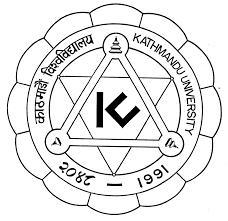
**KATHMANDU UNIVERSITY**

**Dhulikhel, Kavre**



**A Lab Report**

**On**

**“LISP”**

**[ Course title: COMP 301]**

**Submitted by:**

**Bisheshwor Neupane(35)**

**Submitted to:**

**Mr. Nabin Ghimire**

**Department of Computer Science and Engineering**

**Submission Date:**

**August 06, 2021**

# Introduction

**LISP**, in full listprocessing, a [computer programming language](https://www.britannica.com/technology/computer-programming-language) developed about 1960 by [John McCarthy](https://www.britannica.com/biography/John-McCarthy) at the [Massachusetts Institute of Technology](https://www.britannica.com/topic/Massachusetts-Institute-of-Technology) (MIT). LISP was founded on the mathematical theory of [recursive functions](https://www.britannica.com/science/recursive-function) (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](https://www.britannica.com/technology/FORTRAN) and [ALGOL](https://www.britannica.com/technology/ALGOL-computer-language).

# 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

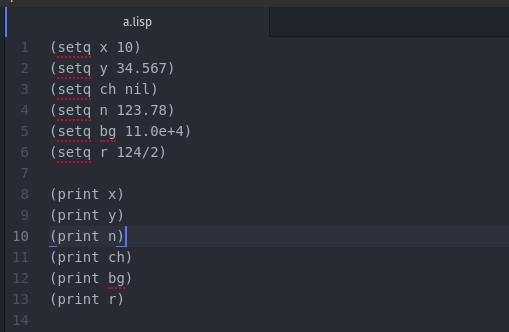
Lists, vectors, bit-vectors, and strings

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.

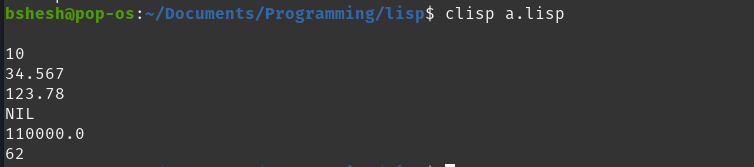
The **typep** predicated is used for finding whether an object belongs to a specific type.

The **type-of** function returns the data type of a given object.

Example:



Output:



# 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 −

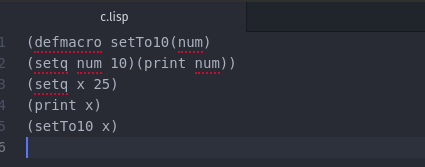
(defmacro macro-name (parameter-list))

"Optional documentation string."

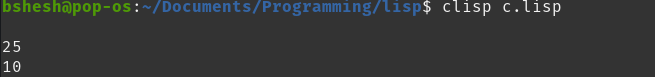
body-form

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:



Output:



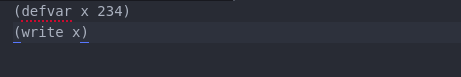
# LISP-Variables

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:



Output:



# 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.

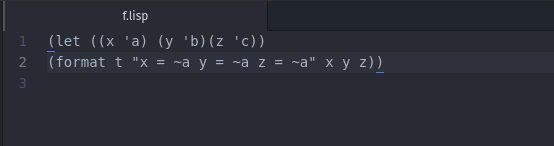
The let construct has the following syntax.

(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:



Output:



# 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

# 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 following table shows all the arithmetic operators supported by LISP.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | (+A B) will give 30 |
| - | Subtracts second operand from the first | (- A B) will give -10 |
| \* | Multiplies both operands | (\* A B) will give 200 |
| / | Divides numerator by de-numerator | (/ B A) will give 2 |
| mod,rem | Modulus Operator and remainder of after an integer division | (mod B A )will give 0 |
| incf | Increments operator increases integer value by the second argument specified | (incf A 3) will give 13 |
| decf | Decrements operator decreases integer value by the second argument specified | (decf A 4) will give 9 |

# 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.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of the operands are all equal or not, if yes then condition becomes true. | (= A B) is not true. |
| /= | Checks if the values of the operands are all different or not, if values are not equal then condition becomes true. | (/= A B) is true. |
| > | Checks if the values of the operands are monotonically decreasing. | (> A B) is not true. |
| < | Checks if the values of the operands are monotonically increasing. | (< A B) is true. |
| >= | Checks if the value of any left operand is greater than or equal to the value of next right operand, if yes then condition becomes true. | (>= A B) is not true. |
| <= | Checks if the value of any left operand is less than or equal to the value of its right operand, if yes then condition becomes true. | (<= A B) is true. |
| max | It compares two or more arguments and returns the maximum value. | (max A B) returns 20 |
| min | It compares two or more arguments and returns the minimum value. | (min A B) returns 10 |

# Logical Operations on Boolean Values

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

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| and | It takes any number of arguments. The arguments are evaluated left to right. If all arguments evaluate to non-nil, then the value of the last argument is returned. Otherwise nil is returned. | (and A B) will return NIL. |
| or | It takes any number of arguments. The arguments are evaluated left to right until one evaluates to non-nil, in such case the argument value is returned, otherwise it returns **nil**. | (or A B) will return 5. |
| not | It takes one argument and returns **t** if the argument evaluates to **nil.** | (not A) will return T. |

# 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 in the following table.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| logand | This returns the bit-wise logical AND of its arguments. If no argument is given, then the result is -1, which is an identity for this operation. | (logand a b)) will give 12 |
| logior | This returns the bit-wise logical INCLUSIVE OR of its arguments. If no argument is given, then the result is zero, which is an identity for this operation. | (logior a b) will give 61 |
| logxor | This returns the bit-wise logical EXCLUSIVE OR of its arguments. If no argument is given, then the result is zero, which is an identity for this operation. | (logxor a b) will give 49 |
| lognor | This returns the bit-wise NOT of its arguments. If no argument is given, then the result is -1, which is an identity for this operation. | (lognor a b) will give -62, |
| logeqv | This returns the bit-wise logical EQUIVALENCE (also known as exclusive nor) of its arguments. If no argument is given, then the result is -1, which is an identity for this operation. | (logeqv a b) will give -50 |

# LISP - Decision Making

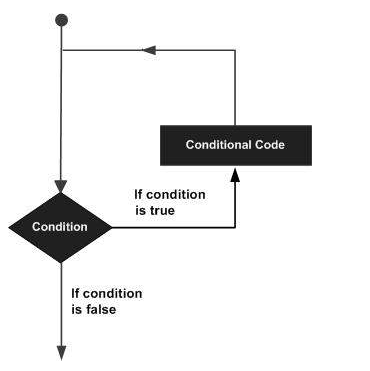
Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statements to be executed if the condition is determined to be true, and optionally other statements to be executed if the condition is determined to be false.

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

|  |  |
| --- | --- |
| **Sr.No.** | **Construct & Description** |
| 1 | **[cond](https://www.tutorialspoint.com/lisp/lisp_cond_construct.htm)**  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](https://www.tutorialspoint.com/lisp/lisp_if_construct.htm)  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](https://www.tutorialspoint.com/lisp/lisp_when_construct.htm)  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](https://www.tutorialspoint.com/lisp/lisp_case_construct.htm)  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. |

# 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(

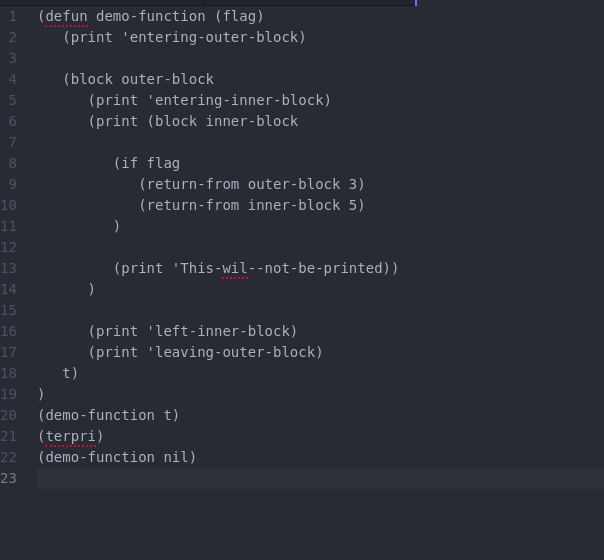
...

...

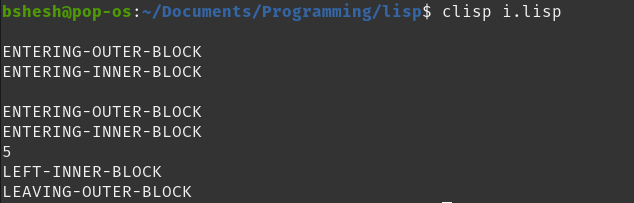
))

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

Example:



Output:



# LISP-Functions

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

# Defining Functions in LISP

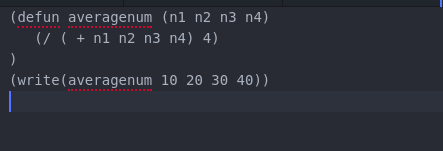
The macro named **defun** is used for defining functions. The **defun** macro needs three arguments −

* Name of the function
* Parameters of the function
* Body of the function

Syntax:

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

Example**:**



Output:



# 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.

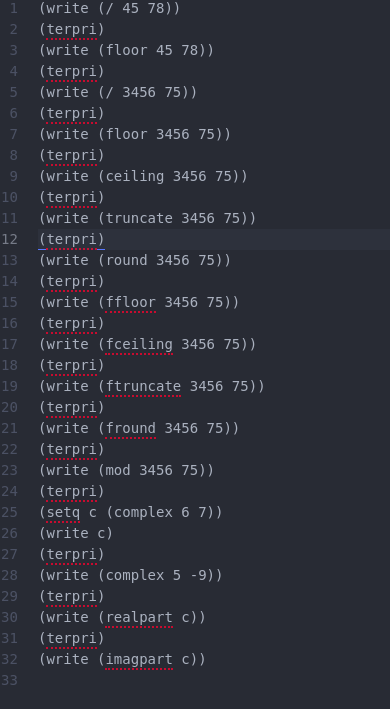
# LISP-Numbers

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

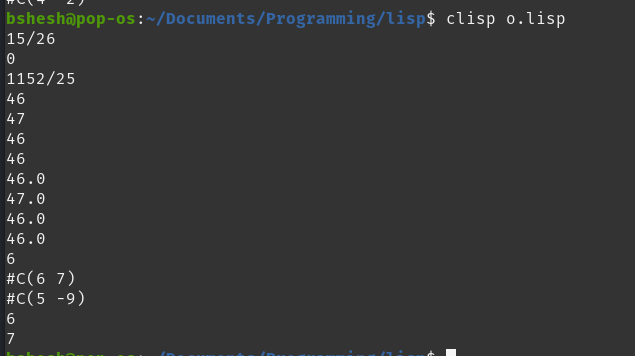
The number types supported by LISP are −

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

Example:



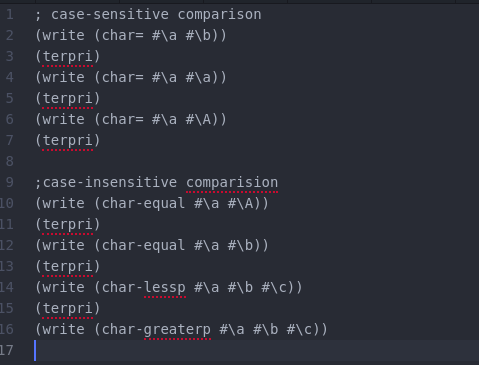
Output:



# LISP-Characters:

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

**Example:**



Output:



# LISP-Arrays:

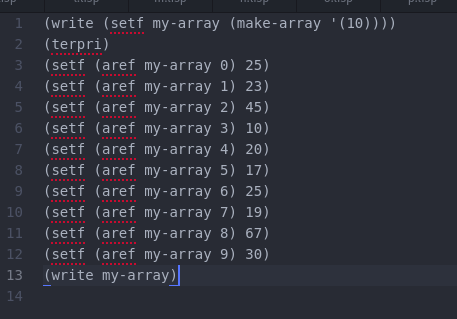
LISP allows you to define single or multiple-decision arrays using the make-array function.

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:



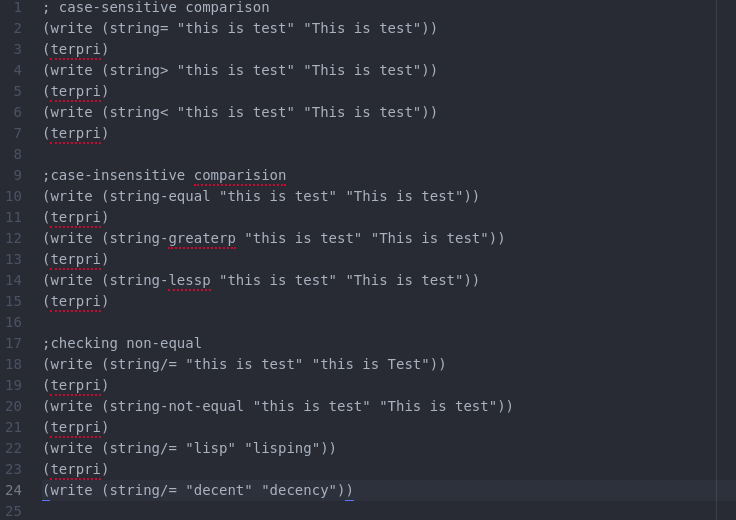
Output:



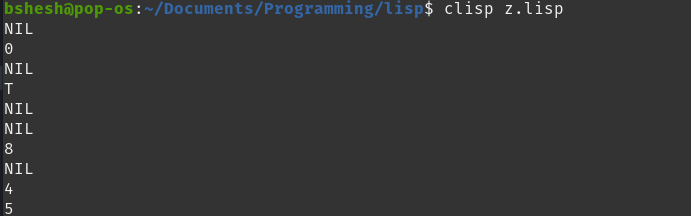
# LISP-Strings:

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:



Output:



# LISP-Sequences:

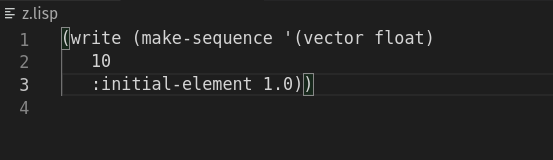
Sequence is an abstract data type in LISP. Vectors and lists are the two concrete subtypes of this data type. All the functionalities defined on sequence data type are actually applied on all vectors and list types.

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

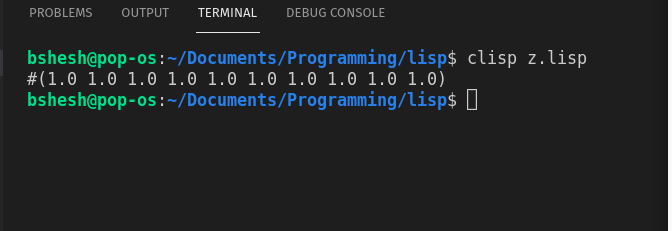
make-sequence sqtype sqsize &key :initial-element

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

Example:



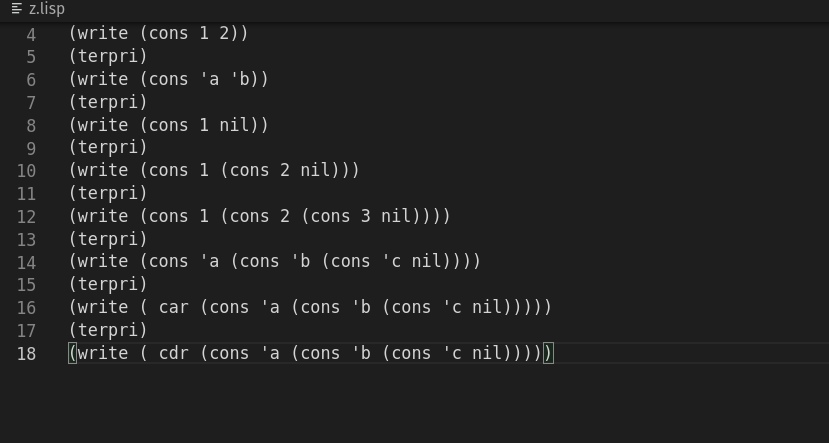
Output:



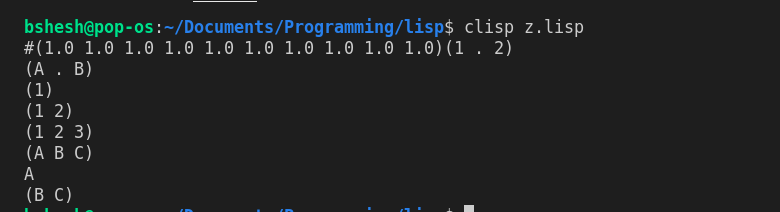
# LISP-Lists:

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:



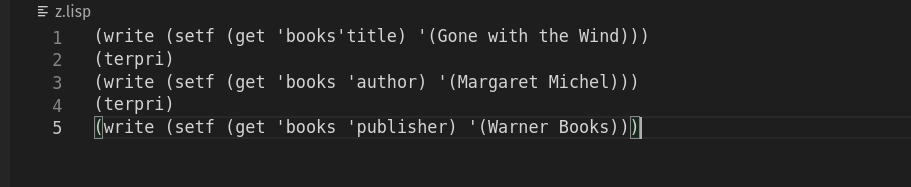
Output:



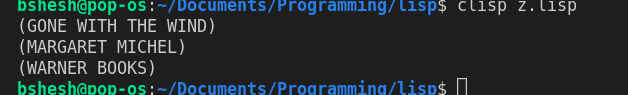
# LISP-Symbols:

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:**



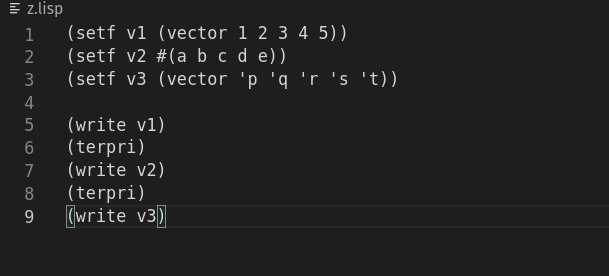
Output:



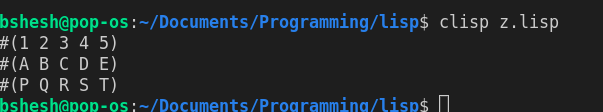
# LISP-Vectors:

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:



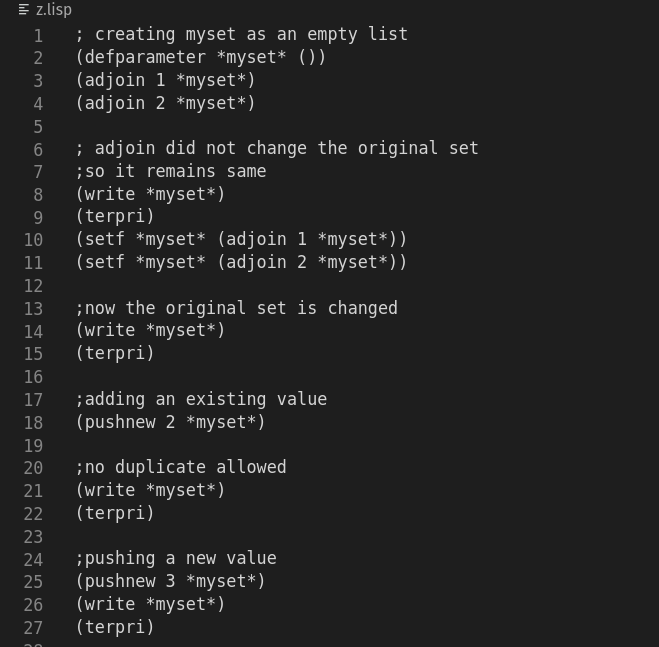
Output:



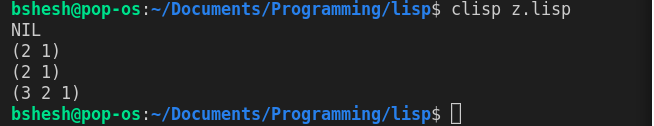
# **LISP-Set**:

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:



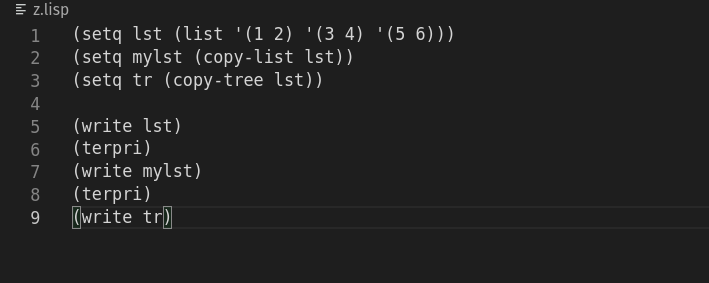
Output:



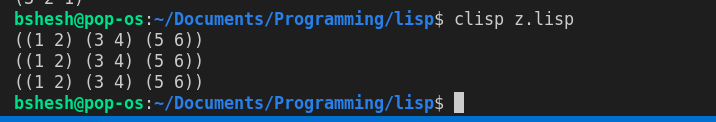
# LISP-Tree:

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:



Output:



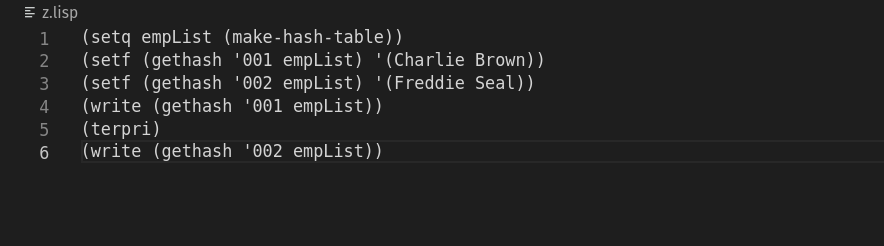
# LISP-Hash Table:

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.

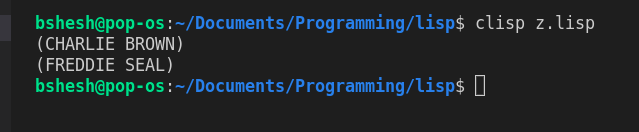
The **make-hash-table** function is used for creating a hash table. Syntax for this function is −

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

Example:



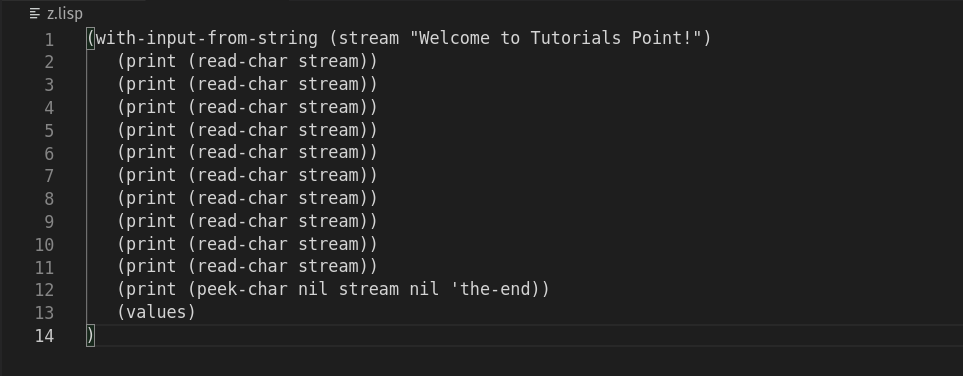
Output:



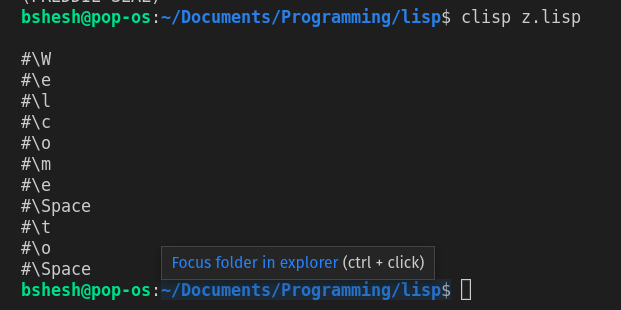
# LISP-Input and Output:

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:

****

Output:

****

# LISP-FIle I/O:

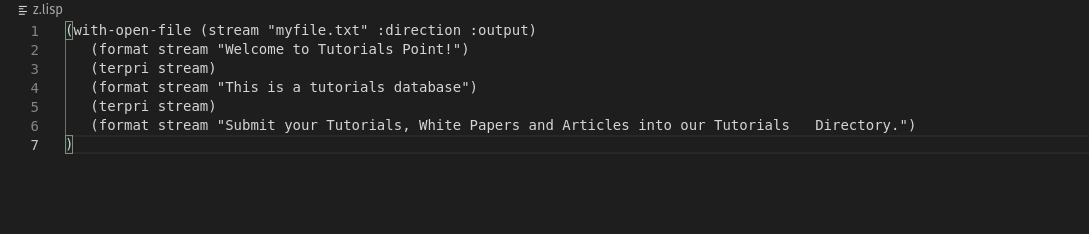
A file represents a sequence of bytes, does not matter if it is a text file or binary file.

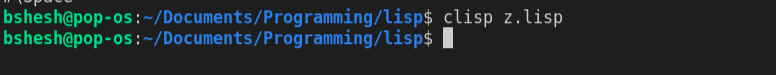
For example: For opening files;

Syntax for the **open** function is −

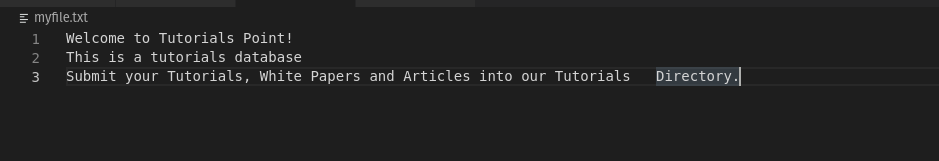
open filename &key :direction :element-type :if-exists :if-does-not-exist :external-format

Example:





Output:

****

# 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

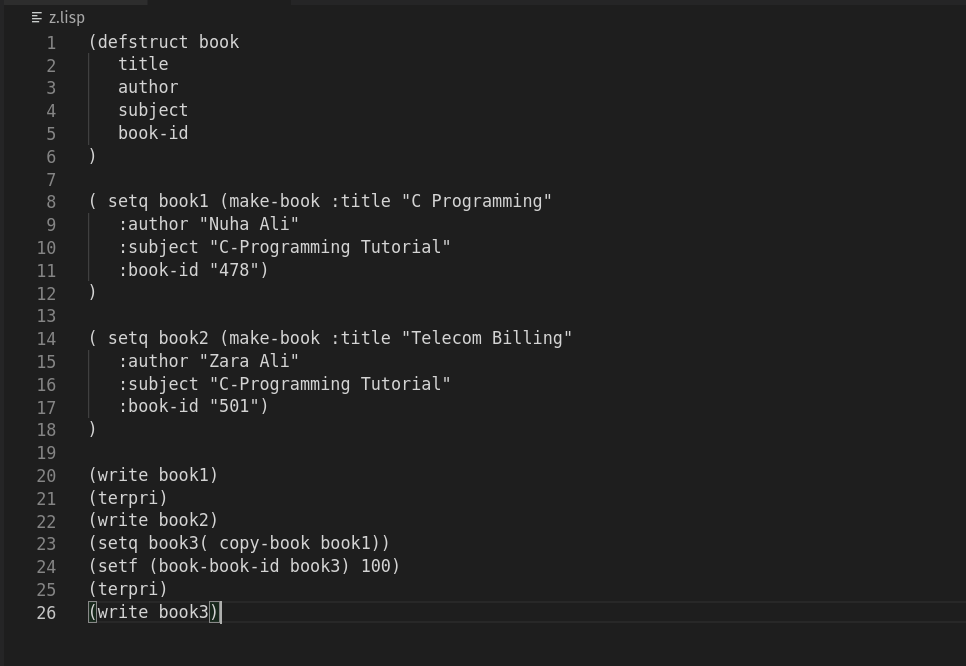
title

author

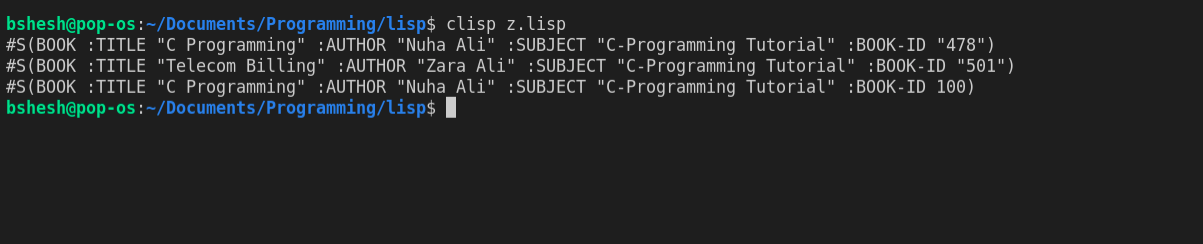
subject

book-id

)

Example:

Output:



# 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

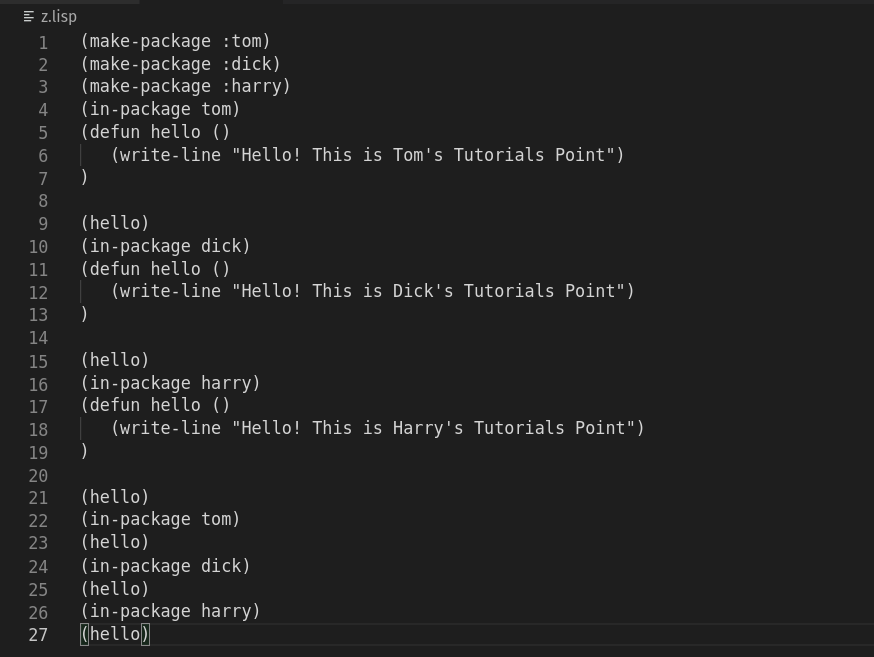
(:use :common-lisp ...)

(:export :symbol1 :symbol2 ...)

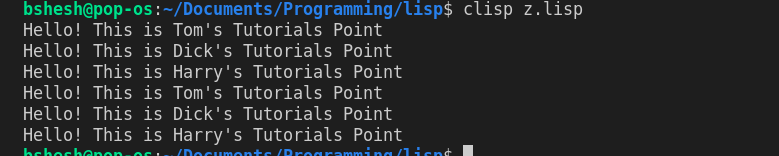
)

Using a Package:

Example:



Output:



# 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"**.**

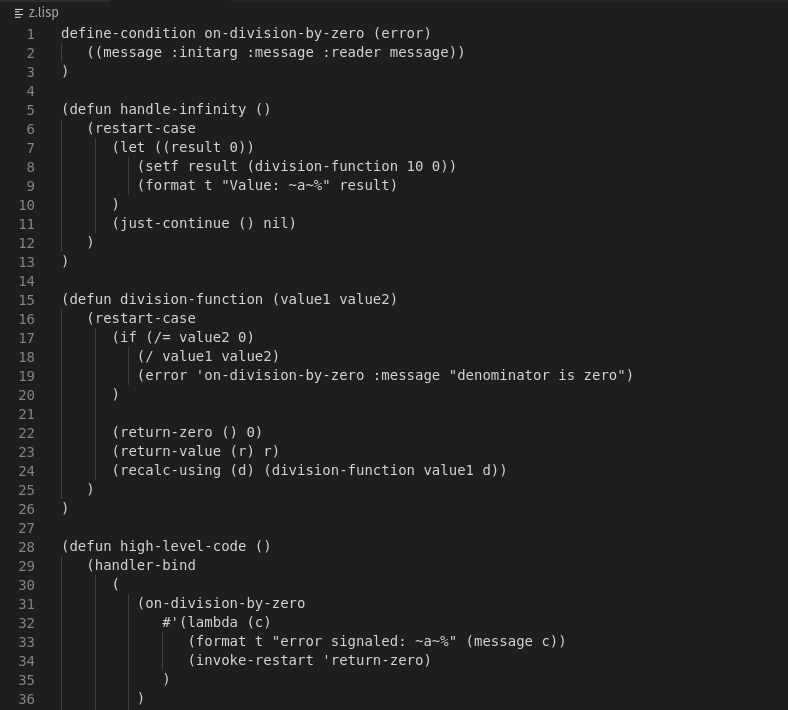
The define-condition macro is used for defining a condition, which has the following syntax −

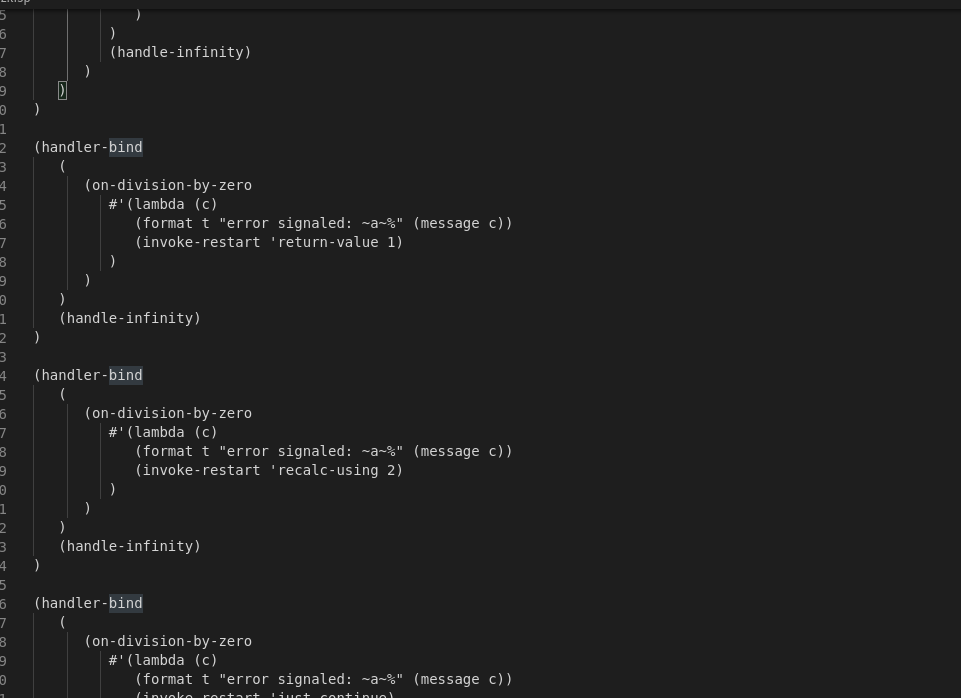
**(define-condition condition-name (error)**

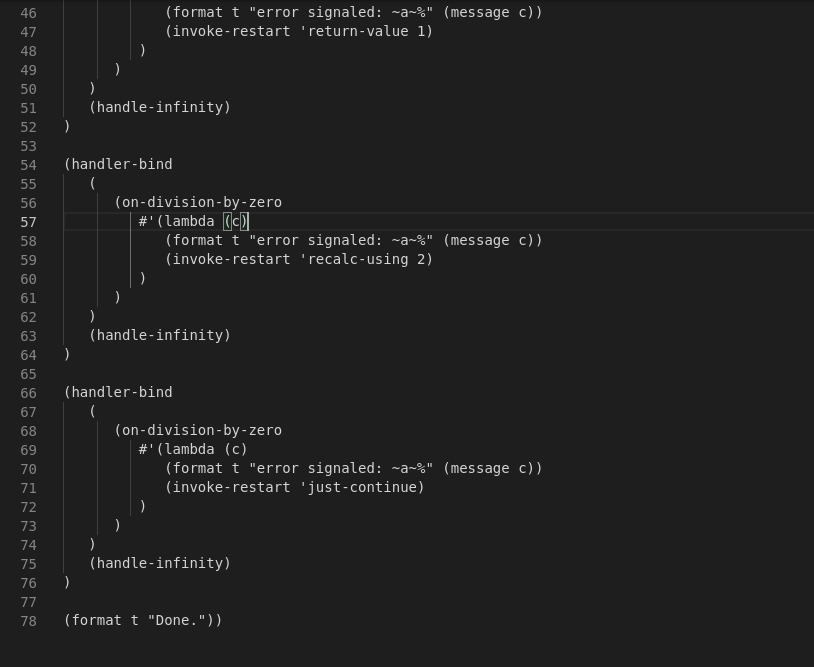
**((text :initarg :text :reader text))**

**)**

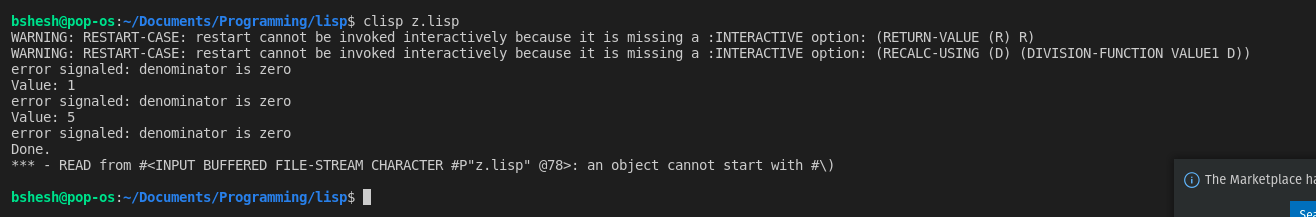
Example:

****

****



Output:



# Conclusion:

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