SRM Institute of Science and Technology
School of Computing

Advanced Programming Practice-18CSC207J

Paradigm types

Unit IV

- Functional Programming Paradigm [Text Book :1]
- Logic Programming Paradigm [Text Book : 1& 3]
- Dependent Type Programming Paradigm
- Network Programming Paradigm [Text Book :4]

Text Book:

1. Elad Shalom, A Review of Programming Paradigms throughout the History: With a suggestion Toward a Future Approach, Kindle Edition, 2018
2. John Goerzen, Brandon Rhodes, Foundations of Python Network Programming: The comprehensive guide to building network applications with Python, 2nd ed., Kindle Edition, 2010
3. Amit Saha, Doing Math with Python: Use Programming to Explore Algebra, Statistics,Calculus and More, Kindle Edition, 2015

Functional Programming Paradigm

Unit-III (15 Session)

Session 11-15 cover the following topics:-

- *Definition - S11-SLO1*
- *Sequence of Commands – S11-SLO2*
- *map(), reduce(), filter(), lambda – S12-SLO1*
- *partial, functools – S12-SLO2*
- *Other languages:F#, Clojure, Haskell – S13-SLO1*
- *Demo: Functional Programming in Python - S13-SLO2*

Lab 9: Functional Programming (Case Study) (S14-15)

Assignment : Comparative study of Functional programming in F#, *Clojure, Haskell*

- **TextBook**: Shalom, Elad. A Review of Programming Paradigms Throughout the History: With a Suggestion Toward a Future Approach, Kindle Edition

Functional Programming Paradigm (TF1)

Definition

- Mainly treat **computation to evaluate mathematical Functions**
- Avoids changing-state and mutable data
- Called as **declarative programming** paradigm
- Output depends on argument passing
- Calling same function several times with same values produces same result
- Uses expressions or declarations rather than statements as in imperative languages

Concepts

- It views all subprograms as **functions** in the mathematical sense
- Take in arguments and return a single solution.
- Solution returned is based entirely on input, and the time at which a function is called has no relevance.
- The computational model is therefore one of function application and reduction.

Characteristics of Functional Programming

- Functional programming method focuses on results, not the process
- Emphasis is on what is to be computed
- Data is immutable
- Functional programming Decompose the problem into 'functions
- It is built on the concept of mathematical functions which uses conditional expressions and recursion to do perform the calculation
- It does not support iteration like loop statements and conditional statements like If-Else

History

- The foundation for Functional Programming is Lambda Calculus. It was developed in the 1930s for the functional application, definition, and recursion
- LISP was the first functional programming language. McCarthy designed it in 1960
- In the late 70's researchers at the University of Edinburgh defined the ML(Meta Language)
- In the early 80's Hope language adds algebraic data types for recursion and equational reasoning
- In the year 2004 Innovation of Functional language 'Scala.'

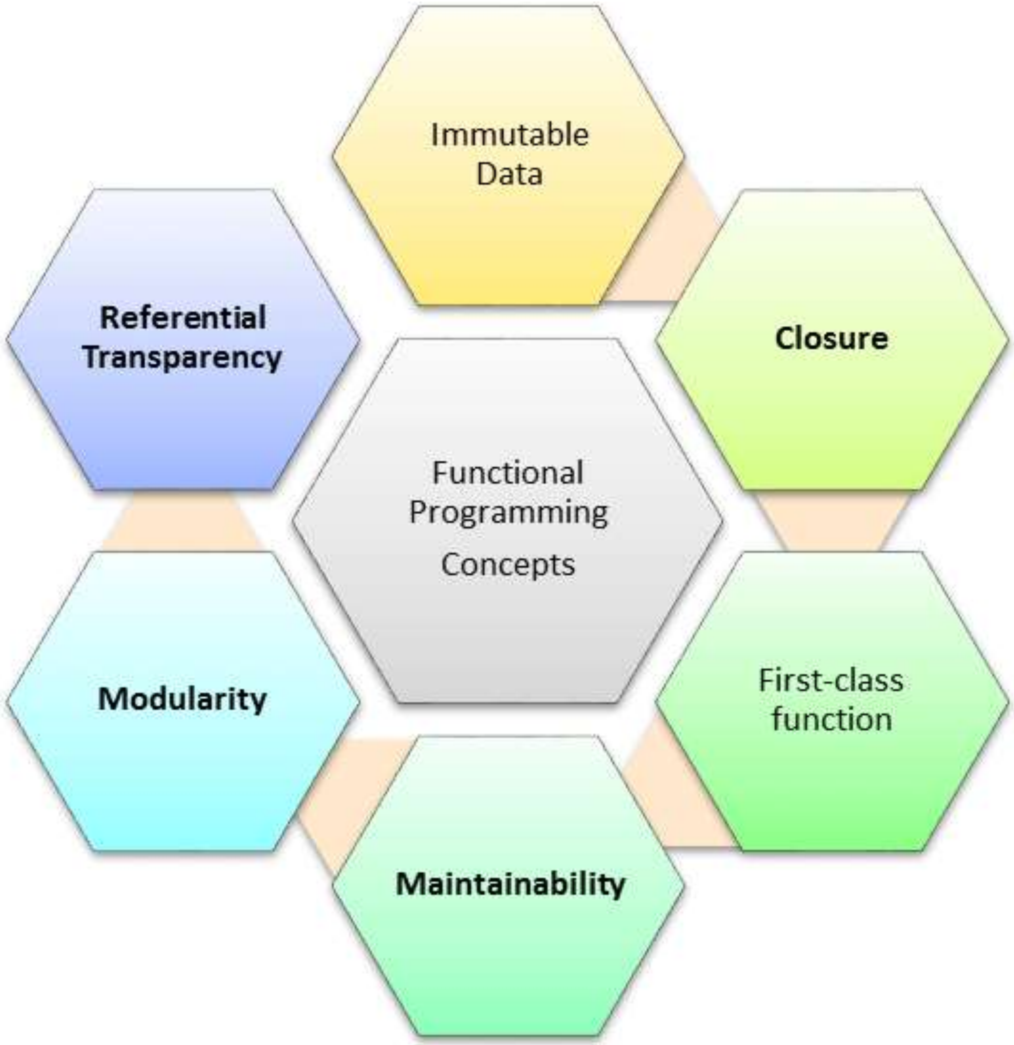
Real Time applications

- Database processing
- Financial modeling
- Statistical analysis and
- Bio-informatics

Functional Programming Languages

- Haskell
- SML
- Clojure
- Scala
- Erlang
- Clean
- F#
- ML/OCaml Lisp / Scheme
- XSLT
- SQL
- Mathematica

Basic Functional Programming Terminology and Concepts



Functional Vs Procedural

S.No	Functional Paradigms	Procedural Paradigm
1	Treats computation as the evaluation of mathematical functions avoiding state and mutable data	Derived from structured programming, based on the concept of modular programming or the *procedure call*
2	Main traits are Lambda calculus, compositionality, formula, recursion, referential transparency	Main traits are Local variables, sequence, selection, iteration, and modularization
3	**Functional** programming focuses on **expressions**	**Procedural** programming focuses on **statements**
4	Often recursive.Always returns the same output for a given input.	The output of a routine does not always have a direct correlation with the input.
5	Order of evaluation is usually undefined.	Everything is done in a specific order.
6	Must be stateless. i.e. No operation can have side effects.	Execution of a routine may have side effects.
7	Good fit for parallel execution, Tends to emphasize a divide and conquer approach.	Tends to emphasize implementing solutions in a linear fashion.

Functional Vs Object-oriented Programming

S.No	Functional Paradigms	Object Oriented Paradigm
1	FP uses Immutable data.	OOP uses Mutable data.
2	Follows Declarative Programming based Model.	Follows Imperative Programming Model.
3	What it focuses is on: "What you are doing. in the programme."	What it focuses is on "How you are doing your programming."
4	Supports Parallel Programming.	No supports for Parallel Programming.
5	Its functions have no-side effects.	Method can produce many side effects.
6	Flow Control is performed using function calls & function calls with recursion.	Flow control process is conducted using loops and conditional statements.
7	Execution order of statements is not very important.	Execution order of statements is important.
8	Supports both "Abstraction over Data" and "Abstraction over Behavior."	Supports only "Abstraction over Data".

Example

Functional Programming

```
num = 1
def function_to_add_one(num):
    num += 1
     return num
function_to_add_one(num)
function_to_add_one(num)
function_to_add_one(num)
function_to_add_one(num)
function_to_add_one(num)

#Final Output: 2
```

Procedural Programming

```
num = 1
def procedure_to_add_one():
    global num
    num += 1
    return num
procedure_to_add_one()
procedure_to_add_one()
procedure_to_add_one()
procedure_to_add_one()
procedure_to_add_one()

 #Final Output: 6
```

Features of Functional paradigms

- **First-class functions** – accept another function as an argument or return a function
- **Pure functions -** they are functions without side effects
- **Recursion -** allows writing smaller algorithms and operating by looking only at the inputs to a function
- **Immutable variables -** variables that cannot be changed
- **Non-strict evaluation -** allows having variables that have not yet been computed
- **Statements -** evaluable pieces of code that have a return value
- **Pattern matching -** allows better type-checking and extracting elements from an object

Functions as first class objects in python

- Using functions as first class objects means to use them in the same manner that you use data.
- So, You can pass them as parameters like passing a function to another function as an argument.

- >>> list(map(int, ["1", "2", "3"])) [1, 2, 3]

Pure Functions

- **is idempotent** — returns the same result if provided the same arguments,
- has no side effects.
- If a function uses an object from a higher scope or random numbers, communicates with files and so on, it might be ***impure***
- Since its result doesn’t depend only on its arguments.

Example

```
def multiply_2_pure(numbers):
    new_numbers = []
    for n in numbers:
        new_numbers.append(n * 2)
    return new_numbers

original_numbers = [1, 3, 5, 10]
changed_numbers =
  multiply_2_pure(original_numbers)
print(original_numbers) # [1, 3, 5, 10]
print(changed_numbers)  # [2, 6, 10, 20]
```

Anonymous Functions

- Anonymous (lambda) functions can be very convenient for functional programming constructs.
- They don't have names and usually are created ad-hoc, with a single purpose.
- Exp1: lambda x, y: x + y
- Exp 2: >>> f = lambda x, y: x + y
  >>> def g(x, y):
  return x + y

Examples

Anonymous function assigned to a variable. Easy to pass around and invoke when needed.

const myVar = function(){console.log('Anonymous function here!')}

myVar()

Anonymous function as argument

- setInterval(function(){console.log(new Date().getTime())}, 1000);

Anonymous functions within a higher order function

```
function mealCall(meal){
    return function(message){
    return console.log(message + " " + meal + '!!') }
    }
```

const announceDinner = mealCall('dinner')

const announceLunch = mealCall('breakfast')

announceDinner('hey!, come and get your')

announceLunch('Rise and shine! time for')

Mathematical Background

- For example, if $f(x)=x^2$and x is 3, the ordered pair is (-3, 9).
- A function is defined by its set of inputs, called the domain.
- A set containing the set of outputs, and possibly additional elements as members, is called its codomain.
- The set of all input-output pairs is called its graph.

Mathematical Background

General Concepts

- **Notation**

  $F(x) = y$

  x , y -> Arguments or parameters

  x -> domain and

  y->codomain

  Types:

  – Injective if $f(a) \neq f(b)$
  – Surjective if $f(X)= Y$
  – Bijective (support both)

Functional Rules

1. $(f+g)(x)=f(x)+g(x)$
2. $(f-g)(x)=f(x)-g(x)$
3. $(f*g)(x)=f(x)*g(x)$
4. $(f/g)(x)=f(x)/g(x)$
5. $(gof)(x)=g(f(x))$
6. $f\,f^{-1}=id_y$

Example using Python

- Anonymous functions are split into two types: lambda functions and closures.
- Functions are made of **four parts**: name, parameter list, body, and return

Example :

```
A = 5
def impure_sum(b):
    return b + A
def pure_sum(a, b):
    return a + b
print(impure_sum(6))
>> 11
print(pure_sum(4, 6))
>> 10
```

Example Code

Function for computing the average of two numbers:

```
(defun avg(X Y) (/ (+ X Y) 2.0))
```

Function is called by:

```
> (avg 10.0 20.0)
```

Function returns:

15.0

Functional style of getting a sum of a list:

```
new_lst = [1, 2, 3, 4]
def sum_list(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return lst[0] + sum_list(lst[1:])
print(sum_list(new_lst))
```

 # or the pure functional way in python using higher order function

```
import functools
print(functools.reduce(lambda x, y: x + y, new_lst))
```

Immutable variables

- Immutable variable (object) is a variable whose state cannot be modified once it is created.
- In contrast, a mutable variable can be modified after it is created
- **Exp**
- String str = “A Simple String.”;
- str.toLowerCase();

Other Examples

- int i = 12; //int is a primitive type
- i = 30; //this is ok
- final int j = 12;
- j = 30;
- final MyClass c = new MyClass();
- m.field = 100; //this is ok;
- m = new MyClass();

First-class function

- First-class functions are functions treated as objects themselves,
- Meaning they can be passed as a parameter to another function, returned as a result of a function
- It is said that a programming language has first-class functions if it treats functions as first-class citizens.

Higher-order function

- takes one or more functions as an input
- outputs a function
- Exp : Map Function
- Consider a function that prints a line multiple times:
- Example Code

```
def write_repeat(message, n):
    for i in range(n):
        print(message)
 write_repeat('Hello', 5)
```

Scenario1

- What if we wanted to write to a file 5 times, or log the message 5 times? Instead of writing 3 different functions that all loop, we can write 1 Higher Order Function that accepts those functions as an argument:

```
def hof_write_repeat(message, n, action):
    for i in range(n):
        action(message)

hof_write_repeat('Hello', 5, print)

# Import the logging library
import logging
# Log the output as an error instead
hof_write_repeat('Hello', 5, logging.error)
```

Scenario2

- Imagine that we're tasked with creating functions that increment numbers in a list by 2, 5, and 10. So instead of creating many different increment functions, we create 1 Higher Order Function:

```
def hof_add(increment):
    # Create a function that loops and adds the increment
    def add_increment(numbers):
        new_numbers = []
        for n in numbers:
            new_numbers.append(n + increment)
        return new_numbers
    # We return the function as we do any other value
    return add_increment
add2=hof_add(2)
print(add2([23, 88])) # [25, 90]
add5 = hof_add(5)
print(add5([23, 88]))   # [28, 93]
add10 = hof_add(10)
print(add10([23, 88]))  # [33, 98]
```

Functional programming tools

- **filter(*function, sequence*)**

  def f(x): return x%2 != 0 and x%3 ==0

  filter(f, range(2,25))
- **map(*function, sequence)***
  - call function for each item
  - return list of return values
- **reduce(*function, sequence*)**
  - return a single value
  - call binary function on the first two items
  - then on the result and next item
  - iterate

Lambda Expression

```
# Using `def` (old way).
def old_add(a, b):
    return a + b

# Using `lambda` (new way).
new_add = lambda a, b: a + b old_add(10, 5) == new_add(10, 5)
>> True

unsorted = [('b', 6), ('a', 10), ('d', 0), ('c', 4)]
print(sorted(unsorted, key=lambda x: x[1]))
>> [('d', 0), ('c', 4), ('b', 6), ('a', 10)]
```

Map Function

- Takes in an iterable (ie. list), and creates a new iterable object, a special map object.
- The new object has the first-class function applied to every element.

```
# Pseudocode for map.

def map(func, seq):
    return Map(
        func(x)
        for x in seq
    )
```

Example:

```
values = [1, 2, 3, 4, 5]

add_10 = list(map(lambda x: x + 10, values))
add_20 = list(map(lambda x: x + 20, values))

print(add_10)
>> [11, 12, 13, 14, 15]

print(add_20)
>> [21, 22, 23, 24, 25]
```

Filter Function

- Takes in an iterable (ie. list), and creates a new iterable object
- The new object has the first-class function applied to every element.

```
# Pseudocode for map.

return Map(
        x for x in seq
        if evaluate(x) is True
    )
```

Example:

```
even = list(filter(lambda x: x % 2 == 0, values))
odd = list(filter(lambda x: x % 2 == 1, values))

print(even)
>> [2, 4, 6, 8, 10]

print(odd)
>> [1, 3, 5, 7, 9]
```

Advantages

- Allows you to avoid confusing problems and errors in the code
- Easier to test and execute Unit testing and debug FP Code.
- Parallel processing and concurrency
- Hot code deployment and fault tolerance
- Offers better modularity with a shorter code
- Increased productivity of the developer
- Supports Nested Functions
- Functional Constructs like Lazy Map & Lists, etc.
- Allows effective use of Lambda Calculus

Disadvantages

- Perhaps less efficiency
- Problems involving many variables or a lot of sequential activity are sometimes easier to handle imperatively
- Functional programming paradigm is not easy, so it is difficult to understand for the beginner
- Hard to maintain as many objects evolve during the coding
- Needs lots of mocking and extensive environmental setup
- Re-use is very complicated and needs constantly refactoring
- Objects may not represent the problem correctly

Transforming code from imperative to functional

1. Introduce higher-order functions.
2. Convert existing methods into pure functions.
3. Convert loops over to recursive/tail-recursive methods (if possible).
4. Convert mutable variables into immutable variables.
5. Use pattern matching (if possible).

Sample Scenario

1) (The MyTriangle module) Create a module named MyTriangle that contains the following two functions:

Returns true if the sum of any two sides is greater than the third side.

def isValid(side1, side2, side3):

Returns the area of the triangle.

def area(side1, side2, side3):

Write a test program that reads three sides for a triangle and computes the area if the input is valid. Otherwise, it displays that the input is invalid.

2) A prime number is called a Mersenne prime if it can be written in the form for some positive integer p. Write a program that finds all Mersenne primes with and displays the output as follows:

p	2^p - 1
2	3
3	7
5	31

Sample Scenarios

3) Write a function named ack that evaluates the Ackermann function. Use your function to evaluate ack(3, 4), which should be 125.

The Ackermann function, $A(m,n)$, is defined:

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Thank you