**Paradigm** can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach.

**Imperative programming paradigm**: It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal.

Advantage: Very simple to implement, It contains loops, variables etc.

Disadvantage: Complex problem cannot be solved, Less efficient and less productive, Parallel programming is not possible

Imperative categories - Procedural, OOP and parallel processing.

* **Procedural programming paradigm**: This paradigm emphasizes on procedure in terms of under lying machine model.
* **Object oriented programming**: The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. Advantages: Data security, Inheritance, Code reusability, abstraction.
* **Parallel processing approach**: Parallel processing is the processing of program instructions by dividing them among multiple processors. A parallel processing system posses many numbers of processor with the objective of running a program in less time by dividing them.

Declarative programming paradigms: The focus is on what needs to be done rather how it should be done basically emphasize on what code code is actually doing. It just declares the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms.

Declarative categories: Logic, Functional, Database.

**Logic programming paradigm**: 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 logical programming the main emphasize is on knowledge base and the problem. Eg prolog.

**Functional programming paradigm**: The functional programming paradigms has its roots in mathematics and it is language independent. The key principal of this paradigms is the execution of series of mathematical functions. Eg perl, haskell

**Database**: This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. Eg SQL.

**Glasgow Haskell Compiler (GHC)** - for real applications

Compiles to native code, Supports parallel execution, and Provides useful performance analysis and debugging tools.

GHC Has three main components:

**ghc**- An optimizing compiler that generates fast native code

**ghci** - An interactive interpreter and debugger

**runghc** - A program for running Haskell programs as scripts, without needing to compile them first

standard library of Haskell: prelude

To use definitions from other modules, we must load them into ghci, using the **:module** command:

ghci> :module + Data.Ratio

or ghci> :m + Data.Ratio (:m is the short form for :module command)

Expressions are written in infix and prefix form, where an operator appears between its operands.

Eg 2 + 2, (+) 2 2

Boolean values: True, False

Logical operators: &&, ||, not

Haskell assigns numeric precedence values to operators.lowest precedence is 1 and Highest precedence is 9. A higher-precedence operator is applied before a lower precedence operator in an expression.

ghci’s :**info** command can be used to check the precedence level of individual operators

Combination of precedence and associativity rules are usually referred to as ***fixity*** Rules. The (+) and (\*) operators are left associative, which is represented as infixl in :info. Right associative is infixr.

String is actually a list of characters

A list is surrounded by square brackets; the elements are separated by commas

All elements of a list must be of the same type

Haskell fills in the rest of the items in the list when **enumeration notation ..** is used.

**Concatenation operator** (++): [1,2,3] ++ [4,5] = [1,2,3,4,5]

**Cons (construct) operator** (:) : Adds an element to the front of a list. The first argument of (:) must be an element, and the second must be a list. Eg 1:[2,3] = [1,2,3]

Variable names must start with lowercase letters

To see type of a variable: use **set +t**

The extra type information can be turned off at any time, using the **unset +t** command

The word **“it’”** is the name of a special variable, in which ghci (not Haskell ! ) stores the result of the last expression we evaluated.

Haskell’s integer type is named **Num**

To construct a rational number, we use the **(%)** operator. Type = Ratio

There are three interesting aspects to types in Haskell

They are strong, They are static and They can be automatically inferred

**Strong types**: The type system guarantees that a program cannot contain certain kinds of errors which doesn’t make sense such as Using an integer as a function or If a function expects to work with integers and if a string is passed, it will be rejected by the Haskell compiler. The benefit of strong typing is that it catches real bugs in our code before they can cause problems

The ability of a Haskell compiler to automatically deduce the types of almost all expressions in a program is known as **Type Inference**

A **static type system** means that the compiler knows the type of every value and expression at compile time

Haskell’s combination of strong and static typing makes it impossible for type errors to occur at runtime.

**take** - Given a number n and a list, take returns the first n elements of the list

**drop** -returns all but the first n elements of the list

For tuples, the **fst** and **snd** functions return the first and second element of a pair, respectively:

**Function signature**: gives a hint as to what the function might actually do

**Function composition**: Suppose f :: Y -> Z and g :: X -> Y are two given functions. We can combine them into a new function f . g :: X -> Z. The order of composition is from right to left because we write functions to the left of the arguments to which they are applied. One of the uses for function composition is making functions on the fly to pass to other functions. Function composition is clearer and more concise than lambda. Function composition is right-associative => f (g (z x)) is equivalent to (f . g . z) x

**Side Effect**: Introduces a dependency between the global state of the system and the behavior of a function. Consider a function that reads and returns the value of a global variable.

If some other code can modify that global variable, then the result of a particular application of our function depends on the current value of the global variable. The function has a side effect, even though it never modifies the variable itself. Side effects are essentially invisible inputs to, or outputs from, functions

In Haskell, the default is for functions to not have side effects: the result of a function depends only on the inputs that we explicitly provide

**Pure Functions**: Functions without Side effects

**Impure Functions**: Functions with Side effects. Identifying an Impure Function: By reading the type signature, The type of the function’s result will begin with IO

Haskell doesn’t have a return keyword, because a function is a single expression, not a sequence of statements. The value of the expression is the result of the function.

In languages that uses **strict evaluation,** the arguments to a function are evaluated before the function is applied. Haskell uses **lazy evaluation**. Lazy evaluation does not evaluate function arguments unless their values are required to evaluate the function call itself.

**Thunk**: The record used to track an unevaluated expression. A thunk is created and the actual evaluation is deferred until it is really needed

On the plus side, Lazy evaluation terminates whenever any reduction order terminates; it never takes more steps than eager evaluation, and sometimes fewer. On the minus side, it can require a lot more space and it is more difficult to understand the precise order in which things happen

three kinds of lists:

* A **finite list**, which is built from (:) and []; for example, 1:2:3:[]
* An **infinite list**, which is built from (:) alone; for example, [1..] is the infinite list of the non-negative integers.
* A **partial list**, which is built from (:) and undefined. For example, the list filter (<4) [1..] is the partial list 1:2:3:undefined

**List comprehension**: used for building more specific lists out of general lists.

**Predicate** is the condition which has to be satisfied for a particular element to appear in output function. Weeding out lists by predicates is also called **filtering.**

The **zip** function takes two lists and “zips” them into a single list of pairs. The resulting list is the same length as the shorter of the two inputs

zip :: [a] -> [b] -> [(a,b)]

zip (x:xs) (y:ys) = (x,y): zip xs ys

zip \_ \_ = []

**zipWith** takes two lists and applies a function to each pair of elements, generating a list that is the same length as the shorter of the two

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys

zipWith f \_ \_ = []

**Init**: takes a list and returns everything except its last element

**Reverse**: reverses a list

**maximum** takes a list of stuff that can be put in some kind of order and returns the biggest element and **minimum** returns the smallest.

**Sum** takes a list of numbers and returns their sum

**Product** takes a list of numbers and returns their product

**elem** takes an element and a list of items and says if the element is an item in the list. 4 `elem` [3,4,5,6] => True

**cycle** takes a list and cycles it into an infinite list. If you just try to display the result, it will go on forever so you have to slice it off somewhere. take 10 (cycle [1,2,3]) => [1,2,3,1,2,3,1,2,3,1]

**repeat** takes an element and produces an infinite list of just that element. It's like cycling a list with only one element. take 10 (repeat 5) => [5,5,5,5,5,5,5,5,5,5]

**replicate** function also replicates some number of the same element in a list. replicate 3 10 =>[10,10,10]

**Pattern Matching** is process of matching specific type of expressions.

\_ means don’t care

Patterns are a way of making sure a value conforms to some form and deconstructing it

**Guards** are a way of testing whether some property of a value (or several of them) are true or false. Guards are similar to if statements. Guards are lot more readable to match several conditions. Guards are indicated by pipes that follow a function's name and its parameters. Usually, they're indented a bit to the right and lined up.

A guard is basically a boolean expression. If it evaluates to True, then the corresponding function body is used.

If it evaluates to False, checking drops through to the next guard and so on.

The **Ord** class is used for types that have an ordering. Ord covers all the standard comparing functions such as >, <, >= and <=. The compare function takes two Ord members of the same type and returns an ordering. Ordering is a type that can be GT, LT or EQ, meaning greater than, lesser than and equal, respectively.

**Where** is a keyword or inbuilt function that can be used at runtime to generate a desired output.

Let Bindings are similar to where bindings. Normally, bindings are a syntactic construct that let you bind to variables at the end of a function and the whole function can see them, including all the guards. Let bindings let you bind to variables anywhere and are expressions themselves, but are very local, so they don't span across guards, whereas, **where** isn’t local. Let bindings are expressions themselves, where bindings are just syntactic constructs.

**Recursion** is actually a way of defining functions in which the function is applied inside its own definition. Recursion is important to Haskell because unlike imperative languages, computations are done in Haskell by declaring what something is instead of declaring how to get it. There are no while loops or for loops in Haskell and instead we many times have to use recursion to declare what something is.

Functions that can take functions as parameters and return functions as return values are called **higher order functions.** Every function in Haskell officially only takes one parameter

**Currying** is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments

**map** is a function that takes two parameters: a function and a list of elements. It will apply the function f to all elements in the list.

**Filter** is a function that takes a predicate and a list and then returns the list of elements that satisfy the predicate

**Lambdas** are anonymous functions. Used when some functions are needed only once Normally lambdas are made with the sole purpose of passing it to a higher-order function. Eg: The expression (\xs -> length xs > 15) returns a function that tells us whether the length of the list passed to it is greater than 15. Lambdas can take any number of parameters.

**fold** (or reduce) is a family of higher order functions that process a list in some order and build a return value. It takes a binary function, a starting value (accumulator) and a list to fold up.

**FOLDL** Folds the list up from the left side The binary function is applied between the starting value and the head of the list. the left fold's binary function has the accumulator as the first parameter and the current value as the second one (so \acc x -> .)

**FOLDR** works in a similar way to the left fold except that the accumulator eats up the values from the right. the right fold's binary function has the current value as the first parameter and the accumulator as the second one (so \x acc -> ...)right fold has the accumulator on the right, because it folds from the right side.

Right folds work on infinite lists, whereas left ones don't

The **FOLDL1** and **FOLDR1** functions work much like FOLDL and FOLDR, only you don't need to provide them with an explicit starting value.

They assume the first (or last) element of the list to be the starting value and then start the fold with the element next to it.

**$$$$$$** A normal function application (putting a space between two things) has a really high precedence, the **$** function has the lowest precedence. Function application with a space is left-associative (so f a b c is the same as ((f a) b) c)) whereas function application with $ is right-associative. sqrt 3 + 4 + 9 is evaluated as (((sqrt 3) + 4) + 9)

To get sqrt of (3+4+9 = 16) we must use as sqrt (3+4+9)