BCS-403. Introduction to Functional & Logic Programming

Unit-I.

1. Distinctive features of functional prog. languages.
2. Functional programming in imperative lang
3. Recursion
4. Tail recursion
5. Higher order functions, lazy evaluation
6. Types in functional programming
7. Mathematics of functional prog: lambda calculus.

What is functional prog.?

* paradigm where functions are treated as basic building block
* focuses on what to solve not how to solve (imperative)

(how) Imperative:- "I see that table located under X sign is empty I and my friend going to sit on it."

Declarative (what):- Table for two please

Imperative:	Declarative	both.
C, C++, Java	HTML, SQL	Java script, python, C#

SQL. SELECT * FROM Users Wher Country = 'Mexico'

HTML: <article>
<header>
<h1> Dec. prog. </h1>
<p> - - - - </p>
<header>
</article>

Javascript ex: # doubling every element

```
function double(arr) {
  let results = [];
    for (let i=0; i<arr.length; i++)
      { results.push(arr[i] * 2);
    }
  return results;
  }
```

above example: imperative example.

① HOW: explicitly iterating over an array then writing steps what we want.

② in functional or declarative fashion this might not be quiet obvious. In each example we are mutating some piece of states.

Declarative example:

↳ adv. 'what' focus on; can't mutate state; should be readable

```
function double(arr) {
    return arr.map((items) => item * 2);
    }
```

→ all mutations are abstracted inside map function

Functional program → a subset of declarative programming

→ Declarative prog. "the act of programming in languages that conform to mental model of the developer rather than operational model of the machine"

Distinctive features of functional program languages

① * Modelled on a concept of mathematical functions.
* It avoids changing state & mutable data.

Properties.

① Pure function.

* Always return the same output for the same input, no side effects.

```
const square = (x)=> x * x;  // pure func.

// impure func

   let counter = 0;
   function increment (value) {
         counter += value;  // modifies external state
                            (side effect)
      return counter; }

   console.log (increment (1));   output 1
     "     "  (  "      (1));     output 2
```

② Immutability.

* Data can't be changed once created. Instead of modifying data new data structures are created.
* Adv: parallelism & hyper threaded environment

Ex.

```
   const numbers = [1, 2, 3];
   # instead of mutating an array, create a new one
   const doubled = numbers.map (n=> n*2);
   console.log (doubled);    // output :- [2,4,6]
     "    "   (numbers);     //   "      [1,2,3] (original unchanged)
```

③ First-Class & Higher-order function.

* functions as first-class citizens, they can be assigned to variable, passed as an argument, or returned from other functions.
* Higher order function take other functions as arguments or return them.

Higher order function

```
const square = (x) => x * x ; // pure function
    # Higher order function takes functions as an arguments
const mapArray = (arr, fn) => arr.map(fn);
# immutability: create a new array, instead modifying original
const numbers = [1, 2, 3, 4, 5];
const squareNumbers = mapArray(numbers, square);
```

$\underbrace{\text{mapArray}}_{\text{Higher order function}}$

④ First class function.

① Assigned to variable:

```
Const greet = function(name) {
    return `Hello, ${name}!`;
};
console.log(greet('Alice')); # Hello, Alice
```

② Passing as an argument:

```
function apply(x, y, operation) {
    return operation(x, y);}
const add = (a,b) => a+b
console.log(apply(5, 3, add)); output = 8
```

③ Returning & storing in data structure:
Javascript, Python, Haskell, LISP treat functions as first-class citizen whereas C, Java have more restrictive functional handling, limiting their functional programming capability

(4) Referential Transparency → As many times a function appear in the code it can be replaced by same value when args are same.

⇒ algorithm ~~aa~~ add (a, b) // referential opaque

referentially opaque [
int a, b
read a, b → since read statement will take different values of a & b as i/p and it will produce different results
return a+b

algorith add (a, b)
{ ~~int a, b~~;
c = a + b;
return c;
}

(5) Avoid side Effects:

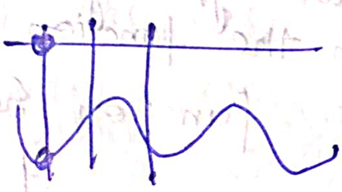
→ functions avoid modifying external state, making code more modular & predictable.

(6) Recursion over loops.

: FP favours recursion over iteration, as loops rely on mutable state.

(7) Function composition – combining simple functions to build complex one.

(8) Declarative over imperative. focuses on what instead of how



Functional Programming in imperative language

* Fun (FP) focuses on writing code using pure func., immutability, avoiding side effects, while imperative prog" relies on mutable state and sequential commands.
* Many imperative languages like Python, C# or JavaScript supports functional programming techniques by incorporating functions like first-class functions, higher order functions etc.

Ex. given a list of numbers, filter out odd numbers and square the remaining even numbers.

Imperative Approach (Python)

```
numbers = [1, 2, 3, 4, 5, 6]
even_squares = []
for num in numbers:
    if num % 2 == 0:
        even_squares.append(num * num)
print(even_squares)
```

Functional approach (Python)

```
numbers = [1, 2, 3, 4, 5, 6]
# filter even numbers and square them using map
even_squares = list(map(lambda x: x * x, filter(
lambda x: x % 2 == 0, numbers)
))
```

Recursion & tail recursion.

Recursion

* When a function calls itself to break a problem into smaller subproblems, with the function call typically occurring anywhere in the function body.

ex.

```
function recsum(x) {
    if (x == 0) {
        return 0;
    } else {
        return x + recsum(x-1);
    }
}
```

tail recursion

* A special case of recursion where the recursive call is the last operation in the function.

```
function tailrecsum(x, run_total = 0) {
    if (x == 0) {
        return run_total;
    } else {
        return tailrecsum(x-1, run_total + x);
    }
}
```

recsum(5) 5 + recsum(4) 5 + (4 + recsum(3)) 5 + (4 + (3 + recsum(2))) 5 + (4 + (3 + (2 + recsum(1)))) 5 + (4 + (3 + (2 + (1 + recsum(0))))) 5 + (4 + (3 + (2 + (1 + 0)))) 5 + 4 + (3 + 3) 5 + (4 + 6) 5 + 10 = 15	tailrecsum (5, 0) tailrecsum (4, 5) " (3, 9) " (2, 12) " (1, 14) " (0, 15) 15 Ans
→ adds a new stack frame per call	* reuses frame stack

Lazy Evaluation

* in FP, Lazy evaluation is a technique where expressions are not evaluated until their results are needed.

Characteristics:

Delayed evaluation - expressions are evaluated only when their value is required.

Memoization : once evaluated, results may be cached for reuse

Examples: Haskell uses lazy evaluation by defaults:

```
nums = [1..]          -- infinite list
first10 = take 10 nums
  -- output = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

⇒ Infinite list is not fully generated. Only the first 10 numbers are computed.

2 - Python - Generators

```
def numbers():
    n = 1
    while True:
        yield n
        n += 1
gen = numbers()
print(next(gen))   # 1
"    "    "        # 2
"    "    "        # 3
```

output

function doesn't generate all numbers at once, it produces them on demand.

③ Short-circuit Evaluation (Boolean logic)

Most languages like C, Java, Python use lazy evaluation in logical operators.

```
x = 0
result = (x != 0) and (10/x > 1)
print(result)  # false
```

the second condition (10/x>1) is not evaluated because, the first (x!=0) is already false.

Types in functional programming.

① Basic Types: Integers, floats, booleans, characters:

Haskell:

```
x :: Int
x = 42
is_True :: Bool
is_True = True
```

② Functions: functions are first class citizens in FP and have types that describe their i/p & o/p.

syntax: a -> b means a function take an i/p of type a and return a type b

haskell:

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

add: Int → Int → Int is a type signature of a function

It takes: Int as first & second arg & returns an Int

→ arrow denote function types in haskell. The last type Int is the return types, and the types before (Int → Int) are the i/p parameters.

Usage:

```
main :: IO()
main = print (add 3 5) -- outputs 8
> add 3 5
  8
```

Records: Structured types with named fields, similar to structs in other languages.

Haskell: data Point = Point { x :: Double , y :: Double } -- defines a record type called Point

```
origin :: Point -- creates a value origin of that type
origin = Point 0.0 0.0 # origin as a Point with values 0.0 0.0.
```

Higher order function. map :: (a→b) → [a] → [b]

map takes function (a→b) and a list [a], returns a list [b].

Lambda Calculus. a formal system for expressing computation via function abstraction and application.

Syntax. Symbols (x, y) → Variables

Abstraction: λx. M defines a function with parameter x and body M.

Application: (M, N) applies function M to argument N

→ Expressions are built using these constructs recursive.

Ex $f(x) = x+1$

In lambda cal. written as $\lambda x . x+1$] lambda function computes in 3 rules

if input 5 output would be 6

① Alpha conversion:
function, $\lambda x . x$ returns the i/p and it is same as $\lambda y . y$

② Beta Reduction: ~~Beta red~~ $(\lambda x . x+2)4$. β-reduction
replaces x with 4 in $x+2$, so it becomes $4+2=6$.

③ Eta conversion: simplifying a function that just passes
its input to another func.

$\lambda x . f x$ (takes x and applies f to it)

λx . square x

eg. $\lambda x . x \bmod 2 = 0$ # to test numbers is even

Intro to Haskell. Module 2