18CSC207J – Advanced Programming Practice

C.Arun, Asst. Prof. Dept of Software Engineering, School of Computing, SRMIST

Functional Programming Paradigm

Introduction

- Functional programming is a programming paradigm in which it is tried to bind each and everything in pure mathematical functions. It is a declarative type of programming style that focuses on what to solve rather than how to solve.
- Functional programming paradigm is based on lambda calculus.
- Instead of statements, functional programming makes use of expressions. Unlike a statement, which is executed to assign variables, evaluation of an expression produces a value.
- Functional programming is a declarative paradigm because it relies on expressions and declarations rather than statements. Unlike procedures that depend on a local or global state, value outputs in FP depend only on the arguments passed to the function.
- Functional programming consists only of PURE functions.
- In functional programming, control flow is expressed by combining function calls, rather than by assigning values to variables.

Example:

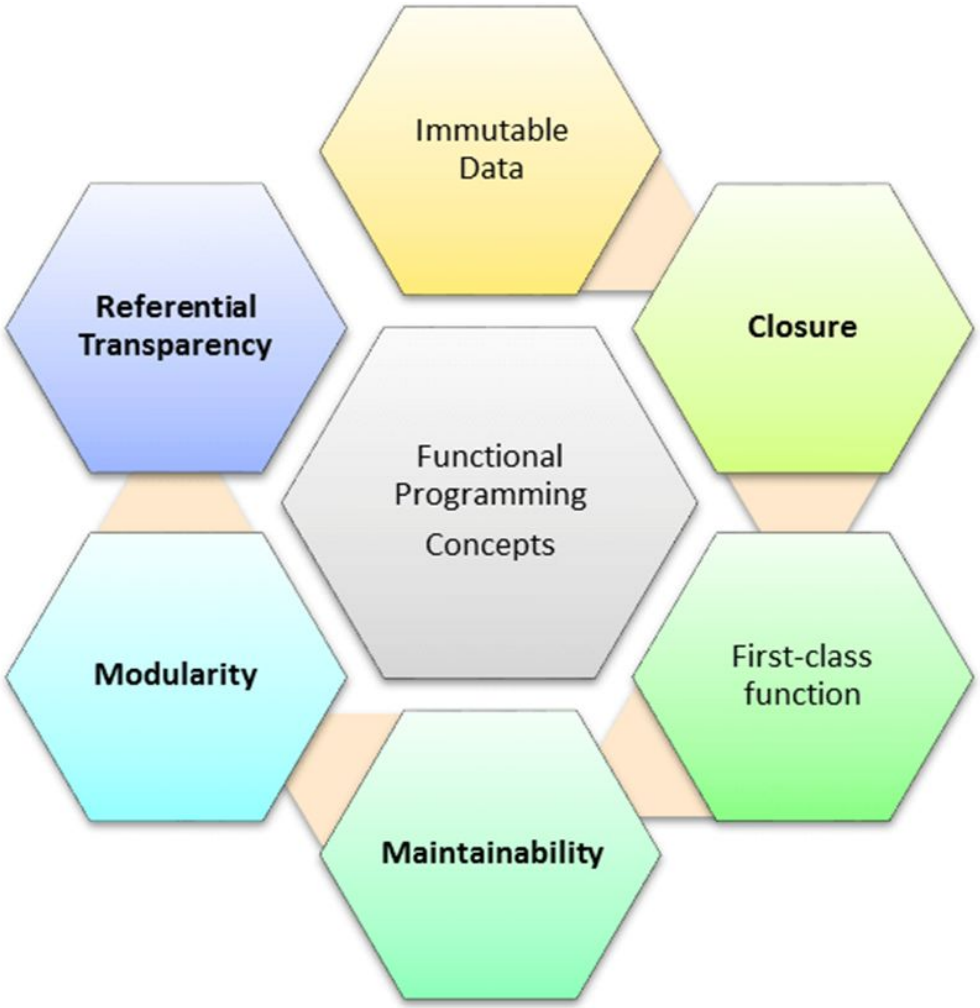
```
sorted(p.name.upper() for p in people if len(p.name) > 5)
```

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
- Functional programming languages are designed on the concept of mathematical functions that use conditional expressions and recursion to perform computation.
- Functional programming supports higher-order functions and lazy evaluation features.
- Functional programming languages don’t support flow Controls like loop statements and conditional statements like If-Else and Switch Statements. They directly use the functions and functional calls.

Concepts of FP

- Pure functions
- Recursion
- Referential transparency
- Functions are First-Class and can be Higher-Order
- Immutability



Functional Programming vs Procedure Programming

```python
#Functional Programming

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

print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
print("Num is :",function_to_add_one(num)) #global num =1
```

```
Num is : 2
Num is : 2
Num is : 2
```

```python
#Procedural  Programming
'''The basic rules for global keyword in Python are:
When we create a variable inside a function, it's local by default.
When we define a variable outside of a function, it's global by default. ...
We use global keyword to read and write a global variable inside a function'''

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()
```

1. Pure functions

- Pure functions always return the same results when given the same inputs. Consequently, they have no side effects.
- Properties of Pure functions are:
  - First, they always produce the same output for same arguments irrespective of anything else.
  - Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
- A simple example would be a function to receive a collection of numbers and expect it to increment each element of this collection.
- We receive the numbers collection, use map with the ***inc*** function to increment each number, and return a new list of incremented numbers.

Example:

```
def inc(x):
        return x+1
list=[8,3,7,5,2,6]
x=map(inc,list)   #print(list)
print(x)
```

```
var z = 15;

function add(x, y) {
    return x + y;
}
```

ding files,
nbers are impure functions

Note: if a function relies on the global variable or class member's data, then it is not pure. And in such cases, the return value of that function is not entirely dependent on the list of arguments received as input and can also have side effects.

A side effect is a change in the state of an application that is observable outside the called function other than its return value

2. Recursion

- In the functional programming paradigm, there is no for and while loops. Instead, functional programming languages rely on recursion for iteration. Recursion is implemented using recursive functions, which repetitively call themselves until the base case is reached.

```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
```

- The above code performs recursion task as the loop by calling itself with a new start and a new accumulator.

Referential transparency

- An expression is said to be referentially transparent if it can be replaced with its corresponding value without changing the program's behaviour. As a result, evaluating a referentially transparent function gives the same value for fixed arguments. If a function consistently yields the same result for the same input, it is referentially transparent.
- Functional programs don’t have any assignment statements. For storing additional values in a program developed using functional programming, new variables must be defined. State of a variable in such a program is constant at any moment in time
  - pure function + immutable date = referential transparency

```
int add(int a, int b)
 {
 return a + b
 }
int mult(int a, int b)
 {
 return a * b;
 }
int x = add(2, mult(3, 4));
```

- Let’s implement a square function:
  - This (pure) function will always have the same output, given the same input.
  - Passing “2” as a parameter of the square function will always returns 4. So now we can replace the (square 2) with 4.

Immutability

- In functional programming you cannot modify a variable after it has been initialized. You can create new variables and this helps to maintain state throughout the runtime of a program.

```
var x = 1;
x = x + 1;
```

- In imperative programming, this means “take the current value of x, add 1 and put the result back into x.” In functional programming, however, x = x + 1 is illegal. That’s because there are technically no variables in functional programming.
- Using immutable data structures, you can make single or multi-valued changes by copying the variables and calculating new values,
- Since FP doesn’t depend on shared states, all data in functional code must be immutable, or incapable of changing

Functions are First-Class and can be Higher-Order

- A programming language is said to have First-class functions when functions in that language are treated like any other variable. For example, in such a language, a function can be passed as an argument to other functions, can be returned by another function and can be assigned as a value to a variable.
- Higher-order functions are functions that take at least one first-class function as a parameter
- Examples:
  - name_lengths = map(len, ["Bob", "Rob", "Bobby"])
- Higher Order functions are map, reduce, filter

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))
```

Functions are First-Class and can be Higher-Order

Map

map() can run the function on each item and insert the return values into the new collection. Example:

```
squares = map(lambda x: x * x, [0, 1, 2, 3, 4])
import random
names = ['Seth', 'Ann', 'Morganna']
team_names = map(lambda x: random.choice(['A Team','B Team']),names)
print names
// ['A Team', 'B Team', 'B Team']
```

reduce()

- reduce() is another higher order function for performing iterations. It takes functions and collections of items, and then it returns the value of combining the items

Example:

```
sum = reduce(lambda a, x: a + x, [0, 1, 2, 3, 4])
print sum        // 10
```

Functions are First-Class and can be Higher-Order ..cont

filter()

- filter function expects a true or false value to determine if the element should or should not be included in the result collection. Basically, if the call-back expression is true, the filter function will include the element in the result collection. Otherwise, it will not.

Example:

```
def f(x):
    return x%2 != 0 and x%3 ==0
filter(f, range(2,25))
```

Functional Programming – Non Strict Evaluation

- In programming language theory, lazy evaluation, or call-by-need[1] is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations
- Allows Functions having variables that have not yet been computed

In Python, the logical expression operators and, or, and if-then-else are all non-strict. We sometimes call them short-circuit operators because they don't need to evaluate all arguments to determine the resulting value.

The following command snippet shows the and operator's non-strict feature:

```
>>> 0 and print("right")
0
>>> True and print("right")
Right
```

When we execute the preceding command snippet, the left-hand side of the and operator is equivalent to False; the right-hand side is not evaluated. When the left-hand side is equivalent to True, the right-hand side is evaluated

Functional Programming – lambda calculus

Lambda expressions in Python and other programming languages have their roots in lambda calculus. Lambda calculus can encode any computation. Functional languages get their origin in mathematical logic and lambda calculus

In Python, we use the lambda keyword to declare an anonymous function, which is why we refer to them as "lambda functions". An anonymous function refers to a function declared with no name.

when you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax:

lambda argument(s): expression

Example:

```
remainder = lambda num: num % 2
print(remainder(5))
```

```
[(lambda x: x*x)(x) for x in [2,6,9,3,6,4,8]]
```

Functional Programming – Closure

Basically, the method of binding data to a function without actually passing them as parameters is called closure. It is a function object that remembers values in enclosing scopes even if they are not present in memory.

Example:.

```
def counter(start=0, step=1):
    x = [start]
    def _inc():
        x[0] += step
        return x[0]
    return _inc
c1 = counter()
c2 = counter(100, -10)
c1()
//1
c2()
90
```

Pure Functions in Python

- If a function uses an object from a higher scope or random numbers, communicates with files and so on, it might be impure

```
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]
```

```
[1, 3, 5, 10]
[2, 6, 10, 20]
```

```
# Example1 : Impure Function
A = 5
def impure_sum(b): # A is out side function ,it has side effect
    return b + A

impure_sum(8)
```

```
13
```

```
# Example2 : Pure Function

def pure_sum(a, b):    #a and b inside function
    return a + b
print(impure_sum(6))
```

```
11
```

Built-in Higher Order Functions

Map

- The map function allows us to apply a function to every element in an iterable object

Filter

- The filter function tests every element in an iterable object with a function that returns either True or False, only keeping those which evaluates to True.

Combining map and filter

- As each function returns an iterator, and they both accept iterable objects, we can use them together for some really expressive data manipulations!

List Comprehensions

- A popular Python feature that appears prominently in Functional Programming Languages is list comprehensions. Like the map and filter functions, list comprehensions allow us to modify data in a concise, expressive way.

Anonymous Function

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the def keyword, in Python anonymous functions are defined using the lambda keyword.

Characteristics of Python lambda functions:

- A lambda function can take any number of arguments, but they contain only a single expression. An expression is a piece of code executed by the lambda function, which may or may not return any value.
- Lambda functions can be used to return function objects.
- Syntactically, lambda functions are restricted to only a single expression.

Syntax of Lambda Function in python

lambda arguments: expression

Example:

```
double = lambda x: x * 2
print(double(5))
# Output: 10
```

```
product = lambda x, y : x * y
print(product(2, 3))
```

Note: you want to pass a function as an argument to higher-order functions, that is, functions that take other functions as their arguments.

map() Function

Example Map with lambda

```
tup= (5, 7, 22, 97, 54, 62, 77, 23, 73, 61)
newtuple = tuple(map(lambda x: x+3 , tup))
print(newtuple)
```

//with multiple iterables

```
list_a = [1, 2, 3]
list_b = [10, 20, 30]
 map(lambda x, y: x + y, list_a, list_b)
```

Example with Map

```
from math import sqrt
map(sqrt, [1, 4, 9, 16])
[1.0, 2.0, 3.0, 4.0]
map(str.lower, ['A', 'b', 'C'])
['a', 'b', 'c']
#splitting the input and convert to int using map
print(list(map(int, input.split(‘ ‘)))
```

```
numbers_list = [2, 6, 8, 10, 11, 4, 12, 7, 13, 17, 0, 3, 21]

mapped_list = list(map(lambda num: num % 2, numbers_list))

print(mapped_list)
```

map() Function

- map() function is a type of higher-order. As mentioned earlier, this function takes another function as a parameter along with a sequence of iterables and returns an output after applying the function to each iterable present in the sequence.

Syntax:

map(function, iterables)

Example without Map

```
my_pets = ['alfred', 'tabitha', 'william', 'arla']
uppered_pets = []
for pet in my_pets:
    pet_=pet.upper()
    uppered_pets.append(pet_)
print(uppered_pets)
```

Example with Map

```
my_pets = ['alfred', 'tabitha', 'william', 'arla']
uppered_pets=list(map(str.upper,my_pets)) print(uppered_pets)
//map with multiple list as input
circle_areas = [3.56773, 5.57668, 4.00914, 56.24241, 9.01344, 32.00013]
result = list(map(round, circle_areas, range(1,7)))
print(result)
```

filter() Function

- filter extracts each element in the sequence for which the function returns True.
- filter(), first of all, requires the function to return boolean values (true or false) and then passes each element in the iterable through the function, "filtering" away those that are false

Syntax:

filter(func, iterable)

The following points are to be noted regarding filter():

- Unlike map(), only one iterable is required.
- The func argument is required to return a boolean type. If it doesn't, filter simply returns the iterable passed to it. Also, as only one iterable is required, it's implicit that func must only take one argument.
- filter passes each element in the iterable through func and returns only the ones that evaluate to true. I mean, it's right there in the name -- a "filter".

Example:

```
def isOdd(x): return x % 2 == 1
filter(isOdd, [1, 2, 3, 4])
# output --->  [1, 3]
```

filter() Function

Example:

Python 3

scores = [66, 90, 68, 59, 76, 60, 88, 74, 81, 65]

def is_A_student(score):

    return score > 75

over_75 = list(filter(is_A_student, scores))

print(over_75)

```
'''
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)
for x in adults:
  print(x)
```

```
# Python 3
dromes = ("demigod", "rewire", "madam", "freer",
"anutforajaroftuna", "kiosk")

palindromes = list(filter(lambda word: word == word[::-1],
dromes))

print(palindromes)
```

```
 #function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False
# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

reduce() Function

- reduce, combines the elements of the sequence together, using a binary function. In addition to the function and the list, it also takes an initial value that initializes the reduction, and that ends up being the return value if the list is empty.
- The “reduce” function will transform a given list into a single value by applying a given function continuously to all the elements. It basically keeps operating on pairs of elements until there are no more elements left.
- reduce applies a function of two arguments cumulatively to the elements of an iterable, optionally starting with an initial argument

Syntax:

reduce(func, iterable[, initial])

Example:

```
reduce(lambda s,x: s+str(x), [1, 2, 3, 4], '')
#output '1234‘
my_list = [3,8,4,9,5]
reduce(lambda a, b: a * b, my_list)
#output 4320 ( 3*8*4*9*5)
```

```
from functools import reduce
y = filter(lambda x: (x>=3), (1,2,3,4))
print(list(y))

reduce(lambda a,b: a+b,[23,21,45,98])

nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,
      29, 21, 60, 27, 62, 59, 86, 56]
sum = reduce(lambda x, y : x + y, nums) / len(nums)
```

map(), filter() and reduce() Function

Using filter() within map():

```
c = map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4)))
print(list(c))
```

Using map() within filter():

```
c = filter(lambda x: (x>=3),map(lambda x:x+x, (1,2,3,4))) #lambda x: (x>=3)
print(list(c))
```

Using map() and filter() within reduce():

```
d = reduce(lambda x,y: x+y,map(lambda x:x+x,filter(lambda x: (x>=3), (1,2,3,4))))
print(d)
```

map(), filter() and reduce() Function

```
from functools import reduce

# Use map to print the square of each numbers rounded# to two decimal places

my_floats = [4.35, 6.09, 3.25, 9.77, 2.16, 8.88, 4.59]

# Use filter to print only the names that are less than or equal to seven letters

my_names = ["olumide", "akinremi", "josiah", "temidayo", "omoseun"]

# Use reduce to print the product of these numbers

my_numbers = [4, 6, 9, 23, 5]

map_result = list(map(lambda x: round(x ** 2, 3), my_floats))

filter_result = list(filter(lambda name: len(name) <= 7, my_names))

reduce_result = reduce(lambda num1, num2: num1 * num2, my_numbers)

print(map_result)

print(filter_result)

print(reduce_result)
```

Examples

```
#print the sum of even numbers from the user input
import functools
l = input("Enter the list of numbers separated by space").split(" ")
l = map(int,l)
def even_fil(x):
    flag = False
    if x%2==0:
        flag=True
    return flag
even = list(filter(even_fil,l))
print(even)


s = functools.reduce(lambda x,y:x+y,even)
```

Examples

```
//Closure
def twice(x):
    def square():
        return x*x
    return square()*square()
def quad(x):
    return twice(x)
print(quad(3))
```

Examples

```
//partial to make some of the parameters as fixed
import functools
def s_total(a,b,c):
    return a*b+c    #a=5,b=15, c=from list
s = functools.partial(s_total,5,15)
l= [13,54,76,89,10]
for i in l:
    print(s(i))
```

Function vs Procedure

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 alculus, 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.

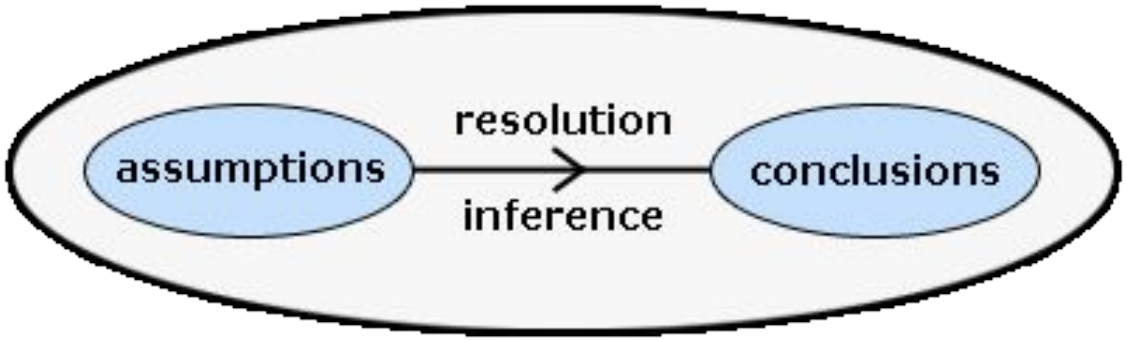
Function vs Object Oriented

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".

Logical Programming Paradigm

Logical Programming Paradigm

Logic programming is a paradigm where computation arises from proof search in a logic according to a fixed, predictable strategy. A logic is a language. It has syntax and semantics. A logic is a language. It has syntax and semantics. More than a language, it has inference rules.



Syntax:

this is the rules about how to form formulas; this is usually the easy part of a logic.

Semantics:

About the meaning carried by the formulas, mainly in terms of logical consequences.

Inference rules:

Inference rules describe correct ways to derive conclusions

Logical Programming Paradigm

Logic :

A Logic program is a set of predicates.

Predicates :

Define relations between their arguments. Logically, a Logic program states what holds. Each predicate has a name, and zero or more arguments. The predicate name is a atom. Each argument is an arbitrary Logic term. A predicate is defined by a collection of clauses.

Clause :

A clause is either a rule or a fact. The clauses that constitute a predicate denote logical alternatives: If any clause is true, then the whole predicate is true.

Logical Programming Paradigm

Fact

A fact must start with a predicate (which is an atom). The predicate may be followed by one or more arguments which are enclosed by parentheses. The arguments can be atoms (in this case, these atoms are treated as constants), numbers, variables or lists.

Facts are axioms; relations between terms that are assumed to be true.

Example facts:

+big('bear')

+big('elephant')

+small('cat')

+brown('bear')

+black('cat')

+grey('elephant')

Consider the 3 fact saying ‘cat’ is a smallest animal and fact 6 saying the elephant is grey in color

Logical Programming Paradigm

Rule

- Rules are theorems that allow new inferences to be made.
- Describe Relationships Using other Relationships.

Example

Two people are sisters if they are both female and have the same parents.

Gives a definition of one relationship given other relationships.

Both must be females.

Both must have the same parents.

If two people satisfy these rules, then they are sisters (according to our simplified relationship)

dark(X)<=black(X)

dark(X)<=brown(X)

Consider rule 1 saying the animal color is black its consider to be dark color animal

Logical Programming Paradigm

Queries

A query is a statement starting with a predicate and followed by its arguments, some of which are variables. Similar to goals, the predicate of a valid query must have appeared in at least one fact or rule in the consulted program, and the number of arguments in the query must be the same as that appears in the consulted program.

```
print(pyDatalog.ask('father_of(X,jess)'))
```

Output:

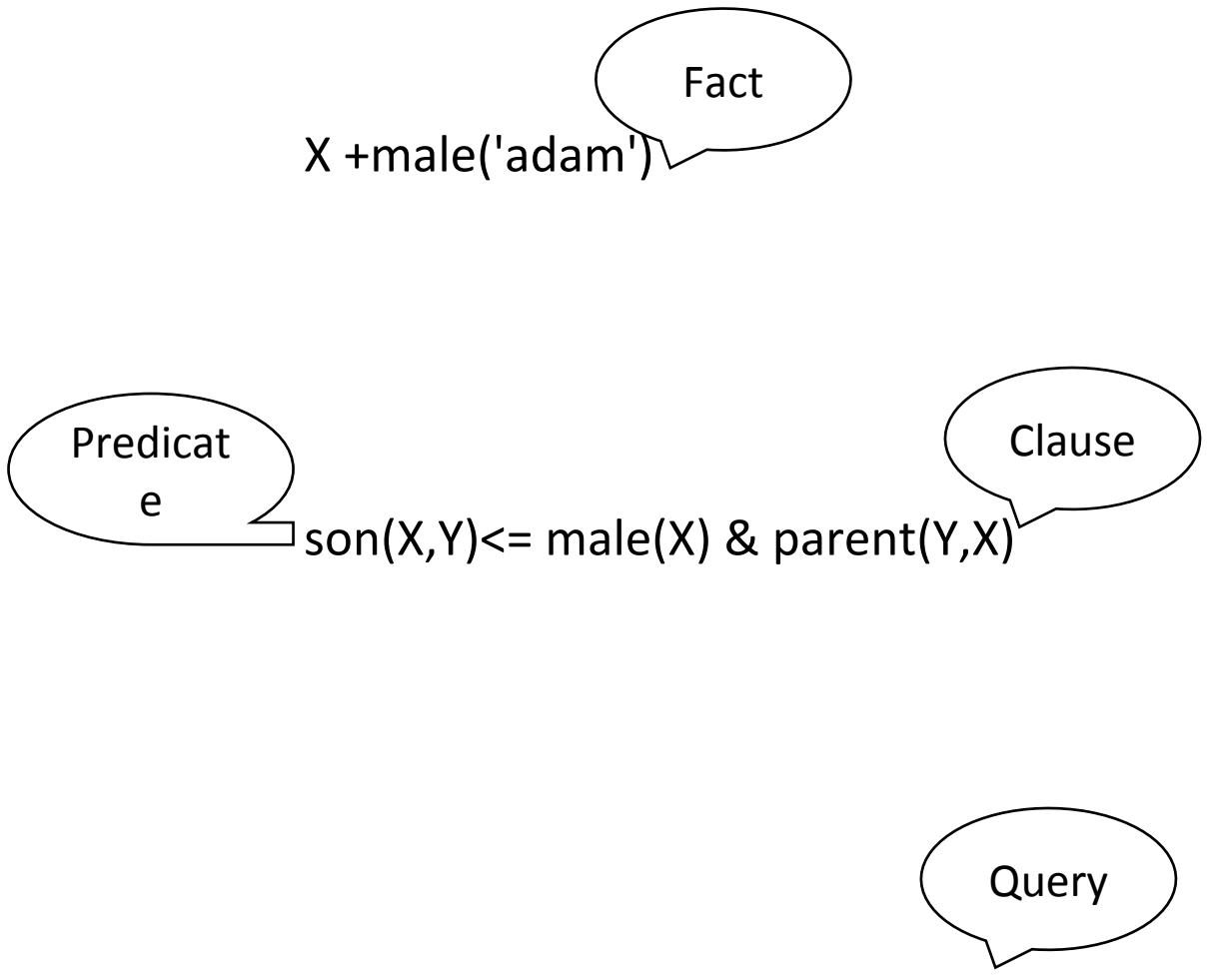
```
{('jack',)}
X
```

```
print(father_of(X,'jess'))
```

Output:

```
jack
X
```

Anatomy Logical Programming Paradigm



Logical Programming Paradigm

```
from pyDatalog import pyDatalog
pyDatalog.create_atoms('parent,male,female,son,daughter,X,Y,Z')
+male('adam')
+female('anne')
+female('barney')
+male('james')
+parent('barney','adam')
+parent('james','anne')

#The first rule is read as follows: for all X and Y, X is the son of Y if there exists X and Y such that Y is the parent of X and X is male.
#The second rule is read as follows: for all X and Y, X is the daughter of Y if there exists X and Y such that Y is the parent of X and X is female.
son(X,Y)<= male(X) & parent(Y,X)
daughter(X,Y)<= parent(Y,X) & female(X)
print(pyDatalog.ask('son(adam,Y)'))
print(pyDatalog.ask('daughter(anne,Y)'))
print(son('adam',X))
```

Logical Programming Paradigm

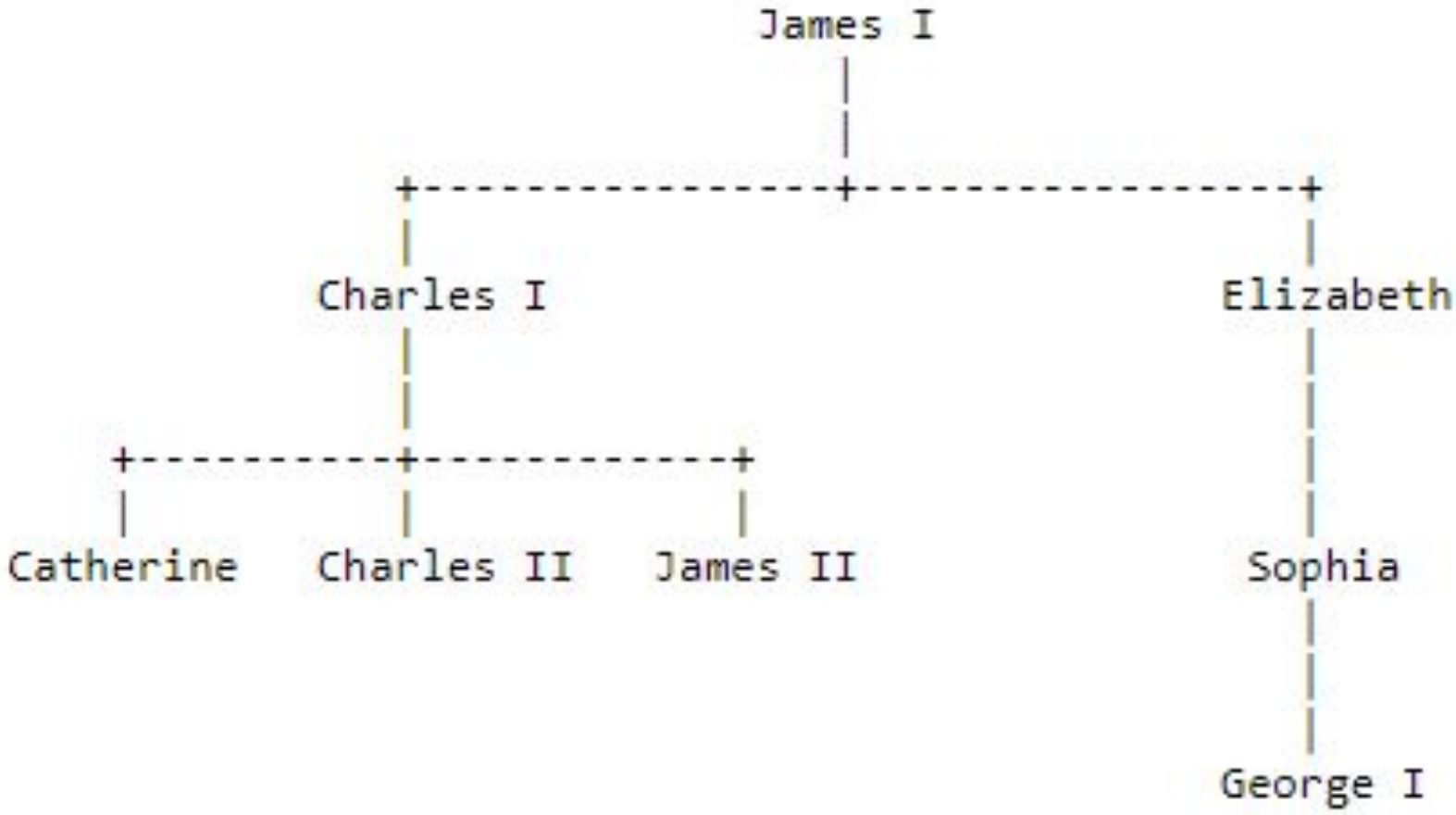
```
pyDatalog.create_terms('factorial, N')

factorial[N] = N*factorial[N-1]

factorial[1] = 1

print(factorial[3]==N)
```

Logical Programming Paradigm



Logical Programming Paradigm

```
from pyDatalog import pyDatalog

pyDatalog.create_terms('X,Y,Z, works_in, department_size, manager, indirect_manager, count_of_indirect_reports')

# Mary works in Production

+works_in('Mary', 'Production')

+works_in('Sam',  'Marketing')

+works_in('John', 'Production')

+works_in('John', 'Marketing')

+(manager['Mary'] == 'John')

+(manager['Sam']  == 'Mary')

+(manager['Tom']  == 'Mary')


indirect_manager(X,Y) <= (manager[X] == Y)

print(works_in(X,  'Marketing'))

indirect_manager(X,Y) <= (manager[X] == Z) & indirect_manager(Z,Y)

print(indirect_manager('Sam',X))
```

Logical Programming Paradigm

Lucy is a Professor

Danny is a Professor

James is a Lecturer

All professors are Dean

Write a Query to reterive all deans?

Soln

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('X,Y,Z,professor,lecturer, dean')
+professor('lucy')
+professor('danny')
+lecturer('james')
dean(X)<=professor(X)
print(dean(X))
```

Logical Programming Paradigm

```
likes(john, susie).                 /* John likes Susie */
likes(X, susie).                    /* Everyone likes Susie */
likes(john, Y).                     /* John likes everybody */
likes(john, Y), likes(Y, john).     /* John likes everybody and everybody likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary */
not(likes(john,pizza)).             /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes Mary.


rules
friends(X,Y) :- likes(X,Y),likes(Y,X).          /* X and Y are friends if they like each other */
hates(X,Y) :- not(likes(X,Y)).                  /* X hates Y if X does not like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if they don't like each other */
```

Dependent Types Paradigms

Dependent Types Paradigms

Unit-IV (15 Session)

Session 6-10 cover the following topics:-

- *Dependent Type Programming Paradigm*- S6-SLO1
- *Logic Quantifier: for all, there exists*- S6-SLO2
- *Dependent functions, dependent pairs*– S7-SLO 1
- *Relation between data and its computation*– S7-SLO 2
- *Other Languages: Idris, Agda, Coq S8*-SLO 1
- Demo: Dependent Type Programming in Python S8-SLO2

Lab 11: Dependent Programming (Case Study) (S8)
Assignment : Comparative study of Dependent programming in *Idris, Agda, Coq*

TextBook:

1) Amit Saha, Doing Math with Python: Use Programming to Explore Algebra, Statistics,Calculus and More, Kindle Edition, 2015

URL :

- https://tech.peoplefund.co.kr/2018/11/28/programming-paradigm-and-python-eng.html

- https://freecontent.manning.com/a-first-example-of-dependent-data-types/

Introduction

- A constant problem:
- Writing a correct computer program is hard
- Proving that a program is correct is even harder
- Dependent Types allow us to write programs and know they are correct before running them.

What is correctness?

- What does it mean to be “correct”?
- Depends on the application domain, but could mean one or more of:
  - **Functionally correct** (e.g. arithmetic operations on a CPU)
  - **Resource safe** (e.g. runs within memory bounds, no memory leaks, no accessing unallocated memory, no deadlock. . .)
  - **Secure** (e.g. not allowing access to another user’s data)

What is Type?

- In **programming,** types are a means of classifying values
- Exp: values 94, "thing", and [1,2,3,4,5] □ classified as an integer, a string, and a list of integers
- For a ***machine***, types describe how bit patterns in memory are to be interpreted.
- For a ***compiler*** **or** ***interpreter***, types help ensure that bit patterns are interpreted consistently when a program runs.
- For a ***programmer***, types help name and organize concepts, aiding documentation and supporting interactive editing environments.

Introductions

- In computer science and logic, a dependent type is a type whose definition depends on a value.
- It is an overlapping feature of type theory and type systems.
- Used to encode logic's quantifiers like "for all" and "there exists".
- Dependent types may help reduce bugs by enabling the programmer to assign types that further restrain the set of possible implementations.
- Exp: Agda, ATS, Coq, F*, Epigram, and Idris

Dependent Type Example

- **Exp matrix arithmetic**
- **Matrix type -** refined it to include the number of rows and columns.
- Matrix 3 4 is the type of 3 × 4 matrices.
- In this type, 3 and 4 are ordinary values.
- A ***dependent type***, such as Matrix, is a type that's calculated from some other values.
- In other words, it *depends on* other values.
- **Definition**
  - A data type is a type which is computed from a ***dependent*** other value.

Elements of dependent types

- **Dependent functions**
  - The return type of a dependent function may depend on the *value* (not just type) of one of its arguments
  - For instance, a function that takes a positive integer n may return an array of length n, where the array length is part of the type of the array.
  - (Note that this is different from polymorphism and generic programming, both of which include the type as an argument.)
- **Dependent pairs**
  - A dependent pair may have a second value of which the type depends on the first value

Formal definition

Π type [edit]

Loosely speaking, dependent types are similar to the type of an indexed family of sets. More formally, given a type $A : \mathcal{U}$ in a universe of types $\mathcal{U}$, one may have a **family of types** $B : A \to \mathcal{U}$, which assigns to each term $a : A$ a type $B(a) : \mathcal{U}$. We say that the type *B(a)* varies with *a*.

A function whose type of return value varies with its argument (i.e. there is no fixed codomain) is a **dependent function** and the type of this function is called **dependent product type**, **pi-type** or **dependent function type**.[3] For this example, the dependent function type is typically written as

$$\prod_{x:A} B(x)$$

or

$$\prod_{x:A} B(x).$$

If $B : A \to \mathcal{U}$ is a constant function, the corresponding dependent product type is equivalent to an ordinary function type. That is, $\prod_{x:A} B$ is judgmentally equal to $A \to B$ when *B* does not depend on *x*.

Formal definition

Σ type [edit]

The dual of the dependent product type is the **dependent pair type**, **dependent sum type**, **sigma-type**, or (confusingly) **dependent product type**.[3] Sigma-types can also be understood as existential quantifiers. Continuing the above example, if, in the universe of types $\mathcal{U}$, there is a type $A : \mathcal{U}$ and a family of types $B : A \to \mathcal{U}$, then there is a dependent pair type

$$\sum_{x:A} B(x).$$

The dependent pair type captures the idea of an ordered pair where the type of the second term is dependent on the value of the first. If

$$(a, b) : \sum_{x:A} B(x),$$

then $a : A$ and $b : B(a)$. If B is a constant function, then the dependent pair type becomes (is judgementally equal to) the product type, that is, an ordinary Cartesian product $A \times B$.

Pseudo-code

- **General Code**

```
float myDivide(float a, float
  b)
 { if (b == 0)
return ???;
Else
 return a / b;
 }
```

- **Dependent Type Code**

```
float myDivide3
(float a, float b, proof(b != 0) p)
{
return a / b;
 }
```

Auto Checking done here

Python Simple Example

```
from typing import Union def return_int_or_str(flag: bool) -> Union[str, int]:
if flag:
 return 'I am a string!‘
 return 0
```

Dependent Type

- » pip install mypy typing_extensions
- from typing import overload
- from typing_extension import Literal

Literal

Literal type represents a specific value of the specific type.

```
from typing_extensions import Literal
 def function(x: Literal[1]) -> Literal[1]:
 return x
function(1)
# => OK!
function(2)
 # => Argument has incompatible type "Literal[2]"; expected "Literal[1]"
```

Python Example

x :: Iterator[T1]

y :: Iterator[T2]

z :: Iterator[T3]

product(x, y) :: Iterator[Tuple[T1, T2]]

product(x, y, z) :: Iterator[Tuple[T1, T2, T3]]

product(x, x, x, x) :: Iterator[Tuple[T1, T1, T1, T1]]

- All the above replaced by
  - def product(*args :Tuple[n]) -> Iterator[Tuple[n]]: pass

A first example: classifying vehicles by power source IDRIS Exampl

Listing 1 Defining a dependent type for vehicles, with their power source in the type (vehicle.idr)

```
data PowerSource = Petrol | Pedal                    ❶
data Vehicle : PowerSource -> Type where             ❷
Bicycle : Vehicle Pedal                    ❸
Car : (fuel : Nat) -> Vehicle Petrol          ❹
  Bus : (fuel : Nat) -> Vehicle Petrol          ❹
```

❶ An enumeration type describing possible power sources for a vehicle
❷ A Vehicle's type is annotated with its power source
❸ A vehicle powered by pedal
❹ A vehicle powered by petrol, with a field for current fuel stocks

IDRIS Second Example

Listing 2 Reading and updating properties of Vehicle

```
wheels : Vehicle power -> Nat                ❶

wheels Bicycle = 2 wheels (Car fuel) = 4 wheels (Bus fuel) = 4

refuel : Vehicle Petrol -> Vehicle Petrol    ❷
refuel (Car fuel) = Car 100 refuel (Bus fuel) = Bus 200
```

❶Use a type variable, power, because this function works for all possible vehicle types.

❷ Refueling only makes sense for vehicles that carry fuel. Restrict the input and output type to Vehicle Petrol.

References

- http://www.cs.ru.nl/dtp11/slides/brady.pdf
- https://freecontent.manning.com/a-first-example-of-dependent-data-types/
- https://en.wikipedia.org/wiki/Dependent_type
- https://livebook.manning.com/book/type-driven-development-with-idris/chapter-1/13
- https://github.com/python/mypy/issues/366