KATHMANDU UNIVERSITY

Dhulikhel, Kavre

A Lab Report

On

"LISP: Rapid Language Learning"

[Course title: COMP 301]

Submitted by:

Bisheshwor Neupane(35)

Submitted to:

Mr. Nabin Ghimire

Department of Computer Science and Engineering

Submission Date:

August 14, 2021

Chapter 1: Introduction

LISP, in full list processing, a computer programming language developed about 1960 by John McCarthy at the Massachusetts Institute of Technology (MIT). LISP was founded on the mathematical theory of recursive functions (in which a function appears in its own definition). A LISP program is a function applied to data, rather than being a sequence of procedural steps as in FORTRAN and ALGOL.

Feature of Common LISP

- It is machine-independent
- It uses iterative design methodology, and easy extensibility.
- It allows updating the programs dynamically.
- It provides high level debugging.
- It provides advanced object-oriented programming.
- It provides a 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 an object-oriented condition system.
- It provides a complete I/O library.
- It provides extensive control structures.

Applications built in LISP

- Emacs
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks:

- Atom
- List
- String

An atom is a number or string of contiguous characters. It includes numbers and special characters.

A list is a sequence of atoms and/or other lists enclosed in parentheses.

A string is a group of characters enclosed in double quotation marks.

Data Types
A data type is a set of LISP objects and many objects may belong to one such set.
LISP data types can be categorized as:

Scalar types:

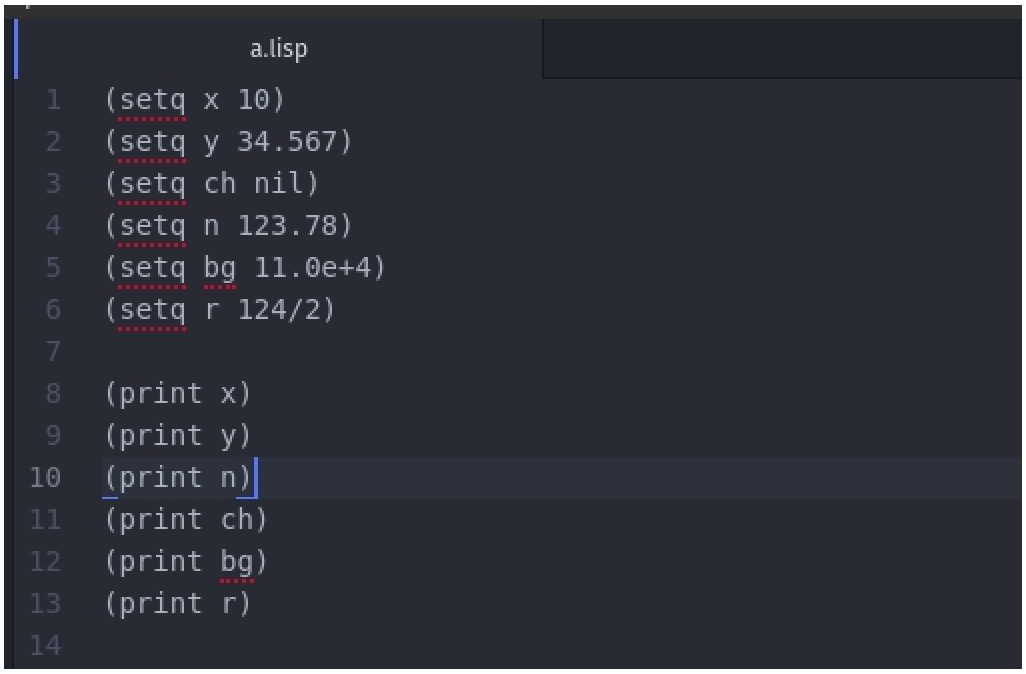
Number types, characters, symbols stc.

Data structures

Lists, vectors, bit-vectors, and strings

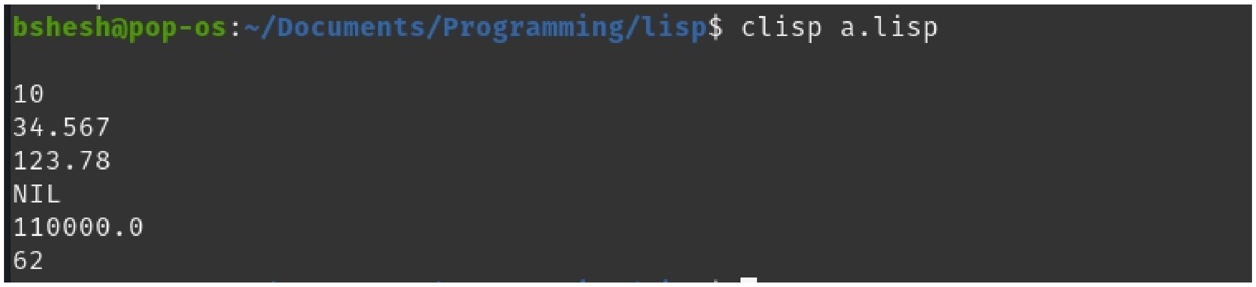
Although, it is not necessary to specify a data type for a LIST variable, it helps in certain loop expansions, in method declarations and some other situations.The data types are arranged into a hierarchy.
The typep predicated is used for finding whether an object belongs to a specific type.
The type-of function returns the data type of a given object.

Example:



```
(setq x 10)
(setq y 34.567)
(setq ch nil)
(setq n 123.78)
(setq bg 11.0e+4)
(setq r 124/2)

(print x)
(print y)
(print n)
(print ch)
(print bg)
(print r)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp a.lisp

10
34.567
123.78
NIL
110000.0
62
```

1.1 LISP Macros

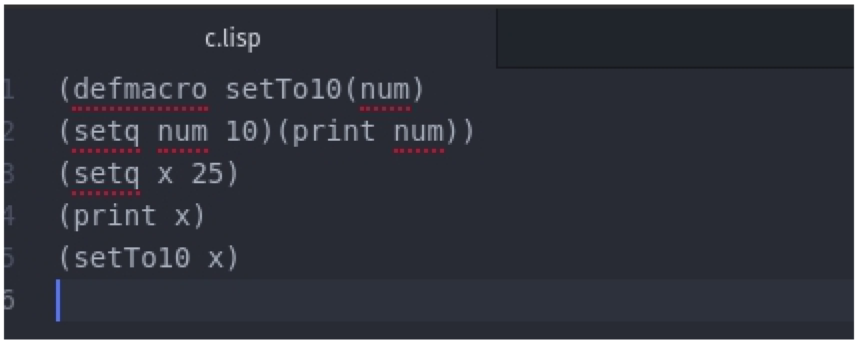
Macros allow you to extend the syntax of standard LISP.

In LISP, a named macro is defined using another macro named defmacro**.** Syntax for defining a macro is –

(defmacro macro-name (parameter-list))
"Optional documentation string."
body-form

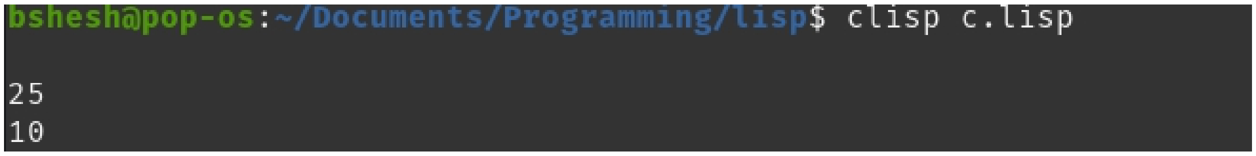
The macro definition consists of the name of the macro, a parameter list, an optional documentation string, and a body of Lisp expressions that defines the job to be performed by the macro.

For example:



```
(defmacro setTo10(num)
(setq num 10)(print num))
(setq x 25)
(print x)
(setTo10 x)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp c.lisp

25
10
```

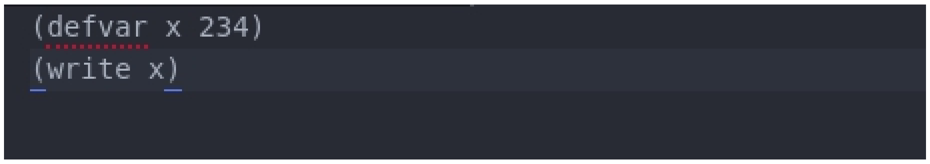
1.2 LISP-Variables

In LISP, each variable is represented by a symbol. The variable's name is the name of the symbol and it is stored in the storage cell of the symbol.

Global Variables

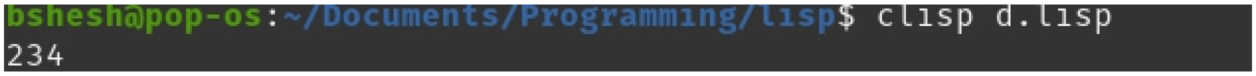
Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified. Global variables are generally declared using the defvar construct.

Example:



```
(defvar x 234)
(write x)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp d.lisp
234
```

Local Variables

Local variables are defined within a given procedure. The parameters named as arguments within a function definition are also local variables. Local variables are accessible only within the respective function.

Like the global variables, local variables can also be created using the setq construct.

There are two other constructs - let and prog for creating local variables.

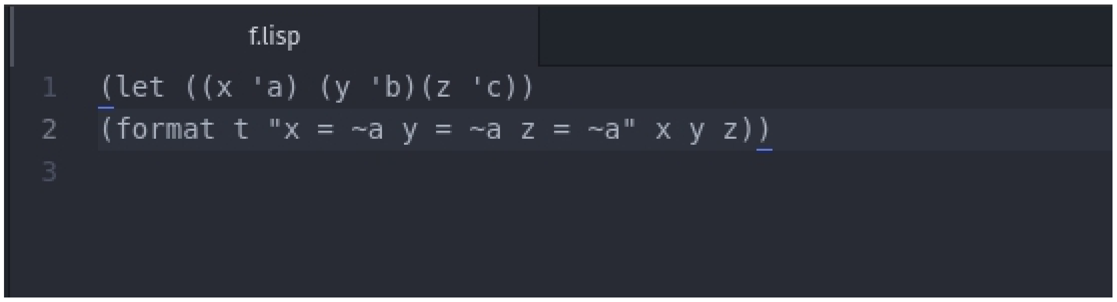
The let construct has the following syntax.

(let ((var1 val1) (var2 val2).. (varn valn))<s-expressions>)

Where var1, var2, ..varn are variable names and val1, val2, .. valn are the initial values assigned to the respective variables.

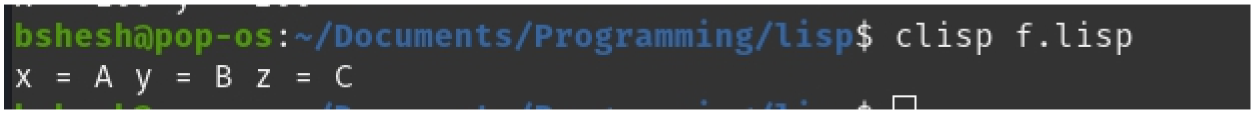
When let is executed, each variable is assigned the respective value and lastly the *s-expression* is evaluated. The value of the last expression evaluated is returned.

Example:



```
(let ((x 'a) (y 'b)(z 'c))
(format t "x = ~a y = ~a z = ~a" x y z))
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp f.lisp
x = A y = B z = C
```

1.3 LISP-Constants

In LISP, constants are variables that never change their values during program execution. Constants are declared using the defconstant construct.

The defun construct is used for defining a function.

```
(defconstant PI 3.141592)

(defun area-circle(rad)

   (terpri)

   (format t "Radius: ~5f" rad)

   (format t "~%Area: ~10f" (* PI rad rad)))

(area-circle 10)
```

Output:

```
Radius:  10.0

Area:   314.1592
```

1.4 LISP-Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. LISP allows numerous operations on data, supported by various functions, macros and other constructs.

The operations allowed on data could be categorized as-

- Arithmetic Operations
- Comparison Operations
- Logical Operations
- Bitwise Operations

Arithmetic Operations

The Airithmetic Operators used in LISP are:
+ , - , *, /, mod, rem, incf , decf

Comparison Operations

Following table shows all the relational operators supported by LISP that compares between numbers. However unlike relational operators in other languages, LISP comparison operators may take more than two operands and they work on numbers only.
The operators are: = , /= , > , < , >= , <= , max , min.

Logical Operations on Boolean Values

Common LISP provides three logical operators: and, or, and not that operates on Boolean values.
Operators: and , or , not

Bitwise Operations on Numbers

Bitwise operators work on bits and perform bit-by-bit operation. The truth tables for bitwise and, or, and xor operations are as follows −

p	q	p and q	p or q	p xor q
0	0	0	0	0
0	1	0	1	1

1	1	1	1	0
1	0	0	1	1

The Bitwise operators supported by LISP are listed below.
logand , logior , logxor , logvor , logeqv .

1.5 LISP - Decision Making
Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statements to be executed if the condition is determined to be true, and optionally other statements to be executed if the condition is determined to be false.

LISP provides following types of decision making constructs. Click the following links to check their detail.

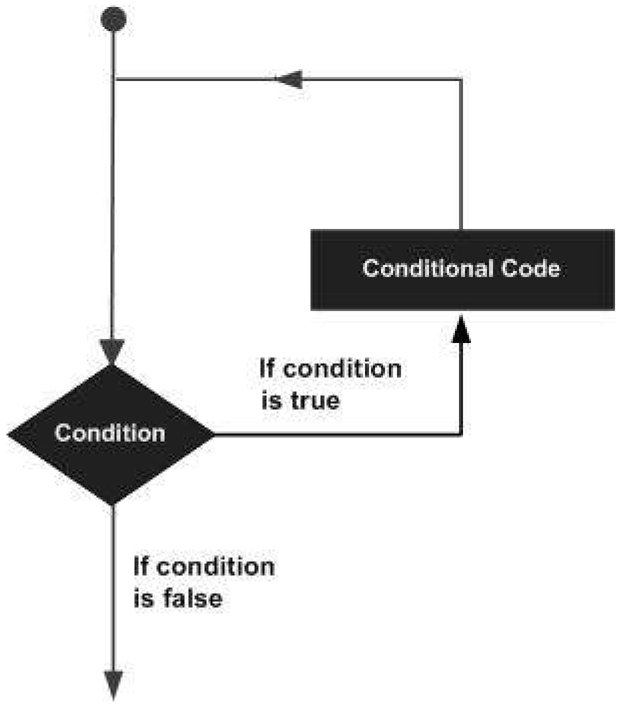
Sr.No.	Construct & Description
1	**cond** This construct is used for used for checking multiple test-action clauses. It can be compared to the nested if statements in other programming languages.
2	**if** The if construct has various forms. In simplest form it is followed by a test clause, a test action and some other consequent action(s). If the test clause evaluates to true, then the test action is executed otherwise, the consequent clause is evaluated.
3	**when** In simplest form it is followed by a test clause, and a test action. If the test clause evaluates to true, then the test action is executed otherwise, the consequent clause is evaluated.

4 **case**

This construct implements multiple test-action clauses like the cond construct. However, it evaluates a key form and allows multiple action clauses based on the evaluation of that key form.

1.6 LISP-Loops

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages.



LISP provides the following types of constructs to handle looping requirements.

- The loop construct is the simplest form of iteration provided by LISP. In its simplest form, it allows you to execute some statement(s) repeatedly until it finds a return statement.
- The loop for construct allows you to implement a for-loop like iteration as most common in other languages.
- The do construct is also used for performing iteration using LISP. It provides a structured form of iteration.
- The dotimes construct allows looping for some fixed number of iterations.
- The dolist construct allows iteration through each element of a list.

The block and return-from allows you to exit gracefully from any nested blocks in case of any error.The block function allows you to create a named block with a body composed of zero or more statements. Syntax is −

(block block-name(

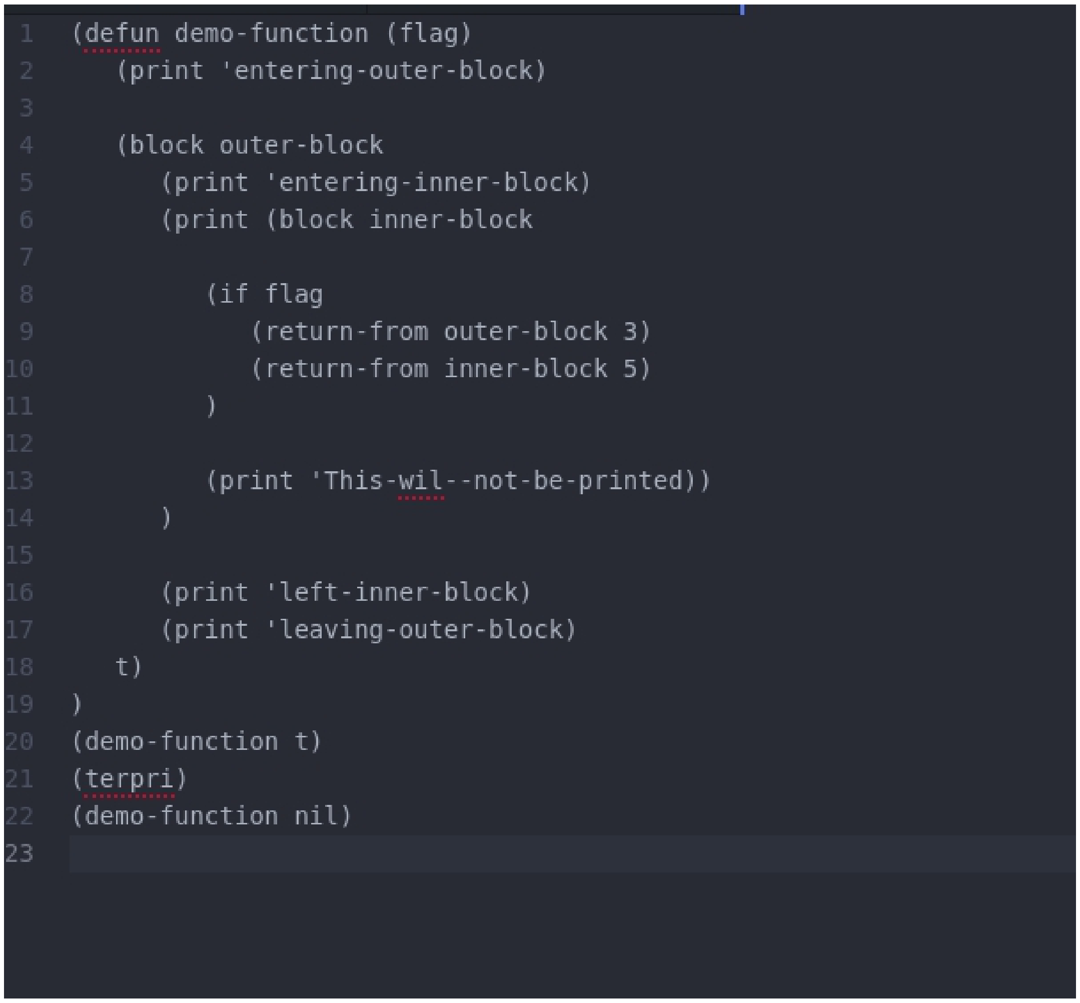
...

...

))

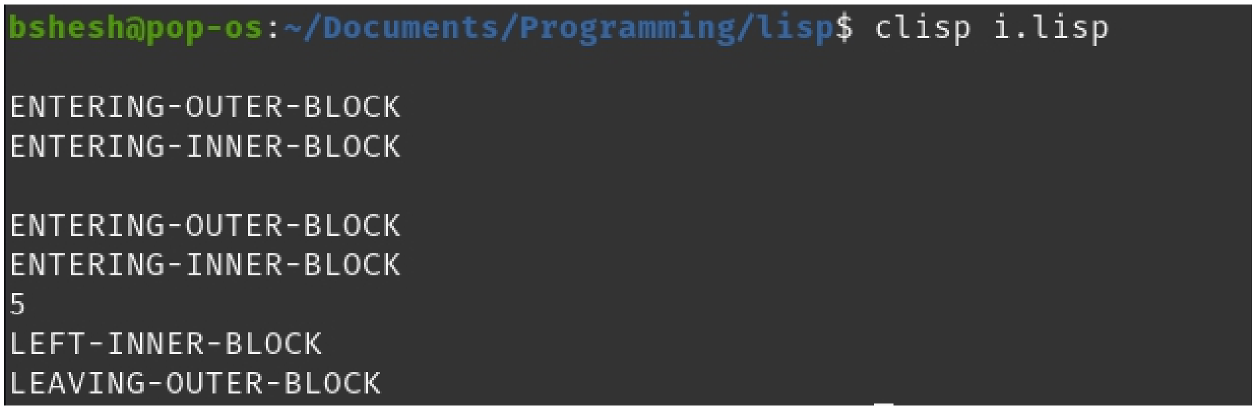
The return-from function takes a block name and an optional (the default is nil) return value.

Example:



```
(defun demo-function (flag)
   (print 'entering-outer-block)

   (block outer-block
      (print 'entering-inner-block)
      (print (block inner-block

         (if flag
            (return-from outer-block 3)
            (return-from inner-block 5)
         )

         (print 'This-wil--not-be-printed))
      )

      (print 'left-inner-block)
      (print 'leaving-outer-block)
   t)
)
(demo-function t)
(terpri)
(demo-function nil)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp i.lisp

ENTERING-OUTER-BLOCK
ENTERING-INNER-BLOCK

ENTERING-OUTER-BLOCK
ENTERING-INNER-BLOCK
5
LEFT-INNER-BLOCK
LEAVING-OUTER-BLOCK
```

1.7 LISP-Functions

A function is a group of statements that together perform a task.

Defining Functions in LISP

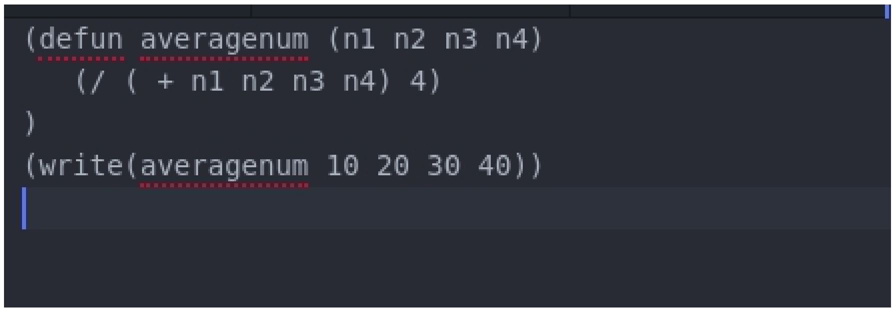
The macro named defun is used for defining functions. The defun macro needs three arguments –

- Name of the function
- Parameters of the function
- Body of the function

Syntax:

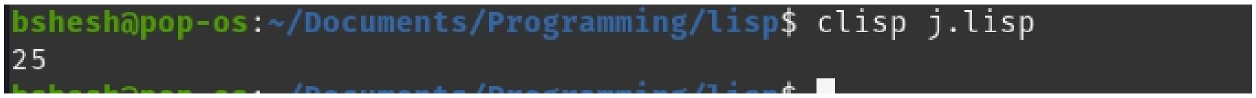
(defun name (parameter-list) "Optional documentation string." body)

Example:



```
(defun averagenum (n1 n2 n3 n4)
   (/ ( + n1 n2 n3 n4) 4)
)
(write(averagenum 10 20 30 40))
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp j.lisp
25
```

1.8 LISP-Predicates

Predicates are functions that test their arguments for some specific conditions and returns nil if the condition is false, or some non-nil value is the condition is true.

- **Atom**

  It takes one argument and returns t if the argument is an atom or nil if otherwise.
- **Equal**

  It takes two arguments and returns t if they are structurally equal or nil otherwise.
- Eq

  It takes two arguments and returns t if they are same identical objects, sharing the same memory location or nil otherwise.

- **Eql**

  It takes two arguments and returns t if the 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, or nil otherwise.

- **evenp**

  It takes one numeric argument and returns t if the argument is even number or nil if otherwise.

- **oddp**

  It takes one numeric argument and returns t if the argument is odd number or nil if otherwise.

- **Zerop**

  It takes one numeric argument and returns t if the argument is zero or nil if otherwise.

- **null**

  It takes one argument and returns t if the argument evaluates to nil, otherwise it returns nil.

- **listp**

  It takes one argument and returns t if the argument evaluates to a list otherwise it returns nil.

- **greaterp**

  It takes one or more argument and returns t if either there is a single argument or the arguments are successively larger from left to right, or nil if otherwise.

- **lessp**

  It takes one or more argument and returns t if either there is a single argument or the arguments are successively smaller from left to right, or nil if otherwise.

- **Numberp**

  It takes one argument and returns t if the argument is a number or nil if otherwise.

- **symbolp**

  It takes one argument and returns t if the argument is a symbol otherwise it returns nil.

- **integerp**

  It takes one argument and returns t if the argument is an integer otherwise it returns nil.

- **rationalp**

  It takes one argument and returns t if the argument is rational number, either a ratio or a number, otherwise it returns nil.

- **floatp**

  It takes one argument and returns t if the argument is a floating point number otherwise it returns nil.

- **Realp**

It takes one argument and returns t if the argument is a real number otherwise it returns nil.

- **complexp**

  It takes one argument and returns t if the argument is a complex number otherwise it returns nil.

- **characterp**

  It takes one argument and returns t if the argument is a character otherwise it returns nil.

- **stringp**

  It takes one argument and returns t if the argument is a string object otherwise it returns nil.

- **arrayp**

  It takes one argument and returns t if the argument is an array object otherwise it returns nil.

- **packagep**

  It takes one argument and returns t if the argument is a package otherwise it returns nil.

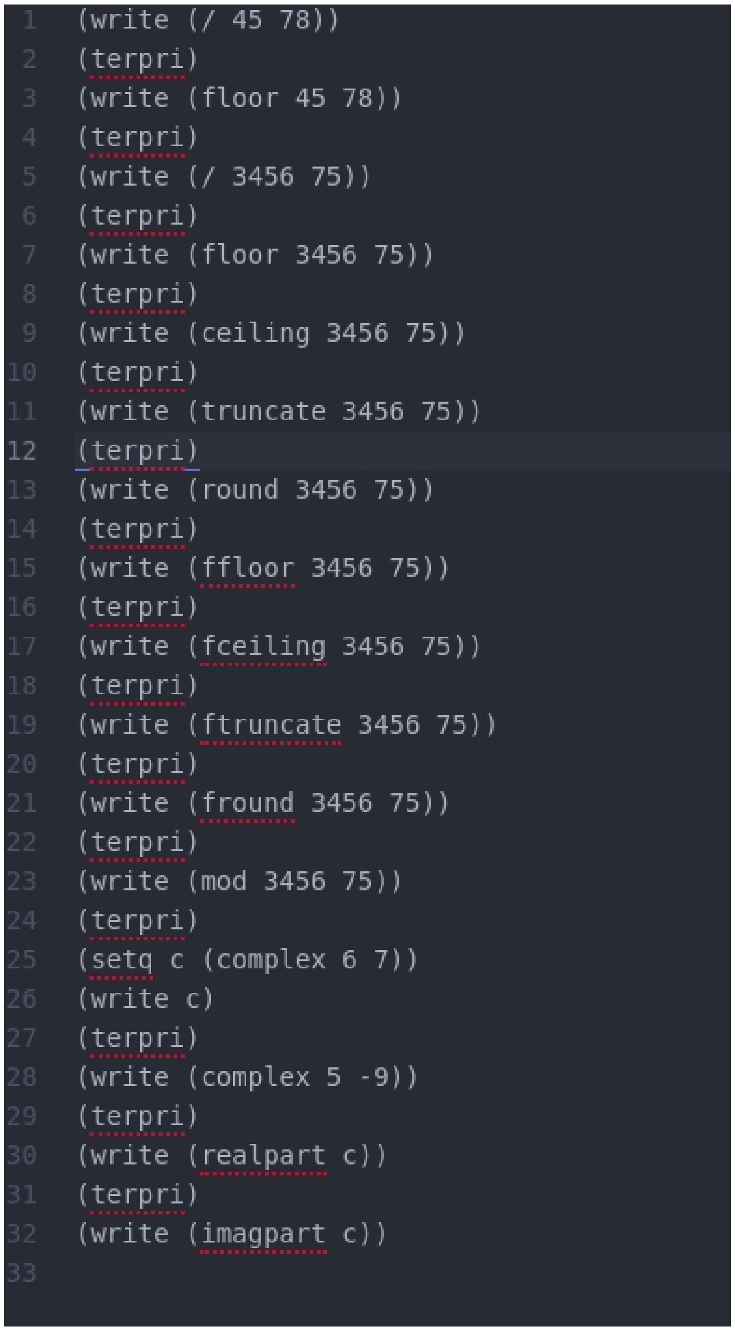
1.9 LISP-Numbers

Common Lisp defines several kinds of numbers. The number data type includes various kinds of numbers supported by LISP.

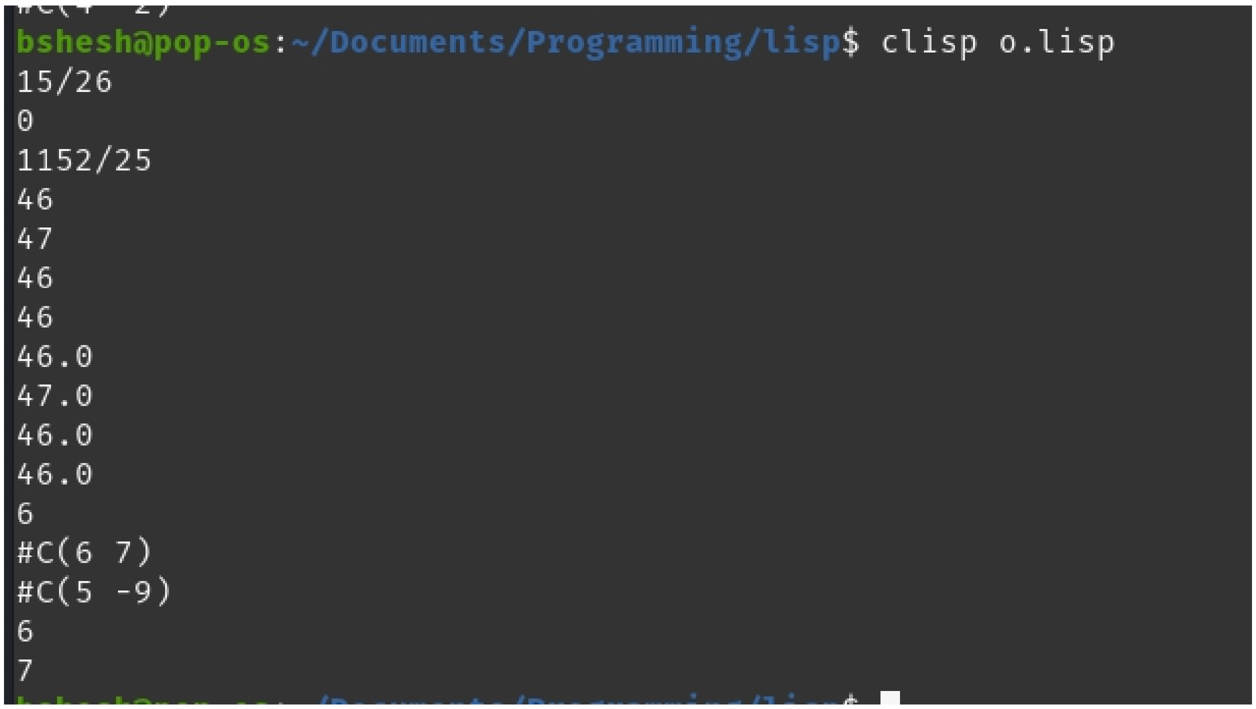
The number types supported by LISP are −

- Integers
- Ratios
- Floating-point numbers
- Complex numbers

Example:


```lisp
(write (/ 45 78))
(terpri)
(write (floor 45 78))
(terpri)
(write (/ 3456 75))
(terpri)
(write (floor 3456 75))
(terpri)
(write (ceiling 3456 75))
(terpri)
(write (truncate 3456 75))
(terpri)
(write (round 3456 75))
(terpri)
(write (ffloor 3456 75))
(terpri)
(write (fceiling 3456 75))
(terpri)
(write (ftruncate 3456 75))
(terpri)
(write (fround 3456 75))
(terpri)
(write (mod 3456 75))
(terpri)
(setq c (complex 6 7))
(write c)
(terpri)
(write (complex 5 -9))
(terpri)
(write (realpart c))
(terpri)
(write (imagpart c))
```

Output:

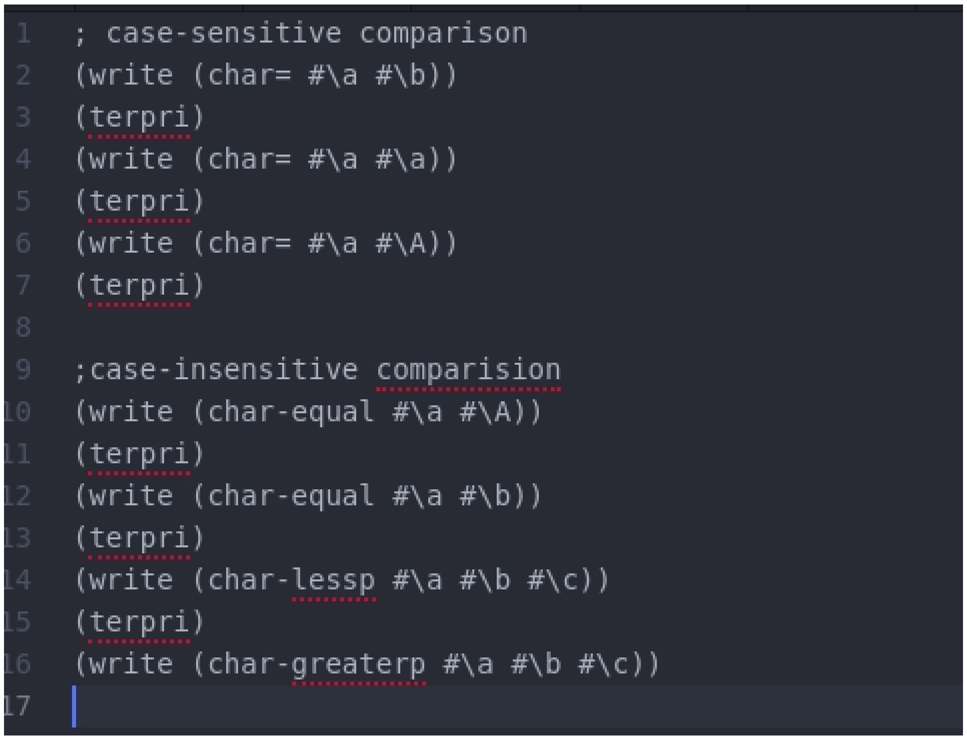


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp o.lisp
15/26
0
1152/25
46
47
46
46
46.0
47.0
46.0
46.0
6
#C(6 7)
#C(5 -9)
6
7
```

1.10 LISP-Characters:

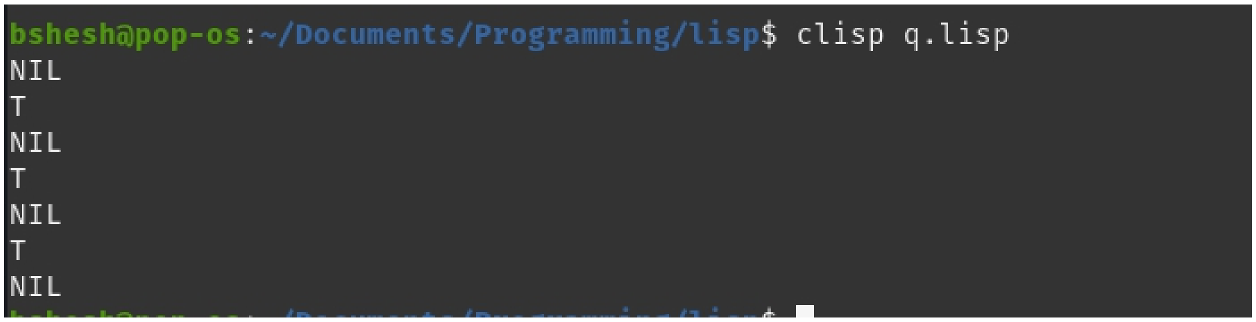
In LISP, characters are represented as data objects of type character.

Example:



```
; case-sensitive comparison
(write (char= #\a #\b))
(terpri)
(write (char= #\a #\a))
(terpri)
(write (char= #\a #\A))
(terpri)

;case-insensitive comparision
(write (char-equal #\a #\A))
(terpri)
(write (char-equal #\a #\b))
(terpri)
(write (char-lessp #\a #\b #\c))
(terpri)
(write (char-greaterp #\a #\b #\c))
```

Output:



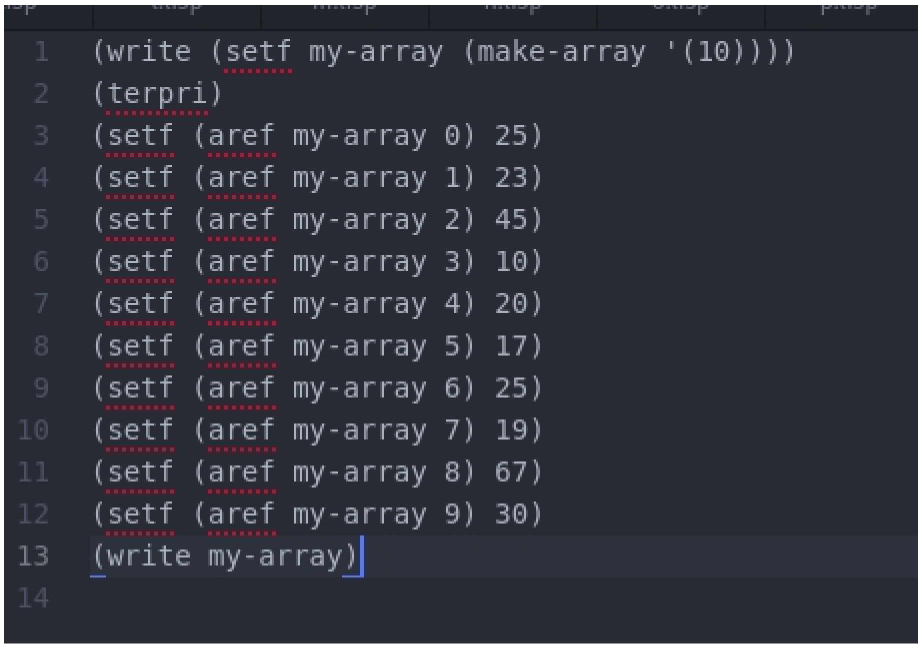
```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp q.lisp
NIL
T
NIL
T
NIL
T
NIL
```

1.11 LISP-Arrays:

LISP allows you to define single or multiple-decision arrays using the make-array function.
An array can store any LISP object as its elements.
For example, to create an array with 10- cells, named my-array, we can write −

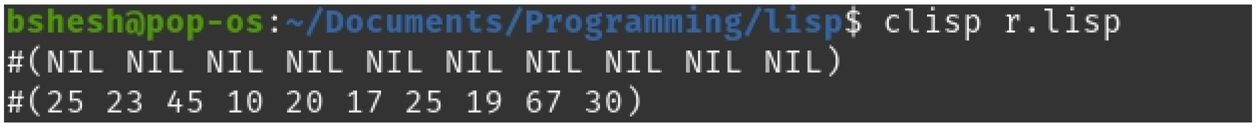
(setf my-array (make-array '(10)))

Example:



```
(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)
```

Output:

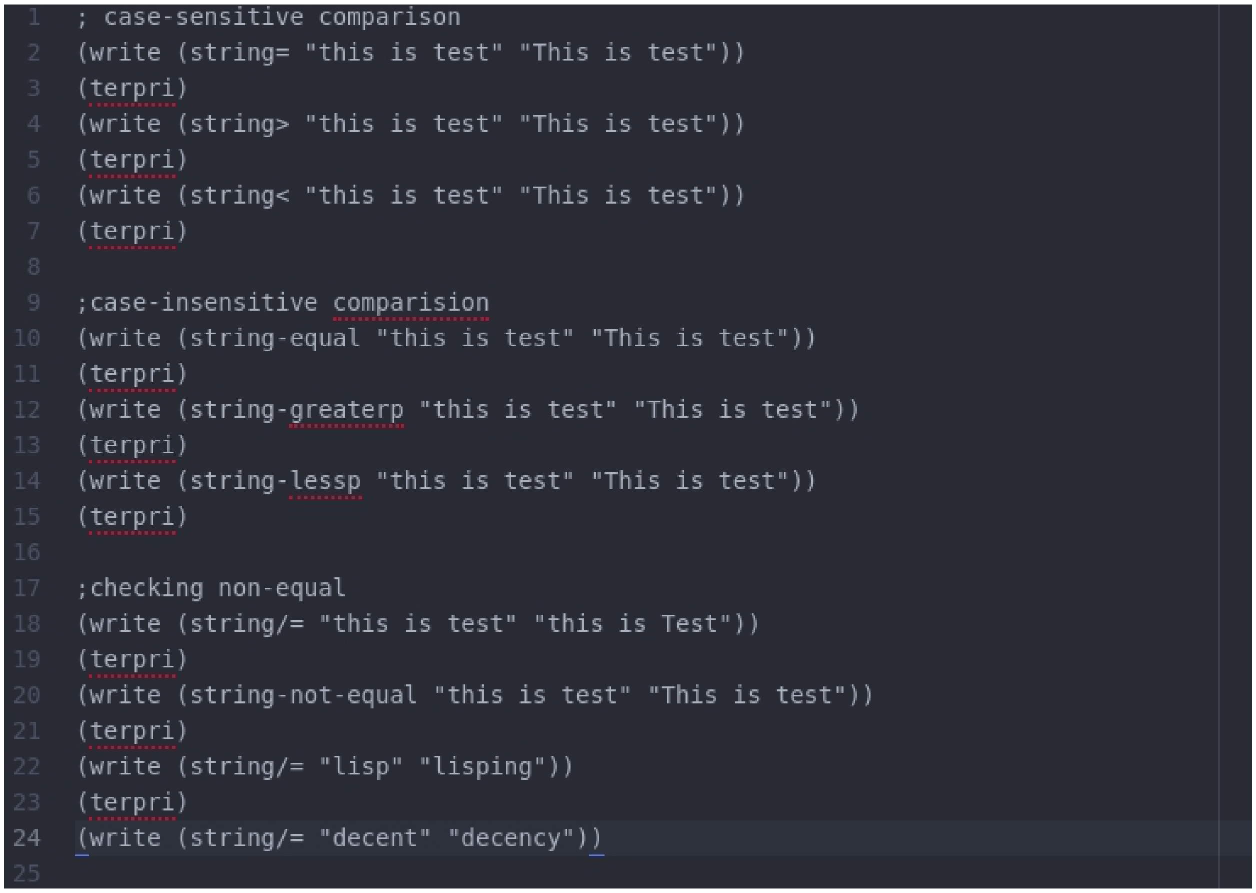


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp r.lisp
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
#(25 23 45 10 20 17 25 19 67 30)
```

1.12 LISP-Strings:

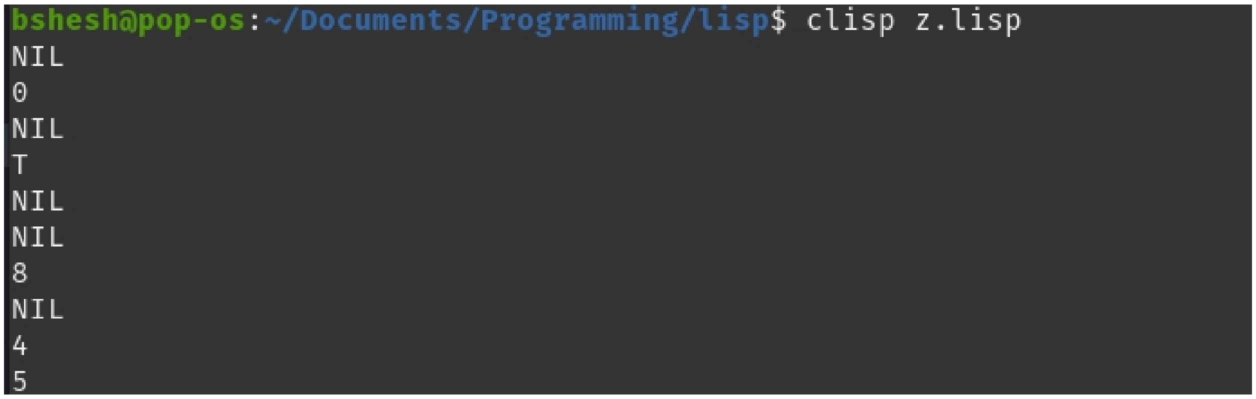
Strings in Common Lisp are vectors, i.e., one-dimensional array of characters.String literals are enclosed in double quotes. Any character supported by the character set can be enclosed within double quotes to make a string, except the double quote character (") and the escape character (\).

Example:



```
; case-sensitive comparison
(write (string= "this is test" "This is test"))
(terpri)
(write (string> "this is test" "This is test"))
(terpri)
(write (string< "this is test" "This is test"))
(terpri)

;case-insensitive comparision
(write (string-equal "this is test" "This is test"))
(terpri)
(write (string-greaterp "this is test" "This is test"))
(terpri)
(write (string-lessp "this is test" "This is test"))
(terpri)

;checking non-equal
(write (string/= "this is test" "this is Test"))
(terpri)
(write (string-not-equal "this is test" "This is test"))
(terpri)
(write (string/= "lisp" "lisping"))
(terpri)
(write (string/= "decent" "decency"))
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
NIL
0
NIL
T
NIL
NIL
8
NIL
4
5
```

1.13 LISP-Sequences:

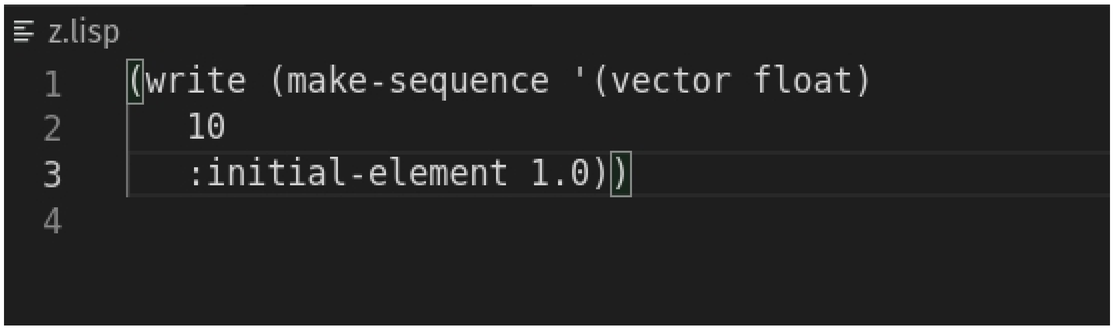
Sequence is an abstract data type in LISP. Vectors and lists are the two concrete subtypes of this data type. All the functionalities defined on sequence data type are actually applied on all vectors and list types.

The function make-sequence allows you to create a sequence of any type. The syntax for this function is −

make-sequence sqtype sqsize &key :initial-element

It creates a sequence of type *sqtype* and of length *sqsize.*

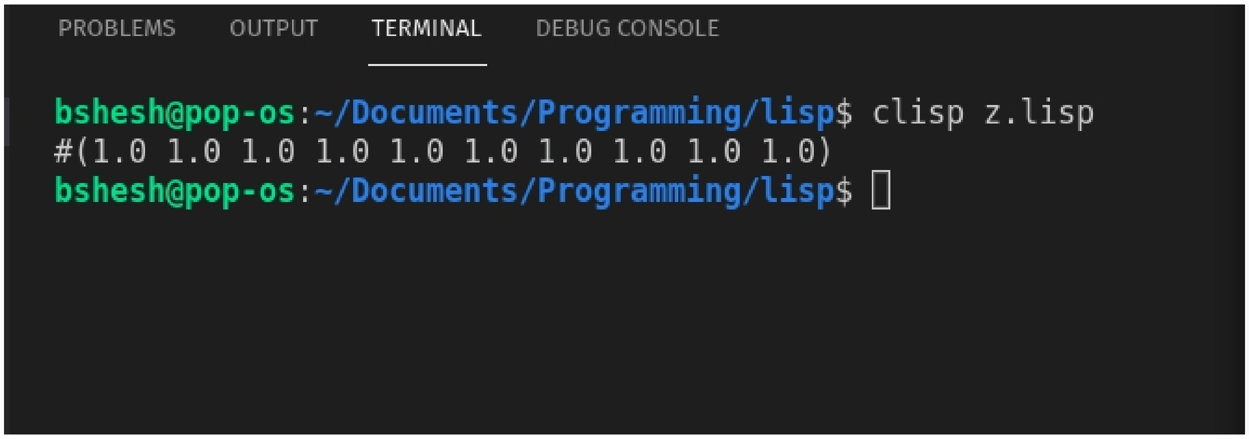
Example:



z.lisp

```
(write (make-sequence '(vector float)
   10
   :initial-element 1.0))
```

Output:

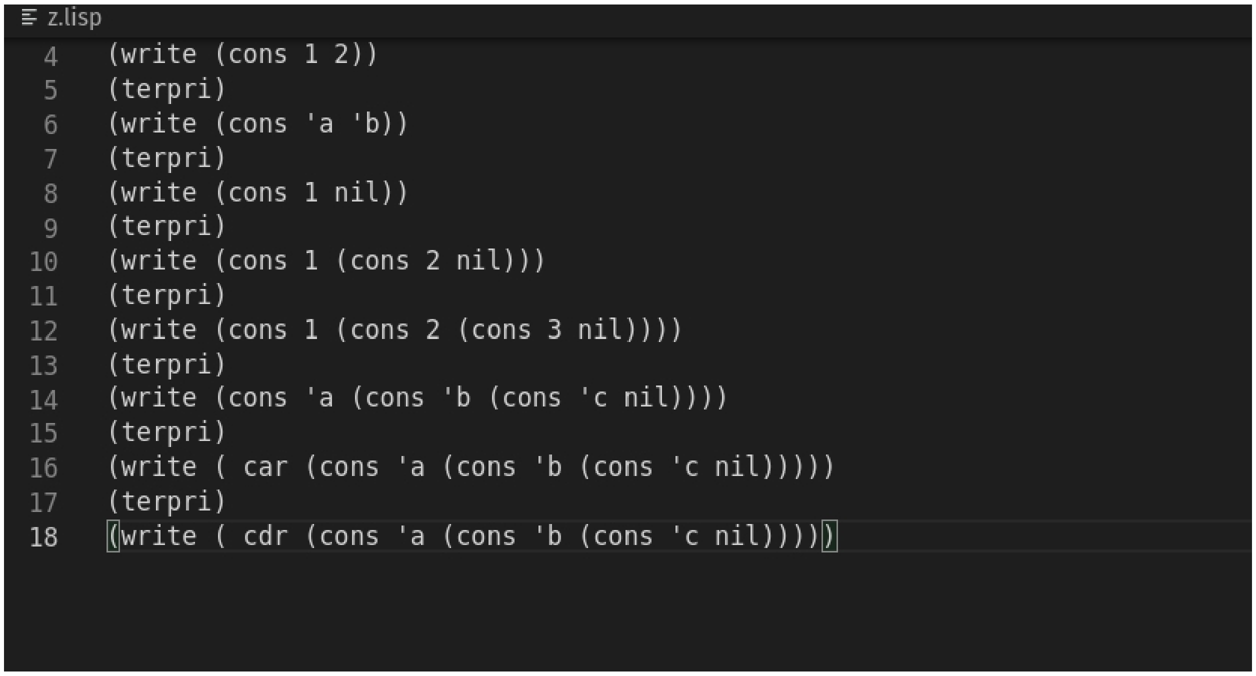


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
#(1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)
bshesh@pop-os:~/Documents/Programming/lisp$
```

1.14 LISP-Lists:

Lists had been the most important and the primary composite data structure in traditional LISP. Present day's Common LISP provides other data structures like, vector, hash table, classes or structures.

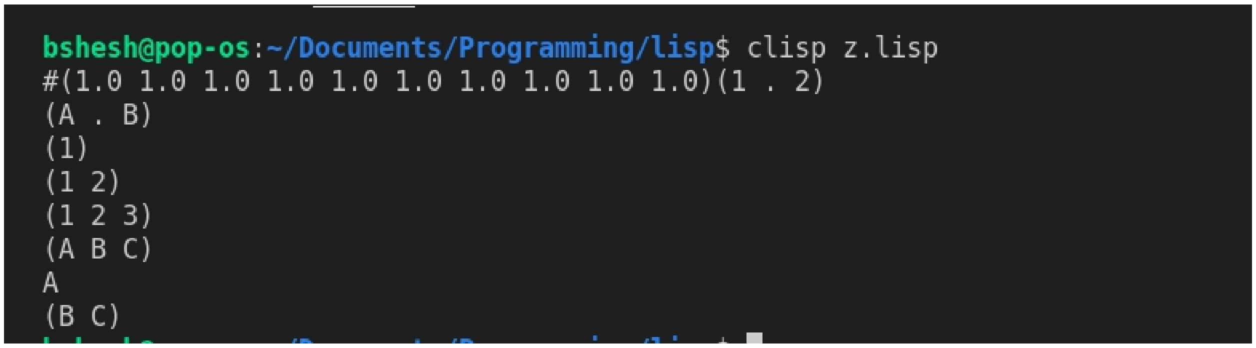
Example:



z.lisp

```
(write (cons 1 2))
(terpri)
(write (cons 'a 'b))
(terpri)
(write (cons 1 nil))
(terpri)
(write (cons 1 (cons 2 nil)))
(terpri)
(write (cons 1 (cons 2 (cons 3 nil))))
(terpri)
(write (cons 'a (cons 'b (cons 'c nil))))
(terpri)
(write ( car (cons 'a (cons 'b (cons 'c nil)))))
(terpri)
(write ( cdr (cons 'a (cons 'b (cons 'c nil)))))
```

Output:

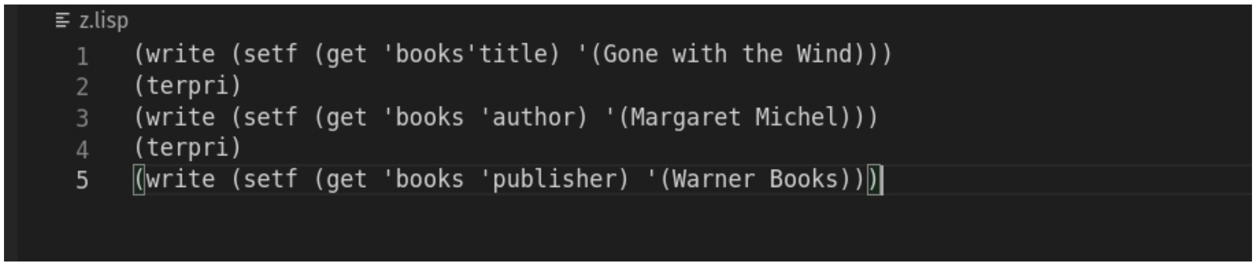


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
#(1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)(1 . 2)
(A . B)
(1)
(1 2)
(1 2 3)
(A B C)
A
(B C)
```

1.15 LISP-Symbols:

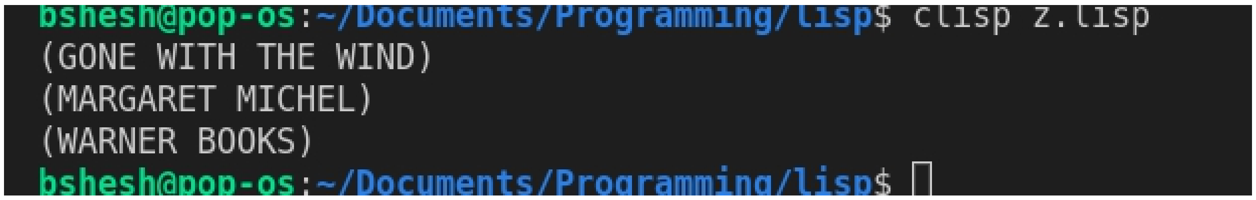
In LISP, a symbol is a name that represents data objects and interestingly it is also a data object.What makes symbols special is that they have a component called the **property list**, or **plist.**

Example:



```
≡ z.lisp
(write (setf (get 'books'title) '(Gone with the Wind)))
(terpri)
(write (setf (get 'books 'author) '(Margaret Michel)))
(terpri)
(write (setf (get 'books 'publisher) '(Warner Books)))
```

Output:

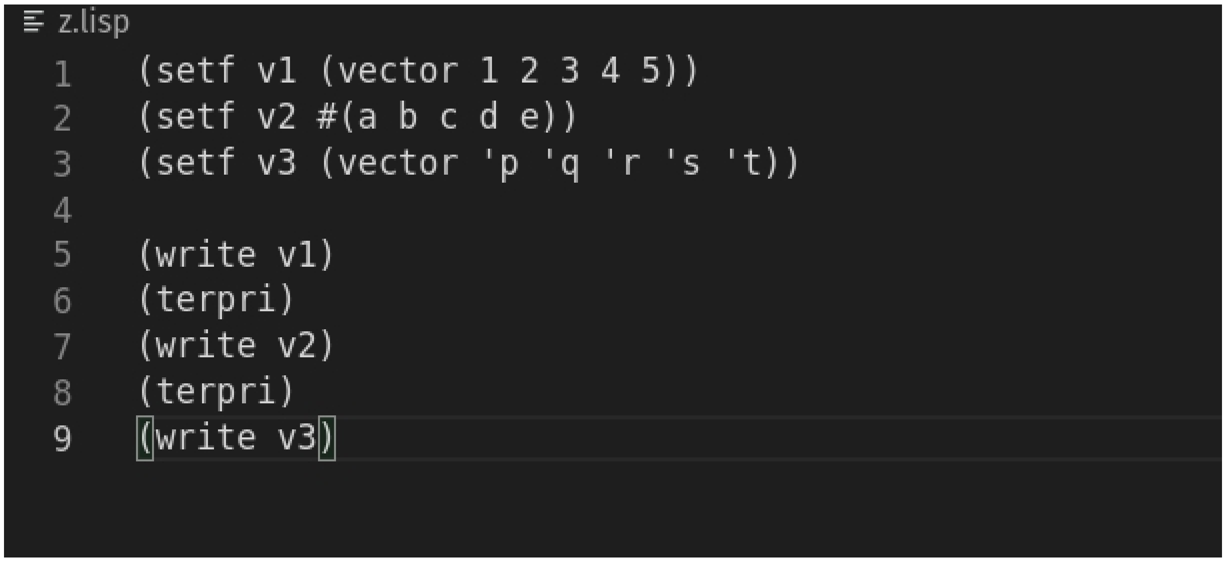


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
(GONE WITH THE WIND)
(MARGARET MICHEL)
(WARNER BOOKS)
bshesh@pop-os:~/Documents/Programming/lisp$
```

1.16 LISP-Vectors:

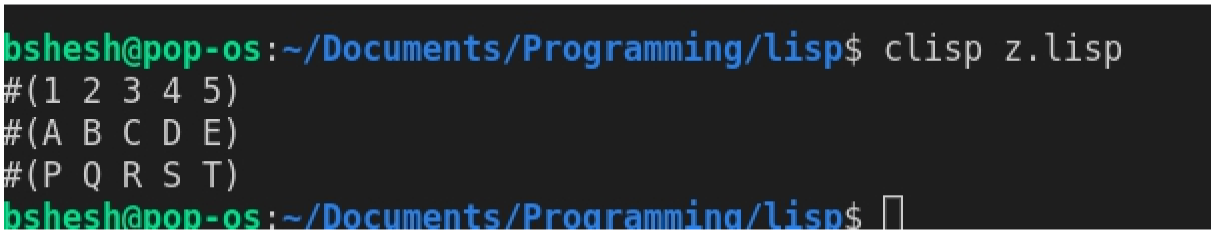
Vectors are one-dimensional arrays, therefore a subtype of array. Vectors and lists are collectively called sequences. Therefore all sequence generic functions and array functions we have discussed so far, work on vectors.

Example:



```
≡ z.lisp
(setf v1 (vector 1 2 3 4 5))
(setf v2 #(a b c d e))
(setf v3 (vector 'p 'q 'r 's 't))

(write v1)
(terpri)
(write v2)
(terpri)
(write v3)
```

Output:

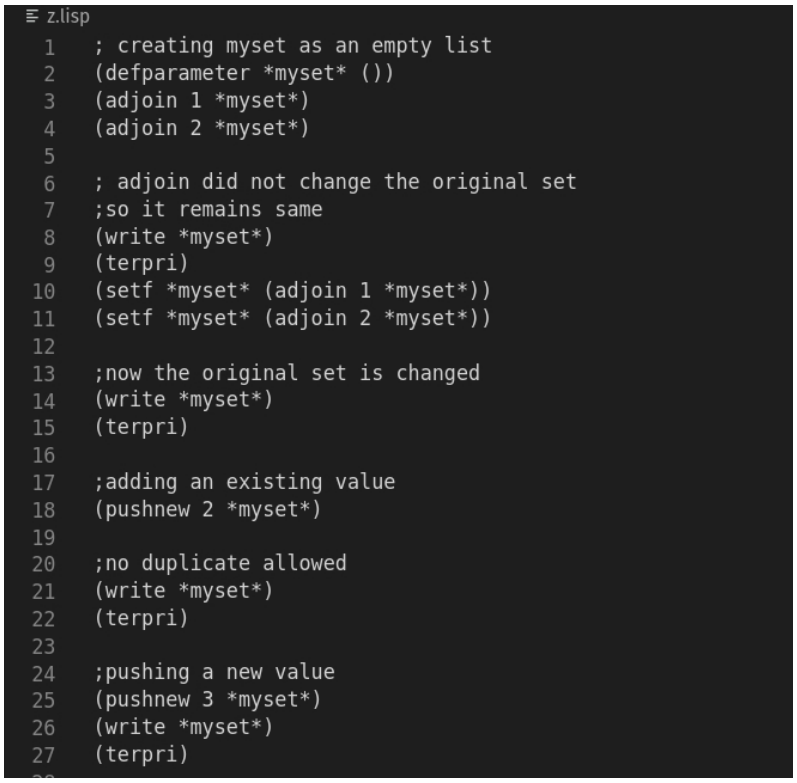


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
#(1 2 3 4 5)
#(A B C D E)
#(P Q R S T)
bshesh@pop-os:~/Documents/Programming/lisp$
```

1.17 LISP-Set:

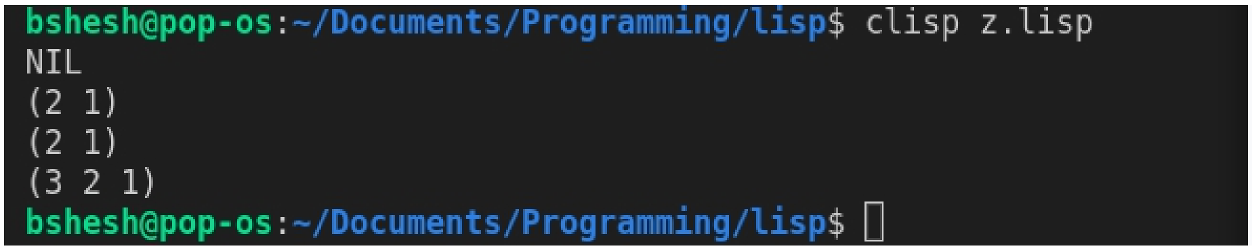
Common Lisp does not provide a set data type. However, it provides number of functions that allows set operations to be performed on a list.You can add, remove, and search for items in a list, based on various criteria. You can also perform various set operations like: union, intersection, and set difference.

Example:



```
z.lisp
; creating myset as an empty list
(defparameter *myset* ())
(adjoin 1 *myset*)
(adjoin 2 *myset*)

; adjoin did not change the original set
;so it remains same
(write *myset*)
(terpri)
(setf *myset* (adjoin 1 *myset*))
(setf *myset* (adjoin 2 *myset*))

;now the original set is changed
(write *myset*)
(terpri)

;adding an existing value
(pushnew 2 *myset*)

;no duplicate allowed
(write *myset*)
(terpri)

;pushing a new value
(pushnew 3 *myset*)
(write *myset*)
(terpri)
```

Output:

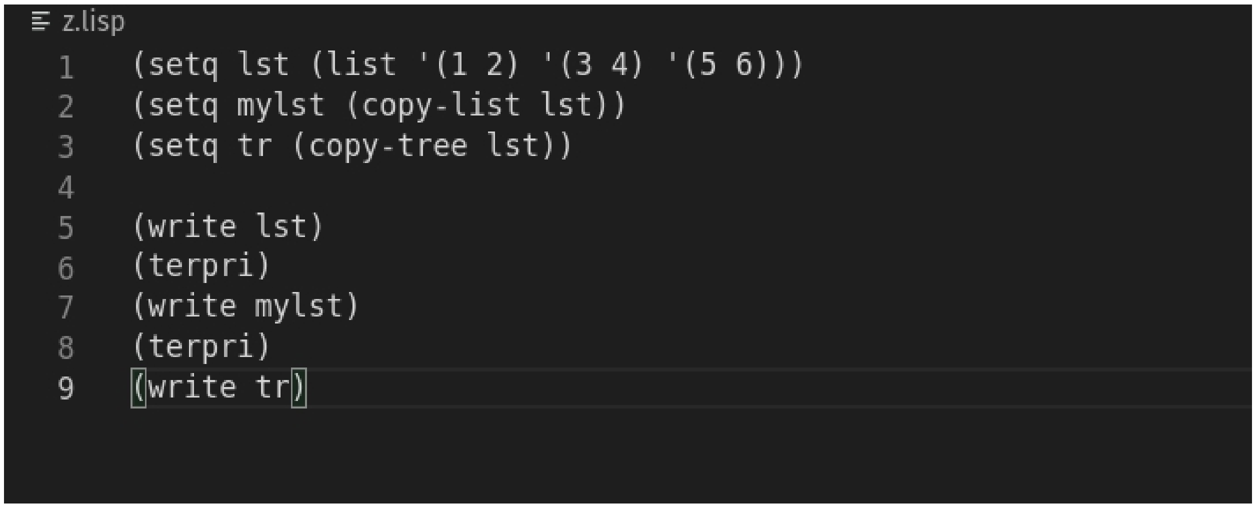


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
NIL
(2 1)
(2 1)
(3 2 1)
bshesh@pop-os:~/Documents/Programming/lisp$ 
```

1.18 LISP-Tree:

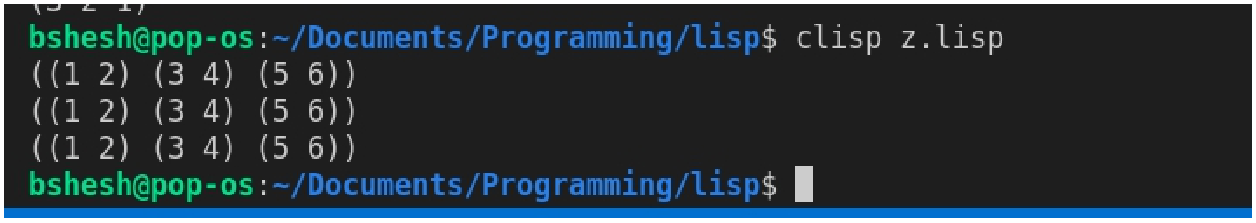
To implement tree structures, you will have to design functionalities that would traverse through the cons cells, in specific order, for example, pre-order, in-order, and post-order for binary trees.

Example:



```
z.lisp
(setq lst (list '(1 2) '(3 4) '(5 6)))
(setq mylst (copy-list lst))
(setq tr (copy-tree lst))

(write lst)
(terpri)
(write mylst)
(terpri)
(write tr)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))
bshesh@pop-os:~/Documents/Programming/lisp$ 
```

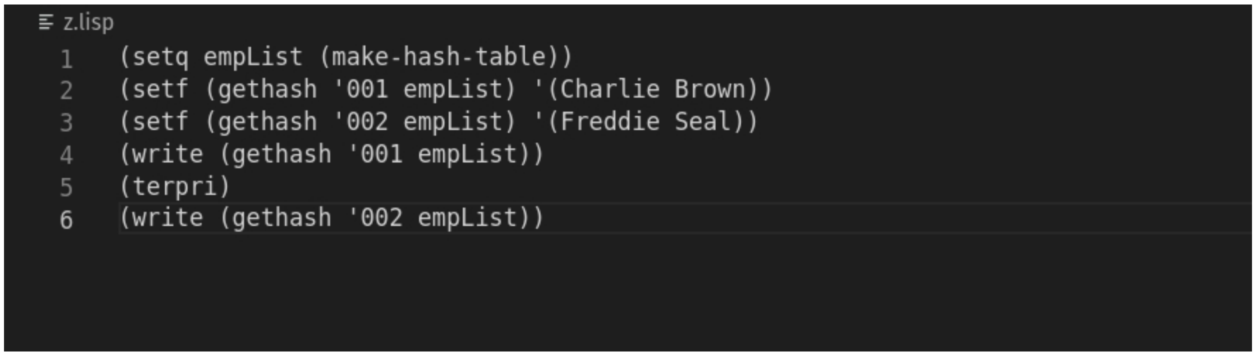
1.19 LISP-Hash Table:

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 −

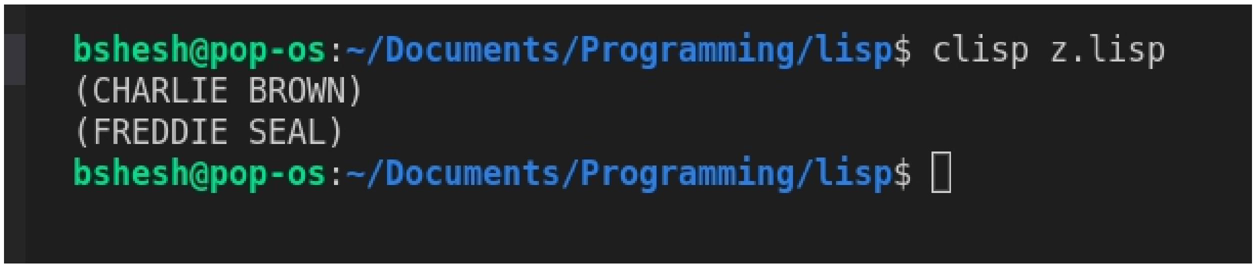
make-hash-table &key :test :size :rehash-size :rehash-threshold

Example:



```
z.lisp
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown))
(setf (gethash '002 empList) '(Freddie Seal))
(write (gethash '001 empList))
(terpri)
(write (gethash '002 empList))
```

Output:

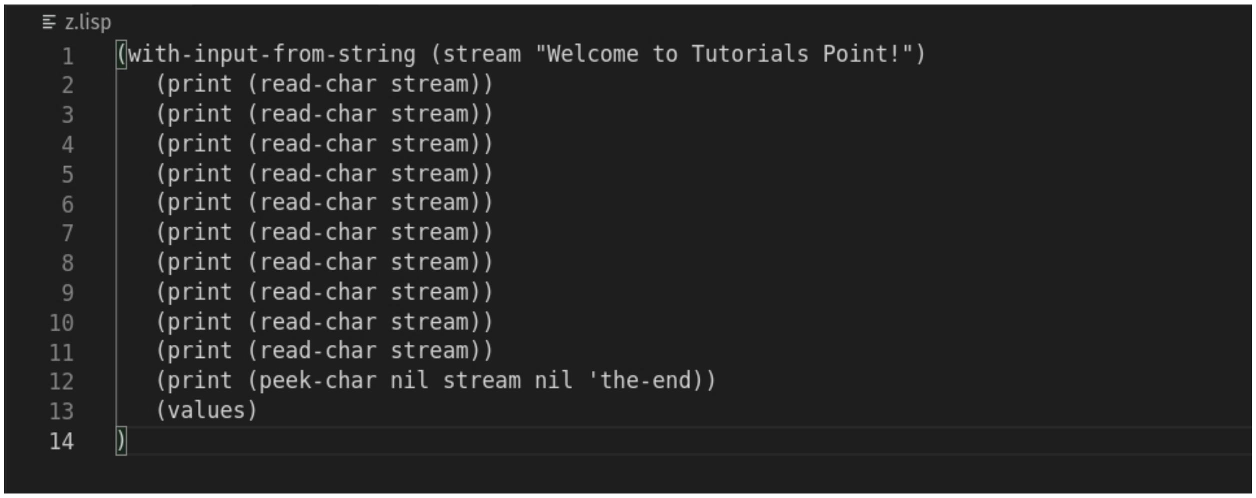


```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
(CHARLIE BROWN)
(FREDDIE SEAL)
bshesh@pop-os:~/Documents/Programming/lisp$ 
```

1.20 LISP-Input and Output:

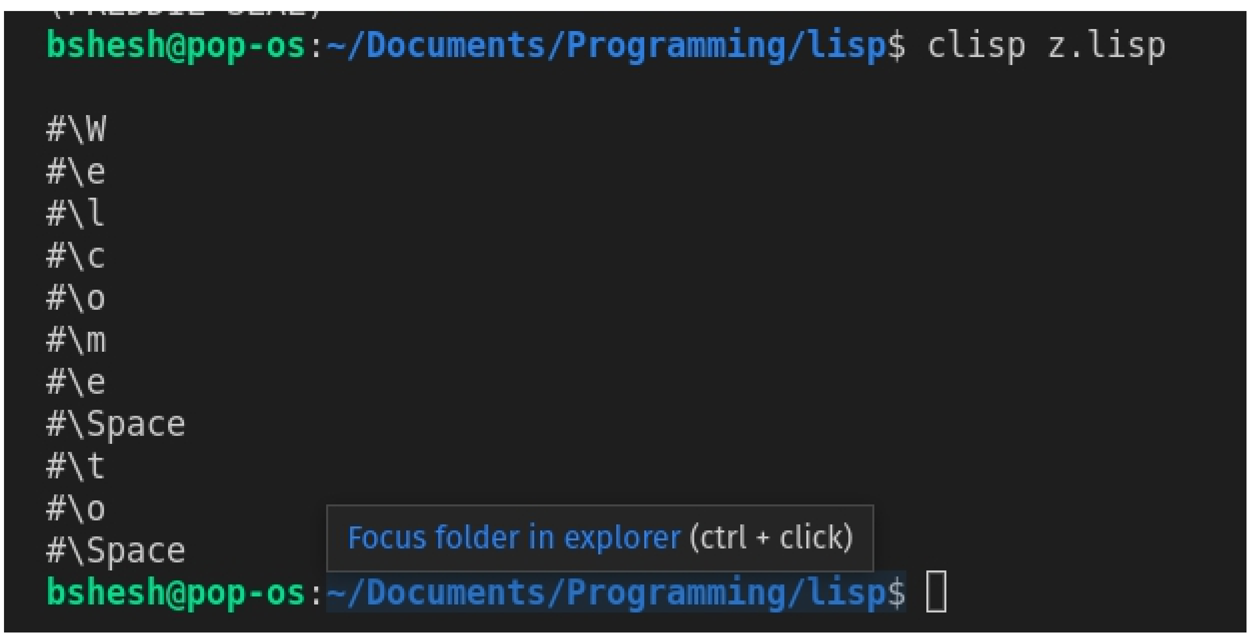
Common LISP provides numerous input-output functions. We have already used the format function, and print function for output. In this section, we will look into some of the most commonly used input-output functions provided in LISP.

Example:



```
z.lisp
(with-input-from-string (stream "Welcome to Tutorials Point!")
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (read-char stream))
   (print (peek-char nil stream nil 'the-end))
   (values)
)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp

#\W
#\e
#\l
#\c
#\o
#\m
#\e
#\Space
#\t
#\o
#\Space
bshesh@pop-os:~/Documents/Programming/lisp$
```

LISP-FIle I/O:

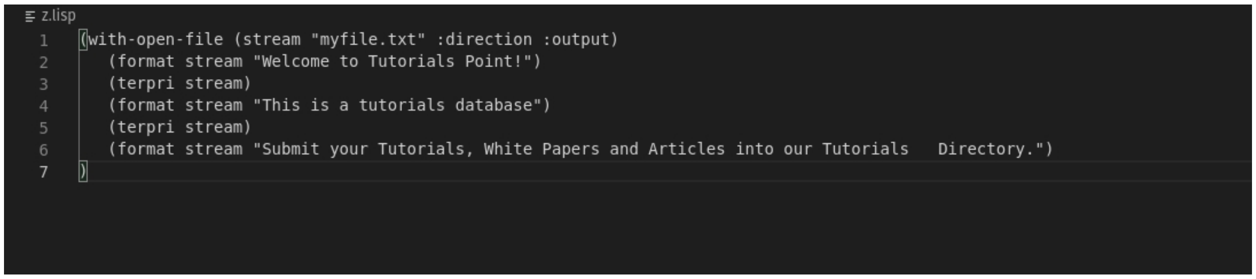
A file represents a sequence of bytes, does not matter if it is a text file or binary file.

For example: For opening files;

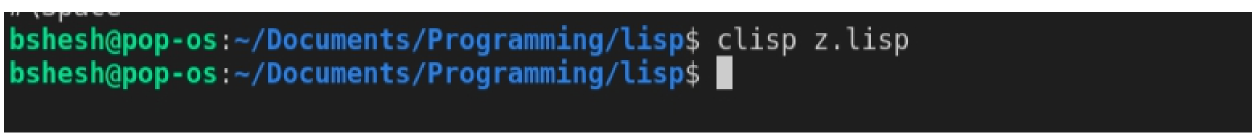
Syntax for the open function is –

open filename &key :direction :element-type :if-exists :if-does-not-exist :external-format

Example:

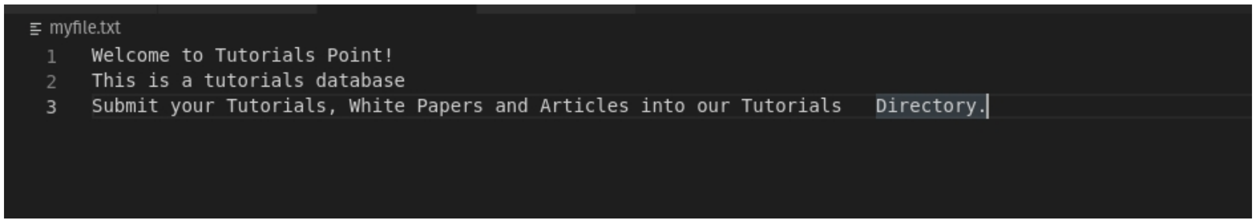


```
z.lisp
(with-open-file (stream "myfile.txt" :direction :output)
  (format stream "Welcome to Tutorials Point!")
  (terpri stream)
  (format stream "This is a tutorials database")
  (terpri stream)
  (format stream "Submit your Tutorials, White Papers and Articles into our Tutorials   Directory.")
)
```



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
bshesh@pop-os:~/Documents/Programming/lisp$ 
```

Output:



```
myfile.txt
Welcome to Tutorials Point!
This is a tutorials database
Submit your Tutorials, White Papers and Articles into our Tutorials   Directory.
```

1.21 LISP-Structures:

Structures are one of the user-defined data type, which allows you to combine data items of different kinds.Structures are used to represent a record.

The defstruct macro in LISP allows you to define an abstract record structure.

To discuss the format of the defstruct macro, let us write the definition of the Book structure. We could define the book structure as –

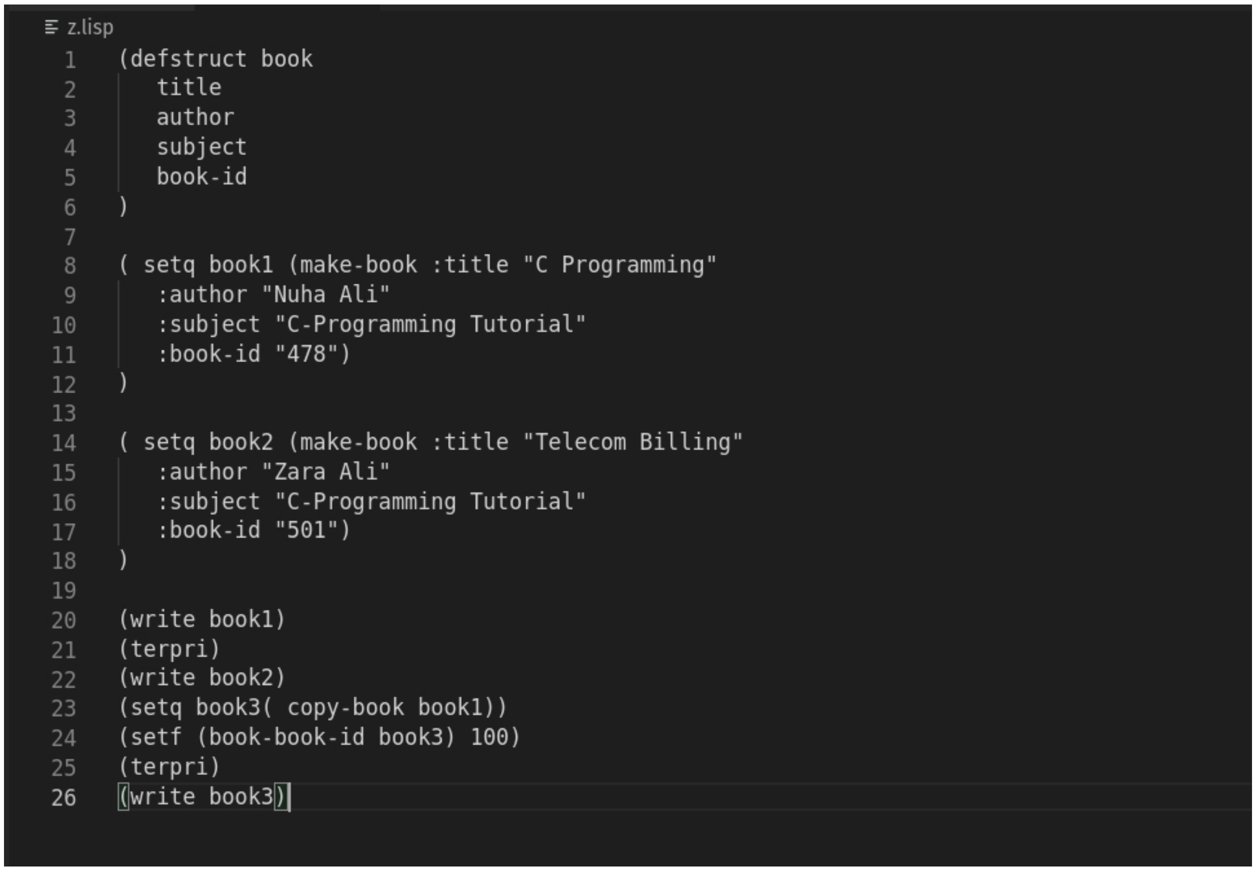
```
(defstruct book
   title
   author
```

subject

book-id

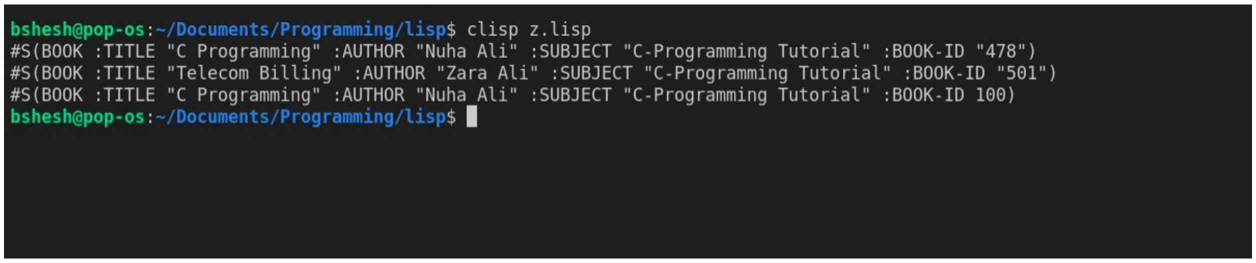
)

Example:



```
z.lisp
(defstruct book
   title
   author
   subject
   book-id
)

( setq book1 (make-book :title "C Programming"
   :author "Nuha Ali"
   :subject "C-Programming Tutorial"
   :book-id "478")
)

( setq book2 (make-book :title "Telecom Billing"
   :author "Zara Ali"
   :subject "C-Programming Tutorial"
   :book-id "501")
)

(write book1)
(terpri)
(write book2)
(setq book3( copy-book book1))
(setf (book-book-id book3) 100)
(terpri)
(write book3)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
#S(BOOK :TITLE "C Programming" :AUTHOR "Nuha Ali" :SUBJECT "C-Programming Tutorial" :BOOK-ID "478")
#S(BOOK :TITLE "Telecom Billing" :AUTHOR "Zara Ali" :SUBJECT "C-Programming Tutorial" :BOOK-ID "501")
#S(BOOK :TITLE "C Programming" :AUTHOR "Nuha Ali" :SUBJECT "C-Programming Tutorial" :BOOK-ID 100)
bshesh@pop-os:~/Documents/Programming/lisp$
```

1.22 LISP-Packages:

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.

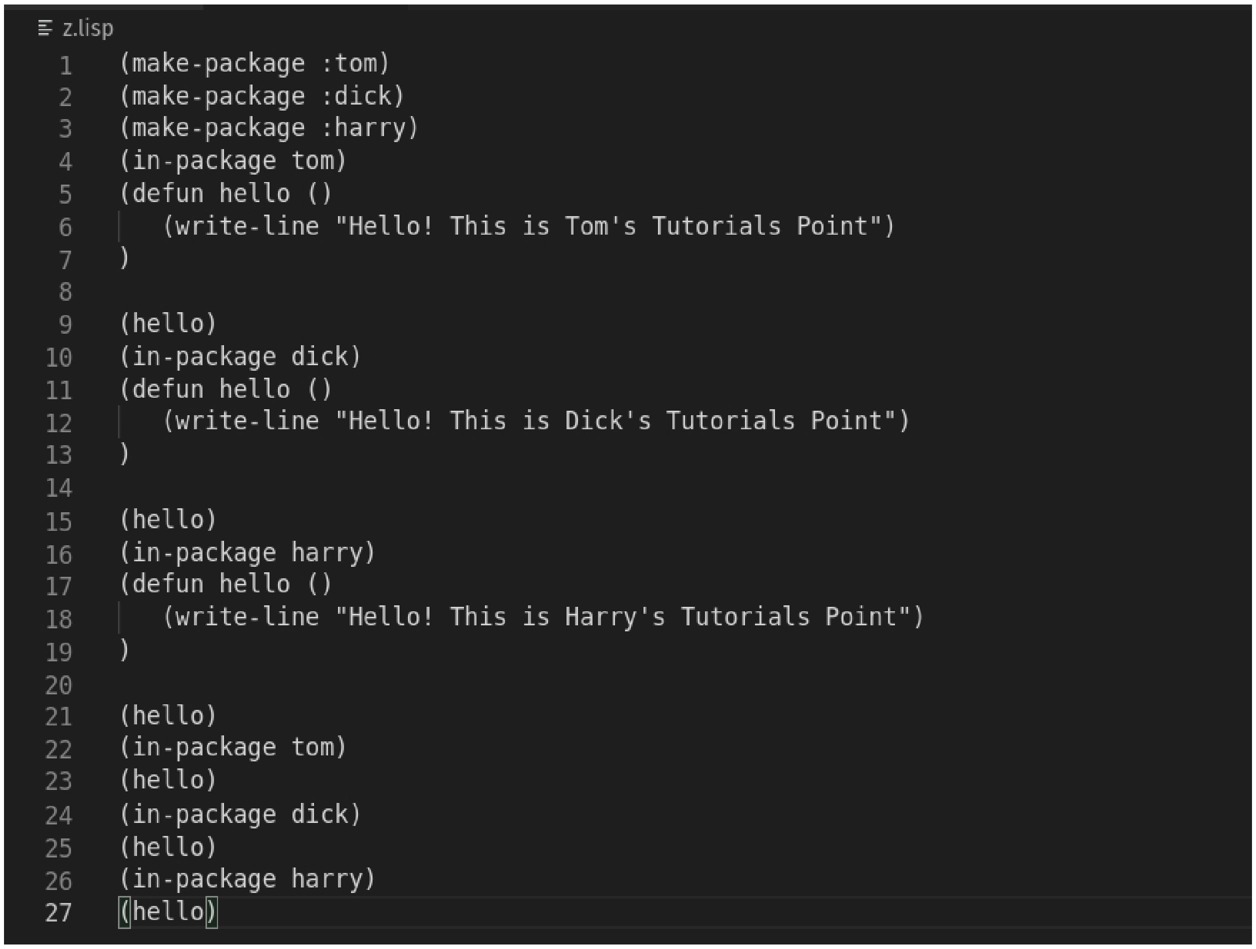
Creating a LISP Package

The defpackage function is used for creating an user defined package. It has the following syntax –

```
(defpackage :package-name

   (:use :common-lisp ...)

   (:export :symbol1 :symbol2 ...)

)
```

Using a Package:

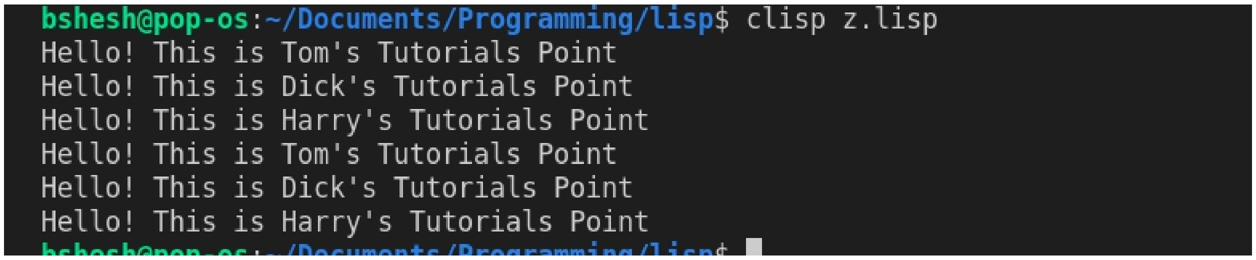
Example:



z.lisp

```lisp
(make-package :tom)
(make-package :dick)
(make-package :harry)
(in-package tom)
(defun hello ()
   (write-line "Hello! This is Tom's Tutorials Point")
)

(hello)
(in-package dick)
(defun hello ()
   (write-line "Hello! This is Dick's Tutorials Point")
)

(hello)
(in-package harry)
(defun hello ()
   (write-line "Hello! This is Harry's Tutorials Point")
)

(hello)
(in-package tom)
(hello)
(in-package dick)
(hello)
(in-package harry)
(hello)
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
bshesh@pop-os:~/Documents/Programming/lisp$
```

1.23 Error Handling:

The condition handling system in LISP has three parts −

- Signalling a condition
- Handling the condition
- Restart the process

"A condition is an object whose class indicates the general nature of the condition and whose instance data carries information about the details of the particular circumstances that lead to the condition being signalled"**.**

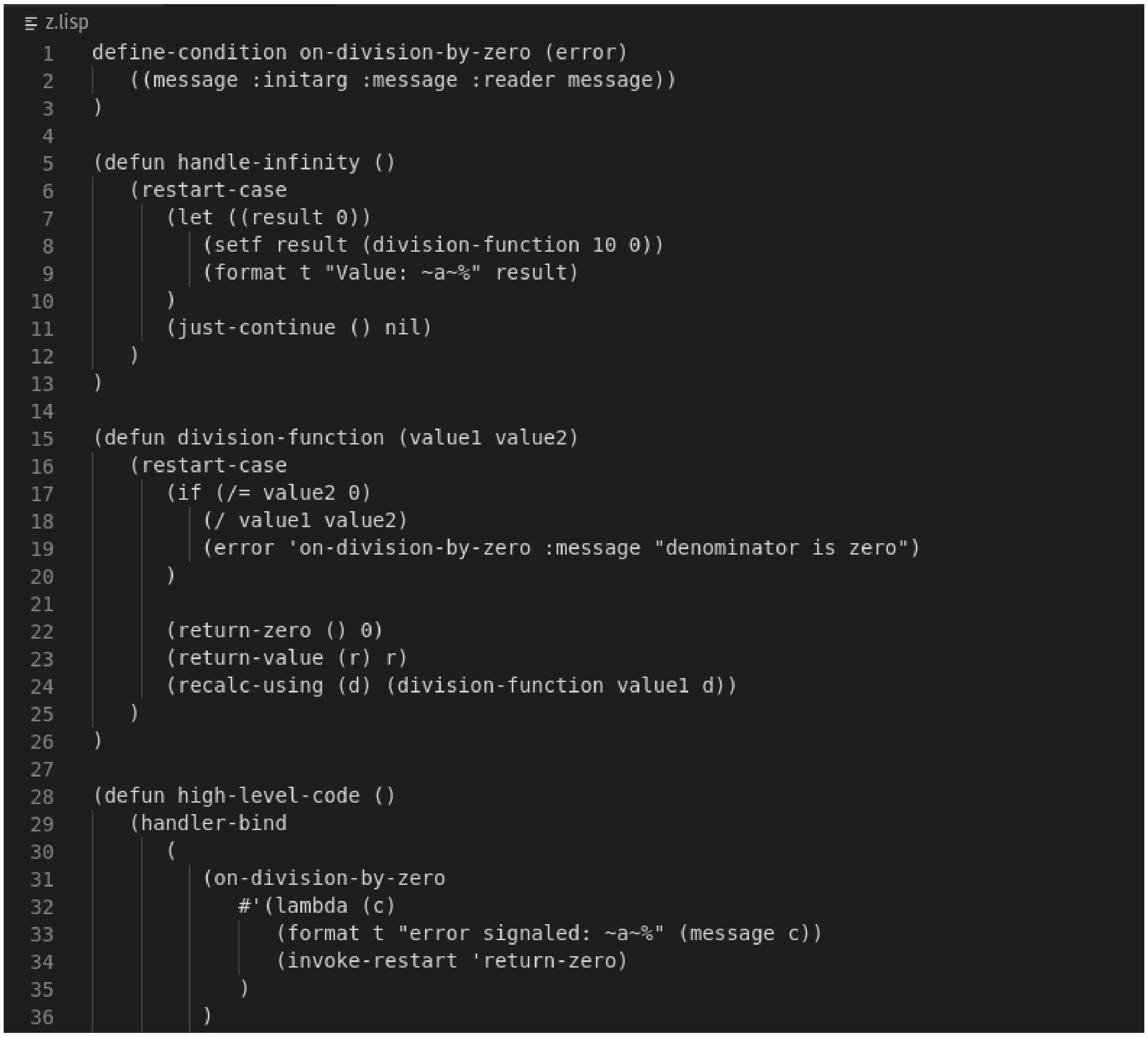
The define-condition macro is used for defining a condition, which has the following syntax −

(define-condition condition-name (error)

((text :initarg :text :reader text))

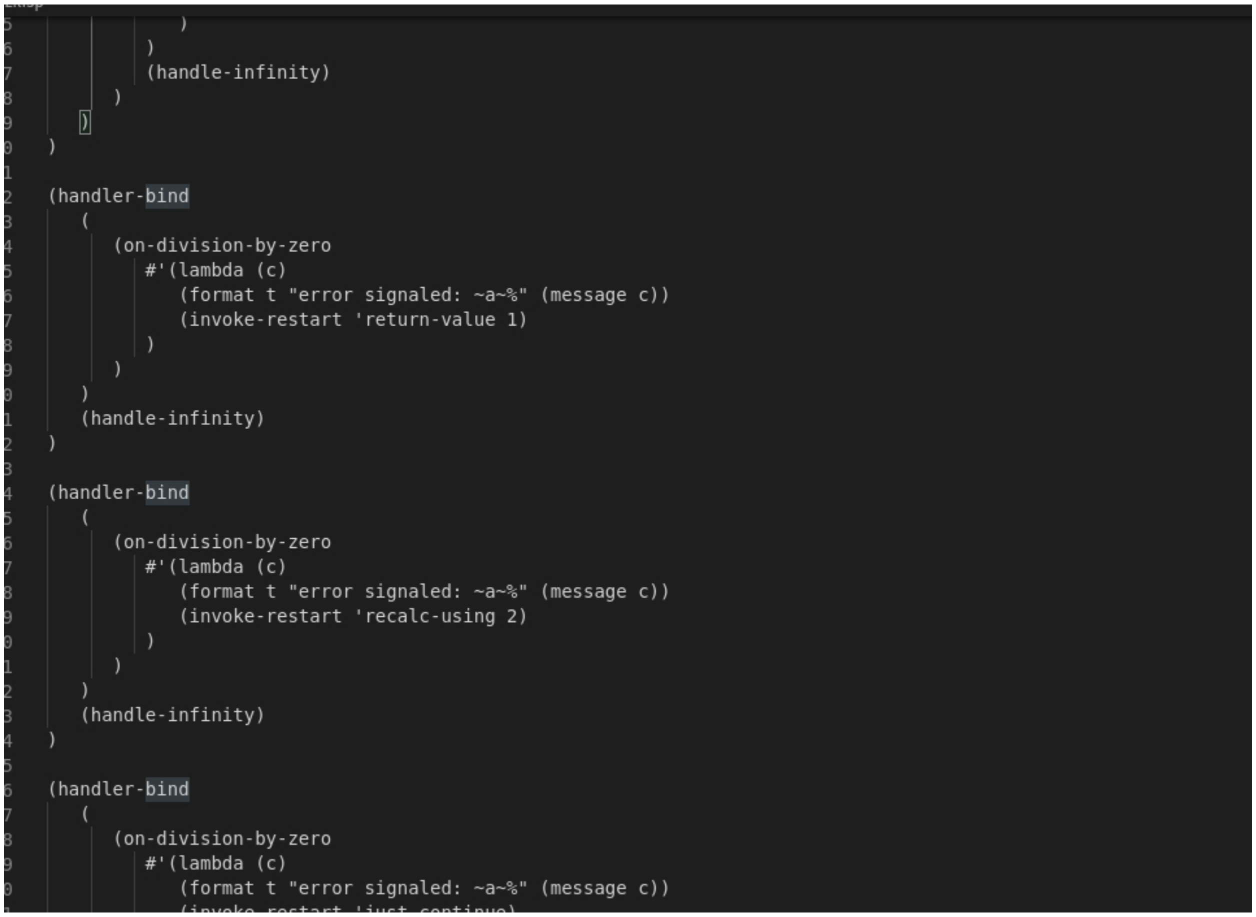
)

Example:

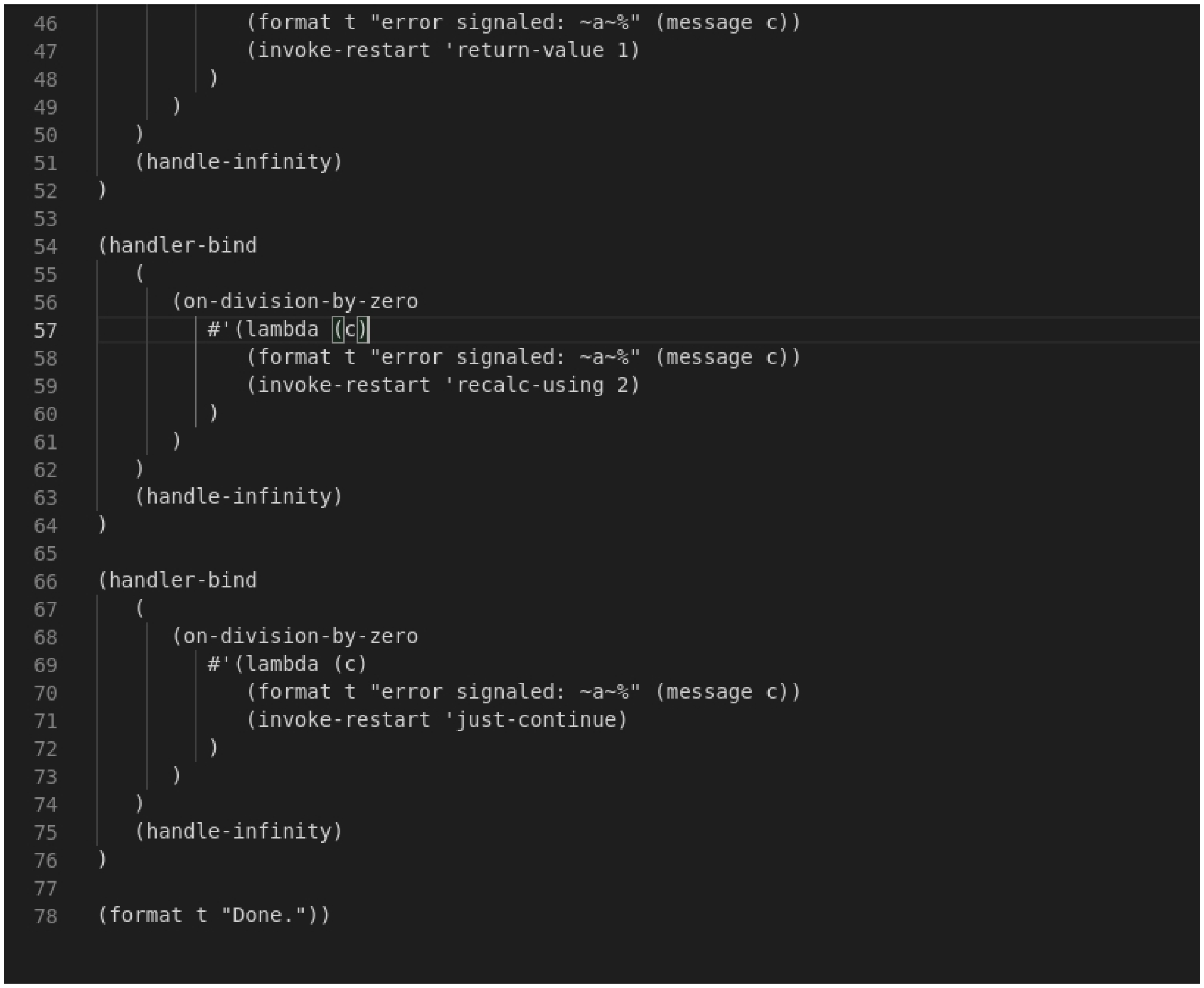


z.lisp

```
define-condition on-division-by-zero (error)
   ((message :initarg :message :reader message))
)

(defun handle-infinity ()
   (restart-case
      (let ((result 0))
         (setf result (division-function 10 0))
         (format t "Value: ~a~%" result)
      )
      (just-continue () nil)
   )
)

(defun division-function (value1 value2)
   (restart-case
      (if (/= value2 0)
         (/ value1 value2)
         (error 'on-division-by-zero :message "denominator is zero")
      )

      (return-zero () 0)
      (return-value (r) r)
      (recalc-using (d) (division-function value1 d))
   )
)

(defun high-level-code ()
   (handler-bind
      (
         (on-division-by-zero
            #'(lambda (c)
               (format t "error signaled: ~a~%" (message c))
               (invoke-restart 'return-zero)
            )
         )
```

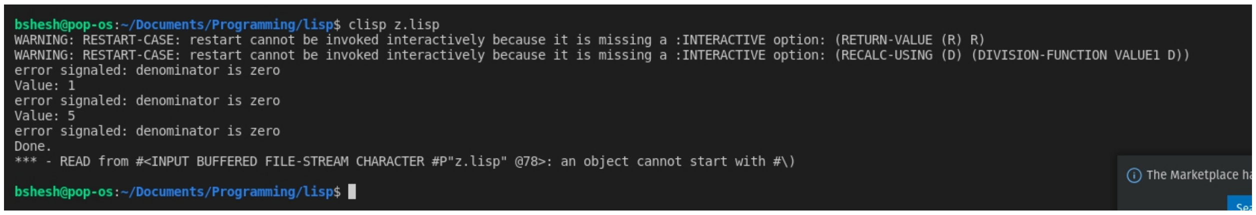


```
            )
         )
         (handle-infinity)
      )
   )
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'return-value 1)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'recalc-using 2)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'just-continue)
```



```
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'return-value 1)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'recalc-using 2)
         )
      )
   )
   (handle-infinity)
)

(handler-bind
   (
      (on-division-by-zero
         #'(lambda (c)
            (format t "error signaled: ~a~%" (message c))
            (invoke-restart 'just-continue)
         )
      )
   )
   (handle-infinity)
)

(format t "Done."))
```

Output:



```
bshesh@pop-os:~/Documents/Programming/lisp$ clisp z.lisp
WARNING: RESTART-CASE: restart cannot be invoked interactively because it is missing a :INTERACTIVE option: (RETURN-VALUE (R) R)
WARNING: RESTART-CASE: restart cannot be invoked interactively because it is missing a :INTERACTIVE option: (RECALC-USING (D) (DIVISION-FUNCTION VALUE1 D))
error signaled: denominator is zero
Value: 1
error signaled: denominator is zero
Value: 5
error signaled: denominator is zero
Done.
*** - READ from #<INPUT BUFFERED FILE-STREAM CHARACTER #P"z.lisp" @78>: an object cannot start with #\)

bshesh@pop-os:~/Documents/Programming/lisp$ 
```

Chapter 2: Conclusion:

LISP programming language was learnt and practised successfully. Hence, different syntax and features of LISP were understood and practically implemented.