LISP

LEARN LISP

list processing

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

LISP is the second-oldest high-level programming language after Fortran and has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely known general-purpose LISP dialects are Common LISP and Scheme.

This tutorial takes you through features of LISP Programming language by simple and practical approach of learning.

Audience

This reference has been prepared for the beginners to help them understand the basic to advanced concepts related to LISP Programming language.

Prerequisites

Before you start doing practice with various types of examples given in this reference, we assume that you are already aware of the fundamentals of computer programming and programming languages.

Copyright & Disclaimer

Table of Contents

1. OVERVIEW

LISP stands for **LIS**t **P**rogramming. John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implemented by Steve Russell on an IBM 704 computer. It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information efficiently.

Common LISP originated during the decade of 1980 to 1990, in an attempt to unify the work of several implementation groups, as a successor of Maclisp like ZetaLisp and New Implementation of LISP (NIL) etc.

It serves as a common language, which can be easily extended for specific implementation. Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

Features of Common LISP

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

Applications Developed in LISP

The following applications are developed in LISP: Large successful applications built in LISP.

- Emacs: It is a cross platform editor with the features of extensibility, customizability, self-document ability, and real-time display.
- G2
- AutoCad
- Igor Engraver
- Yahoo Store

2. ENVIRONMENT SETUP

CLISP is the GNU Common LISP multi-architechtural compiler used for setting up LISP in Windows. The Windows version emulates Unix environment using MingW under Windows. The installer takes care of this and automatically adds CLISP to the Windows PATH variable.

You can get the latest CLISP for Windows at:

http://sourceforge.net/projects/clisp/files/latest/download

It creates a shortcut in the Start Menu by default, for the line-by-line interpreter.

How to Use CLISP

During installation, CLISP is automatically added to your PATH variable if you select the option (RECOMMENDED). It means that you can simply open a new Command window and type "clisp" to bring up the compiler.

To run a *.lisp or *.lsp file, simply use:

```
clisp hello.lisp
```

3. PROGRAM STRUCTURE

LISP expressions are called symbolic expressions or S-expressions. The S-expressions are composed of three valid objects:

- Atoms
- Lists
- Strings

Any S-expression is a valid program. LISP programs run either on an **interpreter** or as **compiled code.**

The interpreter checks the source code in a repeated loop, which is also called the Read-Evaluate-Print Loop (REPL). It reads the program code, evaluates it, and prints the values returned by the program.

A Simple LISP Program

Let us write an s-expression to find the sum of three numbers 7, 9 and 11. To do this, we can type at the interpreter prompt ->:

```
(+7911)
```

LISP returns the following result:

```
27
```

If you would like to execute the same program as a compiled code, then create a LISP source code file named myprog.lisp and type the following code in it:

```
(write(+7911))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
27
```

LISP Uses Prefix Notation

In prefix notation, operators are written before their operands. You might have noted that LISP uses **prefix notation.** In the above program, the '+' symbol works as a function name for the process of summation of the numbers.

For example, the following expression,

```
a * ( b + c ) / d
```

is written in LISP as:

```
(/ (* a (+ b c) ) d)
```

Let us take another example. Let us write code for converting Fahrenheit temperature of 60° F to the centigrade scale:

The mathematical expression for this conversion is:

```
(60 * 9 / 5) + 32
```

Create a source code file named main.lisp and type the following code in it:

```
(write(+ (* (/ 9 5) 60) 32))
```

When you click the Execute button, or type Ctrl+E, MATLAB executes it immediately and the result is:

```
140
```

Evaluation of LISP Programs

The LISP program has two parts:

- Translation of program text into LISP objects by a reader program.
- Implementation of the semantics of the language in terms of LSIP objects by an evaluator program.

The evaluation program takes the following steps:

- The reader translates the strings of characters to LISP objects or **s-expressions.**
- The evaluator defines syntax of LISP **forms** that are built from s-expressions.

- This second level of evaluation defines a syntax that determines which s-expressions are LISP forms.
- The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the LISP expression in parenthesis, because we are sending the entire expression/form to the evaluator as argument.

The 'Hello World' Program

Learning a new programming language does not really take off until you learn how to greet the entire world in that language, right ?

Let us create new source code file named main.lisp and type the following code in it:

```
(write-line "Hello World")

(write-line "I am at 'Tutorials Point'! Learning LISP")
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
Hello World

I am at 'Tutorials Point'! Learning LISP
```

4. BASIC SYNTAX

This chapter introduces you to basic syntax structure in LISP.

Basic Elements in LISP

LISP programs are made up of three basic elements:

- atom
- list
- string

An **atom** is a number or string of contiguous characters. It includes numbers and special characters. The following examples show some valid atoms:

```
hello-from-tutorials-point
name
123008907
*hello*
Block#221
abc123
```

A **list** is a sequence of atoms and/or other lists enclosed in parentheses. The following examples show some valid lists:

```
( i am a list)
(a ( a b c) d e fgh)
(father tom ( susan bill joe))
(sun mon tue wed thur fri sat)
( )
```

A **string** is a group of characters enclosed in double quotation marks. The following examples show some valid strings:

```
" I am a string"
"a ba c d efg #$%^&!"
"Please enter the following details:"
```

```
"Hello from 'Tutorials Point'! "
```

Adding Comments

The semicolon symbol (;) is used for indicating a comment line.

Example

```
(write-line "Hello World") ; greet the world
; tell them your whereabouts
(write-line "I am at 'Tutorials Point'! Learning LISP")
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is:

```
Hello World
I am at 'Tutorials Point'! Learning LISP
```

Notable Points

The following important points are notable:

- The basic numeric operations in LISP are +, -, *, and /
- LISP represents a function call f(x) as (f x), for example cos(45) is written as cos 45
- LISP expressions are not case-sensitive. Means, cos 45 or COS 45 are same.
- LISP tries to evaluate everything, including the arguments of a function. Only three types of elements are constants and always return their own value:
    - Numbers
    - The letter **t**, that stands for logical true
    - The value **nil**, that stands for logical false, as well as an empty list.

LISP Forms

In the previous chapter, we mentioned that the evaluation process of LISP code takes the following steps:

- The reader translates the strings of characters to LISP objects or **s-expressions.**

- The evaluator defines syntax of LISP **forms** that are built from s-expressions. This second level of evaluation defines a syntax that determines which s-expressions are LISP forms.

A LISP form can be:

- An atom
- An empty list or non-list
- Any list that has a symbol as its first element

The evaluator works as a function that takes a valid LISP form as an argument and returns a value. This is the reason why we put the LISP expression in parenthesis, because we are sending the entire expression/form to the evaluator as argument.

Naming Conventions in LISP

Name or symbols can consist of any number of alphanumeric characters other than whitespace, open and closing parentheses, double and single quotes, backslash, comma, colon, semicolon and vertical bar. To use these characters in a name, you need to use escape character (\).

A name can have digits but must not be made of only digits, because then it would be read as a number. Similarly a name can have periods, but cannot be entirely made of periods.

Use of Single Quotation Mark

LISP evaluates everything including the function arguments and list members.

At times, we need to take atoms or lists literally and do not want them evaluated or treated as function calls. To do this, we need to precede the atom or the list with a single quotation mark.

The following example demonstrates this:

Create a file named main.lisp and type the following code into it:

```
(write-line "single quote used, it inhibits evaluation")
(write '(* 2 3))
(write-line " ")
(write-line "single quote not used, so expression evaluated")
(write (* 2 3))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
single quote used, it inhibits evaluation
(* 2 3)
single quote not used, so expression evaluated
6
```

5. DATA TYPES

LISP data types can be categorized as:

Scalar types - numbers, characters, symbols etc.

Data structures - lists, vectors, bit-vectors, and strings.

Any variable can take any LISP object as its value, unless you declare it explicitly. Although, it is not necessary to specify a data type for a LISP variable, however, it helps in certain loop expansions, in method declarations and some other situations that we will discuss in later chapters.

The data types are arranged into a hierarchy. A data type is a set of LISP objects and many objects may belong to one such set.

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

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

Type Specifiers in LISP

Type specifiers are system-defined symbols for data types.

Array	fixnum	package	simple-string
Atom	float	pathname	simple-vector
Bignum	function	random-state	single-float
Bit	hash-table	Ratio	standard-char
bit-vector	integer	Rational	stream
Character	keyword	readtable	string
[common]	list	sequence	[string-char]
compiled-function	long-float	short-float	symbol
Complex	nill	signed-byte	t

Cons	null	simple-array	unsigned-byte
double-float	number	simple-bit-vector	vector

Apart from these system-defined types, you can create your own data types. When a structure type is defined using **defstruct** function, the name of the structure type becomes a valid type symbol.>/p>

Example 1

Create new source code file named *main.lisp* and type the following code in it:

```
(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)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result returned is:

```
10
34.567
123.78
NIL
110000.0
62
```

Example 2

Next let us check the types of the variables used in the previous example. Create new source code file named main.lisp and type the following code in it:

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

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the resultis:

```
(INTEGER 0 281474976710655)
SINGLE-FLOAT
SINGLE-FLOAT
NULL
SINGLE-FLOAT
(INTEGER 0 281474976710655)
```

6. MACROS

This chapter introduces you about macros in LISP.

A macro is a function that takes an s-expression as arguments and returns a LISP form, which is then evaluated. Macros allow you to extend the syntax of standard LISP.

Defining a Macro

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.

Example

Let us write a simple macro named setTo10, which takes a number and sets its value to 10.

Create new source code file named *main.lisp* and type the following code in it:

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

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
25
10
```

7. VARIABLES

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

Global Variables

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

Example

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

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
234
```

As there is no type declaration for variables in LISP, you need to specify a value for a symbol directly with the **setq** construct.

Example

```
->(setq x 10)
```

The above expression assigns the value 10 to the variable x. You can refer to the variable using the symbol itself as an expression.

The **symbol-value** function allows you to extract the value stored at the symbol storage place.

Example

Create new source code file named main.lisp and type the following code in it:

```
(setq x 10)
(setq y 20)
(format t "x = ~2d y = ~2d ~%" x y)
(setq x 100)
(setq y 200)
```

```
(format t "x = ~2d y = ~2d" x y)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
x = 10 y = 20
x = 100 y = 200
```

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 at last, the *s-expression* is evaluated. The value of the last expression evaluated is returned.

If you do not include an initial value for a variable, the variable is assigned to **nil.**

Example

Create new source code file named *main.lisp* and type the following code in it:

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

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
x = A y = B z = C
```

The **prog** construct also has the list of local variables as its first argument, which is followed by the body of the **prog,** and any number of s-expressions.

The **prog** function executes the list of s-expressions in sequence and returns nil unless it encounters a function call named **return.** Then the argument of the **return** function is evaluated and returned.

Example

Create new source code file named *main.lisp* and type the following code in it:

```
(prog ((x '(a b c))
        (y '(1 2 3))
        (z '(p q 10)))
(format t "x = ~a y = ~a z = ~a" x y z))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
x = (A B C) y = (1 2 3) z = (P Q 10)
```

8. CONSTANTS

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

Example

The following example shows declaring a global constant PI and later using this value inside a function named *area-circle* that calculates the area of a circle. The **defun** construct is used for defining a function, we will look into it in the 'Functions' chapter.

Create a new source code file named *main.lisp* and type the following code in it:

```
(defconstant PI 3.141592)
(defun area-circle(rad)
  (terpri)
  (format t "Radius: ~5f" rad)
  (format t "~%Area: ~10f" (* PI rad rad)))
  (area-circle 10)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
Radius:  10.0
Area:   314.1592
```

9. 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. Assume variable **A** = 10 and variable **B** = 20 then:

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

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(setq a 10)
(setq b 20)
(format t "~% A + B = ~d" (+ a b))
(format t "~% A - B = ~d" (- a b))
(format t "~% A x B = ~d" (* a b))
(format t "~% B / A = ~d" (/ b a))
(format t "~% Increment A by 3 = ~d" (incf a 3))
(format t "~% Decrement A by 4 = ~d" (decf a 4))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
A + B = 30
A - B = -10
A x B = 200
B / A = 2
Increment A by 3 = 13
Decrement A by 4 = 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.

Assume variable **A** = 10 and variable **B** = 20, then:

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 20

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(setq a 10)
(setq b 20)
(format t "~% A = B is ~a" (= a b))
(format t "~% A /= B is ~a" (/= a b))
(format t "~% A > B is ~a" (> a b))
(format t "~% A < B is ~a" (< a b))
(format t "~% A >= B is ~a" (>= a b))
(format t "~% A <= B is ~a" (<= a b))
(format t "~% Max of A and B is ~d" (max a b))
(format t "~% Min of A and B is ~d" (min a b))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
A = B is NIL
A /= B is T
```

```
A > B is NIL
A < B is T
A >= B is NIL
A <= B is T
Max of A and B is 20
Min of A and B is 10
```

Logical Operations on Boolean Values

Common LISP provides three logical operators: **and**, **or**, and **not** that operate on boolean values. Assume **A** = nil and **B** = 5, then:

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) returns 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) returns 5.
not	It takes one argument and returns t if the argument evaluates to nil.	(not A) returns T.

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(setq a 10)
(setq b 20)
(format t "~% A and B is ~a" (and a b))
(format t "~% A or B is ~a" (or a b))
(format t "~% not A is ~a" (not a))
(terpri)
(setq a nil)
```

```
(setq b 5)
(format t "~% A and B is ~a" (and a b))
(format t "~% A or B is ~a" (or a b))
(format t "~% not A is ~a" (not a))
(terpri)
(setq a nil)
(setq b 0)
(format t "~% A and B is ~a" (and a b))
(format t "~% A or B is ~a" (or a b))
(format t "~% not A is ~a" (not a))
(terpri)
(setq a 10)
(setq b 0)
(setq c 30)
(setq d 40)
(format t "~% Result of and operation on 10, 0, 30, 40 is ~a" (and a b c d))
(format t "~% Result of and operation on 10, 0, 30, 40 is ~a" (or a b c d))
(terpri)
(setq a 10)
(setq b 20)
(setq c nil)
(setq d 40)
(format t "~% Result of and operation on 10, 20, nil, 40 is ~a" (and a b c d))
(format t "~% Result of and operation on 10, 20, nil, 40 is ~a" (or a b c d))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
A and B is 20
A or B is 10
not A is NIL

A and B is NIL
A or B is 5
not A is T
```

```
A and B is NIL
A or B is 0
not A is T

Result of and operation on 10, 0, 30, 40 is 40
Result of and operation on 10, 0, 30, 40 is 10

Result of and operation on 10, 20, nil, 40 is NIL
Result of and operation on 10, 20, nil, 40 is 10
```

Please note that the logical operations work on Boolean values and secondly, **numeric zero and NIL are not same.**

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

Assume if A = 60; and B = 13; now in binary format they are represented as:

```
A = 0011 1100
B = 0000 1101
-----------------
A and B = 0000 1100
A or B = 0011 1101
A xor B = 0011 0001
not A  = 1100 0011
```

The Bitwise operators supported by LISP are listed in the following table. Assume variable A = 60 and variable B = 13, then:

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)) gives 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) gives 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) gives 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) gives -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) gives -50

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 60)
(setq b 13)
(format t "~% BITWISE AND of a and b is ~a" (logand a b))
(format t "~% BITWISE INCLUSIVE OR of a and b is ~a" (logior a b))
(format t "~% BITWISE EXCLUSIVE OR of a and b is ~a" (logxor a b))
(format t "~% A NOT B is ~a" (lognor a b))
(format t "~% A EQUIVALANCE B is ~a" (logeqv a b))
```

```
(terpri)
(terpri)
(setq a 10)
(setq b 0)
(setq c 30)
(setq d 40)
(format t "~% Result of bitwise and operation on 10, 0, 30, 40 is ~a" (logand
a b c d))
(format t "~% Result of bitwise or operation on 10, 0, 30, 40 is ~a" (logior
a b c d))
(format t "~% Result of bitwise xor operation on 10, 0, 30, 40 is ~a" (logxor
a b c d))
(format t "~% Result of bitwise equivalence operation on 10, 0, 30, 40 is ~a"
(logeqv a b c d))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

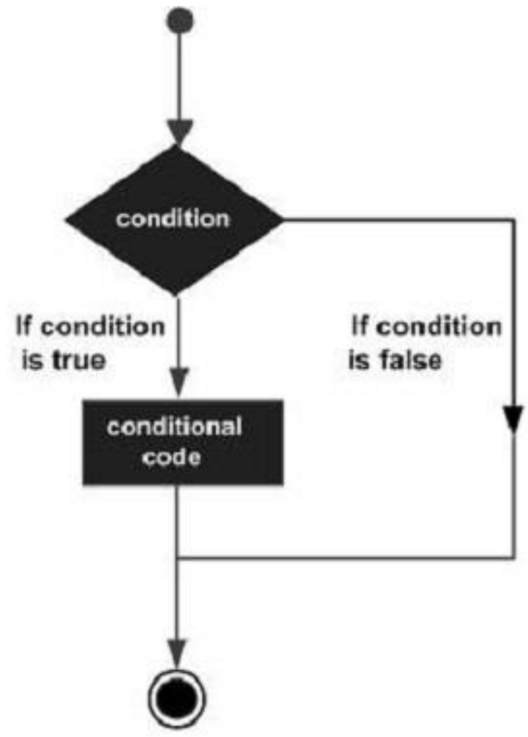
```
BITWISE AND of a and b is 12
BITWISE INCLUSIVE OR of a and b is 61
BITWISE EXCLUSIVE OR of a and b is 49
A NOT B is -62
A EQUIVALANCE B is -50


Result of bitwise and operation on 10, 0, 30, 40 is 0
Result of bitwise or operation on 10, 0, 30, 40 is 62
Result of bitwise xor operation on 10, 0, 30, 40 is 60
Result of bitwise equivalence operation on 10, 0, 30, 40 is -61
```

10. DECISION MAKING

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or 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.

The following diagram shows the general form of a typical decision making structure found in most of the programming languages:



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

Construct	Description
cond	This construct is used for checking multiple test-action clauses. It can be compared to the nested if statements in other programming languages.
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.
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.
case	This construct implements multiple test-action clauses such as cond construct. However, it evaluates a key form and allows multiple action clauses based on the evaluation of that key form.

The *cond* Construct in LISP

The **cond** construct in LISP is most commonly used to permit branching.

Syntax for **cond** is:

```
(cond   (test1    action1)
        (test2    action2)
          ...
         (testn   actionn))
```

Each clause within the **cond** statement consists of a conditional test and an action to be performed.

If the first test following the **cond**, i.e. test$_1$, is evaluated to be true, then the corresponding action part, i.e. action$_1$ is executed. Further, its value is returned and the rest of the clauses are skipped.

If test$_1$ evaluates to be nil, then program control moves to the second clause without executing action$_1$, and the same process is followed.

If none of the test conditions are evaluated to be true, then the **cond** statement returns **nil.**

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 10)
(cond ((> a 20)
    (format t "~% a is less than 20"))
(t (format t "~% value of a is ~d " a)))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
value of a is 10
```

Note that the **t** in the second clause ensures that the last action is performed if none other is fulfilled.

The *if* Construct

The **if** macro is followed by a test clause that evaluates to **t** or nil. If the test clause is evaluated to the **t**, then the action following the test clause is executed. If it is nil, then the next clause is evaluated.

Syntax for if:

(if (test-clause) (<action$_1$) (action$_2$))

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 10)
(if (> a 20)
    (format t "~% a is less than 20"))
(format t "~% value of a is ~d " a)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
value of a is 10
```

Example 2

The **if** clause can be followed by an optional **then** clause:

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 10)
(if (> a 20)
    then (format t "~% a is less than 20"))
(format t "~% value of a is ~d " a)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
a is less than 20
value of a is 10
```

Example 3

You can also create an **if-then-else** type statement using the **if** clause.

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 100)
(if (> a 20)
   (format t "~% a is greater than 20")
    (format t "~% a is less than 20"))
(format t "~% value of a is ~d " a)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
a is greater than 20
value of a is 100
```

The *when* Construct

The **when** macro is followed by a test clause that evaluates to t or nil. If the test clause is evaluated to nil, then no form is evaluated and nil is returned. However, if the test result is **t**, then the action following the test clause is executed.

Syntax for when macro:

(when (test-clause) (<action$_1$))

Example

Create a new source code file named main.lisp and type the following code in it:

```
 (setq a 100)
(when (> a 20)
   (format t "~% a is greater than 20"))
(format t "~% value of a is ~d " a)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
a is greater than 20
value of a is 100
```

The *case* Construct

The **case** construct implements multiple test-action clauses such as the **cond** construct. However, it evaluates a key form and allows multiple action clauses based on the evaluation of that key form.

Syntax for case macro:

The template for CASE is:

```
(case  (keyform)
((key1)   (action1   action2 ...) )
((key2)   (action1   action2 ...) )
...
((keyn)   (action1   action2 ...) ))
```

Example

Create a new source code file named main.lisp and type the following code in it:

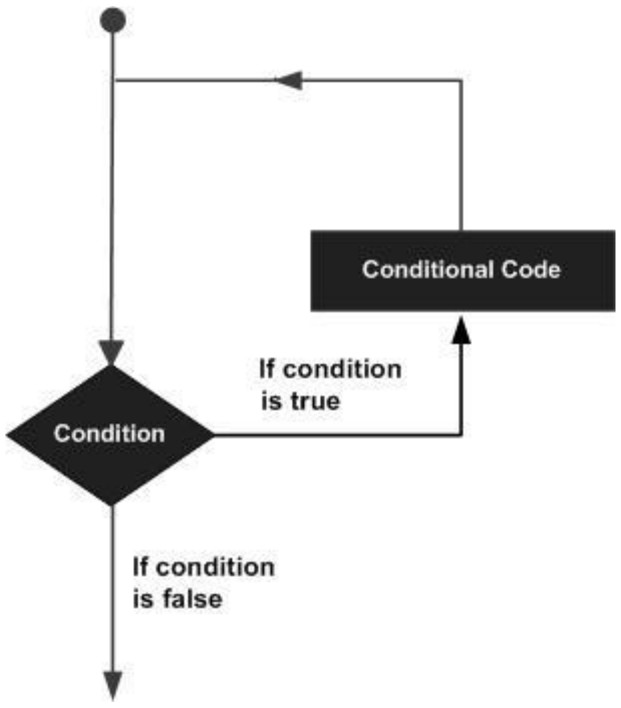
```
(setq day 4)
(case day
(1 (format t "~% Monday"))
(2 (format t "~% Tuesday"))
(3 (format t "~% Wednesday"))
(4 (format t "~% Thursday"))
(5 (format t "~% Friday"))
(6 (format t "~% Saturday"))
(7 (format t "~% Sunday")))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
Thursday
```

11. LOOPS

Consider a situation, where you need to execute a block of code numbers of times. A **loop** statement allows us to execute a statement or group of statements multiple times. The following diagram shows the general form of a **loop** statement in most of the programming languages:



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

Construct	Description
Loop	It 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.
loop for	It allows you to implement a for-loop like iteration as most common in other languages.
do	It is also used for performing iteration using LISP. It provides a structured form of iteration.
dotimes	It allows looping for some fixed number of iterations.

dolist	It allows iteration through each element of a list.

The *loop* Construct

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.

Syntax for loop:

(loop (s-expressions))

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq a 10)
(loop
   (setq a (+ a 1))
   (write a)
   (terpri)
   (when (> a 17) (return a)))
```

When you execute the code, it returns the following result:

```
11
12
13
14
15
16
17
18
```

Note that without the return statement, the loop macro would produce an infinite loop.

The *loop for* Construct

The **loop** for construct allows you to implement a for-loop like iteration as most common in other languages.

It allows you to:

- set up variables for iteration
- specify expression(s) that can conditionally terminate the iteration
- specify expression(s) for performing some job on each iteration
- specify expression(s), and expressions for doing some job before exiting the loop

The **for loop** for construct follows several syntaxes:

```
(loop for loop-variable in <a list>
   do (action))


(loop for loop-variable from value1 to value2
   do (action))
```

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(loop for x in '(tom dick harry)
     do (format t " ~s" x)
)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
TOM DICK HARRY
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(loop for a from 10 to 20
     do (print a)
)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
10
11
12
```

```
13
14
15
16
17
18
19
20
```

Example 3

Create a new source code file named main.lisp and type the following code in it:

```
(loop for x from 1 to 20
  if(evenp x)
  do (print x)
)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
2
4
6
8
10
12
14
16
18
20
```

The *do* Construct

The **do** construct is also used for performing iteration using LISP. It provides a structured form of iteration.

Syntax for do:

```
(do (variable1       value1       updated-value1)
    (variable2       value2      updated-value2)
    (variable3       value3       updated-value3)
    ...
(test     return-value)
(s-expressions))
```

The initial values of each variable is evaluated and bound to the respective variable. The updated value in each clause corresponds to an optional update statement that specifies how the values of the variables will be updated with each iteration.

After each iteration, the test is evaluated.If it returns a non-nil or true, the return-value is evaluated and returned.The last s-expression(s) is optional. If present, they are executed after every iteration, until the test value returns true.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(do ((x 0 (+ 2 x))
    (y 20 ( - y 2)))
    ((= x y)(- x y))
    (format t "~% x = ~d  y = ~d" x y))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
x = 0  y = 20
x = 2  y = 18
x = 4  y = 16
x = 6  y = 14
x = 8  y = 12
```

The *dotimes* Construct

The **dotimes** construct allows looping for some fixed number of iterations.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(dotimes (n 11)
    (print n) (prin1 (* n n)))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
0 0
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
```

The *dolist* Construct

The **dolist** construct allows iteration through each element of a list.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(dolist (n '(1 2 3 4 5 6 7 8 9))
    (format t "~% Number: ~d Square: ~d" n (* n n)))
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
Number: 1 Square: 1
Number: 2 Square: 4
Number: 3 Square: 9
Number: 4 Square: 16
Number: 5 Square: 25
Number: 6 Square: 36
Number: 7 Square: 49
Number: 8 Square: 64
```

```
Number: 9 Square: 81
```

Exiting Gracefully from a Block

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 for block:

```
(block block-name(
...
...
))
```

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

Example

Create a new source code file named main.lisp and type the following code in it:

```
(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)
```

When you click the Execute button, or type Ctrl+E, LISP executes it immediately and the result is:

```
ENTERING-OUTER-BLOCK
ENTERING-INNER-BLOCK


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

12. FUNCTIONS

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

You can divide your code into separate functions. How you divide your code among different functions is up to you, but logically the division is done in such a manner that each function performs a specific task.

Defining Functions in LISP

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

1. Name of the function
2. Parameters of the function
3. Body of the function

Syntax for **defun**:

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

Let us illustrate the concept with simple examples.

Example 1

Let us write a function named *averagenum* to print the average of four numbers. We will send these numbers as parameters.

Create a new source code file named main.lisp and type the following code in it:

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

When you execute the code, it returns the following result:

```
25
```

Example 2

Let us define and call a function that calculates the area of a circle when the radius of the circle is given as an argument.

Create a new source code file named main.lisp and type the following code in it:

```
(defun area-circle(rad)
"Calculates area of a circle with given radius"
  (terpri)
  (format t "Radius: ~5f" rad)
  (format t "~%Area: ~10f" (* 3.141592 rad rad)))
  (area-circle 10)
```

Note:

- You can provide an empty list as parameters, which means the function takes no arguments, the list is empty, written as ().
- LISP also allows optional, multiple, and keyword arguments.
- The documentation string describes the purpose of the function. It is associated with the name of the function and can be obtained using the **documentation** function.
- The body of the function may consist of any number of LISP expressions.
- The value of the last expression in the body is returned as the value of the function.
- You can also return a value from the function using the **return-from** special operator.

Let us discuss the above concepts in brief.

Optional Parameters

You can define a function with optional parameters. To do this, you need to put the symbol **&optional** before the names of the optional parameters.

Let us write a function that displays the parameters it received.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(defun show-members (a b &optional c d) (write (list a b c d)))
(show-members 1 2 3)
```

```
(terpri)
(show-members 'a 'b 'c 'd)
(terpri)
(show-members 'a 'b)
(terpri)
(show-members 1 2 3 4)
```

When you execute the code, it returns the following result:

```
(1 2 3 NIL)
(A B C D)
(A B NIL NIL)
(1 2 3 4)
```

Note that the parameter c and d are the optional parameters in the above example.

Rest Parameters

Some functions need to take a variable number of arguments.

For example, the **format** function we are using needs two arguments, the stream and the control string. However, after the string, it needs a variable number of arguments depending upon the number of values to be displayed in the string. Similarly, the + function, or the * function may also take a variable number of arguments. You can provide for such variable number of parameters using the symbol **&rest**.

The following example illustrates the concept:

Example

Create a new source code file named main.lisp and type the following code in it:

```
(defun show-members (a b &rest values) (write (list a b values)))
(show-members 1 2 3)
(terpri)
(show-members 'a 'b 'c 'd)
(terpri)
(show-members 'a 'b)
(terpri)
(show-members 1 2 3 4)
(terpri)
```

```
(show-members 1 2 3 4 5 6 7 8 9)
```

When you execute the code, it returns the following result:

```
(1 2 (3))
(A B (C D))
(A B NIL)
(1 2 (3 4))
(1 2 (3 4 5 6 7 8 9))
```

Keyword Parameters

Keyword parameters allow you to specify which values go with which particular parameter. They are indicated using **&key** symbol. When you send the values to the function, you must precede the values with **parameter-name.** The following example illustrates the concept:

Example

Create a new source code file named main.lisp and type the following code in it:

```
(defun show-members (&key a b c d ) (write (list a b c d)))
(show-members :a 1 :c 2 :d 3)
(terpri)
(show-members :a 'p :b 'q :c 'r :d 's)
(terpri)
(show-members :a 'p :d 'q)
(terpri)
(show-members :a 1 :b 2)
```

When you execute the code, it returns the following result:

```
(1 NIL 2 3)
(P Q R S)
(P NIL NIL Q)
(1 2 NIL NIL)
```

Returning Values from a Function

By default, a function in LISP returns the value of the last expression evaluated. The following examples demonstrate this:

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(defun add-all(a b c d)
  (+ a b c d))
(setq sum (add-all 10 20 30 40))
(write sum)
(terpri)
(write (add-all 23.4 56.7 34.9 10.0))
```

When you execute the code, it returns the following result:

```
100
125.0
```

However, you can use the **return-from** special operator to immediately return any value from the function.

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(defun myfunc (num)
    (return-from myfunc 10)
    num)
(write (myfunc 20))
```

When you execute the code, it returns the following result:

```
10
```

Change the code a little:

```
(defun myfunc (num)
    (return-from myfunc 10)
    write num)
(write (myfunc 20))
```

It still returns:

```
10
```

Lambda Functions

At times you may need a function in only one place in your program and the function is so trivial that you may not give it a name, or may not like to store it in the symbol table, and would rather write an unnamed or anonymous function.

LISP allows you to write anonymous functions that are evaluated only when they are encountered in the program. These functions are called **Lambda functions.**

You can create such functions using the **lambda** expression. The syntax for the lambda expression is as follows:

```
(lambda (parameters) body)
```

A lambda form cannot be evaluated and it must appear only where LISP expects to find a function.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write ((lambda (a b c x)
      (+ (* a (* x x)) (* b x) c))
   4 2 9 3))
```

When you execute the code, it returns the following result:

```
51
```

Mapping Functions

Mapping functions are a group of functions that could be applied successively to one or more lists of elements. The results of applying these functions to a list are placed in a new list and that new list is returned.

For example, the **mapcar** function processes successive elements of one or more lists.

The first argument of the mapcar function is a function and the remaining arguments are the list(s), to which the function is applied. The argument function is applied to the successive elements and that results into a newly constructed list. If the argument lists are not equal in length, then the process of mapping stops upon reaching the end of the shortest list. The resulting list will have the same number of elements as the shortest input list.

Example 1

Let us start with a simple example. Add the number 1 to each element of the list (23, 34, 45, 56, 67, 78, 89).

Create a new source code file named main.lisp and type the following code in it:

```
(write (mapcar '1+  '(23 34 45 56 67 78 89)))
```

When you execute the code, it returns the following result:

```
(24 35 46 57 68 79 90)
```

Example 2

Let us write a function that returns cube of the elements of a list. Let us use a lambda function for calculating the cube of numbers.

Create a new source code file named main.lisp and type the following code in it:

```
(defun cubeMylist(lst)
 (mapcar #'(lambda(x) (* x x x)) lst))
  (write (cubeMylist '(2 3 4 5 6 7 8 9)))
```

When you execute the code, it returns the following result:

```
(8 27 64 125 216 343 512 729)
```

Example 3

Create a new source code file named main.lisp and type the following code in it:

```
(write (mapcar '+ '(1 3 5 7 9 11 13) '( 2 4 6 8)))
```

When you execute the code, it returns the following result:

```
(3 7 11 15)
```

13. 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 if the condition is true.

The following table shows some of the most commonly used predicates:

Predicate	Description
atom	It takes one argument and returns t if the argument is an atom, nil 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 both arguments are equal, or they are numbers of the same type with the same value, or 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 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 arguments 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 arguments and returns t if 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 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.**

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(write (atom 'abcd))
(terpri)
(write (equal 'a 'b))
(terpri)
(write (evenp 10))
(terpri)
(write (evenp 7 ))
(terpri)
(write (oddp 7 ))
(terpri)
(write (zerop 0.0000000001))
(terpri)
(write (eq 3 3.0 ))
(terpri)
(write (equal 3 3.0 ))
(terpri)
(write (null nil ))
```

When you execute the code, it returns the following result:

```
T
NIL
T
NIL
T
NIL
NIL
NIL
T
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(defun factorial (num)
   (cond ((zerop num) 1)
       (t ( * num (factorial (- num 1))))))
(setq n 6)
(format t "~% Factorial ~d is: ~d" n (factorial n))
```

When you execute the code, it returns the following result:

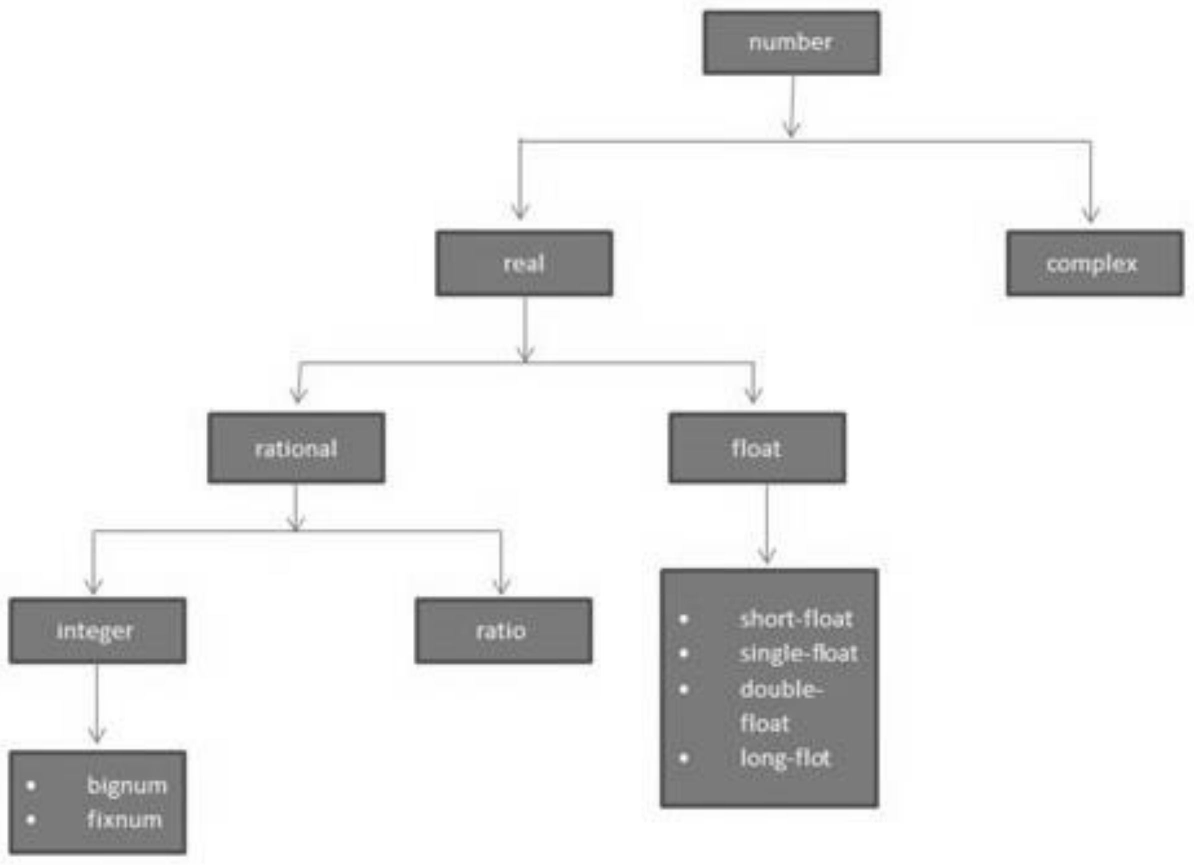
```
Factorial 6 is: 720
```

14. NUMBERS

Common LISP defines several kinds of numbers. The **number** data type includes various kinds of numbers supported by LISP. The following numeric data types are supported by LISP:

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

The following diagram shows the number hierarchy and various numeric data types available in LISP:



Various Numeric Types in LISP

The following table describes various number type data available in LISP:

Data type	Description
fixnum	This data type represents integers which are not too large. Their values range from -215 to 215-1. (it is machine-dependent)
bignum	These are very large numbers with size limited by the amount of memory allocated for LISP, they are not fixnum numbers.
ratio	This represents the ratio of two numbers in the numerator/denominator form. The / function always produces the result in ratios, with integer arguments.
float	It represents non-integer numbers. There are four float data types with increasing precision.
complex	It represents complex numbers, which are denoted by #c. The real and imaginary parts could be both either rational or floating point numbers.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (/ 1 2))
(terpri)
(write ( + (/ 1 2) (/ 3 4)))
(terpri)
(write ( + #c( 1 2) #c( 3 -4)))
```

When you execute the code, it returns the following result:

```
1/2
5/4
#C(4 -2)
```

Number Functions

The following table describes some commonly used numeric functions:

Function	Description
+, -, *, /	Respective arithmetic operations
sin, cos, tan, acos, asin, atan	Respective trigonometric functions.
sinh, cosh, tanh, acosh, asinh, atanh	Respective hyperbolic functions.
exp	Exponentiation function. Calculates e^x
expt	Exponentiation function, takes base and power both.
sqrt	It calculates the square root of a number.
log	Logarithmic function. It one parameter is given, then it calculates its natural logarithm, otherwise the second parameter is used as base.
conjugate	It calculates the complex conjugate of a number. In case of a real number, it returns the number itself.
abs	It returns the absolute value (or magnitude) of a number.
gcd	It calculates the greatest common divisor of the given numbers
lcm	It calculates the least common multiple of the given numbers
isqrt	It gives the greatest integer less than or equal to the exact square root of a given natural number.
floor, ceiling, truncate, round	All these functions take two arguments as a number and returns the quotient; floorreturns the largest integer that is not greater than ratio, ceiling chooses the smaller integer that is larger than ratio, truncate chooses the integer of the same sign as ratio with the largest absolute value that is less than absolute value of ratio, and roundchooses an integer that is closest to ratio.

ffloor, fceiling, ftruncate, fround	Does the same as above, but returns the quotient as a floating point number.
mod, rem	Returns the remainder in a division operation.
float	Converts a real number to a floating point number.
rational, rationalize	Converts a real number to rational number.
numerator, denominator	Returns the respective parts of a rational number.
realpart, imagpart	Returns the real and imaginary part of a complex number.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(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))
```

When you execute the code, it returns the following result:

```
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
```

15. CHARACTERS

In LISP, characters are represented as data objects of type **character**. You can denote a character object preceding #\ before the character itself. For example, #\a means the character a.

Space and other special characters can be denoted by preceding #\ before the name of the character. For example, #\SPACE represents the space character.

The following example demonstrates this:

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write 'a)
(terpri)
(write #\a)
(terpri)
(write-char #\a)
(terpri)
(write-char 'a)
```

When you execute the code, it returns the following result:

```
A
#\a
a
*** - WRITE-CHAR: argument A is not a character
```

Special Characters

Common LISP allows using the following special characters in your code. They are called the semi-standard characters.

- #\Backspace
- #\Tab
- #\Linefeed

- #\Page
- #\Return
- #\Rubout

Character Comparison Functions

Numeric comparison functions and operators, like, < and > do not work on characters. Common LISP provides other two sets of functions for comparing characters in your code. One set is case-sensitive and the other case-insensitive.

The following table provides the functions:

Case-Sensitive Functions	Case-Insensitive Functions	Description
char=	char-equal	Checks if the values of the operands are all equal or not, if yes then condition becomes true.
char/=	char-not-equal	Checks if the values of the operands are all different or not, if values are not equal, then condition becomes true.
char<	char-lessp	Checks if the values of the operands are monotonically decreasing.
char>	char-greaterp	Checks if the values of the operands are monotonically increasing.
char<=	char-not-greaterp	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.
char>=	char-not-lessp	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.

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
; 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))
```

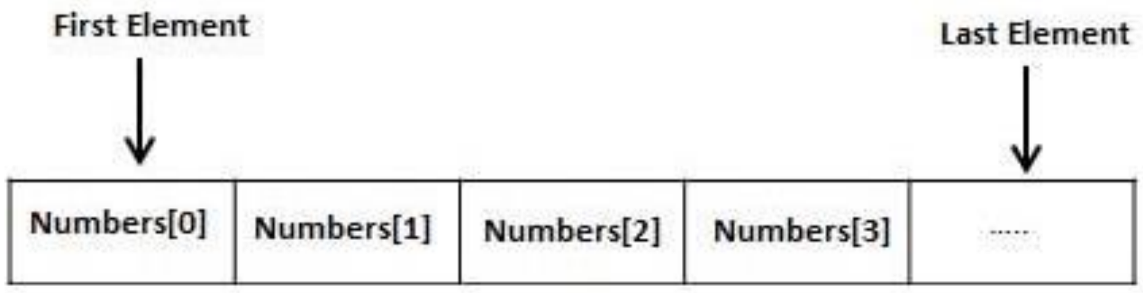
When you execute the code, it returns the following result:

```
NIL
T
NIL
T
NIL
T
NIL
```

16. ARRAYS

LISP allows you to define single or multiple-dimension arrays using the **make-array** function. An array can store any LISP object as its elements.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address corresponds to the last element.



The number of dimensions of an array is called its **rank**.

In LISP, an array element is specified by a sequence of non-negative integer indices. The length of the sequence must equal the rank of the array. Indexing always starts from zero.

Example

To create an array with 10- cells, named my-array, we can write:

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

The **aref** function allows accessing the contents of the cells. It takes two arguments, the name of the array and the index value.

Example

To access the content of the tenth cell, we write:

```
(aref my-array 9)
```

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(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)
```

When you execute the code, it returns the following result:

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

Example 2

Let us create a 3-by-3 array.

Create a new source code file named *main.lisp* and type the following code in it:

```
(setf x (make-array '(3 3)
              :initial-contents '((0 1 2 ) (3 4 5) (6 7 8))))
(write x)
```

When you execute the code, it returns the following result:

```
#2A((0 1 2) (3 4 5) (6 7 8))
```

Example 3

Create a new source code file named main.lisp and type the following code in it:

```
(setq a (make-array '(4 3)))
(dotimes (i 4)
   (dotimes (j 3)
     (setf (aref a i j) (list i 'x j '= (* i j)))))
(dotimes (i 4)
   (dotimes (j 3)
     (print (aref a i j))))
```

When you execute the code, it returns the following result:

```
(0 X 0 = 0)
(0 X 1 = 0)
(0 X 2 = 0)
(1 X 0 = 0)
(1 X 1 = 1)
(1 X 2 = 2)
(2 X 0 = 0)
(2 X 1 = 2)
(2 X 2 = 4)
(3 X 0 = 0)
(3 X 1 = 3)
(3 X 2 = 6)
```

Syntax for the make-array Function

The make-array function takes many other arguments. Let us look at the complete syntax of this function:

```
make-array  dimensions  :element-type  :initial-element  :initial-contents
:adjustable :fill-pointer  :displaced-to :displaced-index-offset
```

Apart from the *dimensions* argument, all other arguments are keywords. The following table provides brief description of the arguments.

Argument	Description
dimensions	It gives the dimensions of the array. It is a number for one-dimensional array, and a list for multi-dimensional array.
:element-type	It is the type specifier, default value is T, i.e. any type
:initial-element	Initial elements value. It makes an array with all the elements initialized to a particular value.
:initial-content	Initial content as object.

:adjustable	It helps in creating a resizable (or adjustable) vector whose underlying memory can be resized. The argument is a Boolean value indicating whether the array is adjustable or not, default value being NIL.
:fill-pointer	It keeps track of the number of elements actually stored in a resizable vector
:displaced-to	It helps in creating a displaced array or shared array that shares its contents with the specified array. Both the arrays should have same element type. The :displaced-to option may not be used with the :initial-element or :initial-contents option. This argument defaults to **nil**.
:displaced-index-offset	It gives the index-offset of the created and shared array.

Example 4

Create a new source code file named main.lisp and type the following code in it:

```
(setq myarray (make-array '(3 2 3)
            :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              ))) 
(setq array2 (make-array 4 :displaced-to myarray
                      :displaced-index-offset 2)) 
(write myarray)
(terpri)
(write array2)
```

When you execute the code, it returns the following result:

```
#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9)))
#(C 1 2 3)
```

If the displaced array is two dimensional:

```
(setq myarray (make-array '(3 2 3)
            :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              )))
(setq array2 (make-array '(3 2) :displaced-to myarray
                      :displaced-index-offset 2))
(write myarray)
(terpri)
(write array2)
```

When you execute the code, it returns the following result:

```
#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9)))
#2A((C 1) (2 3) (D E))
```

Let us change the displaced index offset to 5:

```
(setq myarray (make-array '(3 2 3)
            :initial-contents
            '(((a b c) (1 2 3))
              ((d e f) (4 5 6))
              ((g h i) (7 8 9))
              )))
(setq array2 (make-array '(3 2) :displaced-to myarray
                      :displaced-index-offset 5))
(write myarray)
(terpri)
(write array2)
```

When you execute the code, it returns the following result:

```
#3A(((A B C) (1 2 3)) ((D E F) (4 5 6)) ((G H I) (7 8 9)))
#2A((3 D) (E F) (4 5))
```

Example 5

Create a new source code file named main.lisp and type the following code in it:

```
;a one dimensional array with 5 elements,
;initail value 5
(write (make-array 5 :initial-element 5))
(terpri)
;two dimensional array, with initial element a
(write (make-array '(2 3) :initial-element 'a))
(terpri)
;an array of capacity 14, but fill pointer 5, is 5
(write(length (make-array 14 :fill-pointer 5)))
(terpri)
;however its length is 14
(write (array-dimensions (make-array 14 :fill-pointer 5)))
(terpri)
; a bit array with all initial elements set to 1
(write(make-array 10 :element-type 'bit :initial-element 1))
(terpri)
; a character array with all initial elements set to a
; is a string actually
(write(make-array 10 :element-type 'character :initial-element #\a))
(terpri)
; a two dimensional array with initial values a
(setq myarray (make-array '(2 2) :initial-element 'a :adjustable t))
(write myarray)
(terpri)
;readjusting the array
(adjust-array myarray '(1 3) :initial-element 'b)
(write myarray)
```

When you execute the code, it returns the following result:

```
#(5 5 5 5 5)
#2A((A A A) (A A A))
5
(14)
#*1111111111
"aaaaaaaaaa"
```

```
#2A((A A) (A A))
#2A((A A B))
```

17. 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 (\). However, you can include these by escaping them with a backslash (\).

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write-line "Hello World")
(write-line "Welcome to Tutorials Point")
;escaping the double quote character
(write-line "Welcome to \"Tutorials Point\"")
```

When you execute the code, it returns the following result:

```
Hello World
Welcome to Tutorials Point
Welcome to "Tutorials Point"
```

String Comparison Functions

Numeric comparison functions and operators, like, < and > do not work on strings. Common LISP provides other two sets of functions for comparing strings in your code. One set is case-sensitive and the other case-insensitive.

The following table provides the functions:

Case-Sensitive Functions	Case-insensitive Functions	Description
string=	string-equal	Checks if the values of the operands are all equal or not, if yes then condition becomes true.

string/=	string-not-equal	Checks if the values of the operands are all different or not, if values are not equal then condition becomes true.
string<	string-lessp	Checks if the values of the operands are monotonically decreasing.
string>	string-greaterp	Checks if the values of the operands are monotonically increasing.
string<=	string-not-greaterp	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.
string>=	string-not-lessp	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.

Example

Create a new source code file named main.lisp and type the following code in it:

```
; 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"))
```

When you execute the code, it returns the following result:

```
NIL
0
NIL
T
NIL
NIL
8
NIL
4
5
```

Case Controlling Functions

The following table describes the case controlling functions:

Function	**Description**
string-upcase	Converts the string to upper case
string-downcase	Converts the string to lower case
string-capitalize	Capitalizes each character in the string

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write-line (string-upcase "a big hello from tutorials point"))
(write-line (string-capitalize "a big hello from tutorials point"))
```

When you execute the code, it returns the following result:

```
A BIG HELLO FROM TUTORIALS POINT
A Big Hello From Tutorials Point
```

Trimming Strings

The following table describes the string trimming functions:

Function	Description
string-trim	It takes a string of character(s) as first argument and a string as the second argument and returns a substring where all characters that are in the first argument are removed off the argument string.
String-left-trim	It takes a string of character(s) as first argument and a string as the second argument and returns a substring where all characters that are in the first argument are removed off the beginning of the argument string.
String-right-trim	It takes a string character(s) as first argument and a string as the second argument and returns a substring where all characters that are in the first argument are removed off the end of the argument string

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write-line (string-trim " " "   a big hello from tutorials point   "))
(write-line (string-left-trim " " "   a big hello from tutorials point   "))
(write-line (string-right-trim " " "   a big hello from tutorials point   "))
(write-line (string-trim " a" "   a big hello from tutorials point   "))
```

When you execute the code, it returns the following result:

```
a big hello from tutorials point
a big hello from tutorials point
   a big hello from tutorials point
big hello from tutorials point
```

Other String Functions

Strings in LISP are arrays and thus also sequences. We will cover these data types in coming tutorials. All functions that are applicable to arrays and sequences also apply to strings. However, here we demonstrate some commonly used functions using various examples.

Calculating Length of String

The **length** function calculates the length of a string.

Extracting Sub-string

The **subseq** function returns a sub-string (as a string is also a sequence) starting at a particular index and continuing up to a particular ending index or the end of the string.

Accessing a Character in a String

The **char** function allows accessing individual characters of a string.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (length "Hello World"))
(terpri)
(write-line (subseq "Hello World" 6))
(write (char "Hello World" 6))
```

When you execute the code, it returns the following result:

```
11
World
#\W
```

Sorting and Merging strings

The **sort** function allows sorting a string. It takes a sequence (vector or string) and a two-argument predicate and returns a sorted version of the sequence.

The **merge** function takes two sequences and a predicate and returns a sequence produced by merging the two sequences, according to the predicate.

Example

Create a new source code file named main.lisp and type the following code in it:

```
;sorting the strings
(write (sort (vector "Amal" "Akbar" "Anthony") #'string<))
(terpri)
;merging the strings
(write (merge 'vector (vector "Rishi" "Zara" "Priyanka") (vector "Anju" "Anuj"
"Avni") #'string<))
```

When you execute the code, it returns the following result:

```
#("Akbar" "Amal" "Anthony")
#("Anju" "Anuj" "Avni" "Rishi" "Zara" "Priyanka")
```

Reversing a String

The **reverse** function reverses a string.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write-line (reverse "Are we not drawn onward, we few, drawn onward to new
era"))
```

When you execute the code, it returns the following result:

```
are wen ot drawno nward ,wef ew ,drawno nward ton ew erA
```

Concatenating Strings

The **concatenate** function concatenates two strings. This is generic sequence function and you must provide the result type as the first argument.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write-line (concatenate 'string "Are we not drawn onward, " "we few, drawn
onward to new era"))
```

When you execute the code, it returns the following result:

```
Are we not drawn onward, we few, drawn onward to new era
```

18. 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.

In this chapter, we will discuss most commonly used functions on sequences.

Before starting on various ways of manipulating sequences i.e., vectors and lists, let us have a look at the list of all available functions.

Creating a Sequence

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*.

You may optionally specify some value using the *:initial-element* argument, then each of the elements is initialized to this value.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (make-sequence '(vector float)

               10

               :initial-element 1.0))
```

When you execute the code, it returns the following result:

```
#(1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0)
```

Generic Functions on Sequences

Function	Description
elt	It allows access to individual elements through an integer index.

length	It returns the length of a sequence.
subseq	It returns a sub-sequence by extracting the subsequence starting at a particular index and continuing to a particular ending index or the end of the sequence...
copy-seq	It returns a sequence that contains the same elements as its argument.
fill	It is used to set multiple elements of a sequence to a single value.
replace	It takes two sequences and the first argument sequence is destructively modified by copying successive elements into it from the second argument sequence.
count	It takes an item and a sequence and returns the number of times the item appears in the sequence.
reverse	It returns a sequence containing the same elements of the argument but in reverse order.
nreverse	It returns the same sequence containing the same elements as sequence but in reverse order.
concatenate	It creates a new sequence containing the concatenation of any number of sequences.
position	It takes an item and a sequence and returns the index of the item in the sequence or nil.
find	It takes an item and a sequence. It finds the item in the sequence and returns it. If the item is not found, then it returns nil.
sort	It takes a sequence and a two-argument predicate, and returns a sorted version of the sequence.
merge	It takes two sequences and a predicate, and returns a sequence produced by merging the two sequences according to the predicate.
map	It takes an n-argument function and nsequences and returns a new sequence containing the result of applying the function to subsequent elements of the sequences.

some	It takes a predicate as an argument and iterates over the argument sequence, and returns the first non-NIL value returned by the predicate or returns false if the predicate is never satisfied.
every	It takes a predicate as an argument and iterates over the argument sequence. It terminates by returning false, as soon as the predicate fails. If the predicate is always satisfied, it returns true.
notany	It takes a predicate as an argument and iterates over the argument sequence. It returns false as soon as the predicate is satisfied or true if it is not satisfied.
notevery	It takes a predicate as an argument and iterates over the argument sequence, and returns true as soon as the predicate fails or false if the predicate is always satisfied.
reduce	It maps over a single sequence, applying a two-argument function first to the first two elements of the sequence and then to the value returned by the function and subsequent elements of the sequence.
search	It searches a sequence to locate one or more elements satisfying some test.
remove	It takes an item and a sequence and returns the sequence with instances of item removed.
delete	This also takes an item and a sequence and returns a sequence of the same kind as the argument sequence that has the same elements except the item.
substitute	It takes a new item, an existing item, and a sequence and returns a sequence with instances of the existing item replaced with the new item.
nsubstitute	It takes a new item, an existing item, and a sequence and returns the same sequence with instances of the existing item replaced with the new item.
mismatch	It takes two sequences and returns the index of the first pair of mismatched elements.

Standard Sequence Function Keyword Arguments

Argument	Meaning	Default Value
:test	It is a two-argument function used to compare item or value extracted by :key function to element.	EQL
:key	It is a one-argument function to extract key value from actual sequence element. NIL means use element as it is.	NIL
:start	Starting index (inclusive) of subsequence.	0
:end	Ending index (exclusive) of subsequence. NIL indicates end of sequence.	NIL
:from-end	If true, the sequence will be traversed in reverse order, from end to start.	NIL
:count	Number indicates the number of elements to remove or substitute. NIL indicates all elements to remove or substitute.	NIL

We have just discussed various functions and keywords that are used as arguments in the functions working on sequences. In the next chapters, we will see how to use these functions, with examples.

Finding Length and Element

The **length** function returns the length of a sequence, and the **elt** function allows you to access individual elements using an integer index.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq x (vector 'a 'b 'c 'd 'e))
(write (length x))
(terpri)
(write (elt x 3))
```

When you execute the code, it returns the following result:

```
5
D
```

Modifying Sequences

Some sequence functions allows iterating through the sequence and perform some operations like, searching, removing, counting or filtering specific elements without writing explicit loops.

The following example demonstrates this:

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(write (count 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (substitute 10 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (find 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (position 5 '(1 5 6 7 8 9 2 7 3 4 5)))
```

When you execute the code, it returns the following result:

```
2
(1 6 7 8 9 2 7 3 4)
(1 6 7 8 9 2 7 3 4)
(1 5 6 10 8 9 2 10 3 4 5)
7
1
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(write (delete-if #'oddp '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete-if #'evenp '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove-if #'evenp '(1 5 6 7 8 9 2 7 3 4 5) :count 1 :from-end t))
(terpri)
(setq x (vector 'a 'b 'c 'd 'e 'f 'g))
(fill x 'p :start 1 :end 4)
(write x)
```

When you execute the code, it returns the following result:

```
(6 8 2 4)
(1 5 7 9 7 3 5)
(1 5 6 7 8 9 2 7 3 5)
#(A P P P E F G)
```

Sorting and Merging Sequences

The **sort** function takes a sequence and a two-argument predicate, and it returns a sorted version of the sequence.

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(write (sort '(2 4 7 3 9 1 5 4 6 3 8) #'<))
(terpri)
(write (sort '(2 4 7 3 9 1 5 4 6 3 8) #'>))
(terpri)
```

When you execute the code, it returns the following result:

```
(1 2 3 3 4 4 5 6 7 8 9)
(9 8 7 6 5 4 4 3 3 2 1)
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(write (merge 'vector #(1 3 5) #(2 4 6) #'<))
(terpri)
(write (merge 'list #(1 3 5) #(2 4 6) #'<))
(terpri)
```

When you execute the code, it returns the following result:

```
#(1 2 3 4 5 6)
(1 2 3 4 5 6)
```

Sequence Predicates

The functions **every, some, notany,** and **notevery** are called the sequence predicates.

These functions iterate over sequences and test the boolean predicate. All these functions take a predicate as the first argument and the remaining arguments are sequences.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (every #'evenp #(2 4 6 8 10)))
(terpri)
(write (some #'evenp #(2 4 6 8 10 13 14)))
(terpri)
(write (every #'evenp #(2 4 6 8 10 13 14)))
(terpri)
(write (notany #'evenp #(2 4 6 8 10)))
(terpri)
(write (notevery #'evenp #(2 4 6 8 10 13 14)))
(terpri)
```

When you execute the code, it returns the following result:

```
T
T
NIL
```

```
NIL
T
```

Mapping Sequences

The **map** function allows you to apply a function on to subsequent elements of one or more sequences.

The **map** function takes n-arguments function and n sequences and returns a new sequence after applying the function to subsequent elements of the sequences.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (map 'vector #'* #(2 3 4 5) #(3 5 4 8)))
```

When you execute the code, it returns the following result:

```
#(6 15 16 40)
```

19. LISTS

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

Lists are single linked lists. In LISP, lists are constructed as a chain of a simple record structure named **cons** linked together.

The Cons Record Structure

Cons is a record structure containing two components called the **car** and the **cdr. The cons** cells or **cons** are objects composed of pairs of values that are created using the function **cons.**

The **cons** function takes two arguments and returns a new **cons** cell containing two values. These values can be references to any kind of object. If the second value is not nil, or it is another cons cell, then the values are printed as a dotted pair enclosed by parentheses.

The two values in a **cons** cell are called the **car** and the **cdr.**The **car** function is used to access the first value and the **cdr** function is used to access the second value.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(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)))))
```

When you execute the code, it returns the following result:

```
(1 . 2)
(A . B)
(1)
(1 2)
(1 2 3)
(A B C)
A
(B C)
```

The above example shows how you can use **cons** structures to create a single linked list. Here, the list (A B C) consists of three **cons** cells linked together by their **cdr**s*.*

Diagrammatically, it could be expressed as:

Creating Lists with list Function in LISP

Although cons cells can be used to create lists, however, constructing a list out of nested **cons** function calls cannot be the best solution. The **list** function is rather used for creating lists in LISP.

The **list** function can take any number of arguments and as it is a function, it evaluates its arguments.

The **first** and **rest** functions give the first element and the rest part of a list. The following examples demonstrate the concepts.

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(write (list 1 2))
(terpri)
(write (list 'a 'b))
(terpri)
(write (list 1 nil))
(terpri)
(write (list 1 2 3))
```

```
(terpri)
(write (list 'a 'b 'c))
(terpri)
(write (list 3 4 'a (car '(b . c)) (* 4 -2)))
(terpri)
(write (list (list 'a 'b) (list 'c 'd 'e)))
```

When you execute the code, it returns the following result:

```
(1 2)
(A B)
(1 NIL)
(1 2 3)
(A B C)
(3 4 A B -8)
((A B) (C D E))
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(defun my-library (title author rating availability)
  (list :title title :author author :rating rating :availabilty availability))
(write (getf (my-library "Hunger Game" "Collins" 9 t) :title))
```

When you execute the code, it returns the following result:

```
"Hunger Game"
```

List Manipulating Functions

The following table provides some commonly used list manipulating functions.

Function	Description
car	It takes a list as argument, and returns its first element.
cdr	It takes a list as argument, and returns a list without the first element.

cons	It takes two arguments, an element and a list, and returns a list with the element inserted at the first place.
list	It takes any number of arguments and returns a list with the arguments as member elements of the list.
append	It merges two or more lists into one.
last	It takes a list and returns a list containing the last element.
member	It takes two arguments of which the second must be a list, if the first argument is a member of the second argument, and then it returns the remainder of the list beginning with the first argument.
reverse	It takes a list and returns a list with the top elements in reverse order.

Note that all sequence functions are applicable to lists.

Example 3

Create a new source code file named main.lisp and type the following code in it:

```
(write (car '(a b c d e f)))
(terpri)
(write (cdr '(a b c d e f)))
(terpri)
(write (cons 'a '(b c)))
(terpri)
(write (list 'a '(b c) '(e f)))
(terpri)
(write (append '(b c) '(e f) '(p q) '() '(g)))
(terpri)
(write (last '(a b c d (e f))))
(terpri)
(write (reverse '(a b c d (e f))))
```

When you execute the code, it returns the following result:

```
A
(B C D E F)
(A B C)
(A (B C) (E F))
(B C E F P Q G)
((E F))
((E F) D C B A)
```

Concatenation of *car* and *cdr* Functions

The **car** and **cdr** functions and their combination allows extracting any particular element/member of a list. However, sequences of car and cdr functions could be abbreviated by concatenating the letter a for car and d for cdr within the letters c and r. For example, we can write cadadr to abbreviate the sequence of function calls - car cdr car cdr.

Thus, (cadadr '(a (c d) (e f g))) returns d

Example 4

Create a new source code file named main.lisp and type the following code in it:

```
(write (cadadr '(a (c d) (e f g))))
(terpri)
(write (caar (list (list 'a 'b) 'c)))
(terpri)
(write (cadr (list (list 1 2) (list 3 4))))
(terpri)
```

When you execute the code, it returns the following result:

```
D
A
(3 4)
```

20. SYMBOLS

In LISP, a symbol is a name that represents data objects and interestingly, the name itself is a data object.

What makes symbols special is that they have a component called the **property list,** or **plist.**

Property Lists

LISP allows you to assign properties to symbols. For example, let us have a 'person' object. We would like this 'person' object to have properties like name, gender, height, weight, address, profession etc. A property is nothing but an attribute name.

A property list is implemented as a list with an even number (possibly zero) of elements. Each pair of elements in the list constitutes an entry, in which the first item is the **indicator**, and the second is the **value**. When a symbol is created, its property list is initially empty. Properties are created by using **get** within a **setf** form.

For example, the following statements allow us to assign properties title, author, publisher, and their respective values, to an object (symbol) named 'book'.

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
((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)))
```

When you execute the code, it returns the following result:

```
(GONE WITH THE WIND)
(MARGARET MICHEL)
(WARNER BOOKS)
```

Various property list functions allow you to assign properties as well as retrieve, replace or remove the properties of a symbol.

The **get** function returns the property list of symbol for a given indicator. It has the following syntax:

```
get symbol indicator &optional default
```

The **get** function looks for the property list of the given symbol for the specified indicator. If the symbol is found, then it returns the corresponding value. Otherwise, it returns specified default value. If default value is not specified, it returns nil.

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(setf (get 'books 'title) '(Gone with the Wind))
(setf (get 'books 'author) '(Margaret Micheal))
(setf (get 'books 'publisher) '(Warner Books))
(write (get 'books 'title))
(terpri)
(write (get 'books 'author))
(terpri)
(write (get 'books 'publisher))
```

When you execute the code, it returns the following result:

```
(GONE WITH THE WIND)
(MARGARET MICHEAL)
(WARNER BOOKS)
```

The **symbol-plist** function allows you to see all the properties of a symbol.

Example 3

Create a new source code file named main.lisp and type the following code in it:

```
(setf (get 'annie 'age) 43)
(setf (get 'annie 'job) 'accountant)
(setf (get 'annie 'sex) 'female)
(setf (get 'annie 'children) 3)
(terpri)
(write (symbol-plist 'annie))
```

When you execute the code, it returns the following result:

```
(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT AGE 43)
```

The **remprop** function removes the specified property from a symbol.

Example 4

Create a new source code file named main.lisp and type the following code in it:

```
(setf (get 'annie 'age) 43)
(setf (get 'annie 'job) 'accountant)
(setf (get 'annie 'sex) 'female)
(setf (get 'annie 'children) 3)
(terpri)
(write (symbol-plist 'annie))
(remprop 'annie 'age)
(terpri)
(write (symbol-plist 'annie))
```

When you execute the code, it returns the following result:

```
(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT AGE 43)
(CHILDREN 3 SEX FEMALE JOB ACCOUNTANT)
```

21. 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 work on vectors too.

Creating Vectors

The vector function allows you to make fixed-size vectors with specific values. It takes any number of arguments and returns a vector containing those arguments.

Let us see the example:

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(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)
```

When you execute the code, it returns the following result:

```
#(1 2 3 4 5)
#(A B C D E)
#(P Q R S T)
```

Note that LISP uses the #(...) syntax as the literal notation for vectors. You can use this #(...) syntax to create and include literal vectors in your code.

However, these are literal vectors. Hence, modifying them is not defined in LISP. Therefore,while programming, you should always use the **vector** function, or the more general function **make-array;** to create vectors you plan to modify.

The **make-array** function is more generic way to create a vector. You can access the vector elements using the **aref** function.

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(setq a (make-array 5 :initial-element 0))
(setq b (make-array 5 :initial-element 2))
(dotimes (i 5)
  (setf (aref a i) i))
(write a)
(terpri)
(write b)
(terpri)
```

When you execute the code, it returns the following result:

```
#(0 1 2 3 4)
#(2 2 2 2 2)
```

Fill Pointer Argument

The **make-array** function allows you to create a resizable vector. The **fill-pointer** argument of the function keeps track of the number of elements actually stored in the vector. It is the index of the next position to be filled, when you add an element to the vector.

The **vector-push** function allows you to add an element to the end of a resizable vector. It increases the fill-pointer by 1.

The **vector-pop** function returns the most recently pushed item and decrements the fill pointer by 1.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq a (make-array 5 :fill-pointer 0))
(write a)
(vector-push 'a a)
(vector-push 'b a)
(vector-push 'c a)
(terpri)
(write a)
```

```
(terpri)
(vector-push 'd a)
(vector-push 'e a)
;this will not be entered as the vector limit is 5
(vector-push 'f a)
(write a)
(terpri)
(vector-pop a)
(vector-pop a)
(vector-pop a)
(write a)
```

When you execute the code, it returns the following result:

```
#()
#(A B C)
#(A B C D E)
#(A B)
```

Vectors being sequences, all sequence functions are applicable for vectors. Please refer the sequences chapter for vector functions.

22. SET

Common LISP does not provide a set data type. However, it provides number of functions that allow set operations 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.

Implementing Sets in LISP

Sets, like lists, are generally implemented in terms of **cons** cells. However, for this very reason, the bigger the set becomes, the lesser efficient the set operations becomes. You will understand this once we delve into the matter little deeper.The **adjoin** function allows you to build up a set. It takes an item and a list representing a set and returns a list representing the set containing the item and all the items in the original set.

The **adjoin** function first looks for the item in the given list. If it is found, then it returns the original list; otherwise it creates a new **cons** cell with its ***car*** as the item and ***cdr*** pointing to the original list, and returns this new list.

The **adjoin** function also takes **:key** and **:test** keyword arguments. These arguments are used for checking whether the item is present in the original list.

Since the **adjoin** function does not modify the original list, to make a change in the list itself; you must either assign the value returned by **adjoin** to the original list, or you may use the macro **pushnew** to add an item to the set.

Example

Create a new source code file named main.lisp and type the following code in it:

```
; creating myset as an empty list
(defparameter *myset* ())
(adjoin 1 *myset*)
(adjoin 2 *myset*)
; adjoin didn't 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)
```

When you execute the code, it returns the following result:

```
NIL
(2 1)
(2 1)
(3 2 1)
```

Checking Membership

The **member** group of functions allows you to check whether an element is member of a set or not.

The following syntaxes are used to check membership:

```
member item list &key :test :test-not :key
member-if predicate list &key :key
member-if-not predicate list &key :key
```

These functions search the given list for a given item that satisfies the test. If no such item is found, then the functions return **nil.** Otherwise, the tail of the list with the element as the first element is returned. The search is conducted at the top level only.

These functions can be used as predicates.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(write (member 'zara '(ayan abdul zara riyan nuha)))
(terpri)
(write (member-if #'evenp '(3 7 2 5/3 'a)))
(terpri)
(write (member-if-not #'numberp '(3 7 2 5/3 'a 'b 'c)))
```

When you execute the code, it returns the following result:

```
(ZARA RIYAN NUHA)
(2 5/3 'A)
('A 'B 'C)
```

Set Union

The **union** group of functions allows you to perform set union on two lists provided as arguments to these functions on the basis of a test.

The following syntaxes are used for these functions:

```
union list1 list2 &key :test :test-not :key
nunion list1 list2 &key :test :test-not :key
```

The **union** function takes two lists and returns a new list containing all the elements present in either of the lists. If there are duplications, then only one copy of the member is retained in the returned list.

The **nunion** function performs the same operation but may destroy the argument lists.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq set1 (union '(a b c) '(c d e)))
(setq set2 (union '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch))

(setq set3 (union '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h))))
(write set1)
(terpri)
```

```
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(A B C D E)
(#(F H) #(5 6 7) #(A B) #(G H))
(#(A B) #(5 6 7) #(F H) #(5 6 7) #(A B) #(G H))
```

Note that the union function does not work as expected without :test-not #'mismatch arguments for a list of three vectors. This is because, the lists are made of cons cells and although the values look same to us apparently, the ***cdr*** part of cells does not match, so they are not exactly same to LISP interpreter/compiler. This is the reason for not advising large sets implementation using lists. It works fine for small sets though.

Set Intersection

The **intersection** group of functions allows you to perform intersection on two lists provided as arguments to these functions on the basis of a test.

The following are the syntaxes of these functions:

```
intersection list1 list2 &key :test :test-not :key
nintersection list1 list2 &key :test :test-not :key
```

These functions take two lists and return a new list containing all the elements present in both argument lists. If either list has duplicate entries, the redundant entries may or may not appear in the result.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq set1 (intersection '(a b c) '(c d e)))
(setq set2 (intersection '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch))

(setq set3 (intersection '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h))))
(write set1)
```

```
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(C)
(#(A B) #(5 6 7))
NIL
```

The **nintersection** function is the destructive version of intersection, i.e., it may destroy the original lists.

Set Difference

The **set-difference** group of functions allows you to perform set difference on two lists provided as arguments to these functions on the basis of a test.

The following syntaxes are used for these functions:

```
set-difference list1 list2 &key :test :test-not :key
nset-difference list1 list2 &key :test :test-not :key
```

The set-difference function returns a list of elements of the first list that do not appear in the second list.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq set1 (set-difference '(a b c) '(c d e)))
(setq set2 (set-difference '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h)) :test-not #'mismatch))
(setq set3 (set-difference '(#(a b) #(5 6 7) #(f h))
       '(#(5 6 7) #(a b) #(g h))))
(write set1)
(terpri)
(write set2)
(terpri)
(write set3)
```

When you execute the code, it returns the following result:

```
(A B)
(#(F H))
(#(A B) #(5 6 7) #(F H))
```

23. TREE

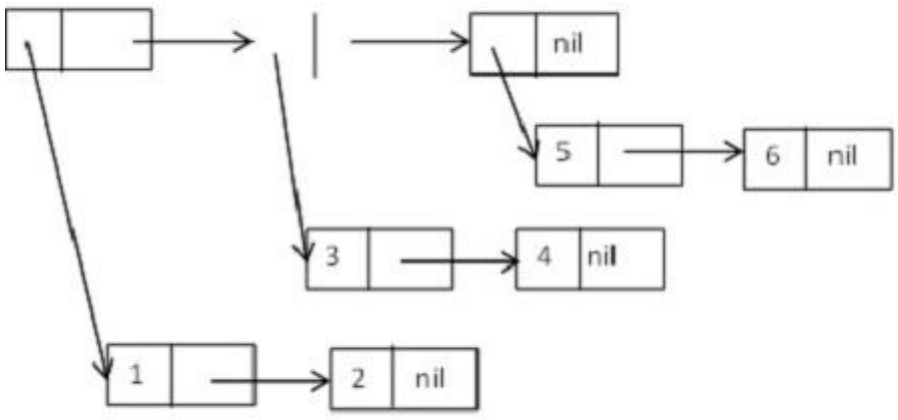
You can build tree data structures from cons cells, as lists of lists. To implement tree structure, you need to design functionalities that can traverse through the **cons** cells, in a specific order. For example, pre-order, in-order, and post-order for binary trees.

Tree as List of Lists

Let us consider a tree structure made up of cons cell that forms the following list of lists:

((1 2) (3 4) (5 6)).

Diagrammatically, it can be expressed as:



Tree Functions in LISP

Although mostly you will need to write your own tree-functionalities according to your specific need, LISP provides some tree functions that you can use.

Apart from all the list functions, the following functions work especially on tree structures:

Function	Description
copy-tree x &optional vecp	It returns a copy of the tree of cons cells x. it recursively copies both the car and the cdr directions. If x is not a cons cell, the function simply returns x unchanged. If the optional vecp argument is true, this function copies vectors (recursively) as well as cons cells.

tree-equal x y &key :test :test-not :key	It compares two trees of cons cells. If x and y are both cons cells, their cars and cdrs are compared recursively. If neither x nor y is a cons cell, they are compared by eql, or according to the specified test. The :key function, if specified, is applied to the elements of both trees.
subst new old tree &key :test :test-not :key	It substitutes occurrences of given old item with *new* item, in *tree*, which is a tree of cons cells.
nsubst new old tree &key :test :test-not :key	It works same as subst, but it destroys the original tree.
sublis alist tree &key :test :test-not :key	It works like subst, except that it takes an association list *alist* of old-new pairs. Each element of the tree (after applying the :key function, if any), is compared with the cars of alist; if it matches, it is replaced by the corresponding cdr.
nsublis alist tree &key :test :test-not :key	It works same as sublis, but a destructive version.

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(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)
```

When you execute the code, it returns the following result:

```
((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))
((1 2) (3 4) (5 6))
```

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(setq tr '((1 2 (3 4 5) ((7 8) (7 8 9)))))
(write tr)
(setq trs (subst 7 1 tr))
(terpri)
(write trs)
```

When you execute the code, it returns the following result:

```
((1 2 (3 4 5) ((7 8) (7 8 9))))
((7 2 (3 4 5) ((7 8) (7 8 9))))
```

Building Your Own Tree

Let us try to build our own tree, using the list functions available in LISP.

Creating a new node that contains some data

```
(defun make-tree (item)
  "it creates a new node with item."
  (cons (cons item nil) nil))
```

Adding a Child Node into a Tree

It takes two tree nodes and adds the second tree as a child of the first.

```
(defun add-child (tree child)
  (setf (car tree) (append (car tree) child))
  tree)
```

This function returns the first child of a given tree. It takes a tree node and returns the first child of that node. It returns **nil**, if this node does not have any child node.

```
(defun first-child (tree)
  (if (null tree)
    nil
    (cdr (car tree))))
```

This function returns the next sibling of a given node. It takes a tree node as argument, and returns a reference to the next sibling node, or **nil**, if the node does not have any sibling.

```
(defun next-sibling (tree)
   (cdr tree))
```

Lastly, we need a function to return the information in a node.

```
(defun data (tree)
  (car (car tree)))
```

Example

Create a new source code file named main.lisp and type the following code in it:

```
(defun make-tree (item)
  "it creates a new node with item."
  (cons (cons item nil) nil))
 (defun first-child (tree)
  (if (null tree)
    nil
    (cdr (car tree))))
 (defun next-sibling (tree)
   (cdr tree))
(defun data (tree)
   (car (car tree)))
 (defun add-child (tree child)
  (setf (car tree) (append (car tree) child))
  tree)

(setq tr '((1 2 (3 4 5) ((7 8) (7 8 9)))))
(setq mytree (make-tree 10))
(write (data mytree))
(terpri)
(write (first-child tr))
```

```
(terpri)
(setq newtree (add-child tr mytree))
(terpri)
(write newtree)
```

When you execute the code, it returns the following result:

```
10
(2 (3 4 5) ((7 8) (7 8 9)))


((1 2 (3 4 5) ((7 8) (7 8 9)) (10)))
```

24. 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.

A hash table is used when you need to access elements by using a key, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.

Creating Hash Table in LISP

In Common LISP, has table is a general-purpose collection. You can use arbitrary objects as a key or indexes.

When you store a value in a hash table, you make a key-value pair, and store it under that key. Later, you can retrieve the value from the hash table using the same key. Each key maps to a single value, though you can store a new value in a key.

Hash tables, in LISP, could be categorized into three types, based on the way the keys are compared - **eq, eql** or **equal.** If the hash table is hashed on LISP objects then the keys are compared with **eq** or **eql**. If the hash table hash on tree structure, then it is compared using **equal**.

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

Syntax for make-hash-table:

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

Where,

- The key argument provides the key.
- The :test argument determines how keys are compared - it should have one of three values #'eq, #'eql, or #'equal, or one of the three symbols **eq, eql,** or **equal**. If not specified, **eql** is assumed.
- The :size argument sets the initial size of the hash table. This must be an integer greater than zero.
- The :rehash-size argument specifies the amount by which the size of the hash table can be increased when it becomes full. This can be an integer greater than zero, which is the number of entries to add, or it can be a floating-point number

greater than 1, which is the ratio of the new size to the old size. The default value for this argument is implementation-dependent.

- The :rehash-threshold argument specifies how full the hash table can get before it must grow. This must be an integer greater than zero and less than the :rehash-size (it is scaled whenever the table is grown), or it can be a floating-point number between zero and 1. The default value for this argument is implementation-dependent.

You can also call the **make-hash-table** function with no arguments.

Retrieving Items from Hash Table

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**.

Syntax for gethash function:

```
gethash key hash-table &optional default
```

where,

- key: is the associated key
- hash-table: is the hash-table to be searched
- default: is the value to be returned, if the entry is not found, which is nil, if not specified.

The **gethash** function actually returns two values, the second being a predicate value that is true if an entry was found, and false if no entry was found.

Adding Items into Hash Table

For adding an item to the hash table, you can use the **setf** function along with the **gethash** function.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(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))
```

When you execute the code, it returns the following result:

```
(CHARLIE BROWN)
(FREDDIE SEAL)
```

Removing an Entry from Hash Table

The **remhash** function removes any entry for a specific key in hash-table. This is a predicate that is true if there was an entry, or false if there was not.

Syntax for remhash function:

```
remhash key hash-table
```

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown))
(setf (gethash '002 empList) '(Freddie Seal))
(setf (gethash '003 empList) '(Mark Mongoose))
(write (gethash '001 empList))
(terpri)
(write (gethash '002 empList))
(terpri)
(write (gethash '003 empList))
(remhash '003 empList)
(terpri)
(write (gethash '003 empList))
```

When you execute the code, it returns the following result:

```
(CHARLIE BROWN)
(FREDDIE SEAL)
(MARK MONGOOSE)
NIL
```

Applying a Specified Function on Hash Table

The **maphash** function allows you to apply a specified function on each key-value pair on a hash table.

It takes two arguments - the function and a hash table. It invokes the function once for each key/value pair in the hash table.

Example

Create a new source code file named main.lisp and type the following code in it:

```
(setq empList (make-hash-table))
(setf (gethash '001 empList) '(Charlie Brown))
(setf (gethash '002 empList) '(Freddie Seal))
(setf (gethash '003 empList) '(Mark Mongoose))
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) empList)
```

When you execute the code, it returns the following result:

```
3 => (MARK MONGOOSE)
2 => (FREDDIE SEAL)
1 => (CHARLIE BROWN)
```

25. INPUT & OUTPUT

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

Input Functions

The following table provides the most commonly used input functions of LISP:

Sr. No.	Functions and Descriptions
1	**read &optional *input-stream eof-error-p eof-value recursive-p*** It reads in the printed representation of a LISP object from input-stream, builds a corresponding LISP object, and returns the object.
2	**read-preserving-whitespace &optional *in-stream eof-error-p eof-value recursive-p*** It is used in some special situations where it is desirable to determine precisely which character terminated the extended token.
3	**read-line &optional *input-stream eof-error-p eof-value recursive-p*** It reads a line of text terminated by a newline.
4	**read-char &optional *input-stream eof-error-p eof-value recursive-p*** It takes one character from input-stream and returns it as a character object.
5	**unread-char *character &optional input-stream*** It puts the character most recently read from the input-stream, onto the front of input-stream.
6	**peek-char &optional *peek-type input-stream eof-error-p eof-value recursive-p*** It returns the next character to be read from input-stream, without actually removing it from the input stream.

7	**listen &optional *input-stream*** The predicate listen is true if there is a character immediately available from input-stream, and is false if not.
8	**read-char-no-hang &optional *input-stream eof-error-p eof-value recursive-p*** It is similar to read-char, but if it does not get a character, it does not wait for a character, but returns nil immediately.
9	**clear-input &optional *input-stream*** It clears any buffered input associated with *input-stream.*
10	**read-from-string *string* &optional *eof-error-p eof-value* &key *:start :end :preserve-whitespace*** It takes the characters of the string successively and builds a LISP object and returns the object. It also returns the index of the first character in the string not read, or the length of the string (or, length +1), as the case may be.
11	**parse-integer *string &key :start :end :radix :junk-allowed*** It examines the substring of string delimited by :start and :end (default to the beginning and end of the string). It skips over whitespace characters and then attempts to parse an integer.
12	**read-byte *binary-input-stream* &optional *eof-error-p eof-value*** It reads one byte from the binary-input-stream and returns it in the form of an integer.

Reading Input from Keyboard

The **read** function is used for taking input from the keyboard. It may not take any argument.

Example

Consider the following code snippet:

```
(write ( + 15.0 (read)))
```

Assume the user enters 10.2 from the STDIN Input. Then the code returns:

```
25.2
```

The read function reads characters from an input stream and interprets them by parsing as representations of LISP objects.

Example

Create a new source code file named main.lisp and type the following code in it:

```
; the function AreaOfCircle
; calculates area of a circle
; when the radius is input from keyboard

(defun AreaOfCircle()
(terpri)
(princ "Enter Radius: ")
(setq radius (read))
(setq area (* 3.1416 radius radius))
(princ "Area: ")
(write area))
(AreaOfCircle)
```

When you execute the code, it returns the following result:

```
Enter Radius: 5 (STDIN Input)
Area: 78.53999
```

Example

Create a new source code file named main.lisp and type the following code in it:

```
(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))
```

When you execute the code, it returns the following result:

```
#\W
#\e
#\l
#\c
#\o
#\m
#\e
#\Space
#\t
#\o
#\Space
```

Output Functions

All output functions in LISP take an optional argument called *output-stream,* where the output is sent. If not mentioned or *nil,* output-stream defaults to the value of the variable *standard-output*.

The following table provides the most commonly used output functions of LISP:

Sr. No.	Functions and Descriptions
1	write *object* &key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array write *object* &key :stream :escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch Both write the object to the output stream specified by :stream, which defaults to the value of *standard-output*. Other values default to the corresponding global variables set for printing.

2	prin1*object* &optional *output-stream* print *object* &optional *output-stream* pprint *object* &optional *output-stream* princ *object* &optional *output-stream* All these functions output the printed representation of object to *output-stream*. However, the following differences are there: prin1 returns the object as its value. print prints the object with a preceding newline and followed by a space. It returns object. pprint is just like print except that the trailing space is omitted. princ is just like prin1 except that the output has no escape character
3	write-to-string *object &key* :escape :radix :base :circle :pretty :level :length :case :gensym :array write-to-string*object* &key :escape :radix :base :circle :pretty :level :length :case :gensym :array :readably :right-margin :miser-width :lines :pprint-dispatch prin1-to-string *object* princ-to-string *object* The object is effectively printed and the output characters are made into a string, which is returned.
4	write-char *character* &optional *output-stream* It outputs the character to *output-stream,* and returns character.
5	write-string *string* &optional *output-stream* &key :start :end It writes the characters of the specified substring of *string* to the *output-stream.*
6	write-line *string* &optional *output-stream* &key :start :end It works the same way as write-string, but outputs a newline afterwards.
7	terpri &optional *output-stream*

	It outputs a newline to *output-stream.*
8	fresh-line &optional *output-stream* It outputs a newline only if the stream is not already at the start of a line.
9	finish-output &optional *output-stream* force-output &optional *output-stream* clear-output &optional *output-stream* The function finish-output attempts to ensure that all output sent to output-stream has reached its destination, and only then returns nil. The function force-output initiates the emptying of any internal buffers but returns nil without waiting for completion or acknowledgment. The function clear-output attempts to abort any outstanding output operation in progress in order to allow as little output as possible to continue to the destination.
10	write-byte *integer binary-output-stream* It writes one byte, the value of the *integer.*

Example

Create a new source code file named main.lisp and type the following code in it:

```
; this program inputs a numbers and doubles it
(defun DoubleNumber()
(terpri)
(princ "Enter Number : ")
(setq n1 (read))
(setq doubled (* 2.0 n1))
(princ "The Number: ")
(write n1)
(terpri)
(princ "The Number Doubled: ")
(write doubled)
)
(DoubleNumber)
```

When you execute the code, it returns the following result:

```
Enter Number : 3456.78 (STDIN Input)
The Number: 3456.78
The Number Doubled: 6913.56
```

Formatted Output

The function **format** is used for producing well formatted text. It has the following syntax:

```
format destination control-string &rest arguments
```

where,

- *destination* is standard output
- *control-string* holds the characters to be output and the printing directive.

A **format directive** consists of a tilde (~), optional prefix parameters separated by commas, optional colon (:) and at-sign (@) modifiers, and a single character indicating what kind of directive this is.

The prefix parameters are generally integers, notated as optionally signed decimal numbers. The following table provides brief description of the commonly used directives:

Directive	Description
~A	It is followed by ASCII arguments
~S	It is followed by S-expressions
~D	For decimal arguments
~B	For binary arguments
~O	For octal arguments
~X	For hexadecimal arguments
~C	For character arguments
~F	For Fixed-format floating-point arguments.

~E	Exponential floating-point arguments
~$	Dollar and floating point arguments.
~%	A new line is printed
~*	Next argument is ignored
~?	Indirection. The next argument must be a string, and the one after it a list.

Example

Let us rewrite the program for calculating area of circle:

Create a new source code file named main.lisp and type the following code in it:

```
(defun AreaOfCircle()
(terpri)
(princ "Enter Radius: ")
(setq radius (read))
(setq area (* 3.1416 radius radius))
(format t "Radius: = ~F~% Area = ~F" radius area)
)
(AreaOfCircle)
```

When you execute the code, it returns the following result:

```
Enter Radius: 10.234 (STDIN Input)
Radius: = 10.234
Area = 329.03473
```

26. FILE I/O

We discussed about how standard input and output is handled by common LISP. All file iput-output functions work for reading from and writing into text and binary files. Only difference is, the stream we use is not standard input or output, but it is a stream created for the specific purpose of writing into or reading from files.

In this chapter, we will see how LISP can create, open, close text or binary files for their data storage.

Opening Files

A file represents a sequence of bytes, be it a text file or binary file.

You can use the **open** function to create a new file or to open an existing file. It is the most basic function for opening a file. However, the **with-open-file** is usually more convenient and more commonly used.

When a file is opened, a stream object is constructed to represent it in the LISP environment. All operations on the stream are equivalent to operations on the file.

Syntax for the **open** function is:

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

where,

- The *filename* argument is the name of the file to be opened or created.
- The keyword arguments specify the type of stream and error handling ways.
- The **:direction** keyword specifies whether the stream should handle input, output, or both. It takes the following values:
    - :input - for input streams (default value)
    - :output - for output streams
    - :io - for bidirectional streams
    - :probe - for checking a existence of a file; the stream is opened and then closed.
- The **:element-type** specifies the type of the unit of transaction for the stream.

- The **:if-exists** argument specifies the action to be taken if the **:direction** is **:output** or **:io** and a file of the specified name already exists. If the direction is**:input** or **:probe**, this argument is ignored. It takes the following values:
- **:error** - it signals an error.
- **:new-version** - it creates a new file with the same name but larger version number.
- **:rename** - it renames the existing file.
- **:rename-and-delete** - it renames the existing file and then deletes it.
- **:append** - it appends to the existing file.
- **:supersede** - it supersedes the existing file means, new file is created to replace old one.
- **nil** - it does not create a file or even a stream just returns nil to indicate failure.
- The **:if-does-not-exist** argument specifies the action to be taken if a file of the specified name does not already exist. It takes the following values:
- **:error** - it signals an error.
- **:create** - it creates an empty file with the specified name and then uses it.
- **nil** - it does not create a file or even a stream. Instead, it simply returns nil to indicate failure.
- The **:external-format** argument specifies an implementation-recognized scheme for representing characters in files.

Example

Let us open a file named myfile.txt stored in the /tmp folder as:

```
(open "/tmp/myfile.txt")
```

Writing to and Reading from Files

The function **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.

It has the following syntax:

```
with-open-file (stream filename {options}*)
       {declaration}* {form}*
```

Where,

- *filename* is the name of the file to be opened. It may be a string, a pathname, or a stream.
- The *options* are same as the keyword arguments to the function open.

Example 1

Create a new source code file named main.lisp and type the following code in it:

```
(with-open-file (stream "/tmp/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."))
```

Please note that all input-output functions discussed in the previous chapter such as **terpri** and **format** are working for writing into the file we created here.

When you execute the code, it does not return anything. However, the data is written into the file. The **:direction :output** keywords allows us do this.

However, we can read from this file using the **read-line** function.

Example 2

Create a new source code file named main.lisp and type the following code in it:

```
(let ((in (open "/tmp/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)))
```

When you execute the code, it returns the following result:

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

Closing a File

If you do not close an open file when you are done with it, every time program will keep on leaking a new file handle and soon you will discover you are not able to create new files anymore. Hence, it is important to close a file that you opened as a best programming practice.

The **close** function closes a stream.

27. STRUCTURES

Structures are the user-defined data type. They allow you to combine related data items of different data types. You can access any of the components of a structure.

Structures are used to represent a record. Let us say you need to keep track of your books in a library. You might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Structure

The **defstruct** macro in LISP allows you to define an abstract record structure. The **defstruct** statement defines a new data type, with more than one member for your program.

To discuss the format of the **defstruct** macro, let us write the definition of the Book structure. You can define the book structure as:

```
(defstruct book
  title
  author
  subject
  book-id
 )
```

Note:

- The above declaration creates a book structure with four named components namely title, author, subject and book-id. Thus, every object created as type book will have all these components.
- It defines four functions named *book-title, book-author, book-subject* and *book-book-id*, which take one argument, a book structure, and returns the field title, author, subject and book-id of the book object. These functions are called the access functions.

- The symbol book becomes a data type and you can check it using the **typep** predicate.
- There is an implicit function named book-p, which is a predicate. It is true if its argument is a book and is false otherwise.
- Another implicit function named make-book is created, which is a constructor. When it is invoked, it creates a data structure with four components, suitable to be uses with the access functions.
- The #S syntax refers to a structure, and you can use it to read or print instances of a book.
- An implicit function named copy-book of one argument is also defined that. It takes a book object and creates another book object, which is a copy of the first one. This function is called the copier function.
- You can use setf to alter the components of a book as given below:

```
(setf (book-book-id book3) 100)
```

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(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)
```

When you execute the code, it returns the following result:

```
#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)
```

28. PACKAGES

In general term of 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 do not conflict with the same symbols declared in another. This way, packages reduce the naming conflicts between independent code modules.

The LISP reader maintains a table of all the symbols it has found. When it finds a new character sequence, it creates a new symbol and stores in the symbol table. This table is called a package.

The current package is referred by the special variable *package*. There are two predefined packages in LISP:

common-lisp - it contains symbols for all the defined functions and variables.

common-lisp-user - it uses the common-lisp package and all other packages with editing and debugging tools. It is called cl-user in short.

Package Functions in LISP

The following table provides most commonly used functions used for creating, using and manipulating packages:

Sr. No.	Functions and Descriptions
1	**make-package *package-name* &key :nicknames :use** It creates and returns a new package with the specified package name.
2	**in-package *package-name* &key :nicknames :use** I makes the package as current.
3	**in-package *name*** This macro causes *package* to be set to the package named name, which must be a symbol or string.
4	**find-package *name*** It searches for a package. The package with that name or nickname is returned; if no such package exists, find-package returns nil.

5	**rename-package** ***package new-name*** **&optional** ***new-nicknames*** It renames a package.
6	**list-all-packages** This function returns a list of all packages that currently exist in the LISP system.
7	**delete-package** ***package*** It deletes a package.

Creating a Package

The **defpackage** function is used for creating a user defined package. It has the following syntax:

```
defpackage :package-name
  (:use :common-lisp ...)
  (:export :symbol1 :symbol2 ...))
```

Where,

- **package-name** is the name of the package.
- The **:use** keyword specifies the packages needed by this package i.e., packages that define functions used by code in this package.
- The **:export** keyword specifies the symbols that are external in this package.

The **make-package** function is also used for creating a package. The syntax for this function is:

```
make-package package-name &key :nicknames :use
```

The arguments and keywords has same meaning as explained before.

Using a Package

Once you have created a package, you can use the code in this package, by making it the current package. The **in-package** macro makes a package current in the environment.

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(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)
```

When you execute the code, it returns the following result:

```
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
```

Deleting a Package

The delete-package macro allows you to delete a package. The following example demonstrates this:

Example

Create a new source code file named *main.lisp* and type the following code in it:

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

When you execute the code, it returns the following result:

```
Hello! This is Tom's Tutorials Point
Hello! This is Dick's Tutorials Point
Hello! This is Harry's Tutorials Point
```

```
*** - EVAL: variable TOM has no value
```

29. ERROR HANDLING

This chapter explains Object Oriented condition handling and error handling in LISP.

In Common LISP terminology, exceptions are called conditions. In fact, conditions are more general than exceptions in traditional programming languages. A condition represents any occurrence of an event or an error, which might affect various levels of function call stack. Conditions are capable of altering program flow or halting program execution. They enforce programmers or users to take decisions. Errors are a subset of conditions. The errors can be fatal, if they are not managed appropriately.

The condition handling system in LISP has three responsibilities:

1. Signaling a condition
2. Handling the condition
3. Restarting or continuing the program

Signaling a Condition

Signaling a condition means making the user aware of some occurrence of an event through program execution. A condition can be announced by displaying message on the screen or by beeping sound. When a condition is signaled, the signaling mechanism finds the most recently established handler that is compatible with the condition type and calls its function. Signaling a condition does not affect the program flow.

Handling a Condition

Let us take an example of handling a condition arising out of divide by zero condition.

You need to take the following steps for handling a condition:

- **Define the condition** - "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 signaled". 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)))
```

MAKE-CONDITION macro creates new condition objects. It initializes the slots of the new condition based on **:initargs** argument.

The following code defines the condition:

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

- **Writing the Handlers** - a condition handler is a code that is used for handling the signaled condition thereon. It is generally written in one of the higher level functions that call the erring function. When a condition is signaled, the signaling mechanism searches for an appropriate handler based on the class of the condition.

  Each handler consists of:

  - Type specifier that indicates the type of condition it can handle
  - A function that takes a single argument, the condition

The macro **handler-case** establishes a condition handler. The basic form of a handler-case:

```
(handler-case expression
  error-clause*)
```

Where, each error clause is of the form:

```
condition-type ([var]) code)
```

Restarting or Continuing the Program Execution

This is the code, which actually recovers your program from errors. The condition handlers can then handle a condition by invoking an appropriate restart. The restart code is generally place in middle-level or low-level functions and the condition handlers are placed into the upper levels of the application.

The **handler-bind** macro allows you to provide a restart function, and to continue at the lower level functions without unwinding the function call stack. In other words, the flow of control still remains at the lower level function.

The basic form of **handler-bind** is as follows:

```
(handler-bind (binding*) form*)
```

Where each binding is a list of the following:

- a condition type
- a handler function of one argument

The **invoke-restart** macro finds and invokes the most recently bound restart function with the specified name as argument. You can have multiple restarts.

Example

In this example, we demonstrate the above concepts by writing a function named division-function, which creates an error condition if the divisor argument is zero. We have three anonymous functions that provide three ways to come out of it - by returning a value 1, by sending a divisor 2 and recalculating, or by returning 1.

Create a new source code file named *main.lisp* and type the following code in it:

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

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

When you execute the code, it returns the following result:

```
error signaled: denominator is zero
Value: 1
error signaled: denominator is zero
Value: 5
```

```
error signaled: denominator is zero
Done.
```

Apart from the condition system as discussed above, Common LISP also provides various functions that may be called for signaling an error. Handling of an error, when signaled, is however, implementation-dependent.

Error Signaling Functions in LISP

The user program specifies an error message. The functions process this message and may or may not display it to the user.

The error messages should be constructed by applying the **format** function, should not contain a newline character at either the beginning or end, and need not indicate error, as the LISP system will take care of all these according to its preferred style.

The following table provides commonly used functions signaling warnings, breaks, non-fatal and fatal errors.

Sr. No.	Functions and Descriptions
1	**error** *format-string* &rest *args* It signals a fatal error. It is impossible to continue from this kind of error; thus error will never return to its caller.
2	**cerror** *continue-format-string error-format-string* &rest *args* It signals an error and enters the debugger. However, it allows the program to be continued from the debugger after resolving the error.
3	**warn** *format-string* &rest *args* It prints an error message but normally does not go into the debugger.
4	**break** &optional *format-string* &rest *args* It prints the message and goes directly into the debugger without allowing any possibility of interception by programmed error-handling facilities.

Example

In this example, the factorial function calculates factorial of a number. However, if the argument is negative, it raises an error condition.

Create a new source code file named *main.lisp* and type the following code in it:

```
(defun factorial (x)
   (cond ((or (not (typep x 'integer)) (minusp x))
          (error "~S is a negative number." x))
         ((zerop x) 1)
         (t (* x (factorial (- x 1))))))


(write(factorial 5))
(terpri)
(write(factorial -1))
```

When you execute the code, it returns the following result:

```
120
*** - -1 is a negative number.
```

30. COMMON LISP OBJECT SYSTEMS

Common LISP predated the advance of object-oriented programming by couple of decades. However, the object-orientation was incorporated into it at a later stage.

Defining Classes

The **defclass** macro allows creating user-defined classes. It establishes a class as a data type. It has the following syntax:

```
(DEFCLASS class-name (superclass-name*)
  (slot-description*)
  class-option*)
```

The slots are variables that store data, or fields.

A slot-description has the form (slot-name slot-option*), where each option is a keyword followed by a name, expression and other options. Most commonly used slot options are:

- **:accessor** function-name
- **:initform** expression
- **:initarg** symbol

For example, let us define a Box class, with three slots length, breadth, and height.

```
(defclass Box ()
(length
breadth
height))
```

Providing Access and Read/Write Control to a Slot

Unless the slots have values that can be accessed, read or written to, classes are pretty useless. You can specify **accessors** for each slot when you define a class. For example, refer our Box class:

```
(defclass Box ()
  ((length :accessor length)
   (breadth :accessor breadth)
   (height :accessor height)))
```

You can also specify separate **accessor** names for reading and writing a slot.

```
(defclass Box ()
    ((length :reader get-length :writer set-length)
     (breadth :reader get-breadth :writer set-breadth)
     (height :reader get-height :writer set-height)))
```

Creating Instance of a Class

The generic function **make-instance** creates and returns a new instance of a class.

It has the following syntax:

```
(make-instance class {initarg value}*)
```

Example

Let us create a Box class, with three slots, length, breadth and height. We will use three slot accessors to set the values in these fields.

Create a new source code file named *main.lisp* and type the following code in it:

```
(defclass box ()
  ((length :accessor box-length)
   (breadth :accessor box-breadth)
   (height :accessor box-height)))
(setf item (make-instance 'box))
(setf (box-length item) 10)
(setf (box-breadth item) 10)
(setf (box-height item) 5)
(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))
```

When you execute the code, it returns the following result:

```
Length of the Box is 10
Breadth of the Box is 10
```

```
Height of the Box is 5
```

Defining a Class Method

The **defmethod** macro allows you to define a method inside the class. The following example extends our Box class to include a method named volume.

Create a new source code file named *main.lisp* and type the following code in it:

```
(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))
```

When you execute the code, it returns the following result:

```
Length of the Box is 10
Breadth of the Box is 10
Height of the Box is 5
Volume of the Box is 500
```

Inheritance

LISP allows you to define an object in terms of another object. This is called **inheritance.** You can create a derived class by adding features that are new or different. The derived class inherits the functionalities of the parent class.

The following example explains this:

Example

Create a new source code file named *main.lisp* and type the following code in it:

```
(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))
```

When you execute the code, it returns the following result:

```
Length of the Wooden Box is 10
Breadth of the Wooden Box is 10
Height of the Wooden Box is 5
Volume of the Wooden Box is 500
Price of the Wooden Box is 1000
```