Unit VI

Logical and Functional Programming

UNIT VI
LOGICAL AND FUNCTIONAL PROGRAMMING

Syllabus

Functional Programming Paradigm: Understanding symbol manipulation, Basic LISP functions, definitions, predicates, conditionals and scoping, Recursion and iteration, Properties List array and access functions, Using lambda definitions, printing, reading and atom manipulation.

Logic Programming Paradigm: An Overview of Prolog, Syntax and Meaning of Prolog Programs, Lists, Operators, Arithmetic, Using Structures.

FUNCTIONAL PROGRAMMING PARADIGM

Functional programming languages are specially designed to handle symbolic computation and list processing applications. Functional programming is based on mathematical functions. Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

Functional programming languages are categorized into two groups, i.e. −

- **Pure Functional Languages** − These types of functional languages support only the functional paradigms. For example − Haskell.
- **Impure Functional Languages** − These types of functional languages support the functional paradigms and imperative style programming. For example − LISP.

Functional Programming – Characteristics

- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports **higher-order functions** and **lazy evaluation** features.
- Functional programming languages don't support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.
- Like OOP, functional programming languages support popular concepts such as Abstraction, Encapsulation, Inheritance, and Polymorphism.

Functional Programming – Advantages

- **Bugs-Free Code** − Functional programming does not support **state**, so there are no side-effect results and we can write error-free codes.
- **Efficient Parallel Programming** − Functional programming languages have NO Mutable state, so there are no state-change issues. One can program "Functions" to work parallel as "instructions". Such codes support easy reusability and testability.
- **Efficiency** − Functional programs consist of independent units that can run concurrently. As a result, such programs are more efficient.
- **Supports Nested Functions** − Functional programming supports Nested Functions.
- **Lazy Evaluation** − Functional programming supports Lazy Functional Constructs like Lazy Lists, Lazy Maps, etc.

Functional Programming – disadvantages

As a downside, functional programming requires a large memory space. As it does not have state, you need to create new objects every time to perform actions.

Functional Programming – Applications

Functional Programming is used in situations where we have to perform lots of different operations on the same set of data.

- Lisp is used for artificial intelligence applications like Machine learning, language processing, Modeling of speech and vision, etc.
- Embedded Lisp interpreters add programmability to some systems like Emacs.

Functional Programming vs. Object-oriented Programming

Functional Programming	OOP
Uses Immutable data.	Uses Mutable data.
Follows Declarative Programming Model.	Follows Imperative Programming Model.
Focus is on: “What you are doing”	Focus is on “How you are doing”
Supports Parallel Programming	Not suitable for Parallel Programming
Its functions have no-side effects	Its methods can produce serious side effects.
Flow Control is done using function calls & function calls with recursion	Flow control is done using loops and conditional statements.

It uses "Recursion" concept to iterate Collection Data.	It uses "Loop" concept to iterate Collection Data. For example: For-each loop in Java
Execution order of statements is not so important.	Execution order of statements is very important.
Supports both "Abstraction over Data" and "Abstraction over Behavior".	Supports only "Abstraction over Data".

Understanding Symbol Manipulations

Symbol Manipulation Is Like Working with Words and Sentences

Everything in a computer is a string of binary digits, ones and zeros, that everyone calls bits. From one perspective, sequences of bits can be interpreted as a code for ordinary decimal digits. But from another perspective, sequences of bits can be interpreted as a code for wordlike objects and sentencelike objects.

- In Lisp, the fundamental things formed from bits are wordlike objects called *atoms*.
- Groups of atoms form sentencelike objects called *lists*. Lists themselves can be grouped together to form higher-level lists.
- Atoms and lists collectively are called *symbolic expressions*, or more succinctly, *expressions*. Working with symbolic expressions is what symbol manipulation using Lisp is about.

A symbol-manipulation program uses symbolic expressions to remember and work with data and procedures, just as people use pencil, paper, and human language to remember and work with data and procedures. A symbol-manipulation program typically has procedures that recognize particular symbolic expressions, tear existing ones apart, and assemble new ones.

Example

The parentheses mark where lists begin and end. It is a description of a certain university.

```
(mit (a-kind-of university)
     (location (cambridge massachusetts))
     (phone 253-1000)
     (schools (architecture
               business
               engineering
               humanities
               science))
     (founder (william barton rogers)))
```

Certainly these are not scary. Both just describe something according to some conventions about how to arrange symbols.

LISP programming

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 effectively.

Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups that were successors to Maclisp, like ZetaLisp and NIL (New Implementation of Lisp) 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, and easy extensibility.
- It allows updating the programs dynamically.
- It provides high level debugging.
- It provides advanced object-oriented programming.
- It provides a convenient macro system.
- It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
- It is expression-based.
- It provides an object-oriented condition system.
- It provides a complete I/O library.
- It provides extensive control structures.

Applications Built in LISP

Large successful applications built in Lisp.

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

Simple Program

- LISP expressions are called symbolic expressions or s-expressions. The s-expressions are composed of three valid objects, atoms, lists and 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.

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.

```
(+ 7 9 11)
```

LISP returns the result −

```
27
```

If you would like to run 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 (+ 7 9 11))
```

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

```
27
```

Basic Building Blocks in LISP

LISP programs are made up of three basic building blocks −

- atom
- list
- string

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

Following are examples of some valid atoms −

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

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

Following are examples of 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)
( )
```

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

Following are examples of 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.

For Example,

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

LISP Functions

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)
```

Let us illustrate the concept with simple examples.

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
```

Example 2

Let's define and call a function that would calculate 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)
```

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

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

Please note that −

- 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.

Predicates

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

Sr.No.	Predicate & Description
1	**atom** It takes one argument and returns t if the argument is an atom or nil if otherwise.
2	**equal** It takes two arguments and returns **t** if they are structurally equal or **nil** otherwise.
3	**eq** It takes two arguments and returns **t** if they are same identical objects, sharing the same memory location or **nil** otherwise.
4	**eql** It takes two arguments and returns **t** if the arguments are **eq**, or if they are numbers of the same type with the same value, or if they are character objects that represent the same character, or **nil** otherwise.
5	**evenp** It takes one numeric argument and returns **t** if the argument is even number or **nil** if otherwise.
6	**oddp** It takes one numeric argument and returns **t** if the argument is odd number or **nil** if otherwise.
7	**zerop** It takes one numeric argument and returns **t** if the argument is zero or **nil** if otherwise.
8	**null** It takes one argument and returns **t** if the argument evaluates to nil, otherwise it returns **nil**.
9	**listp**

	It takes one argument and returns **t** if the argument evaluates to a list otherwise it returns **nil**.
10	**greaterp** It takes one or more argument and returns **t** if either there is a single argument or the arguments are successively larger from left to right, or **nil** if otherwise.
11	**lessp** It takes one or more argument and returns **t** if either there is a single argument or the arguments are successively smaller from left to right, or **nil** if otherwise.
12	**numberp** It takes one argument and returns **t** if the argument is a number or **nil** if otherwise.
13	**symbolp** It takes one argument and returns **t** if the argument is a symbol otherwise it returns **nil**.
14	**integerp** It takes one argument and returns **t** if the argument is an integer otherwise it returns **nil**.
15	**rationalp** It takes one argument and returns **t** if the argument is rational number, either a ratio or a number, otherwise it returns **nil**.
16	**floatp** It takes one argument and returns **t** if the argument is a floating point number otherwise it returns **nil**.
17	**realp** It takes one argument and returns **t** if the argument is a real number otherwise it returns **nil**.
18	**complexp** It takes one argument and returns **t** if the argument is a complex number otherwise it returns **nil.**
19	**characterp** It takes one argument and returns **t** if the argument is a character otherwise it returns **nil**.
20	**stringp** It takes one argument and returns **t** if the argument is a string object otherwise it returns **nil**.
21	**arrayp** It takes one argument and returns **t** if the argument is an array object otherwise it returns **nil**.

22	**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
```

Conditional Statements in LISP

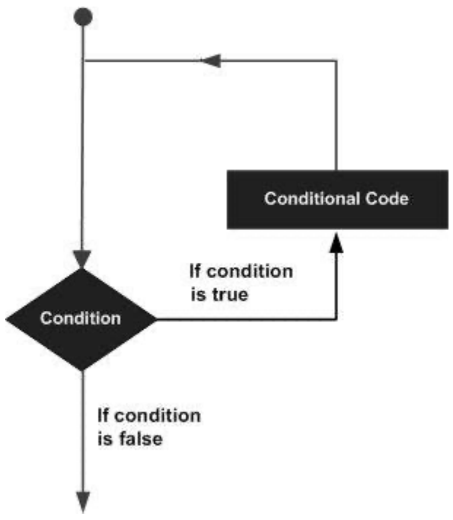
Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined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

LISP Iterations

There may be a situation, when 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 and following is the general form of a loop statement in most of the programming languages.



LISP provides the following types of constructs to handle looping requirements. Click the following links to check their detail.

Sr.No.	Construct & Description	example

1	loop
	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.
		`(setq a 10)`
		`(loop`
		`   (setq a (+ a 1))`
		`   (write a)`
		`   (terpri)`
		`   (when (> a 14) (return a))`
		`)`
		`Output`
		`11`
		`12`
		`13`
		`14`
		`15`

2	loop for
	The loop for construct allows you to implement a for-loop like iteration as most common in other languages.
		`(loop for x in '(tom dick harry)`
		`   do (format t " ~s" x)`
		`)`
		Output
		`TOM DICK HARRY`

3	do	`(do ((x 0 (+ 2 x))`
		`   (y 20 ( - y 2)))`
		`   ((= x y)(- x y))`

	The do construct is also used for performing iteration using LISP. It provides a structured form of iteration.	
		```
		   (format t "~% x = ~d  y = ~d" x y)
		)
		```
		Output
		```
		x = 0  y = 20
		x = 2  y = 18
		x = 4  y = 16
		x = 6  y = 14
		x = 8  y = 12
		```

4	dotimes
	The dotimes construct allows looping for some fixed number of iterations.
		```
		(dotimes (n 5)
		   (print n) (prin1 (* n n))
		)
		```
		Output
		```
		0 0
		1 1
		2 4
		3 9
		4 16
		```

5	dolist
	The dolist construct allows iteration through each element of a list.
		```
		(dolist (n '(1 2 3 4 5))
		   (format t "~% Number: ~d Square: ~d" n (* n n))
		)
		```
		Output
		```
		Number: 1 Square: 1
		 Number: 2 Square: 4
		 Number: 3 Square: 9
		 Number: 4 Square: 16
		 Number: 5 Square: 25
		```

Recursion in Lisp

In pure Lisp there is **no looping**; recursion is used instead. A recursive function is defined in terms of: 1. One or more base cases 2. Invocation of one or more simpler instances of itself. Note that recursion is directly related to mathematical induction.

Example

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)
```

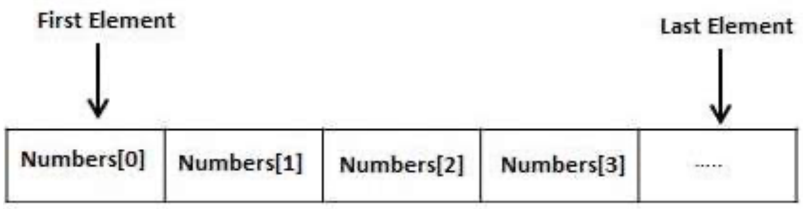
```
    (format t "~% Factorial ~d is: ~d" n (factorial n))
```

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

```
    Factorial 6 is: 720
```

LISP 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.



- 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 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)
```

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))
```

Property Lists

- In LISP, a symbol is a name that represents data objects and interestingly it is also a data object.
- What makes symbols special is that they have a component called the **property list**, or **plist.**
- 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, sex, height, weight, address, profession etc. A property is like 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; 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 and publisher, and respective values, to an object named (symbol) 'book'.

Example

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 found then it returns the corresponding value; otherwise default is returned (or nil, if a default value is not specified).

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
```

Access Functions

Reading Functions in LISP

Reading Input from Keyboard

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

For example, consider the code snippet −

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

Assume the user enters 10.2 from the STDIN Input, it returns,

```
25.2
```

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

Writing/ Printing Functions in LISP

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*.

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
```

Logic Programming Paradigm

It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.
In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

sum of two number in prolog:

```
  predicates
  sumoftwonumber(integer, integer)
clauses
  sum(0, 0).
   sum(n, r):-
        n1=n-1,
        sum(n1, r1),
        r=r1+n
```

Prolog

Prolog or **PRO**gramming in **LOG**ics is a logical and declarative programming language. It is one major example of the fourth generation language that supports the declarative programming paradigm. This is particularly suitable for programs that involve **symbolic** or **non-numeric computation**. This is the main reason to use Prolog as the programming language in **Artificial Intelligence**, where **symbol manipulation** and **inference manipulation** are the fundamental tasks.

In Prolog, we need not mention the way how one problem can be solved, we just need to mention what the problem is, so that Prolog automatically solves it. However, in Prolog we are supposed to give clues as the **solution method**.

Prolog language basically has three different elements −

- **Facts** − The fact is predicate that is true, for example, if we say, "Tom is the son of Jack", then this is a fact.
- **Rules** − Rules are extinctions of facts that contain conditional clauses. To satisfy a rule these conditions should be met. For example, if we define a rule as −
  `grandfather(X, Y) :- father(X, Z), parent(Z, Y)`
  This implies that for X to be the grandfather of Y, Z should be a parent of Y and X should be father of Z.
- **Questions** − And to run a prolog program, we need some questions, and those questions can be answered by the given facts and rules.

Facts

We can define fact as an explicit relationship between objects, and properties these objects might have. So facts are unconditionally true in nature. Suppose we have some facts as given below −

- Tom is a cat
- Kunal loves to eat Pasta
- Hair is black
- Nawaz loves to play games
- Pratyusha is lazy.

So these are some facts, that are unconditionally true. These are actually statements, that we have to consider as true.

Following are some guidelines to write facts −

- Names of properties/relationships begin with lower case letters.
- The relationship name appears as the first term.
- Objects appear as comma-separated arguments within parentheses.
- A period "." must end a fact.
- Objects also begin with lower case letters. They also can begin with digits (like 1234), and can be strings of characters enclosed in quotes e.g. color(penink, 'red').
- phoneno(agnibha, 1122334455). is also called a predicate or clause.

Syntax

The syntax for facts is as follows −

```
relation(object1,object2...).
```

Example

Following is an example of the above concept −

```
    cat(tom).
    loves_to_eat(kunal,pasta).
    of_color(hair,black).
    loves_to_play_games(nawaz).
    lazy(pratyusha).
```

Rules

We can define rule as an implicit relationship between objects. So facts are conditionally true. So when one associated condition is true, then the predicate is also true. Suppose we have some rules as given below −

- Lili is happy if she dances.
- Tom is hungry if he is searching for food.
- Jack and Bili are friends if both of them love to play cricket.
- will go to play if school is closed, and he is free.

So these are some rules that are **conditionally** true, so when the right hand side is true, then the left hand side is also true.

Here the symbol (:-) will be pronounced as "If", or "is implied by". This is also known as neck symbol, the LHS of this symbol is called the Head, and right hand side is called Body. Here we can use comma (,) which is known as conjunction, and we can also use semicolon, that is known as disjunction.

Syntax

```
     rule_name(object1, object2, ...) :- fact/rule(object1,
      object2, ...)
     Suppose a clause is like :
     P :- Q;R.
     This can also be written as
     P :- Q.
     P :- R.

     If one clause is like :
     P :- Q,R;S,T,U.

     Is understood as
     P :- (Q,R);(S,T,U).
     Or can also be written as:
     P :- Q,R.
     P :- S,T,U.
```

Example

```
     happy(lili) :- dances(lili).
     hungry(tom) :- search_for_food(tom).
     friends(jack, bili) :- lovesCricket(jack), lovesCricket(bili).
     goToPlay(ryan) :- isClosed(school), free(ryan).
```

Queries

Queries are some questions on the relationships between objects and object properties. So question can be anything, as given below −

- Is tom a cat?
- Does Kunal love to eat pasta?
- Is Lili happy?
- Will Ryan go to play?

So according to these queries, Logic programming language can find the answer and return them.

Some Applications of Prolog

Prolog is used in various domains. It plays a vital role in automation system. Following are some other important fields where Prolog is used −

- Intelligent Database Retrieval
- Natural Language Understanding
- Specification Language
- Machine Learning
- Robot Planning
- Automation System
- Problem Solving

Operators in Prolog

1. **Comparison Operators**

Comparison operators are used to compare two equations or states.

Operator	**Meaning**
X > Y	X is greater than Y
X < Y	X is less than Y
X >= Y	X is greater than or equal to Y
X =< Y	X is less than or equal to Y
X =:= Y	the X and Y values are equal
X =\= Y	the X and Y values are not equal

You can see that the '=<' operator, '=:=' operator and '=\=' operators are syntactically different from other languages. Let us see some practical demonstration to this.

Example

```
| ?- 1+2=:=2+1.
yes
| ?- 1+2=2+1.
no
| ?- 1+A=B+2.
A = 2
B = 1
yes
| ?- 5<10.
```

```
yes
| ?- 5>10.
no
| ?- 10=\=100.
yes
```

Here we can see 1+2=:=2+1 is returning true, but 1+2=2+1 is returning false. This is because, in the first case it is checking whether the value of 1 + 2 is same as 2 + 1 or not, and the other one is checking whether two patterns '1+2' and '2+1' are same or not. As they are not same, it returns no (false). In the case of 1+A=B+2, A and B are two variables, and they are automatically assigned to some values that will match the pattern.

Arithmetic Operators in Prolog

Arithmetic operators are used to perform arithmetic operations.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Power
//	Integer Division
mod	Modulus

Let us see one practical code to understand the usage of these operators.

Program

```
calc :- X is 100 + 200,write('100 + 200 is '),write(X),nl,
        Y is 400 - 150,write('400 - 150 is '),write(Y),nl,
        Z is 10 * 300,write('10 * 300 is '),write(Z),nl,
        A is 100 / 30,write('100 / 30 is '),write(A),nl,
        B is 100 // 30,write('100 // 30 is '),write(B),nl,
        C is 100 ** 2,write('100 ** 2 is '),write(C),nl,
        D is 100 mod 30,write('100 mod 30 is '),write(D),nl.
```

Note − The nl is used to create new line.

Output

```
| ?- change_directory('D:/TP Prolog/Sample_Codes').

yes
| ?- [op_arith].
compiling D:/TP Prolog/Sample_Codes/op_arith.pl for byte code...
D:/TP Prolog/Sample_Codes/op_arith.pl compiled, 6 lines read -
2390 bytes written, 11 ms

yes
| ?- calc.
100 + 200 is 300
400 - 150 is 250
10 * 300 is 3000
100 / 30 is 3.3333333333333335
100 // 30 is 3
100 ** 2 is 10000.0
100 mod 30 is 10

yes
| ?-
```

Prologs List

It is a data structure that can be used in different cases for non-numeric programming. Lists are used to store the atoms as a collection.

The list is a simple data structure that is widely used in non-numeric programming. List consists of any number of items, for example, red, green, blue, white, dark. It will be represented as, [red, green, blue, white, dark]. The list of elements will be enclosed with **square brackets**.

A list can be either **empty** or **non-empty**. In the first case, the list is simply written as a Prolog atom, []. In the second case, the list consists of two things as given below −

- The first item, called the **head** of the list;
- The remaining part of the list, called the **tail**.

Suppose we have a list like: [red, green, blue, white, dark]. Here the head is red and tail is [green, blue, white, dark]. So the tail is another list.

Now, let us consider we have a list, L = [a, b, c]. If we write Tail = [b, c] then we can also write the list L as L = [a | Tail]. Here the vertical bar (|) separates the head and tail parts.

So the following list representations are also valid −

- `[a, b, c] = [x | [b, c] ]`
- `[a, b, c] = [a, b | [c] ]`
- `[a, b, c] = [a, b, c | [ ] ]`

For these properties we can define the list as −

A data structure that is either empty or consists of two parts − a head and a tail. The tail itself has to be a list.

Basic Operations on Lists

Operations	**Definition**
Membership Checking	During this operation, we can verify whether a given element is member of specified list or not?
Length Calculation	With this operation, we can find the length of a list.
Concatenation	Concatenation is an operation which is used to join/add two lists.
Delete Items	This operation removes the specified element from a list.
Append Items	Append operation adds one list into another (as an item).
Insert Items	This operation inserts a given item into a list.

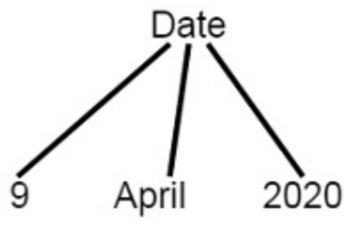
Structures in prolog

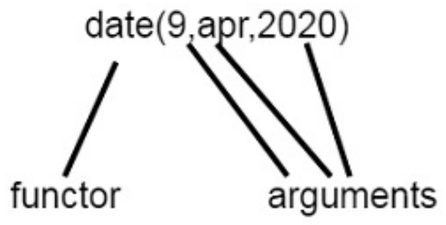
Structures are Data Objects that contain multiple components.

For example, the date can be viewed as a structure with three components — day, month and year. Then the date 9th April, 2020 can be written as: date(9, apr, 2020).

Note − Structure can in turn have another structure as a component in it.

So we can see views as tree structure and **Prolog Functors**.





Now let us see one example of structures in Prolog. We will define a structure of points, Segments and Triangle as structures.

To represent a point, a line segment and a triangle using structure in Prolog, we can consider following statements −

- p1 − point(1, 1)
- p2 − point(2,3)
- S − seg(Pl, P2): seg(point(1,1), point(2,3))
- T − triangle(point(4,Z), point(6,4), point(7,1))

Note − Structures can be naturally pictured as trees. Prolog can be viewed as a language for processing trees.

Comparisons between Functional programming and Logical Programming

Functional Programming	Logical Programming
It is totally based on functions.	It is totally based on formal logic.
In this programming paradigm, programs are constructed by applying and composing functions.	In this programming paradigm, program statements usually express or represent facts and rules related to problems within a system of formal logic.
These are specially designed to manage and handle symbolic computation and list processing applications.	These are specially designed for fault diagnosis, natural language processing, planning, and machine learning.
Its main aim is to reduce side effects that are accomplished by isolating them from the rest of the software code.	Its main aim is to allow machines to reason because it is very useful for representing knowledge.
Some languages used in functional programming include Clojure, Wolfram Language, Erland, OCaml, etc.	Some languages used for logic programming include Absys, Cycl, Alice, ALF (Algebraic logic functional programming language), etc.

Functional Programming	Logical Programming
It reduces code redundancy, improves modularity, solves complex problems, increases maintainability, etc.	It is data-driven, array-oriented, used to express knowledge, etc.
It usually supports the functional programming paradigm.	It usually supports the logic programming paradigm.
Testing is much easier as compared to logical programming.	Testing is comparatively more difficult as compared to functional programming.
It simply uses functions.	It simply uses predicates. Here, the predicate is not a function i.e., it does not have a return value.

****