHS:
"Body/fun_random.html"
```

Lisp systems can either be interpreters or compilers, though most modern Lisp systems are compilers and automaticall compile your code as soon as you enter it. However *how* they compile the code varies. For example, CLISP compiles to a P-code rather than to machine code (kind of like how javac compiles to .class files rather than to machine code). SBCL and LispWorks compile to machine code.

If you're interested in seeing what the resulting compiled code looks like, you can disassemble a function with **(disassemble *function-pointer*)**

```
[CLISP on zeus...]
[1]> (defun factorial (n)
          (if (<= n 0)
            1
            (* n (factorial (- n 1)))))
FACTORIAL
[3]> (disassemble #'factorial)

Disassembly of function FACTORIAL
(CONST 0) = 1
1 required argument
0 optional arguments
No rest parameter
No keyword parameters
11 byte-code instructions:
0     L0
0     (LOAD&PUSH 1)
1     (CALLS2&JMPIFNOT 174 L14)           ; MINUSP
4     (LOAD&PUSH 1)
5     (LOAD&DEC&PUSH 2)
7     (JSR&PUSH L0)
9     (CALLSR 2 57)                       ; *
12    (SKIP&RET 2)
14    L14
14    (CONST 0)                           ; 1
15    (SKIP&RET 2)
NIL
```

```
[SBCL on your laptop...]
* (defun factorial (n)
          (if (<= n 0)
            1
            (* n (factorial (- n 1)))))
FACTORIAL
* (disassemble #'factorial)

; disassembly for FACTORIAL
; Size: 158 bytes. Origin: #x1004B55BFD
; BFD:       498B4C2460       MOV RCX, [R12+96]              ; thread.binding-stack-pointer
                                                             ; no-arg-parsing entry point
; C02:       48894DF8         MOV [RBP-8], RCX
; C06:       488B4DF0         MOV RCX, [RBP-16]
; C0A:       8D41F1           LEA EAX, [RCX-15]
; C0D:       A801             TEST AL, 1
; C0F:       750E             JNE L0
; C11:       3C0A             CMP AL, 10
; C13:       740A             JEQ L0
; C15:       A80F             TEST AL, 15
; C17:       7575             JNE L4
; C19:       8079F11D         CMP BYTE PTR [RCX-15], 29
; C1D:       776F             JNBE L4
; C1F: L0:   488B55F0         MOV RDX, [RBP-16]
; C23:       31FF             XOR EDI, EDI
; C25:       B930040020       MOV ECX, 536871984             ; GENERIC->
; C2A:       FFD1             CALL RCX
; C2C:       7E0B             JLE L3
; C2E: L1:   BA02000000       MOV EDX, 2
; C33: L2:   488BE5           MOV RSP, RBP
; C36:       F8               CLC
; C37:       5D               POP RBP
; C38:       C3               RET
; C39: L3:   488B55F0         MOV RDX, [RBP-16]
; C3D:       31FF             XOR EDI, EDI
; C3F:       B9B0040020       MOV ECX, 536872112             ; GENERIC-=
; C44:       FFD1             CALL RCX
```

```
; C46:       74E6             JEQ L1
; C48:       488B55F0         MOV RDX, [RBP-16]
; C4C:       BF02000000       MOV EDI, 2
; C51:       41BB40020020     MOV R11D, 536871488            ; GENERIC--
; C57:       41FFD3           CALL R11
; C5A:       488D5C24F0       LEA RBX, [RSP-16]
; C5F:       4883EC18         SUB RSP, 24
; C63:       488B0536FFFFFF   MOV RAX, [RIP-202]             ; #<FDEFINITION for FACTORIAL>
; C6A:       B902000000       MOV ECX, 2
; C6F:       48892B           MOV [RBX], RBP
; C72:       488BEB           MOV RBP, RBX
; C75:       FF5009           CALL QWORD PTR [RAX+9]
; C78:       480F42E3         CMOVB RSP, RBX
; C7C:       488BFA           MOV RDI, RDX
; C7F:       488B55F0         MOV RDX, [RBP-16]
; C83:       41BBB0020020     MOV R11D, 536871600            ; GENERIC-*
; C89:       41FFD3           CALL R11
; C8C:       EBA5             JMP L2
; C8E: L4:   488B45F0         MOV RAX, [RBP-16]
; C92:       0F0B0A           BREAK 10                       ; error trap
; C95:       02               BYTE #X02
; C96:       13               BYTE #X13                      ; OBJECT-NOT-REAL-ERROR
; C97:       1B               BYTE #X1B                      ; RAX
; C98:       0F0B10           BREAK 16                       ; Invalid argument count trap

NIL
```

```
[LispWorks running on mason / osf1...]

CL-USER 39 >  (defun factorial (n)
          (if (<= n 0)
            1
            (* n (factorial (- n 1)))))
FACTORIAL

CL-USER 40 > (disassemble #'factorial)
.L00:   ld      [%g1 + 692], %g3
4       cmp     %o5, %g3
8       bleu    .L01
12      noop
16      cmp     %g5, 1
20      be,a    .L02
24      ld      [%o4 + 6], %o4
.L01:   jmp     [%g1 + eb6]    ;;  global fun: SYSTEM::*%INTERNAL-WRONG-NUMBER-OF-ARGUMENTS
32      noop
36      ld      [%o4 + 6], %o4
.L02:   andn    %o5, 7, %g2
44      save    %g2, -40, %o6
48      sub     %g2, 40, %o5
52      ld      [%i4 + 1d], %g3    ;;  call counter
56      add     %g3, 4, %g3
60      st      %g3, [%i4 + 1d]
64      andcc   %i0, 3, %g3
68      bne,a   .L13
72      mov     %g0, %o1
76      cmp     %i0, 0
80      bge,a   .L14
84      mov     1, %g5
88      tsubcc  %i0, 4, %g6
.L03:   bvs,a   .L15
96      mov     4, %o1
100     mov     %g6, %o0
104     ld      [%i4 + 2d], %o4     ;;  FACTORIAL
.L04:   ld      [%o4 + 2], %g2
112     jmpl    [%g2 + 5], %o7
116     mov     1, %g5
120     or      %i0, %o0, %g6
124     andcc   %g6, 3, %g3
128     bne,a   .L16
132     mov     %o0, %i1
136     sra     %i0, 2, %g5
140     andncc  %o0, 3ff, %g0
144     bne     .L08
148     wr      %o0, %g0, %y
152     mov     %g5, %o0
156     noop
160     noop
164     rd      %y, %g5
168     andcc   %g0, %g0, %g2
172     call    .L07
```

```
176     mulscc  %g2, %o0, %g2
180     mulscc  %g2, %g0, %g2
184     mov     %o0, %g3
188     mov     %g5, %o0
192     mov     %g3, %g5
.L05:   rd      %y, %g3
200     sll     %g2, a, %o4
204     srl     %g3, 16, %g3
208     orcc    %o4, %g3, %o4
212     sra     %g2, 16, %g2
216     bge,a   .L11
220     addcc   %g2, %g0, %g0
224     ba      .L11
228     cmp     %g2, -1
.L06:   mulscc  %g2, %o0, %g2
.L07:   mulscc  %g2, %o0, %g2
240     mulscc  %g2, %o0, %g2
244     mulscc  %g2, %o0, %g2
248     mulscc  %g2, %o0, %g2
252     mulscc  %g2, %o0, %g2
256     mulscc  %g2, %o0, %g2
260     mulscc  %g2, %o0, %g2
264     jmp     [%o7 + 4]
268     mulscc  %g2, %o0, %g2
.L08:   andncc  %g5, 3ff, %g0
276     wr      %g5, %g0, %y
280     bne     .L09
284     andcc   %g0, %g0, %g2
288     call    .L07
292     mulscc  %g2, %o0, %g2
296     ba      .L05
300     mulscc  %g2, %g0, %g2
.L09:   call    .L06
308     mulscc  %g2, %o0, %g2
312     call    .L06
316     mulscc  %g2, %o0, %g2
320     call    .L07
324     mulscc  %g2, %o0, %g2
328     mulscc  %g2, %g0, %g2
332     orcc    %g5, %g0, %g0
336     rd      %y, %o4
340     bge     .L10
344     orcc    %o4, %g0, %g0
348     sub     %g2, %o0, %g2
.L10:   bge     .L11
356     addcc   %g2, %g0, %g2
360     cmp     %g2, -1
.L11:   bne,a   .L16
368     mov     %o0, %i1
372     mov     1, %g5
.L12:   mov     %o4, %i0
380     jmp     [%i7 + 8]
384     restore %g0, %g0, %g0
388     noop
392     mov     %g0, %o1
.L13:   jmpl    [%g1 + b6e], %o7    ;;  global fun: SYSTEM::*%>=$ANY-STUB
400     mov     %i0, %o0
404     cmp     %o0, %g1
408     be,a    .L03
412     tsubcc  %i0, 4, %g6
416     mov     1, %g5
.L14:   ba      .L12
424     mov     4, %o4
428     mov     4, %o1
.L15:   jmpl    [%g1 + c8e], %o7    ;;  global fun: SYSTEM::*%-$ANY-STUB
436     mov     %i0, %o0
440     ba      .L04
444     ld      [%i4 + 2d], %o4     ;;  FACTORIAL
448     mov     %o0, %i1
.L16:   jmp     [%g1 + cd6]    ;;  global fun: SYSTEM::*%*$ANY-STUB
456     restore %g0, %g0, %g0
115
```