# COMP 348
# PRINCIPLES OF PROGRAMMING LANGUAGES

**Tutorial #4**

Functional Programming

LISP

# LISP - LIST PROCESSING LANGUAGE

- An AI language developed in 1958 (J. McCarty at MIT)
- Special focus on symbolic processing and symbol manipulation
  - ❑Linked list structures
  - ❑Programs, functions are represented as lists

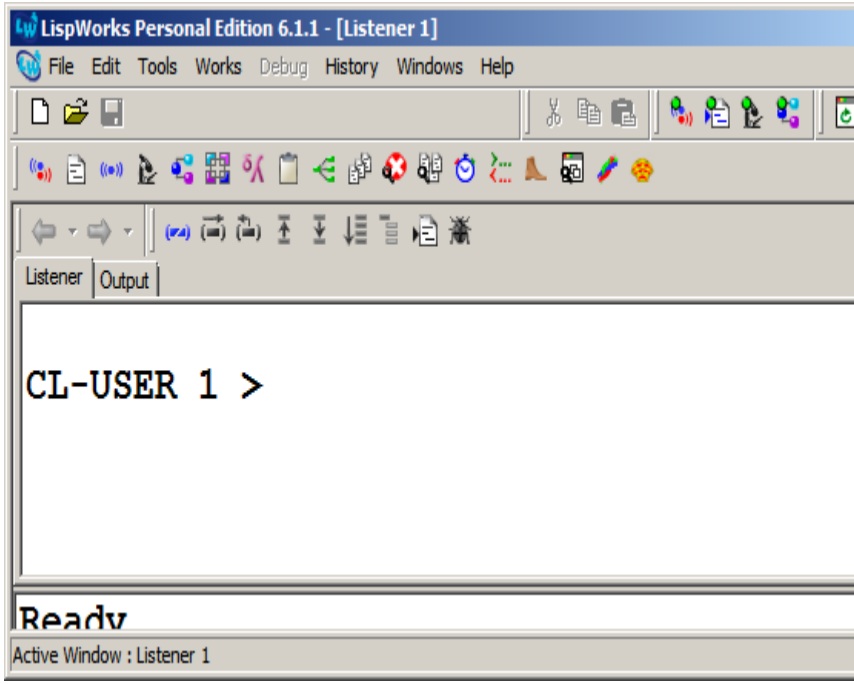
- Many AI programs now are written in C, C++, Java
  - ❑List manipulation libraries are available

- LISP programming languages
  - ❑Scheme – widely-known general-purpose Lisp dialects; supports functional and imperative programming
  - ❑Common Lisp – supports a combination of procedural, functional, and object-oriented programming (includes CLOS)

# INTERACTIVE PROGRAMMING

- Lisp prompt
- Lisp reads Lisp expressions, evaluates them according to the rules of Lisp, and prints the result.
- It's called the read-eval-print loop, or REPL for short.

It's also referred to as the top-level, the top-level listener, or the Lisp listener.



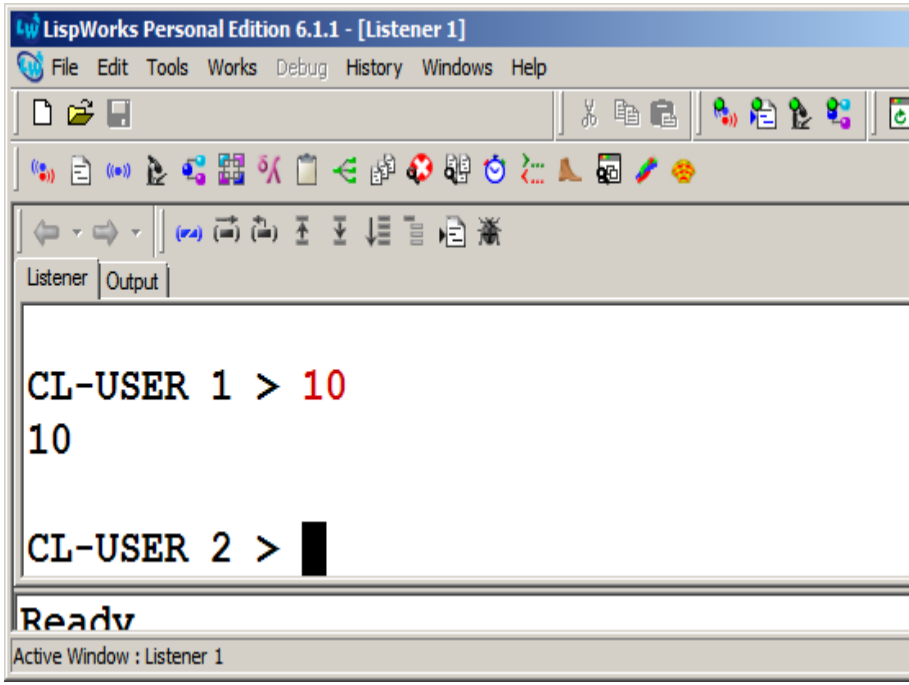
# REPL CONCEPT

- A user type 10 followed by Return and should see the output as shown in the Fig.

- The Lisp reader, the R in REPL, reads the text "10" and creates a Lisp object representing the number 10.

- This object is a self-evaluating object, which means that when given to the evaluator, the E in REPL, it evaluates to itself.

- This value is then given to the printer, which prints the 10 on the line by itself.



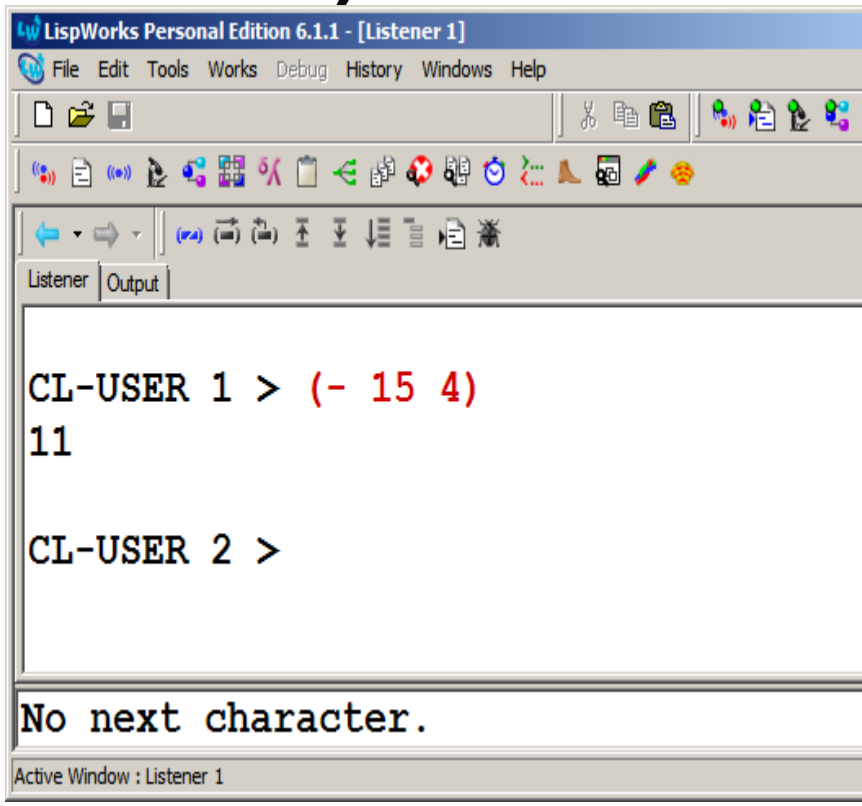
# REPL CONCEPT (CONT.)

- When a user type (- 15 4) at the Lisp prompt.

- Here we have a list of three elements, the symbol -, and the numbers 15 and 4.

- Lisp, in general, evaluates lists by treating the first element as the name of a function and the rest of the elements as expressions to be evaluated to yield the arguments to the function.

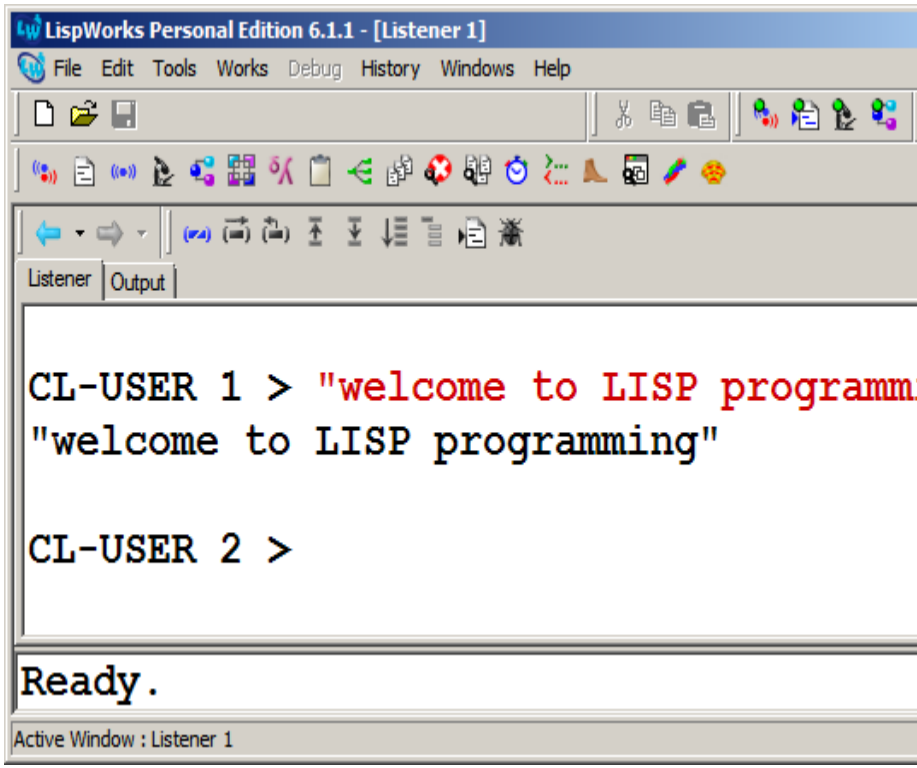
- So here, the symbol - names a function that performs subtraction. 15 and 4 evaluate to themselves and are then passed to the subtraction function, which returns 11. The value 11 is passed to the printer, which prints it.



# REPL CONCEPT (CONT.)

- Lisp reads the double-quoted string and instantiates a string object in memory that, when evaluated, evaluates to itself and is then printed in the same literal syntax.

- The quotation marks aren't part of the string object in memory--they're just the syntax that tells the reader to read a string.



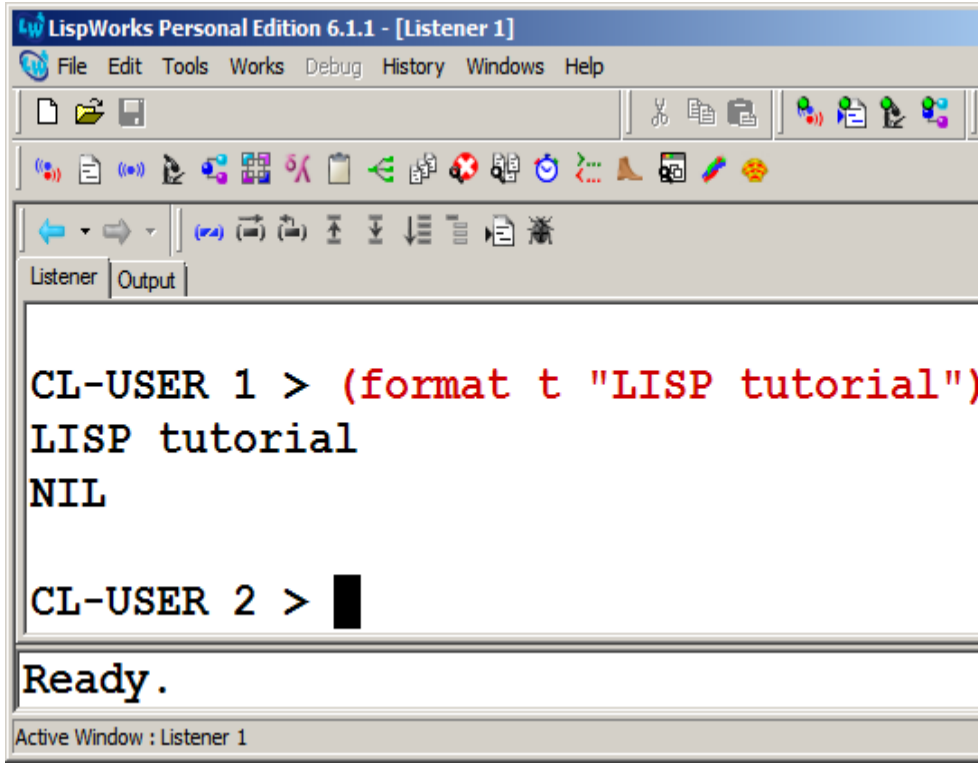
# FORMAT FUNCTION

- FORMAT takes a variable number of arguments, but the only two required arguments are the place to send the output and a string.

- If you pass t as its first argument, it sends its output to standard output.

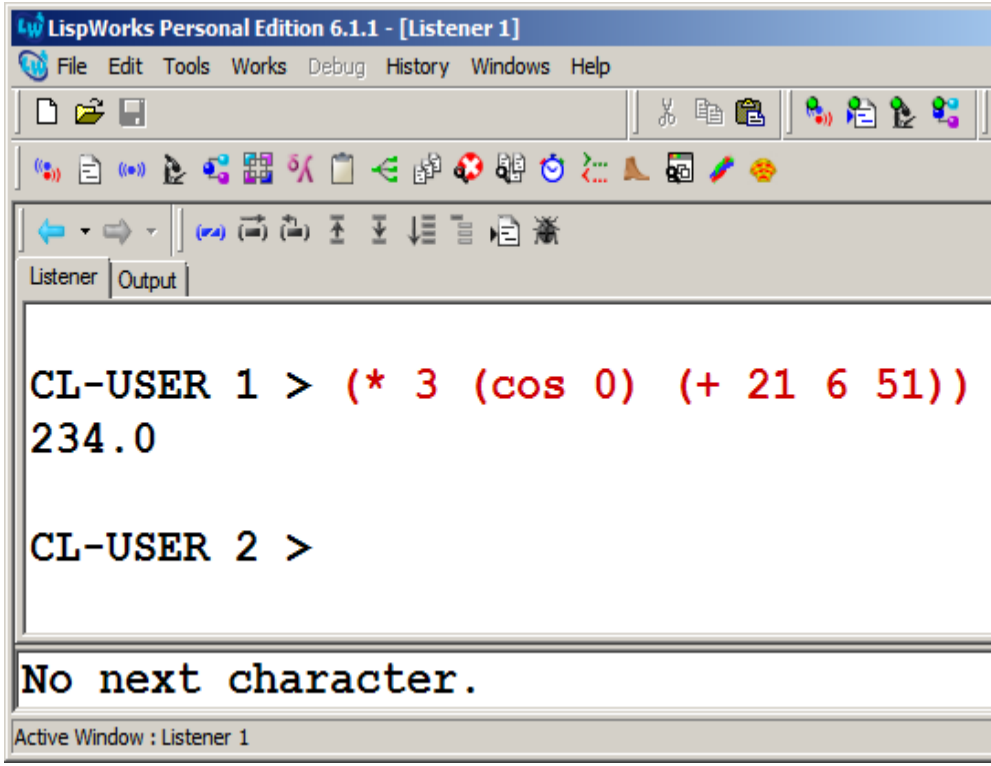
- NIL is the result of evaluating the FORMAT expression, printed by the REPL. (NIL is Lisp's version of false and/or null)

- There are other ways as well to emit output.



# LISP EXPRESSIONS

- The most common LISP form is function application.
- LISP represents a function call f(x) as (f x). For example, cos(0) is written as (cos 0).
- LISP expressions are case-insensitive. It makes no difference whether we type (cos 0) or (COS 0).
- functions, like "+" and "*", could take an arbitrary number of arguments
- Math representation: 3*Cos(0)*(21+6+51)



# LISP EXPRESSIONS (CONT.)

LispWorks Personal Edition 6.1.1 - [Listener 1]

File Edit Tools Works Debug History Windows Help

Listener | Output

```
CL-USER 1 > (+ 2/4 2/8)
3/4

CL-USER 2 > (* 2.5 1.25 -9.5 2/4)
-14.84375

CL-USER 3 > (sqrt 9)
3.0

CL-USER 4 > (reverse "sunny day")
"yad ynnus"

CL-USER 5 > (length "hello world")
11

CL-USER 6 > (not (and (= (+ 1 2) 3) (< 4 3) (= (+ 2 3) 7) ))
T

CL-USER 7 >
```

Ready.

Active Window : Listener 1

# BOOLEAN OPERATIONS

```
CL-USER 1 > (not nil)
T

CL-USER 2 > (not t)
NIL

CL-USER 3 > (and t t nil)
NIL

CL-USER 4 > (or t nil t)
T

CL-USER 5 > (and t t ())
NIL

CL-USER 6 > (and t t (or t nil))
T
```

# BOOLEAN OPERATIONS (CONT.)

```
CL-USER 7 > (setq Z '(a1 a2 a3))
(A1 A2 A3)

CL-USER 8 > (and (member 'a3 z)(member 'a2 z))
(A2 A3)

CL-USER 9 > (not (member 'a4 z))
T

CL-USER 10 > (member 'a5 Z)
NIL

CL-USER 11 > (or (member 'a4 z)(member 'a5 z))
NIL

CL-USER 12 > (or (member 'a4 z)(member 'a3 z))
(A3)

CL-USER 13 > (and (member 'a3 z)(member 'a5 z))
NIL
```

# CONSTRUCTING LISTS

Three built-in functions to create a list.

- cons: creates a list by adding an element as the head of an existing list
- list: creates a list comprised of its arguments.
- append: creates a list by concatenating existing lists.

# FUNCTION CONS

A list in Lisp is singly-linked where each node is a pair of two pointers, the first one pointing to a data element and the second one pointing to the tail of the list with the last node's second pointer pointing to the empty list.

The image cannot be displayed. Your computer may not have enough memory to open the image, or the image may have been corrupted. Restart your computer, and then open the file again. If the red x still appears, you may have to delete the image and then insert it again.

# FUNCTION LIST

Lists can be created directly with the list function, which takes any number of arguments, and it returns a list composed of these arguments.

```
CL-USER 1 > (list 1 2 'a 3)
(1 2 A 3)

CL-USER 2 > (list 1 '(2 3) 4)
(1 (2 3) 4)

CL-USER 3 > (list '(+ 2 1) (+ 2 1))
((+ 2 1) 3)

CL-USER 4 > (list 1 2 3 (list 'a 'b 4) 5)
(1 2 3 (A B 4) 5)

CL-USER 5 > (list '(+ (2 + 4))(* 5 4))
((+ (2 + 4)) 20)
```

# FUNCTION APPEND

The append function takes any number of list arguments and it returns a list which is the concatenation (join) of its arguments:

```
CL-USER 1 > (append '(1 2) '(3 4))
(1 2 3 4)

CL-USER 2 > (append '(1 2 3) '() '(a) '(5 6))
(1 2 3 A 5 6)

CL-USER 3 > (append '(1 2 3 '(a b c)) '() '(d) '(4 5))
(1 2 3 (QUOTE (A B C)) D 4 5)

CL-USER 4 > (append '(1 '2 3 a) '() '(d) '(4 5))
(1 (QUOTE 2) 3 A D 4 5)

CL-USER 5 > (append a '(1 b 2))

Error: The variable A is not of type LIST

CL-USER 6 > (append (list 'a) '(4 5 6))
(A 4 5 6)
```

We must transform **a** into a list

# ACCESSING A LIST

Only two operations are available. We can only access either the head of a list, or the tail of a list.

Operation car (also: first) takes a list as an argument and returns the head of the list.

```
CL-USER 1 > (car '((a b d) c f))
(A B D)

CL-USER 2 > (car '(a e f))
A
```

Operation cdr (also: rest) takes a list as an argument and returns the tail of the list.

```
CL-USER 3 > (cdr '((a b d) c f))
(C F)

CL-USER 4 > (cdr '(a e f))
(E F)

CL-USER 5 > (cdr '((a 1) (b 2) (c 3)))
((B 2) (C 3))
```