Functional Programming

- Functional programming is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects.
- Functional programming is declarative (telling the computer what you want to do) rather than imperative (telling the computer exactly how to do that), and application state flows through pure functions.
- 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,
- 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.
- In programming terms, a function is a block of statements that performs a specific task.
- Functions accept data, process it, and return a result.
- Functions are written primarily to support the concept of reusability.
- Once a function is written, it can be called easily, without having to write the same code again and again.
- Why Functional Programming?
- It's generally more concise
- It's generally more predictable
- It's easier to test than object-oriented code

Lambda Calculus

- Lambda calculus is a framework developed by Alonzo Church in 1930s to study computations with functions.
- The λ calculus can be called the smallest universal programming language of the world.
- The central concept in the lambda calculus is an expression which we can think of as a program that when evaluated returns a result consisting of another lambda calculus expression. Ex: λx. (+ x 1)

  expr → λ variable . expr | expr expr | variable | (expr) | constant

– A variable is an identifier or “name” can be any of the letters a, b, c,…

– A constant is a built-in function such as an integer or boolean.

– The λ calculus consists of a single transformation rule (variable substitution) and a single function definition scheme

– An expression is defined recursively as follows:

  <expression>:= <name> | <function> | <application>

  <function>:= λ <name>.<expression>

  <application>:= <expression><expression>

– λ expression with a single identifier is called as the identity function **λx.x**

  Function Abstraction or Creation

- A function abstraction, often called a lambda abstraction, is a lambda expression that defines a function.
- A function abstraction: λx.expr consists of four parts, a lambda followed by a variable, a period, and then an expression (Body).
- In the function abstraction λx.expr the variable x is the formal parameter of the function also called placeholder and expr is the body of the function.
- For example, the function abstraction λx. + x 1 defines a function of x that adds x to 1. Parentheses can be added to lambda expressions

for clarity. Thus, we could have written this function abstraction as λx.(+ x 1) or even as (λx. (+ x 1)).

- In C this function definition might be written as int addOne (int x){ return (x + 1); }
- Unlike C, the lambda abstraction does not give a name to the function. The lambda expression itself is the function and have single argument.
- λx.expr binds the variable x in expr and that expr is the scope of the variable.

Function Application

- Function application −
- A function application, often called a lambda application, consists of an expression followed by an expression: expr expr.
  - The notation E1.E2 to denote the application of function E1 to actual argument E2.
- The first expression is a function abstraction and the second expression is the argument to which the function is applied.
- Expressions can be thought of as programs in the language of lambda calculus.
- All functions in lambda calculus have exactly one argument.
- Multiple-argument functions are represented by currying
  - For example, the lambda expression λx. (+ x 1) 2 is an application of the function λx. (+ x 1) to the argument 2.
  - This function application λx. (+ x 1) 2 can be evaluated by substituting the argument 2 for the formal parameter x in the body (+ x 1).
  - Doing this we get (+ 2 1). This substitution is called a beta reduction.

– Beta reductions are like macro substitutions in C. To do beta reductions correctly we may need to rename bound variables in lambda expressions to avoid name clashes.

- Function application associates left-to-right; thus, f g h = (f g)h.
- Function application binds more tightly than λ; thus, λx. f g x = (λx. (f g)x).

– Multiple expressions: E1E2E3 . . . En (. . .((E1E2)E3). . . En)

- Functions in the lambda calculus are first-class citizens; that is to say, functions can be used as arguments to functions and functions can return functions as results.

Free and bound variables

- Bound variables:
  – In the function definition λx.x the variable x in the body of the definition (the second x) is bound because its first occurrence in the definition is λx where x is preceded by λ.
- Free variables:
  – A variable or a name not preceded by a λ is called a "free variable".
  – In the function (λx.xy), the variable x in the body of the function is

bound and the variable y is free.

- Ex: In the expression (λx.x)(λy.yx):
  – The variable x in the body of the leftmost expression is bound to the first lambda.
  – The variable y in the body of the second expression is bound to the second lambda.
  – The variable x in the body of the second expression is free.

- The x in second expression is independent of the x in first expression.

Practice Problems

1.Lambda calculus Make all parentheses explicit in the following

λ-expressionsa.λx.xz λy.xy (λx.((x z) (λy.(x y))))

b.(λx.xz)λy.wλw.wyzx ((λx.(x z)) (λy.(w (λw.((((w y) z) x))))))

c.λx.xyλx.yx (λx.((x y) (λx.(y x))))

Find all free (unbound) variables in the following λ-expressions

d. λx.x zλy.x y (λx.((xz) (λy.(x y))))

e. (λx. x z)λy. wλw. w y z x ((λx.(xz)) (λy.(w(λw.((((w y)z)x))))))

f. λx. x yλx. y x (λx.((xy) (λx.(yx))))

Q1. Explicit Parenthesis Make the parentheses in the following lambda expressions explicit: λx. x y λy. y y z Note: You may use λ, \, or L to denote the lambda symbol.

(λx. ((x y) (λy. (y y) z)))

Correctly parenthesize each of these lambda expressions:

a) λx . x λy . y x

Make all parentheses explicit in the following expressions:

λx.xz λy.xy

(λx.xz λy.xy)

(λx.(xz λy.xy))

(λx.(xz (λy.xy)))

(λx.((xz) (λy.xy)))

(λx.((xz) (λy.(xy))))

Correctly parenthesize each of these lambda expressions:

a) λx . x λy . y x
b) (λx . x) (λy . y) λx . x (λy . y) z
c) (λf . λy . λz . f z y z) p x
d) λx . x λy . y λz . z λw . w z y x

Find the set of free variables for each of the following lambda expressions:

a) λx . x y λz . x z
b) (λx . x y) λz . w λw . w z y x
c) x λz . x λw . w z y
d) λx . x y λx . y x