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 result is:

```
(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.