- is one of the oldest languages (1958)
- is the second high level language developed (behind FORTRAN (1957))
- was introduced for symbolic manipulation
- is a functional language
- has a uniform and simple syntax

Upon starting the LISP IDE, the user sees a listener (>).
The user then gives a LISP expression of the form:

```
(op arg1 ... argn) ; where op is a function name or special operator
```

Types of expressions:

1. Numeric Expressions
2. Symbolic Expressions
3. Boolean Expressions

Other Material:

1. Branching
2. Function Definition

and that is LISP!
Lisp uses prefix notation for operations:

```
; Mixing of types in an expression is allowed:
> (+ 2.7 10)
> 12. 7
; There is no limit on the argument count
> (+ 21 35 12 7)
> 75
; Expressions can be recursive
> (+ (* 3 51) (- 10 6))
> 19
```

We introduce two special operators:

1. The quote operator gets an expression without evaluation.
   We use this since sometimes lists represent data rather than expressions.

   ```
   > (quote (+ 3 1))
   > (+ 3 1)
   ; this can also be written "> '(+ 3 1)"
   ```

2. The setq operator assigns a value to a variable.

   ```
   > (setq x 3)
   > 3
   > (setq y (+ 1 3))
   > y
   > 4
   ; we can use this to bind an expression to a variable
   > (setq z '(+ 1 3))
   > (+ 1 3)
   ; expressions in expressions are evaluated
   > (setq z y)
   > 4
   ```

More generally, a LISP expression is either

1. an atom (number, symbol, string, etc)
2. a list (>= 0 expressions)

A common way to use lists/expression is to represent data; we then need:

1. accessors (car, cdr) = (first, rest)

2. constructors (cons, list)

The accessors car and cdr behave as follows:

```
> (setq x '(a b c d)) // this would be an error without the quote
> (a b c d)
; car (or first) gives the first item in a list
> (car x)
> a
; cdr (or rest) gives the rest of the list
> (cdr x)
> (b c d)
; how would I print the second element?
> (car (cdr x)) // shorthand: (cadr x)
> b
; we can recurse indefinitely here
> (caddr x)
> c
```

The empty list is denoted "NIL" and evaluates to false.
NIL functions expectedly with car/cdr

```
> (car NIL)
> NIL
> (cdr NIL)
> NIL
> (cdr '(c))
> NIL
```

The constructors list and cons behave as follows:

```
; list joins the following elements
> (list 1 2 3)
> (1 2 3)
; cons joins the two parameters, such that arg1=car \& arg2=cdr
> (cons 'a '(b c))
> (a b c)
> (cons (+ 1 2) '(b c))
> (3 b c)
> (cons 'a NIL)
> (a)
> (cons '(a b) '(c d))
> ((a b) c d)
> (cons 1 (cond 2 NIL))
> (1 2)
```

Boolean expressions use NIL for false and t for true

```
> (> 3 1)
> t
> (< 3 1)
> NIL
; listp evaluates type of element
> (not (listp 3))
> t
> (listp '(a b))
> t
; there is an =NIL operator 'null'
> (null (cdr '(2)))
> t
; note that it uses the most recent true value for return
> (OR NIL 3)
> 3
> (OR NIL (cdr '(c)) 7)
> 7
```

We can evaluate equality in one of three ways:

- '=' compares integers
- 'equal' compares elements (or direct values)
- 'eqL' compares underlying pointers

Now onto the big stuff. . . . . .

Functions

```
> (defun name (parameters)
      operations))
```

A very simple function to start with is the "square" function

```
> (defun square (x)
      (* x x))
> (square 3)
> 9
```

We can use the "cond" keyword to emulate a switch

```
> (defun odd (x)
      (cond ((= x 0) NIL)
        ((= x 1) t)
        (t (odd (- x 2)))))
```

In Lisp, we tend to write functions that fall into one of three types:

1. numeric (as above)
2. list
   (a) use accessors
   (b) use constructors

A quick note on numeric functions:

```
; we bind variables with the keyword "let
> (defun foo (x y)
      (let ((a (+ x y))
        (b (* x y)))
        (/ a b)))
; this binding is local to the scope of the operation
> (setq x 5)
> (+ (let ((x 3)) (+ x (* x 10))) x)
> 38
; binding is done in parallel
> (setq x 2)
> (let ((a (+ x -1))
      (b (+ a 3)}
; GIVES AN ERROR: let* binds in series \& would allow the above:
> (let ((x 3) (y (+ x 2))) (x + y)
> 12
> (let* ((x 3) (y (+ x 2))) (x + y)
> 15
```

We consider some accessor functions:

1. compute the sum of all the elements of a list:

   ```
   > (defun sumlist (L)
   (cond ((null L) 0)
     (t (+ (first L) (sumlist (rest L)}
   ; a curly brace denotes "close all paren"
   > (sumlist '(1 2 3))
   > 6
   ```

2. determine whether an element is a member of a list:

```
> (defun member (x L)
      (cond ((null t) NIL)
        ((equal x (first L)) t)
        (t (member x (rest L)}
> (member NIL ‘(a b))
> NIL
> (member ‘(a b) ‘(1 7 (a b) d))
> t
```

3. return the last element in a list

```
> (defun last (L)
      (cond ((null (rest L)) (first L))
        (t (last (rest L)}
```

We consider some constructor functions:

1. remove one occurrence of an element from a list

```
> (defun remove (elm list)
      (cond ((null list) NIL)
        ((equal elm (first list)) (rest list)
        (t (cons (first list) (remove elm (rest list)}
```

2. append a list to another

```
> (defun append (L1 L2)
      (cond ((null (last L2)) L2)
        (t (cons (first L2) (append L1 (rest L2)}
```

3. reverse a list

```
> (defun reverse (L)
      (cond ((null L) NIL)
        ((null (rest L) (first L))
        (t (append (reverse (rest L) (first L)}
```