1. Write a “Hello World” program in LISP

```
CL-USER> (defun hello ()
           (format t "Hello, World!~%"))
HELLO

CL-USER> (hello)
Hello, World!
NIL
```

After we have defined the `hello` function, we call it with no parameters by writing `(hello)`. This in turn will call the `format` function with the parameters `t` and `"Hello, World!~%"`. The `format` function produces formatted output based on the arguments which it is given (a bit like an advanced version of `printf` in C). The first argument tells it where to output to, with `t` meaning standard-output. The second argument tells it what to print (and how to interpret any extra parameters). The directive (special code in the second argument) `~%` tells format to print a newline (i.e. on UNIX it might write `\n` and on windows `\r\n`).

2. Write a LISP program to demonstrate the global and local variable and constant.

Variables are declared with the `let` special operator:

```
(let ((str "Hello, world!"))
  (string-upcase str))


;; => "HELLO, WORLD!"
```

You can define multiple variables:

```
(let ((x 1)
      (y 5))
  (+ x y))
```

```
;; => 6
```

To define variables whose initial values depend on previous variables in the same form, use `let*`:

```
(let* ((x 1)
       (y (+ x 1)))
  y)


;; => 2
```

You can define multiple variables:

```
(let ((x 1)
      (y 5))
  (+ x y))

;; => 6
```

To define variables whose initial values depend on previous variables in the same form, use `let*`:

```
(let* ((x 1)
       (y (+ x 1)))
  y)


;; => 2
```

3. Write a LISP program to demonstrate the macros.

Common Lisp has no `while` loop, rather, there's a `loop` macro directive for iterating while a condition is true. For brevity, we can define:

```
(defmacro while (condition &body body)
  `(loop while ,condition do (progn ,@body)))
```

And use it like this:

```
(while (some-condition)
  (do-something)
  (do-something-else))
```

This expands to:

```
(loop while (some-condition) do
  (progn
    (do-something)

    (do-something-else)))
```

Example:
Let us write a simple macro named setTo10, which will take a number and set its value to 10.

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

Answer is:

```
25
10
```

4. Write a LISP program to demonstrate the process of assigning the value in variable and showing them in the console.

Global Variables

Global variables have permanent values throughout the LISP system and remain in effect until a new value is specified.

Global variables are generally declared using the defvar construct.

For example

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

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

```
234
```

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

For 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.

For 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 returned 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 lastly the *s-expression* is evaluated. The value of the last expression evaluated is returned.

If you don't include an initial value for a variable, it 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 returned 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 returned is.

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

5. Write a LISP program to demonstrate the arithmetic expression.

Assume variable A holds 10 and variable B holds 20 then :

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

Example:

```
(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))
```

```
A + B = 30
A - B = -10
A x B = 200
B / A = 2
Increment A by 3 = 13
Decrement A by 4 = 9
```

6. Write a LISP program to demonstrate the comparison and logical operator.

Assume variable A holds 10 and variable B holds 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

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.
min	It compares two or more arguments and returns the minimum value.	(min A B) returns 10

Example:

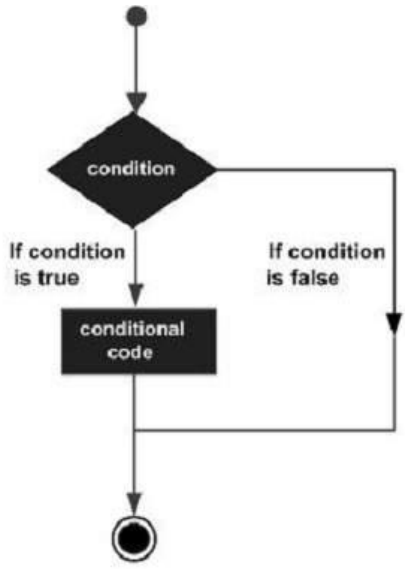
```
(setq a 10)
(setq b 20)
(format t "~% A = B is ~a" (= a b))
```

```
(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))
```

```
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
```

7. Write a LISP program to demonstrate the decision making.

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



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

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

if Statement

The if is a decision-making statement used to check whether the condition is right or wrong. The “if” the condition is right, then it will go inside the if block and execute the statements under the “if” block. Otherwise, the statements are not executed.

Syntax:

```
(if (condition) then (statement 1).....(statement  n))
```

Here, then is an optional keyword used inside the if statement.

Example 1: LISP Program to check conditions with operators

```
;define value to 100
(setq val1 100)

;check the number is equal to 100
(if (= val1 100)
(format t "equal to 100"))

(terpri)

;check the number is greater than to 50
(if (> val1 50)
(format t "greater than 50"))

(terpri)

;check the number is less than to 150
(if (< val1 150)
(format t "less than 150"))
```

Output:

```
equal to 100
greater than 50
less than 150
```

cond Statement

The cond is a decision-making statement used to make n number of test conditions. It will check all the conditions.

Syntax:

```
(cond   (condition1 statements)
(condition2 statements)
(condition3 statements)
...
(conditionn statements)
)
```

Here,

- The conditions specify different conditions – if condition1 is not satisfied, then it goes for the next condition IE condition until the last condition.
- The statements specify the work done based on the condition.

Example 1: LISP program to check whether a number is greater than 200 or not.

```
;set value1 to 500
(setq val1 500)

;check whether the val1 is greater than 200
(cond ((> val1 200)
   (format t "Greater than 200"))
   (t (format t "Less than 200")))
```

when Statement

The when is a decision-making statement used to specify the decisions. It is similar to conditional statements.

Syntax:

```
(when (condition) (statements) )
```

where,

1. condition is a test statement used to test
2. statements are the actions that will depend on the condition

Example 1: LISP Program to check the number is equal to 50 or not

```
;set number to 50
(setq number 50)

;condition check the given number is equal to 50
(when (= number 50)

;statement
(format t "Equal to 50")
)
```

8. Write a LISP program to demonstrate the looping operations.

The ***loop for construct*** in common LISP is used to iterate over an iterable, similar to the *for* loop in other programming languages. It can ber used for the following:

1. This is used to set up variables for iteration.
2. It can be used for conditionally terminate the iteration.
3. It can be used for operating on the iterated elements.

The loop For construct can have multiple syntaxes.

- **To iterate over a list:**

```
(loop for variable in input_list
  do (statements/conditions)
)
```

Here,

1. ***input_list*** is the list that is to be iterated.
2. The ***variable*** is used to keep track of the iteration.
3. The ***statements/conditions*** are used to operate on the iterated elements.

- **To iterate over the given range:**

```
(loop for variable  from number1 to number2
 do (statements/conditions)
)
```

Here,

- The ***number1*** is the starting number and ***number 2*** is the ending number from the range.
- The ***variable*** is used to keep track of the iteration.

- The ***statements/conditions*** are used to operate on the iterated elements.

```
;range from 1 to 5
(loop for i from 1 to 5

;display each number
do (print i)
)
```

Output:

```
1
2
3
4
5
```

9. Write a LISP program to demonstrate the function.

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

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so 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 −

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

Syntax for defun is −

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

Example 1

Let's write a function named *averagenum* that will 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
```

- 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.

10. Write a LISP program to calculate factorial by using a function.

Factorial of a whole number 'n' is defined as the product of that number with every whole number till 1. For example, the factorial of 4 is 4×3×2×1, which is equal to 24. It is represented using the symbol '!' So, 24 is the value of 4!

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))) ) )

(loop for i from 0 to 16
   do (format t "~D! = ~D~%" i (factorial i)) )
```

The factorial of a number is the function that multiplies the number by every natural number below it. Symbolically, factorial can be represented as "!". So, n factorial is the product of the first n natural numbers and is represented as n!

The formula for n factorial is:

$$n!=n\times(n-1)!$$

11. Write a LISP program to find a maximum of three numbers.

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

The number types supported by LISP are −

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

```
(defun max3(a b c)
    (cond((>a b) (cond((>a c)
                          (format t "Max ~a"
                           (t(format t "Max ~a"
                      ((>b c> (format t "Max ~a"
                                  (t format t "MAX ~a" c))))
```

The following table describes some commonly used numeric functions −

Sr.No.	Function & Description
1	+, -, *, / Respective arithmetic operations
2	sin, cos, tan, acos, asin, atan

	Respective trigonometric functions.
3	sinh, cosh, tanh, acosh, asinh, atanh Respective hyperbolic functions.
4	exp Exponentiation function. Calculates e^x
5	expt Exponentiation function, takes base and power both.
6	sqrt It calculates the square root of a number.
7	log Logarithmic function. It one parameter is given, then it calculates its natural logarithm, otherwise the second parameter is used as base.

8	conjugate It calculates the complex conjugate of a number. In case of a real number, it returns the number itself.
9	abs It returns the absolute value (or magnitude) of a number.
10	gcd It calculates the greatest common divisor of the given numbers.
11	lcm It calculates the least common multiple of the given numbers.
12	isqrt It gives the greatest integer less than or equal to the exact square root of a given natural number.

13	floor, ceiling, truncate, round All these functions take two arguments as a number and returns the quotient; floor returns 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 round chooses an integer that is closest to ratio.
14	ffloor, fceiling, ftruncate, fround Does the same as above, but returns the quotient as a floating point number.
15	mod, rem Returns the remainder in a division operation.
16	float Converts a real number to a floating point number.
17	rational, rationalize Converts a real number to rational number.

18	numerator, denominator Returns the respective parts of a rational number.
19	realpart, imagpart Returns the real and imaginary part of a complex number.

12.Write a LISP program to find the GCD of two numbers.

GCD (Greatest Common Divisor) or HCF (Highest Common Factor) of two numbers is the largest number that divides both of them.

```
(DEFUN GCD(X Y)
    (COND((OR(=X 0)(=Y 0)) 0)
          ((<X 0) (GCD(-X) Y)))
          ((<Y 0) (GCD X (-Y)))
          (T(GCD-POSITIVE(X Y)
             (COND((=XY)X)
                   ((>X Y) (GCD-POSITIVE(-X Y)Y))
                   (T(GCD-POSITIVE X(-Y X)))))

           (DEFUN MAIN(X Y)  # Main Program
                (GCD X Y))
```

13. Write a lisp program to implement Fibonacci Series.

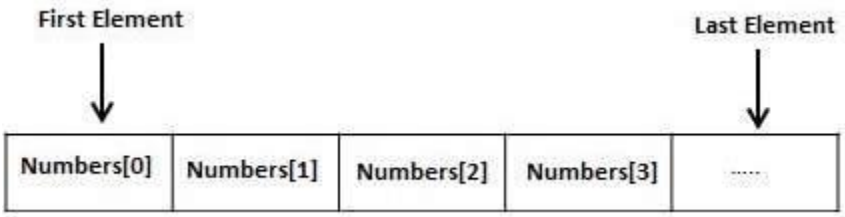
In mathematics, the Fibonacci numbers, commonly denoted F , form a sequence, the Fibonacci sequence, in which each number is the sum of the two preceding ones. The sequence commonly starts from 0 and 1, although some authors omit the initial terms and start the sequence from 1 and 1 or from 1 and 2.

```
;;Find the nth Fibonacci number for any n > 0.
;; Precondition: n > 0, n is an integer. Behavior undefined otherwise.
(defun fibonacci (n)
    (cond
        (                                   ;; Base case.
;; The first two Fibonacci numbers (indices 1 and 2) are 1 by definition.
            (<= n 2)                        ;; If n <= 2
            1                               ;; then return 1.
        )
        (t                                  ;; else
            (+                              ;; return the sum of
                                            ;; the results of calling
                (fibonacci (- n 1))         ;; fibonacci(n-1) and
                (fibonacci (- n 2))         ;; fibonacci(n-2).

;; This is the recursive case.
            )
        )
    )
)
```

14.Write a program in LISP to demonstrate the concept of 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 to the last element.



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 starts from zero.

For 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.

For example, to access the content of the tenth cell, we write −

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

Sr.No.	Argument & Description
1	dimensions It gives the dimensions of the array. It is a number for one-dimensional array, and a list for multi-dimensional array.
2	:element-type It is the type specifier, default value is T, i.e. any type
3	:initial-element Initial elements value. It will make an array with all the elements initialized to a particular value.
4	:initial-content Initial content as object.
5	:adjustable It helps in creating a resizeable (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.

6	:fill-pointer It keeps track of the number of elements actually stored in a resizeable vector.
7	: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.
8	:displaced-index-offset It gives the index-offset of the created shared array.

15.Write a LISP program to demonstrate the string processing.

Strings in Common Lisp are vectors, i.e., 1-D 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 (\).

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 −

Sr.No.	Function & Description
1	string-upcase Converts the string to upper case
2	string-downcase Converts the string to lower case
3	string-capitalize Capitalizes each word 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 −

Sr.No.	Function & Description
1	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.
2	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.
3	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, we will demonstrate some commonly used functions using various examples.

Calculating Length

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 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 of 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.

For 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.

For 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
```