Common LISP Tutorial 1 (Basic)

CLISP Download

https://sourceforge.net/projects/clisp/

IPPL Course Materials (UST sir only) Download

https://silp.iiita.ac.in/wordpress/?page_id=494

Introduction

- Lisp (1958) is the second-oldest high-level programming language after Fortran.
  - *FORTRAN language invented in 1957. C & PROLOG invented in 1972.*
  - *C++ invented in 1979. JAVA invented in 1995.*
- Lisp has changed a great deal since its early days, and a number of dialects (variation or extension) have existed over its history. Today, the most widely known general-purpose Lisp dialects are Common Lisp (1980's onwards).
- It is suitable for Artificial Intelligence programs. Why?

  It processes symbolic information effectively.

Features of Common LISP

- It is machine-independent.
- It uses iterative design methodology, and easy extensibility.
- It provides convenient macro system.
- It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
- It is expression-based.
- It provides complete I/O library.
- It provides extensive control structures.
- It provides advanced object-oriented programming.

Properties of CLISP

- The basic elements of expression: **lists** and **atoms**
- Lisp program code takes the form of **lists**. A list begins with a parenthesis, contains any number of whitespace-separated elements, then a closing parenthesis. (2 3 4 5 6 7)
- Space separated elements of a list are known as **ATOMS.**
- functional programming language.
- It is both an interpreter and compiler.
- Case Insensitive

Contd….

- Expressions are evaluated in CLISP and a value is **“always”** returned.
- In Lisp, the special constant **nil** (case insensitive) all by itself represents "false". Every other expression but **nil** is considered to be “true”.
- Expressions always in Prefix form.
- In Lisp, nearly everything is a function. Even the mathematical operators. For example:

  (+ (* 2 3) 1) equals 7.
- As you can see, the functions open and close with parenthesis. The format for calling functions in Lisp is (function arg1 arg2)
- **(= 4 3)** outputs : NIL
- **(< 3 6)** outputs : T

- x ; the symbol X
- () ; the empty list
- (1 2 3) ; a list of three numbers
- ("foo" "bar") ; a list of two strings
- (x y z) ; a list of three symbols
- (x 1 "foo") ; a list of a symbol, a number, and a string
- (+ (* 2 3) 4) ; a list of a symbol, a list, and a number.
- ; This is a comment! (To comment in Lisp, prefix the line with ;)

REPL (Read–eval–print loop)

- CLISP takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user.
- The **read function** accepts an expression from the user, and parses it into a data structure in memory.
- The **eval function** takes this internal data structure and evaluates it.
- The **print function** takes the result yielded by eval, and prints it out to the user.
- The development environment then returns to the read state, creating a **loop**, which terminates when the program is closed.
- (+ 1 2 3) ➔ 6
- (* 2 (+ 1 2 3)) ➔ 12

Handling of Numbers

- Numbers (Integers, Floating-point, fractions, complex numbers, giant numbers etc.) are also regarded as expressions.
- Complex number *a*+*b*i is represented as **#C(a b)**

(+ #c(2 1) #c(3 4)) ➔ (+ #c(5 5)

- Basic Math Operation functions: +, -, *, and /
- x/y does not always return a floating point value. The result depends on the data type of “x” and “y” used.
  1. **(/ 8 4)** will return 2 (as 8 is divisible by 4)
  2. **(/ 8 3)** will return 8/3 (as 8 is not divisible by 3)
  3. **(/ 8 4.0)** will return 2.0
  4. **(/ 8 3.0)** will return 2.66667

Math Functions

Truncating functions:

- **TRUNCATE** truncates toward zero. (TRUNCATE 1.9) ➔ 1; 0.9
- **FLOOR** truncates toward negative. (floor (/ 5.0 3)) ➔1; 0.6666666
- **CEILING** truncates toward positive. (CEILING (/ 5.0 3)) ➔ 2; -0.33333

incf and **decf** are used for incrementing and decrementing the value of place, respectively.

(setq n 0)
(incf n) => 1
(decf n 3) => -2

Numeric Comparisons

- The function = is the numeric equality predicate. It compares numbers by mathematical value, ignoring differences in type.
- **(= 1 1)** ==> T
- **(= 10 20/2)** ==> T
- The /= function, conversely, returns true only if all its arguments are different values.
- **(/= 1 1)** ==> NIL
- **(/= 1 2)** ==> T
- **(/= 1 2 3)** ==> T
- **(/= 1 2 3 1)** ==> NIL
- **(/= 1 2 3 1.0)** ==> NIL

Equality Predicates

- Common Lisp provides predicates for testing for equality of two objects: **eq** (the most specific), **eql, equal**, and **equalp** (the most general).
- (eq x y) is true if and only if **x and y are the same identical object**. (Implementationally, x and y are usually eq **if and only if they address the same identical memory location**.)

(setq a '(1 2 3))

(setq b '(1 2 3))

- **(eq 'a 'b)** is false.
- **(eq 'a 'a)** is true.
- **(eq 3 3)** might be true or false, depending on the implementation.
- **(eq 3 3.0)** is false.

- The eql predicate is true if its arguments are eq, or if they are numbers of the **same type with the same value**, or if they are character objects that represent the same character.
  - **(eql 'a 'b)** is false.
  - **(eql 'a 'a)** is true.
  - **(eql 3 3)** is true.
  - **(eql 3 3.0)** is false.
  - **(eql 3.0 3.0)** is true.
  - **(eql "Fun" "Fun")** might be true or false.
- The equal predicate is true if its arguments are **structurally similar** (isomorphic) objects. i.e. their printed representations are the same.
  - **(equal 'a 'b)** is false.
  - **(equal 'a 'a)** is true.
  - **(equal 3 3)** is true.
  - **(equal 3 3.0)** is false.
  - **(equal 3.0 3.0)** is true.
  - **(equal "Fun" "Fun")** is true.

- Two hash tables are considered the same by equalp.
  - **(equalp 'a 'b)** is false.
  - **(equalp 'a 'a)** is true.
  - **(equalp 3 3)** is true.
  - **(equalp 3 3.0)** is true.
  - **(equalp 3.0 3.0)** is true.
  - **(equalp "Fun" "Fun")** is true.
  - **(equalp "FUN" "fun")** is true.

In-Built Functions

1. **subseq**: takes a string followed by one *or* two numbers. If two numbers *i* and *j* are provided, then **subseq** returns the substring starting a position *i* in the string and ending at position *j-1*.
   (subseq "Four score and seven years ago" 9 16) ==> e and s
2. **Print: (print expression-to-print). (print (+ 2 3 4 1))** will print 10,10. If print is used to evaluate an expression, the final result will be printed twice: **print** prints its argument, then returns it, and Lisp always prints [again] the final return value of the expression.
3. **Boolean Operators**:
   1. **(= 4 3)** -> [4 = = 3] ➔ nil
   2. **(< 3 9)** -> [3<9] ➔T
   3. **(numberp 9)** -> [is 9 a number?] ➔T
   4. **(oddp 9)** -> [is 9 an odd number?] ➔T

Control Structures

- **setq is used to set variables.**

  (setq var 32)
  (setq str "Texas")
  (setq lst '(1 2 3))

  The three types of data here are numbers, strings, and lists. Notice that **Lisp,** unlike C, C++, Java, **is dynamically typed.**

- **setf:** it just evaluates *expression*, and stores its value in the variable associated with *variable-symbol*. Then it returns the value of *expression*.

  (setf x (* 3 2)) -> sets x to “6”

- **if** : General Form

  (if test-expression
  then-expression
  optional-else-expression)

E.g. **(if (<= 3 2) (* 3 9) (+ 4 2 3))** will return 9. **if** only allows one test-expression, one then-expression, and one optional-else-expression.

- **progn:** It is used to make a **block** (a group of expressions executed one-by-one) in an **if else** statement. Eg. **(if (> 3 2) (progn (print "hello") (print "yo") (print "whassup?") 9) (+ 4 2 3))** will print "hello“ "yo" "whassup?" 9.

let: let declares local variables with each declaration.

- Syntax

(let (dec1 declaration2 ...)
    expr1
    expr2
    ...)

- **Example**

(let ((x 3)) ;[x declared local]
    (print x)
    (setf x 9) ;[the local x is set]
    (print x)
    (print "hello"))

output: 3
9
"hello"
"hello"

Example :

- **(let ((x 3))**
- **(print x)**
- **(let (x)**
- **(print x)**
- **(let ((x "hello")) (print x))**
- **(print x))**
- **(print x)**
- **(print "yo"))**

- Outputs :

  3

  NIL

  "hello"

  NIL

  3

  "yo"

  "yo"

- **dotimes**: It first evaluates the expression *high-val*, which should return a positive integer. Then it sets the variable *var.* to 0. Then it evaluates the zero or more expressions one by one. Then it increments *var* by 1 and does this until *var* reaches *high-val*. At this time, optional-return-val is evaluated and returned, or nil is returned if *optional-return-val* is missing.

```
(dotimes (var high-val
  optional-return-val)
  expr1
  expr2
  ...)
```

```
(dotimes (x 3 "you") (print "hello"))
        "hello"
        "hello"
        "hello"
        "you"
```

Arrays and Vectors

make-array: This form makes a one-dimensional fixed-length array length elements long. The elements are each initialized to nil.

(make-array 4)

#(NIL NIL NIL NIL)

A multidimensional array is created as follows:

(make-array dimension-list).

(make-array '(2 8))

#2A((NIL NIL NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL NIL NIL))

- Key word : initial-element
- (make-array '(dim1 dim2) :initial-element i)
- The general function for extracting the element of any array is aref.
- It takes the form:
- (aref array index1 index2 ...)
- **(setf x (make-array '(3 3) :initial-contents '((0 1 2) (3 4 5) (6 7 8))))(write x)**

```
(write (setf my-array (make-array '(10))))
(terpri)
(setf (aref my-array 0) 25)
(setf (aref my-array 1) 23)
(setf (aref my-array 2) 45)
(setf (aref my-array 3) 10)
(setf (aref my-array 4) 20)
(setf (aref my-array 5) 17)
(setf (aref my-array 6) 25)
(setf (aref my-array 7) 19)
(setf (aref my-array 8) 67)
(setf (aref my-array 9) 30)
(write my-array)
```

```
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
#(25 23 45 10 20 17 25 19 67 30)
```

More operations on list:

- ‘Car’ => first
- ‘Cdr’ => rest

- (car (list)) returns first element of list.
- (cdr (list)) returns except first all rest elements of list
- caadr is the same as

  (car (car (cdr ...)))

- **(car ‘(a b c))** outputs: A
- **(cdr ‘(a b c))** outputs: (B C)
- **(caar '((a b c) ((d)) e f g h))**

  outputs : A
- **(cdar '((a b c) ((d)) e f g h))**

  outputs : (B C)

Functional Programming

- One of Lisp's most powerful features is the ability to pass functions to other functions. Most of these functions take two arguments, a function and a list.
- mapcar (map): Returns the list the results from applying the function to each of the items in the list.

  (mapcar #'car '((1 a) (2 b) (3 c))) => (1 2 3)

  (mapcar (function car) '((1 a) (2 b) (3 c))) => (1 2 3)
- remove-if (filter): Removes items from the list if the item, when plugged into the function, returns true.

  (remove-if #'oddp '(1 2 3 4 5)) => '(2 4)
- reduce: Reduces a list to a single value by applying the function to each of the items.

  (reduce #'+ '(1 2 3 4 5)) => 15

Writing Functions

defun: defun builds a function of zero or more arguments of the local-variable names given by the parameter symbols, then evaluates the expressions one by one, then returns the value of the last expression

- Syntax

(defun function-name-symbol
    (param1 param2 param3 ...)
  expr1
  expr2
  expr3
  ...)

- Example
- **(defun add-four (x)**
  (+ x 4))

-> (add-four 7)

11

Question: Write a tail recursive function for factorial of n?

Answer

```
(defun factorial (n)
 "Compute the factorial of N."
        (if (= n 1)
          1
          (* n (factorial (- n 1)))))
```

- The definition of factorials:

$n! = 1$	if n = 1
$n! = n * (n - 1)!$	if n > 1

```
(defun sum (x y)
"Sum any two numbers after printing a message."
(format t "Summing ~d and ~d." x y)
(+ x y))
```

➔**Summing 3 and 4.**
7

- Optional Parameters: **(defun fun (a b &optional c d) (list a b c d))**
  (fun 1 2) ==> (1 2 NIL NIL)
  (fun 1 2 3) ==> (1 2 3 NIL)
  (fun 1 2 3 4) ==> (1 2 3 4)
- Function Return Values:

```
(defun fn (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from fn (list i j))))))
(fn 5)        ==> (1 6)
```