Brian's Concise Common Lisp Reference Sheet

Basic List Stuff

List Constuction

```
> (list 'a 'b 'c)                   -->  (A B C)
> (list 'my (+ 2 1) "sons")         -->  (MY 3 "sons")
> (cons 'a nil)                     -->  (A)
```

List Alteration

```
> (cons 'a '(b c d))                -->  (A B C D)
> (append '(a b) '(c d) '(e))       -->  (A B C D E)
> (adjoin 'z '(a b c))              -->  (Z A B C)
> (adjoin 'b '(a b c))              -->  (A B C)

> (reverse '(a b c))                -->  (C B A)
> (sort '(0 2 1 3 8) #'>)           -->  (8 3 2 1 0)
> (substitute 'b 'a '(a x a y))     -->  (B X B Y)

1. sort is destructive, so use w/ copy-list!!
```

List Access

```
> (length '(a b c))                 -->  3
> (count 'a '(a b b a a))           -->  3

> (car '(a b c))                    -->  A
> (cdr '(a b c))                    -->  (B C)
> (third '(a b c d))                -->  C    ; to tenth
> (last '(a b c))                   -->  (C)
> (butlast '(a b c d))              -->  (A B C)
> (butlast '(a b c d) 2)            -->  (A B)
> (nth 0 '(a b c))                  -->  A    ; 0-indexed!
> (nthcdr 2 '(a b c))               -->  (C)  ; 0-indexed!

> (subseq '(a b c d e) 1)           -->  (B C D E)
> (subseq '(a b c d e) 1 3)         -->  (B C)
```

Removing Items

```
> (setf lst '(c a r a t))           -->  (C A R A T)
> (remove 'a lst)                   -->  (C R T)
> lst                               -->  (C A R A T)
> (setf lst (remove 'a lst))        -->  (C R T)
> lst                               -->  (C R T)
```

Stack Ops

```
1. These are all *destructive*!!!

> (setf x '(b))                     -->  (B)
> (push 'a x)                       -->  (A B)
> x                                 -->  (A B)
> (setf y x)                        -->  (A B)
> (pop x)                           -->  A
> x                                 -->  (B)
> y                                 -->  (A B)
```

```
> (setf x '(a b))                    -->  (A B)
> (pushnew 'c x)                     -->  (C A B)
> x                                  -->  (C A B)
> (pushnew 'a x)                     -->  (C A B)
```

<u>List Copying</u>

```
> (setf x (copy-list lst))
```

<u>Set Ops</u>

```
> (member 'b '(a b c))                        --> (B C)
> (union '(a b c) '(c b s))                   --> (A C B S)
> (intersection '(a b c) '(b b c))            --> (B C)
> (set-difference '(a b c d e) '(b e))        --> (A C D)

> (member 'a '((a b) (c d)) : key #'car)
((A B) (C D))
```

1. If T, member returns list starting w/ looked-for item
2. W/ 'key' in member, the fn is applied to every member of list *before* the 'member' comparison!

Control Structures, Loops, Code Blocks

<u>if-then Stmts</u>

```
(if [test] [then do this] [else do this])
> (if (listp '(a b c))
      (+ 1 2)
      (+ 5 6))                  -->  3
```

<u>when/unless Statements</u>

```
(when ([test])
   ([do this])
   ([do this])
   ([do this, which returns  value]))
```

1. Equivalent to an if stmt + a progn

<u>Case Statements</u>

```
(cond ((= x 1)  (format t "One.     ~%"))
       ((= x 2)  (format t "Two.     ~%"))
       ((= x 3)  (format t "Three.   ~%"))
       (t        (format t "Default. ~%")))
```

1. Each case has an implicit progn!

<u>Case Statements With Constants</u>

```
(defun month-length (mon)
   (case mon
      ((jan mar may jul aug oct dec)  31)
      ((apr jun sep nov)  30)
      (feb  (if (leap-year) 29 28))
      (otherwise "Unknown Month.")))
```

Simple 'do' Iteration

```
(do (([a var] [initial val] [how to update it each time]))
    ([test for stop] [what to eval when stops])
   ([stuff to do each iteration]))

1. can include multiple ([var] [init] [update]) stmts!

(defun showSquares (start end)
   (do ((i start (+ i 1))
       ((> i end) 'done)
      (format t "~A ~A ~%" i (* i i))))

> (showSquares 2 5)
2 4
3 9
4 16
5 25
DONE
```

Iterate Over Integers

```
> (dotimes (x 5 'done)          -->   0 1 2 3 4
     (format t "~A " x))              DONE
```

Iterate Over List Elements

```
> (dolist (x '(a b c d) 'done)   -->   A B C D
     (format t "~A " x))              DONE
```

Forced Exit From Loop

```
> (dolist (x '(a b c d e))       -->   A B C
     (format t "~A " x)               DONE
     (if (eql x 'c)
         (return 'done)))
```

Block That Only Returns *Last* Expression

```
(progn
   ([do this])
   ([do this])
   ([do this]))
```

Forced Exit From a Block

```
> (block head
     (format t "Here we go.")
     (return-from head 'idea)
     (format t "We'll never see this."))
Here we go.
IDEA
```

Pause Evaluation

```
(sleep 6)  ; pauses for 6 seconds
```

Variables

Value Assignment

```
(setf [var] [value])
> (setf foo (list 'a 'b 'c))
> (setf (car foo) 'n)
> foo                          -->  (N B C)

(setf  a b                     (setf a b)
       c d      equiv to       (setf c d)
       e f)                    (setf e f)
```

Pointer Behavior	`> (setf x '(a b c))`	-->	(A B C)
	`> (setf y x)`	-->	(A B C)
	`> (eql x y)`	-->	T
Local Vars	`> (let ((x 1) (y 2))` `    (+ x y))`	-->	3

Local Vars With Dependence	`> (let* ((x 1)`
	`         (y (+ x 1)))`
	`    (+ x y))`
		-->	3

Global Vars	`(defparameter *maxHits* 93)  ; * notation = convention`		
Global Constants	`(defconstant limit 30)`		

Functions

Basic Functions

```
(defun name (parameters)
   (what to do))

(defun is-a-member (obj lst)
   (if (null lst)
       nil
       (if (eql obj (car lst))
           lst
           (is-a-member obj (cdr lst)))))
```

Multiple Variable Arg Lists

```
(defun foo (x &rest y)
   (dolist (num y)
     (format t "~A * ~A = ~A ~%" x num (* x num))))
```

1. y is set to a list of the rest of the args after x

```
> (foo 4 5)
4 * 5 = 20

> (foo 4 5 10 20)
4 * 5 = 20
4 * 10 = 40
4 * 20 = 80
```

Optional Arguments

```
(defun foo (x &optional y)
   (list x 'is y))

(defun bar (x &optional (y 'green))
   (list x 'is y))

> (foo 'lisp)              -->   (LISP IS NIL)
> (bar 'lisp)              -->   (LISP IS GREEN)
> (foo 'lisp 'fun)         -->   (LISP IS FUN)
> (bar 'lisp 'fun)         -->   (LISP IS FUN)
```

Lambda Expressions

```
> ((lambda (x) (+ x 100)) 1)
101
> (funcall #'(lambda (x) (+ x 100))
           1)
101
```

Apply a Fn to Multiple Args

```
> (apply #'+ '(1 2 3))        -->  6
> (funcall #'+ 1 2 3)         -->  6
```

Apply Fns to Lists

```
> (mapcar #'(lambda (x) (+ x 10))
          '(1 2 3))
(11 12 13)

1. Works only 'til *any* list runs out!

> (mapcar #'list '(a b c) '(1 2 3 4))
((A 1) (B 2) (C 3))
```

Apply Fn to Successive cdr's

```
> (maplist #'(lambda (x) x) '(a b c))
((A B C) (B C) (C))
```

Return Multiple Values

```
> (values 'a nil (+ 2 4))
A
NIL
6
```

Receive Multiple Values

```
> (multiple-value-bind (x y z)
                          (values 1 2 3)
     (list x y z))
(1 2 3)
```

Pass on Multiple Values

```
> (multiple-value-call #'+ (values 1 2 3))
6
```

Math, Equality and Category Tests

Incrementing & Decrementing

```
> (setf x 5)                  -->  5
> (incf x)                    -->  6
> x                           -->  6
> (decf x 4)                  -->  2
```

Basic Math Stuff

```
> (abs -6.3)                  -->  6.3
> (mod 23 5)                  -->  3
> (mod 25 5)                  -->  0
> (max 1 2 3 4 5)             -->  5
> (min 1 2 3 4 5)             -->  1
> (expt 2 5)                  -->  32
> (float 1)                   -->  1.0
> (float .5)                  -->  0.5
> (float 2/3)                 -->  0.6666667
> (numerator 2/3)             -->  2
> (denominator 2/3)           -->  3
```

```
> (round 2.5)                          -->  2
                                            0.5
> (round 1.5)                          -->  2
                                            -0.5
> (floor 2.6)                          -->  2
                                            .5999999
> (ceiling 2.6)                        -->  3
                                            -0.4000001
> (truncate 1.3)                       -->  1
                                            0.29999995

1. If equidistant, round returns nearest *even* integer
```

Trig Functions

```
sin   cos   tan       sinh   cosh   tanh
asin  acos  atan      asinh  acosh  atanh
```

Random #s

```
1. (random n) returns a number x, where 0 <= x <= n,
     of the same type as n

> (random 6)                           -->  4
> (random 6.0)                         -->  3.0013876
```

Equality Tests

```
> (eql (cons 'a nil) (cons 'a nil))      -->  NIL
> (equal (cons 'a nil) (cons 'a nil))    -->  T

1. eql   : same object in memory
   equal : lists have same members
```

Type Tests

```
> (typep 27 'integer)                  -->  T
```

Every & Some

```
> (every #'oddp '(1 3 5))              -->  T
> (some #'evenp '(1 2 3))              -->  T
> (every #'> '(1 3 5) '(0 2 4))        -->  T
```

is-a Tests

```
null                numberp
listp               floatp
zerop               integerp
plusp               graphic-char-p
minusp              alphanumericp
upper-case-p        alpha-char-p
lower-case-p
```

Arrays and Vectors

Basic Arrays

```
> (setf arr (make-array '(2 3) :initial-element nil))
#2A((NIL NIL NIL) (NIL NIL NIL))
> (aref arr 0 0)
NIL                       ; zero-indexed!
> (setf (aref arr 0 0) 'b)
B
> arr
#2A((B NIL NIL) (NIL NIL NIL))
> (aref arr 0 0)
B
```

Basic Vectors

```
> (setf vec (make-array 4 :initial-element nil))
#(NIL NIL NIL NIL)
> (setf (svref vec 1) 'blah)
BLAH
> vec
#(NIL BLAH NIL NIL)
> (svref vec 1)
BLAH

> (vector 'blah 34 "hello")
#(BLAH 34 "hello")
```

Structs

Basic Structs

```
> (defstruct point x y)
POINT
> (setf p (make-point :x 0 :y 0))
#S(POINT :X 0 :Y 0)
> (point-x p)
0
> (setf (point-x p) 2)
2
> (point-x p)
2
```

1. The defstruct in the example above implicitly defines make-point, point-p, copy-point, point-x, and point-y!

Custom Print Fns for Structs

```
(defstruct (point (:print-function print-point))
   (x 0)
   (y 0))

(defun print-point (point stream depth)
   (format stream "<~A, ~A>" (point-x p) (point-y p)))
```

1. This defines how to print a structure when it's gotta be displayed, e.g. by the top-level

```
> (make-point)
<0,0>
```

Strings

Basic String Fns

```
> (format nil "~A or ~A" "x" "y")       -->  "x or y"
> (concatenate 'string "x " "y")        -->  "x y"

> (char "abc" 1)                        -->  #\b
> (char "a b c" 1)                      -->  #\space
> (position #\a "fantasia")             -->  1
```

```
> (setf str "merlin")
> (setf (char str 3) #\k)                  -->  #\k
> str                                      -->  "merkin"

> (string-capitalize "foo")                -->  "Foo"
> (string-upcase "foo")                    -->  "FOO"
> (string-downcase "Foo")                  -->  "foo"

> (equal "fred" "fred")                    -->  T
> (equal "fred" "Fred")                    -->  NIL
> (string-equal "fred" "Fred")             -->  T

> (sort "elbow" #'char<)                   -->  "below"
> (remove-duplicates "abracadabra")        -->  "cdbra"
```

String Comparisons

string=, string/=, string<, string>, string<=, string>=

1. Case-sensitive!

Printing

Basic Printing

(**format** [where to output] [a $ template] [other args])

```
> (format t "~A plus ~A equals ~A.~%" 2 3 (+ 2 3))
2 plus 3 equals 5.
NIL  ; returned since format was called at top level
```

Printing Decimals

1. Use ~F template
2. Rounding behavior not guaranteed!

5 Args

1.	total # chars to print	[all]
2.	# digits after decimal	[all]
3.	# digits to shift decimal to left	[none]
4.	char to print if too long for #1	[ignore #1]
5.	char to print to left at start	[blank]

```
> (format nil "~,2F"  26.21875)
"26.22"
```

Other String Templates

~%	newline	~A	variable
~3%	3 newlines	~S	print as string

Input, Files, System Commands

read-line

1. Reads input up to newline, returning it as a string
2. Second return val is T iff readline ran out of input before seeing newline

```
> (progn
     (format t "Your name: ")
     (read-line))
Your name: Brian Scholl
"Brian Scholl"
NIL
```

read

1. Reads exactly one lisp expression
2. Could be less or more than one line
3. Must read valid LISP syntax!

```
> (read)
(a
b
c)
(A B C)
```

Read From a String

1. Takes a string and returns 1st expression it sees
2. 2nd return val = position where stopped reading

```
> (read-from-string "a b c")
A
2
```

Read Specific Input Type

```
(defun askForANumber ()
   (format t "Enter a number: ")
   (let ((foo (read)))
      (if (numberp foo)
          foo
          (askForANumber))))
> (askForANumber)
Please enter a number: a
Please enter a number: (ho hum)
Please enter a number: 52
52
```

File I/O

```
(setf mypath (make-pathname : name "myfile.txt"))
(setf mystream (open mypath :direction : output
                             :if-exists : supersede))
(format mystream "Boo! ~2%")
(close mystream)

% cat myfile.txt               -->  Boo!
```

1. ":direction : output" if only writing to file
   ":direction : input " if only reading a file
   ":direction : io   " if both

System File Functions

```
(setf mypath  (make-pathname :name "myfile.txt"))
(setf newpath (make-pathname :name "newfile.txt"))
(delete-file mypath)
(rename-file mypath newpath)
```

Transcribe a Lisp Session

```
(dribble mypath)
```

Macros

Define a Macro

```
(defmacro nil! (x)
   (list 'setf x nil))
```

1. Typing "(nil! x)" is the same as typing "(setf x nil)".

```
> (setf x 3)                         -->  3
> x                                  -->  3
> (nil! x)                           -->  NIL
> x                                  -->  NIL
```

Expand a Macro

```
> (macroexpand-1 '(nil! x))
(SETF X NIL)
T
```

My Library Functions

My Library Fns

```
> (nthmost 2 '(0 2 1 3 8))           -->  3
> (starts-with '(a b c) 'a)          -->  T
> (starts-with '(a b c) 'b)          -->  NIL
> (rotate-left '(a b c d))           -->  (B C D A)
> (rotate-right '(a b c d))          -->  (D A B C)
> (insert-between 'x '(a b c d))     -->  (A X B X C X D)

> (permutations '(1 2))
((1 2) (2 1))
> (permutations '(1 2 3))
((3 1 2) (3 2 1) (2 1 3) (2 3 1) (1 2 3) (1 3 2))

> (palindrome-p '(a))                -->  T
> (palindrome-p '(a a))              -->  T
> (palindrome-p '(a b a))            -->  T
> (palindrome-p '(a b b a))          -->  T
```

Not Included (Yet)!

Not Included Yet!

- Assoc Lists
- Dotted Lists
- Property Lists
- Hash Tables
- Garbage
- Exceptions
- Symbol Names w/ Whitespace
- Tree Functions
- Coerce
- Compile
- Loop
- Packages
- Binary Streams
- Type Specifiers
- Explicit Type Declarations
- CLOS
- pprint
- Backquotes
- Macro Design Issues