Intro to Lisp Part II

loosely based on

The Little Schemer

by Donald Friedman & Matthias Felleisen

T or Nil: (latp *l*)

- Where *l* is (Jack Sprat could eat no chicken fat)

  T - each S-expression in *l* is an atom

T or Nil: (latp *l*)

- Where *l* is ((Jack) Sprat could eat no chicken fat)

nil - since the (car *l*) is a list

T or Nil: (latp *l*)

- Where *l* is (Jack (Sprat could) eat no chicken fat)

  nil - since one of the S-expressions in list *l* is a list

T or Nil: (latp *l*)

- Where *l* is ( )

  T - since it does not contain a list

latp return t when its argument is a list of atoms

latp return t when its argument is a list of atoms

Why end in p? p stands for predicate. A predicate is a function that returns t or nil based on testing. (t or non-nil)

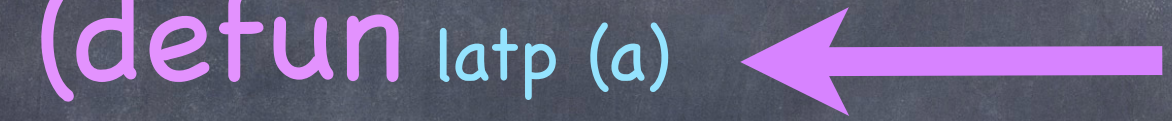
Now let's write the function latp

Defining Functions

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

Defining Functions

defining a function and its arguments

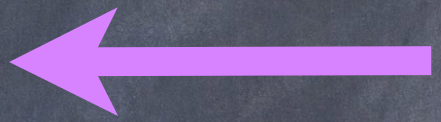


```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

Defining Functions

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

asks questions



Defining Functions

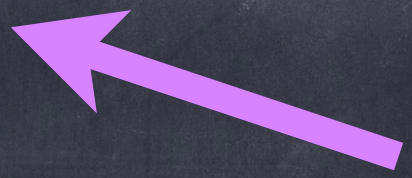
```
(defun latp (a)
 (cond
  ((null a) t)
  ((atom (car a)) (latp (cdr a)))
  (t nil)))
```

Asks if the argument a is empty.
If a is empty, the function returns t. If a is not empty, then move on to ask the next question.

Defining Functions

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

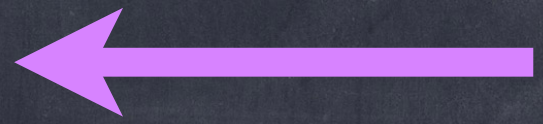
Asks if (car a) is an atom. If t, the function is called again with the (cdr a). If (car a) is not an atom, then move on to ask the next question.



Defining Functions

```
(defun latp (a)
 (cond
  ((null a) t)
  ((atom (car a)) (latp (cdr a)))
  (t nil)))
```

Since all other questions returned nil, then the function returns nil.



(latp '(1 2 3))

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

(latp '(1 2 3))

```
(defun latp (a)
 (cond
  ((null a) t)
  ((atom (car a)) (latp (cdr a)))
  (t nil)))
```

So, (latp '(1 2 3)) returns t

```
(latp '(1 2 3))
 (null '(1 2 3))                    nil
 (atom (car '(1 2 3))               t
  (latp '(2 3))
   (null '(2 3))                    nil
   (atom (car '(2 3))               t
    (latp '(3)
     (null '(3))                    nil
     (atom (car '(3))               t
      (latp '())
       (null '())                   t
      (latp '())                    t
    (latp '(3)                      t
  (latp '(2 3))                     t
(latp '(1 2 3))                     t
```

(latp '(1 (2) 3))

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

(latp '(1 (2) 3))

```
(defun latp (a)
  (cond
    ((null a) t)
    ((atom (car a)) (latp (cdr a)))
    (t nil)))
```

```
(latp '(1 (2) 3))
  (null '(1 (2) 3))                nil
  (atom (car '(1 (2) 3))           t
    (latp '((2) 3))
      (null '((2) 3))              nil
      (atom (car '((2) 3))         nil
    (latp '((2) 3))                nil
(latp '(1 (2) 3))                  nil
```

So, (latp '(1 (2) 3)) returns nil

latp is a list of atoms
what is a list of lists?

How about llip

Write llip

What Lisp primitive do we need?

listp - (listp arg) - returns t when arg is a list; otherwise, nil

Write llip

```
(defun llip (a)
  (cond
    ((null a) t)
    ((listp (car a)) (llip (cdr a)))
    (t nil)))
```

(llip '((1) (2) (3)))

```
(defun llip (a)
  (cond
    ((null a) t)
    ((listp (car a)) (llip (cdr a)))
    (t nil)))
```

So, (llip '((1) (2) (3))) returns t

```
(llip '((1) (2) (3)))
  (null '((1) (2) (3)))                nil
  (list? (car '((1) (2) (3))))          t
    (llip '((2) (3)))
      (null '((2) (3)))                nil
      (list? (car '((2) (3)))           t
        (llip '((3))
          (null '((3)))                nil
          (list? (car '((3)))           t
            (llip '())
              (null '())                t
            (llip '())                  t
        (llip '((3))                    t
    (llip '((2) (3)))                   t
(llip '((1) (2) (3)))                   t
```

Write x! - (fact x)

Factorial

4! = 4 * 3 * 2 * 1

x is a non-negative integer

remember by definition 0! = 1

Write x! - (fact x)

```
(defun fact (x)
  (cond
    ((equalp x 0) 1)
    (t (* x (fact (- x 1))))))
```

Write (fibon n)

Fibonacci Sequence: 1, 1, 2, 3, 5, 8, ...

(fibon n) returns the nth term in the sequence

n is a positive integer

What new Lisp primitives do we need?

Write (fibon n)

What new Lisp primitives do we need?

or - logical operation

(or arg1 arg2 arg3 .... argn)

where each argument returns nil or non-nil

or returns non-nil when at least one argument is non-nil; otherwise, nil

Write (fibon n)

```
(defun fibon (n)
  (cond
    ((or (equalp n 1) (equalp n 2)) 1)
    (t (+ (fibon (- n 1)) (fibon (- n 2))))))
```

Write (remlist l)

(remlist '(1 2 3)) returns (1 2 3)

(remlist '(1 (2) 3)) returns (1 3)

(remlist '((1) (2) (3)) returns ()

Write (remlist l)

(remlist '(1 2 3)) returns (1 2 3)

(remlist '(1 (2) 3)) returns (1 3)

(remlist '((1) (2) (3)) returns ()

Remlist returns its argument with all lists removed.

l is a list

Write (remlist l)

```
(defun remlist (l)
  (cond
    ((null l) nil)
    ((listp (car l)) (remlist (cdr l)))
    (t (cons (car l) (remlist (cdr l))))))
```

New Lisp Primitives

defun	cond
t - new use	
listp	
and, or, not	

Next Step?

read "Defining Lisp Functions" in Lisp Primer
read "Recursion" section in Lisp Primer
try out the exercises at the end of the "Defining" section
write some functions!!!!
keep track of your questions & confusions