Common LISP Tutorial 2 (Advance)

CLISP Download

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

IPPL Course Materials (UST sir only) Download

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

Lambda Expressions:

- lambda is the symbol for an anonymous function, a function without a name.

(lambda (arg-variables...)

[interactive-declaration]

body-forms...)

((lambda (x y) (* x y)) 2 3) ➔ 6

(mapcar (lambda (x) (* x x)) '(2 3 1 6)) ➔ ?

Hash Table:

- The hash table data structure represents a collection of **key-and-value** pairs that are organized based on the hash code of the key. It uses the key to access the elements in the collection.
- The make-hash-table function is used for creating a hash table. Syntax for this function is:
- **make-hash-table &key :test :size :rehash-size :rehash-threshold**

key argument provides the key.

:test argument determines how keys are compared - it should have one of three values #'eq, #'eql, or #'equal. If not specified, eql is assumed.

:size argument sets the initial size of the hash table.

:rehash-size argument specifies how much to increase the size of the hash table when it becomes full.

:rehash-threshold argument specifies how full the hash table can get before it must grow.

The make-hash-table function can be called with no arguments.

- The gethash function retrieves an item from the hash table by searching for its key. If it does not find the key, then it returns nil.

gethash key hash-table &optional default

- The remhash function removes any entry for a specific key in hash-table.

remhash key hash-table

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(lisp prg.))
(setf (gethash '002 empList) '(java prg.))
(setf (gethash '003 empList) '(prolog prg.))

(print (gethash '001 empList))
(print (gethash '002 empList))
(print (gethash '003 empList))
 (remhash '003 empList)
(print (gethash '003 empList))
(terpri)
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) empList)
```

Packages:

- In programming languages, a package is designed for providing a way to keep one set of names separate from another. The symbols declared in one package will not conflict with the same symbols declared in another. This way packages reduce the naming conflicts between independent code modules.

```
(make-package :A) (make-package :B) (make-package :C)
(in-package A)
(defun hello ()    (write-line "Hello! This is A line"))
(in-package B)
(defun hello ()    (write-line "Hello! This is B line"))
(in-package C)
(defun hello ()    (write-line "Hello! This is C line"))
(in-package A)
(hello)
(in-package B)
(hello)
(in-package C)
(hello)
(delete-package C)
(in-package C)
(hello)
```

File Write:

- The **open function** is used to create a new file or to open an existing file.

(open “c:/clisp/myfile.txt”)

- *The **with-open-file** allows reading or writing into a file, using the stream variable associated with the read/write transaction. Once the job is done, it automatically closes the file. It is extremely convenient to use.*

```
(with-open-file (stream "c:/clisp/myfile.txt" :direction :output)
    (format stream "Welcome to PPL class!")
    (terpri stream)
   (format stream "This is a demo file.")
)
```

File Read:

```
(let ((in (open "c:/clisp/myfile.txt" :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
          while line do (format t "~a~%" line))
    (close in)
  )
)
```

LISP – CLOS (Common Lisp Object System)

- In CLOS, Object-oriented system is based on the concept of "objects", which may contain data, in the form of fields, often known as **slots**; and code, in the form of procedures, often known as **functions**.
- **Class:** A class is a "template" that describes the structure and behavior of its *instances*. Every kind of Lisp data is an instance of some class. There are built-in classes, such as the class of integers, or the class of strings. You can use the function class-of to determine the class of some Lisp object.

(class-of 5) => #<BUILT-IN-CLASS INTEGER>

(class-of “IIITA") => #<BUILT-IN-CLASS STRING>

Defining Classes:

(defclass name (superclasses)

(slots)

options)

The options are:

:initarg - a keyword what would be used to supply slot values when an instance of a class is created.

:initform - if no value for a slot was supplied, it would be initialized with the result of evaluating initform. If there is no initform, an error would be signaled.

:reader - specifies a function to read a particular slot.

:writer - specifies a function to write to a particular slot.

:accessor - specifies a function to read and write a value of a slot.

Example:

```
(defclass book ()
   ((author :initform "" :accessor author)
    (year :initform 0 :accessor year))
   (:documentation "Describes a book"))
(setf *my-book1* (make-instance 'book))
(print (class-of *my-book1*))
(print (author *my-book1*))
(print (year *my-book1*))
(setf (slot-value *my-book1* 'year) 1995)
(print (year *my-book1*))
```

```
(defclass book2 ()
   ((author :initarg :author :accessor author)
    (year :initarg :year :accessor year))
   (:documentation "Describes a book"))
(setf *my-book2*(make-instance 'book2
                :author "ABC"
                :year 1995))
(print (author *my-book2*))
(setf (slot-value *my-book2* 'author) "AAA")
(print (author *my-book2*))
```

Class function:

```
(defclass box ()
  ((length :accessor box-length)
    (breadth :accessor box-breadth)
    (height :accessor box-height)
    (volume :reader volume)
  )
)
; method calculating volume
(defmethod volume ((object box))
  (* (box-length object) (box-breadth object)(box-height object))
)
```

```
;setting the values
(setf item (make-instance 'box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
; displaying values
(format t "Length of the Box is ~d~%" (box-length item))
(format t "Breadth of the Box is ~d~%" (box-breadth item))
(format t "Height of the Box is ~d~%" (box-height item))
(format t "Volume of the Box is ~d~%" (volume item))
```

Inheritance:

```
(defclass box ()
  ((length :accessor box-length)
    (breadth :accessor box-breadth)
    (height :accessor box-height)
    (volume :reader volume)
  )
)
; method calculating volume
(defmethod volume ((object box))
  (* (box-length object) (box-breadth object)(box-height object))
)
;wooden-box class inherits the box class
(defclass wooden-box (box)
((price :accessor box-price)))
```

```
;setting the values
(setf item (make-instance 'wooden-box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
(setf (box-price item) 1000)
; displaying values
(format t "Length of the Wooden Box is ~d~%" (box-length item))
(format t "Breadth of the Wooden Box is ~d~%" (box-breadth item))
(format t "Height of the Wooden Box is ~d~%" (box-height item))
(format t "Volume of the Wooden Box is ~d~%" (volume item))
(format t "Price of the Wooden Box is ~d~%" (box-price item))
```

ELIZA:

- ELIZA is an natural language processing computer program created from 1964 to 1966 at the MIT Artificial Intelligence Laboratory by Joseph Weizenbaum. Eliza simulated conversation by using a '**pattern matching**' and substitution methodology that gave users an illusion of understanding. ELIZA was one of the first chatterbots, but was also regarded as one of the first programs capable of passing the **Turing Test**.