CS334 Lisp Cheat Sheet

Lisp (**Lis**t **P**rocessing)
> • uses linked lists as fundamental data structure;
> • lists are especially flexible as their elements may be virtually anything!

Numbers

- integers
- reals (decimal and scientific notation)
- rationals

Standard arithmetic functions

+, -, *, /,
floor, ceiling, mod,
sin, cos, tan, sqrt,
etc.

- always **prefix notation**
- think "application of a function to its arguments"
- return a number according to type contagion

Examples.

```
> (+ 3 3/4)
  15/4

> (/ 10.0 5)
  2.0

> (/ 10 5 5 1)
  2/5                    ;the functions +-*/ all accept multiple arguments
```

Symbols

- a symbol is just a string of characters.
- As long as you stick to letters, digits, and hyphens, you'll be safe.

Examples.
    T1
    Trouble-234
    a
    b

Lisp is not case sensitive: Trouble-234 is like TROUBLE-234 is like trouble-234, etc.

Special symbols

There are two special symbols, **t** and **nil**.

-    the value of t is defined always to be t.
-    the value of nil is defined always to be nil.

Note that: **nil means false; anything else means true**.

nil is also used in other contexts.

Conses

A **cons** is just a **two-field record**.

        CAR = "contents of address register"
        CDR = "contents of decrement register"

Examples.

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

In the examples above, 4 and 5 are **atoms**. The conses are **dotted lists**. Usually we just use cons to create lists, as below.

Lists

- built out of conses
- car is the head of the list
- cdr is the rest – either another cons or nil

Examples.

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

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

The last example is exactly equivalent to the call (list 4 5 6).

"nil" as an empty list

In addition to having the meaning "false", nil also represents an empty list.

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

Can also use '( ) to mean an empty list.

A special form: quote

(cons 'a 'b) → (A . B)

(cons a b) has the meaning "cons the values of the variables a and b"

More examples.

(cons 'a ' (b c d)) → (A B C D)

(cons a (b c d)) would be taken to mean "cons the value of the variable a with the result of applying the function b to variables c and d"

Can also say:

(cons (quote a) (quote b))

and

(cons (quote a) (quote (b c d)))

Defining simple functions

To define the function: $f(x, y) = x^2 + y^2$

> (defun sum-of-squares (x y) (+ (* x x) (* y y)))
 SUM-OF-SQUARES

> (sum-of-squares 2 3)
  13

Line breaks are ok!
> (defun sum-of-squares (x y)
        (+ (* x x)
           (* y y)))

The return value of any function is the value of the last function/line executed.

We will not use the following, but gives you some idea of Lisp's flexibility:
Note that you can define functions with optional arguments and default values for arguments.

You can also define a function to accept any number of arguments by ending its argument list with an &rest parameter. Lisp will collect all arguments not otherwise accounted for into a list and will bind the &rest parameter to that list.

For example,
> (defun give-me-the-rest (x &rest y) y)
  GIVE-ME-THE-REST

> (give-me-the-rest 3)
  NIL

> (give-me-the-rest 4 5 6)
  (5 6)

Forms and the top-level loop

The things that you type to the Lisp interpreter are called forms

Read-eval-print loop:
- if form = atom(integer, string, symbol – i.e., not a list), then evaluate immediately
- if form = list, the car of the list is a function, which is applied to its arguments

Special forms:
- these look like function calls, but aren't
- they include defun and quote, among others (if, do, setq and others that we'll see later)

Now, before we can write more interesting functions, we need to know more about the functions Lisp provides for us…

Functions that get at list elements

- **car** returns the first element of a list (recall that it returns the first element of a cons)
- **cdr** returns the rest

Note that you can use the functions **first** and **rest** instead of car and cdr.

Examples.
(car ' (a b c d)) -> A
(cdr ' (a b c d)) -> (B C D)
(car ' ((a b c) d)) -> (A B C)
(cdr ' ((a b c) d)) -> (D)

car and cdr of nil, ' ( ) and ' (nil) are defined to be nil, but while nil and ' ( ) are equal, '(nil) is not equal to either.

---

caar, caaar, caaar, cddr, cadr, caddr, …

It is common to want to apply car or cdr to the result of an application of car or cdr. For this reason, there are shorthand function names that will allow you to do so.

- cadr returns the car of the cdr of the argument

- caar returns the car of the car of the argument
- cdar returns the cdr of the car

Examples.
(cadr ' (a b c d)) -> B
(cdar ' ((a b c) d)) -> (B C)

You can combine up to 4 cars/cdrs in this way.

More ways to get at list elements

If x is a list,
- (nth 4 x) returns the element at index 4 (assuming 0-indexing); substitute any valid number for 4, of course.
- (fifth x) returns the same value – i.e., the fifth element in the list; can also use first, second, …, tenth.

Building lists

- cons
- list
- append – concatenates two lists

Examples of append.
(append ' (1 2 3) '(4 5 6)) -> (1 2 3 4 5 6)
(append ' (1 2) '((3 4) 5 6)) -> (1 2 (3 4) 5 6)

Some other useful list functions

- length, list-length – both of these return the length of a list; they differ in the way they handle circular lists (list-length returns nil, while length might not return at all).
- reverse – reverses a list

Examples.
(length ' (a b (c d))) -> 3
(reverse ' (a b (c d))) -> ((C D) B A)

Conditionals

There are several ways to write conditionals in Lisp. For example, we can use the **cond** macro:

```
(cond   (test-1 consequent-1)
        (test-2 consequent-2)
        (test-3 consequent-3)
…)
```

Examples.
(cond ((eq 0 1) 2) (t 3)) -> 3
(cond ((eq 0 0) 2) (t 3)) -> 2

The first clause whose test evaluates to non-nil is selected. All others are ignored.

Remember that the tests are not returning "true" and "false". They return either nil, which is taken to mean false or a non-nil value that is presumed to mean true.

_____

Predicates

-   predicates are functions that return either nil or a non-nil value to mean "true"
-   predicates can be viewed as boolean functions

Examples of useful predicates.
-   eq – returns whether two operands are equals (see next section).
-   atom – returns whether its argument is an atom or not.
-    <, <=, etc.- the usual math comparisons.

(< 1 2) -> t
(< 2 1) -> nil

(atom 'A)   -> t
(atom nil) -> t
(atom (car '(A nil)))   -> t
(atom (cdr '(A nil)))   -> nil

_____

The many faces of equality…

Different functions test for different kinds of equality. For us,
eq will be sufficient. (Note, however, that list comparison should be done with **equal** if
you ever want to compare the contents of two lists.)

> (eq 'a 'a)
T
> (eq 'a 'b)
NIL
> (eq 3 3)
T

The **equal** predicate is equivalent to eql for symbols and numbers. It is true for two
conses if and only if their cars are equal and their cdrs are equal. It is true for two
structures if and only if the structures are the same type and their corresponding fields are
equal.

;; lisp comparison rather than standard comparison
> (eq '(a b c) '(a b c))
NIL
> (equal '(a b c) '(a b c))
T

More interesting functions

Now we have all the basic tools to write more interesting functions. We know about
Lisp's basic data types, we know a few functions that can manipulate them, and we have
the ability to express conditionals, which also allow us to write recursive functions.

Example. A simple function to calculate factorial:

```
(defun factorial (n)
    (cond ((= n 0) 1)
          (T (* n (factorial (- n 1)))))))
```

Interesting fact: if a functions is tail-recursive, then the tail-recursion is automatically
eliminated by Lisp.

How to write and run a Lisp program

This quick summary gives everything you need to write and run a Lisp program.

Using CMU Common Lisp on a unix machine:

Type **lisp** at the shell prompt
- start typing lisp functions **or**
- **[BETTER]** write your functions in a file (good to use .lisp extension) and then within Lisp: (**load** "filename.lisp")
- **[ALSO GOOD]** just pipe the file into lisp: lisp < "filename.lisp"

To get information about a function, type (**describe** 'fctn-name)

To get out of lisp, type **(quit)**

If you've made an error and have been thrown into debugging mode, you can either just keep going, or type :q

Using clisp on a Mac:

Everything is the same as above. Just type **clisp**, rather than lisp.

Functions as arguments

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

Examples.

> (mapcar 'not '(t nil t nil t nil))
  (NIL T NIL T NIL T)

> (mapcar #'(lambda (x) (+ x 2)) '(1 2 3 4 5))
  (3 4 5 6 7)

Note that **lambda** can be used to create temporary, nameless functions.