**1. Distinctive Features of Functional Programming Languages**

**Functional programming (FP) is a paradigm where programs are built by composing pure functions.**

**Key Features:**

- **Pure Functions: No side effects, output depends only on input.**
- **Immutability: Data cannot be changed once created.**
- **First-class functions: Functions can be stored in variables, passed as arguments, or returned.**
- **Declarative style: Focuses on *what* to solve, not *how to solve*.**
- **Recursion instead of loops: Repeated tasks are handled via recursive functions.**
- **Lazy evaluation: Expressions are evaluated only when needed.**
- **Higher-order functions: Functions that take other functions as arguments or return them.**

✅ **JS Example (pure vs impure)**

```
// Pure function
function add(a, b) {
  return a + b;  // Always same output for same input
}


// Impure function (depends on external state)
let counter = 0;
function increase() {
  counter++;     // Side effect
  return counter;
}
```

---

**2. Functional Programming in Imperative Languages**

- **Even though JavaScript is multi-paradigm (supports OOP + imperative + functional), we can write in FP style.**
- **Imperative style changes *state* step by step, while FP style avoids mutation.**

✅ **JS Example**

```
// Imperative style
let nums = [1, 2, 3, 4];
```

```
let squares1 = [];
for (let i = 0; i < nums.length; i++) {
  squares1.push(nums[i] * nums[i]);
}

// Functional style
let squares2 = nums.map(x => x * x);

console.log(squares1); // [1, 4, 9, 16]
console.log(squares2); // [1, 4, 9, 16]
```

---

### 3. Recursion

- A function that calls itself until a base condition is met.
- Used instead of loops in FP.

✅ JS Example

```
// Factorial using recursion
function fact(n) {
  if (n === 0) return 1;
  return n * fact(n - 1);
}

console.log(fact(5)); // 120
```

---

### 4. Tail Recursion

- Special form of recursion where the recursive call is the last operation.
- Helps optimize memory (no need to keep previous stack frames).

✅ JS Example

```
function factTail(n, acc = 1) {
  if (n === 0) return acc;
  return factTail(n - 1, n * acc); // tail recursive
}
```

**console.log(factTail(5)); // 120**

---

**5. Higher Order Functions (HOFs)**

- **Functions that take other functions as arguments or return them.**

✅ **JS Example**

**// Map, Filter, Reduce (common HOFs)**

let nums = [1, 2, 3, 4, 5];

**// map**

let doubled = nums.map(x => x * 2);

**// filter**

let evens = nums.filter(x => x % 2 === 0);

console.log(doubled); // [2, 4, 6, 8, 10]

console.log(evens);   // [2, 4]

**What is Lazy Evaluation?**

- **Lazy Evaluation** (also called *call-by-need*) is a strategy where **expressions are not evaluated until their values are actually required**.
- Instead of computing everything immediately (*eager evaluation*), the program delays computation, which can improve efficiency.

---

**2. Key Points**

1. **Avoids unnecessary computation**: If a value is never used, it is never computed.
2. **Supports infinite data structures**: Lazy evaluation allows working with potentially infinite lists (streams).
3. **Improves performance**: Saves time if not all branches of a program are needed.
4. **Memoization**: Once computed, results are stored, so repeated requests don't recompute.

---

**4. Example in Haskell (True Lazy Language)**

```
-- Infinite list of natural numbers
naturals = [1..]

-- Take first 5 numbers (Haskell computes only as needed)
main = print (take 5 naturals)   -- [1,2,3,4,5]
```

🔹 Here, [1..] is an **infinite list**, but Haskell only computes first 5 elements because of lazy evaluation.

**Types in Functional Programming**

---

### 1. What are Types?

- A **type** defines the *kind of data* a value or function can hold or operate on.
- In Functional Programming (FP), types ensure:
  - **Correctness**: Functions only work on valid data.
  - **Predictability**: Behavior is fixed by type.
  - **Safety**: Many errors are caught at compile-time (in strongly typed FP languages).

👉 Example: An Int type only stores integers, a Bool only stores True or False.

---

### 2. Importance of Types in FP

1. **Express program behavior** clearly.
2. **Avoid runtime errors** by catching mismatches at compile time.
3. **Encourage modular design** (functions defined by type signatures).
4. **Enable polymorphism** (general functions that work for multiple types).
5. **Basis for reasoning** in lambda calculus & proofs.

---

### 3. Type Systems in FP

#### (a) Primitive Types

Basic data types like Int, Float, Char, Bool.

✅ **Haskell Example**

```
x :: Int
x = 10

y :: Bool
y = True
```

✅ **JavaScript (dynamic, no explicit types)**

```
let x = 10;      // number
let y = true;    // boolean
```

---

**(b) Function Types**

Functions themselves have types → (InputType -> OutputType).

✅ **Haskell Example**

```
add :: Int -> Int -> Int
add x y = x + y
```

✅ **JavaScript Example**

```
const add = (x, y) => x + y;  // works for numbers
```

---

**(c) Composite Types**

- **Tuples**: Group multiple values of different types.
- **Lists**: Ordered collection of same type.

✅ **Haskell Example**

```
pair :: (Int, String)
pair = (10, "hello")

nums :: [Int]
nums = [1, 2, 3, 4]
```

✅ **JavaScript Example**

```
let pair = [10, "hello"];   // tuple-like (array with mixed types)
let nums = [1, 2, 3, 4];    // array
```

---

#### (d) Polymorphic Types

A function that works for *any type*.

✅ **Haskell Example**

```haskell
-- works for any type 'a'
identity :: a -> a
identity x = x
```

✅ **JavaScript Example**

```javascript
const identity = x => x;   // works for any type
console.log(identity(5));      // 5
console.log(identity("hi"));   // hi
```

---

#### (e) Algebraic Data Types (ADTs)

Types built using combinations of other types.

- **Sum types** (either/or)
- **Product types** (and)

✅ **Haskell Example**

```haskell
-- Sum type (can be Int OR Bool)
data MyType = number Int | truth Bool


-- Product type (record with both fields)
data Person = Person String Int
```

⚠️ JavaScript doesn't have ADTs directly, but objects/union types in TypeScript simulate them.

---

#### (f) Type Inference

- Some FP languages (like Haskell, ML) **automatically deduce types** even if not written.

✅ **Haskell Example**

```haskell
double x = 2 * x   -- compiler infers: Int -> Int (if used with Ints)
```

---

### 4. Strong vs Weak Typing in FP

- **Haskell**: Strongly typed, static → all types checked before execution.

- **JavaScript**: Weakly typed, dynamic → types decided at runtime.

Example in JS (danger):

console.log(5 + "5"); // "55" (string concatenation, not numeric addition)

---

#### 5. Types in Lambda Calculus (Mathematical FP Basis)

- **Simply Typed Lambda Calculus** introduces types to pure λ-expressions.
- Example:
  - Identity: λx: Int. x
  - Function: (λx: Int. x+1) : Int -> Int

---

#### 6. Advantages of Types in FP

1. Prevents errors early.
2. Helps in reasoning and proofs.
3. Enables abstraction & polymorphism.
4. Guides program design.

## 📘 Mathematics of Functional Programming: Lambda Calculus

---

#### 1. Introduction

- **Lambda Calculus (λ-calculus)** is a **formal system in mathematical logic** developed by Alonzo Church (1930s).
- It is the **theoretical foundation** of Functional Programming (FP).
- In λ-calculus, *everything is a function*.
- FP languages like **Haskell** are based directly on λ-calculus principles.

---

#### 2. Components of Lambda Calculus

Lambda calculus has only **three constructs**:

1. **Variables** → placeholders for values.
   Example: x, y, z.
2. **Abstraction** → function definition using λ.

- Syntax: λx. expression
- Meaning: "a function of x that returns expression".

Example: λx. x+1 means a function that adds 1 to its input.

3. **Application** → applying a function to an argument.
   - Syntax: (f a)
   - Meaning: apply function f to argument a.

Example: (λx. x+1) 5 → 6.

---

**3. Rules of Lambda Calculus**

1. **α-conversion (Renaming variables)**
   - Variables can be renamed to avoid clashes.
   - Example: λx. x+1 ≡ λy. y+1.
2. **β-reduction (Function application)**
   - Applying a function to an argument by substituting the variable with the value.
   - Example: (λx. x*2) 3 → 6.
3. **η-conversion (Extensionality)**
   - A function λx. f x is equivalent to f if x is not free in f.
   - Example: λx. (add x) ≡ add.

---

**4. Examples of Lambda Calculus**

**Example 1: Identity Function**

-- Haskell

identity x = x

-- JS

const identity = x => x;

console.log(identity(5)); // 5

λ-calculus: λx. x

---

**Example 2: Increment Function**

inc x = x + 1

λ-calculus: λx. x+1
Application: (λx. x+1) 5 → 6

---

**Example 3: Function Composition**

compose f g x = f (g x)

-- JS

const compose = (f, g) => x => f(g(x));

λ-calculus: λf. λg. λx. f (g x)

---

**Example 4: Boolean Logic in λ-calculus**

- TRUE: λx. λy. x
- FALSE: λx. λy. y
- AND: λp. λq. p q p

-- Haskell

true  x y = x

false x y = y

A **lambda expression** in Haskell is an **anonymous function** (a function without a name).
It is written using a **backslash (\)** followed by parameters, an arrow (->), and then the function body.

**General Syntax:**

\x y -> expression

- \ means *lambda (anonymous function)*.
- x y are the parameters.
- -> separates parameters from the function body.
- expression is the computation.

---

**Example: Lambda function to add two numbers**

```haskell
main :: IO ()

main = do

  let add = \x y -> x + y   -- lambda function

  print (add 5 7)
```

A **side effect** happens when a function does something **other than just returning a value**.
Examples:

- Modifying a variable outside its scope

**JavaScript Examples**

❌ **Impure Function (with side effects)**

```javascript
let count = 0;  // global variable


function increment() {

  count++;  // modifies external state (side effect)

  console.log("Count is now:", count); // side effect (printing)

  return count;

}
```

increment(); // Count is now: 1

increment(); // Count is now: 2

- Changes count (global state).
- Prints to console (another side effect).
- Same input → different results.

---

✅ **Pure Function (no side effects)**

function add(a, b) {

 return a + b;  // only depends on inputs

}

console.log(add(3, 4)); // 7

console.log(add(3, 4)); // 7 (always same)

- No external state change.
- No printing/logging.
- Always predictable.

🔹 **First-Class Functions**

In JavaScript, **functions are treated as first-class citizens**.
That means:

1. You can **store functions in variables**.
2. You can **pass functions as arguments** to other functions.
3. You can **return functions** from other functions.
4. You can **store them in data structures** (like arrays, objects).

👉 **Example 1: Store in a variable**

const greet = function(name) {

 return "Hello, " + name;

};

```
console.log(greet("Bablu")); // Hello, Bablu
```

Here greet is a variable holding a function.

---

👉 **Example 2: Pass function as argument**

```
function callFunction(fn, value) {
  return fn(value);
}

function square(x) {
  return x * x;
}

console.log(callFunction(square, 5)); // 25
```

Here square function is **passed as an argument** to another function.

---

👉 **Example 3: Return function from another function**

```
function multiplier(factor) {
  return function(x) {
    return x * factor;
  };
}

const double = multiplier(2);
const triple = multiplier(3);

console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Here multiplier **returns another function**.

---

👉 **Example 4: Store functions in data structures**

```
const operations = [
```

```
 (a, b) => a + b,
 (a, b) => a - b,
 (a, b) => a * b
];

console.log(operations[0](5, 3)); // 8  (addition)
console.log(operations[1](5, 3)); // 2  (subtraction)
console.log(operations[2](5, 3)); // 15 (multiplication)
```

✅ This proves that functions in JS are **first-class** (treated like any other value).

---

### 🔹 Higher-Order Functions (HOFs)

A **Higher-Order Function** is a function that:

1. **Takes one or more functions as arguments**, OR
2. **Returns another function**.

All higher-order functions are possible **because functions are first-class**.

---

### 👉 Example 1: Function taking another function

```
function applyOperation(a, b, operation) {
 return operation(a, b);
}

const add = (x, y) => x + y;
const multiply = (x, y) => x * y;

console.log(applyOperation(4, 5, add));     // 9
console.log(applyOperation(4, 5, multiply)); // 20
```

Here applyOperation is a **higher-order function** since it takes another function (add or multiply) as input.

---

### 👉 Example 2: Function returning a function

```
function power(exponent) {
```

```
  return function(base) {
    return base ** exponent;
  };
}

const square = power(2);
const cube = power(3);

console.log(square(5)); // 25
console.log(cube(2));   // 8
```

Here power is a **higher-order function** since it **returns another function**.

---

👉 **Example 3: Built-in Higher-Order Functions in JS**

JavaScript arrays come with many HOFs: map, filter, reduce, forEach, etc.

```
const numbers = [1, 2, 3, 4, 5];

// map → applies function to each element
const squares = numbers.map(n => n * n);
console.log(squares); // [1, 4, 9, 16, 25]

// filter → keeps only even numbers
const evens = numbers.filter(n => n % 2 === 0);
console.log(evens); // [2, 4]

// reduce → accumulates values into one
const sum = numbers.reduce((acc, n) => acc + n, 0);
console.log(sum); // 15
```