UNIT-III: Declarative Programming Paradigm: Functional Programming

Faculty In-charge

Mrinmoyee Mukherjee
Assistant Professor (IT Dept.)
email: mrinmoyeemukherjee@sfit.ac.in
Mob: 9324378409

Academic Year: 2021-22

OUTLINE OF UNIT-3

Sub-Unit	Contents
3.0	Introduction to functional programming
3.1	Introduction to Lambda Calculus
3.2	Functional Programming Concepts
3.3	Evaluation order
3.4	Higher order functions
3.5	I/O- Streams and Monads

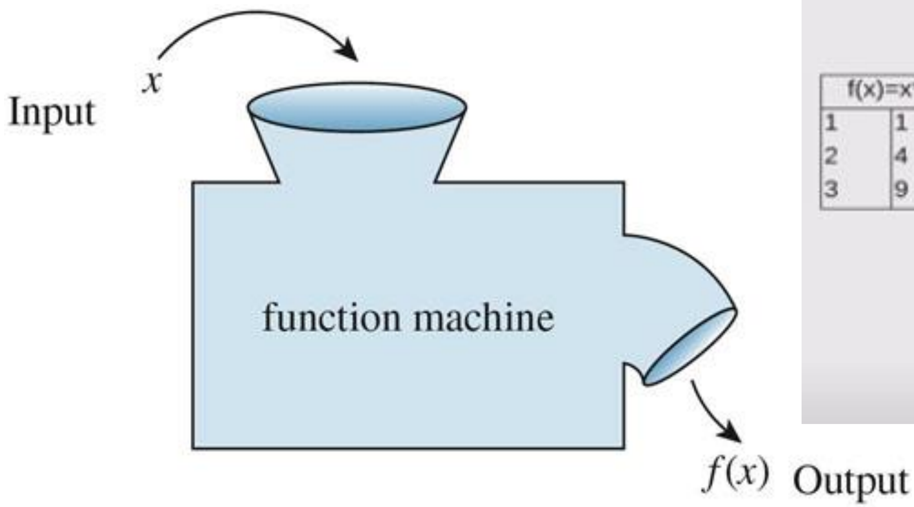
3.0: Introduction to Functional Programming

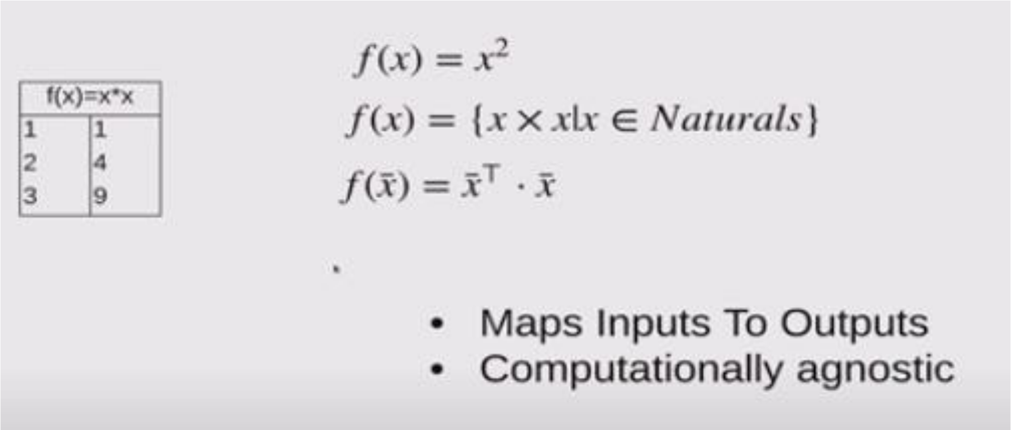
INTRODUCTION TO FUNCTIONAL PROGRAMMING

1. Functional programming starts with a point of view that a program is a function

2. Function can be thought of as a black box that takes input and produces output

3. A program is a function that transforms input to output





4. So when we are writing a program in a functional programming language, we are

- ✓ Specifying the rules on how to generate the output from a given input
- ✓ In computation we apply the rules to generate the expected output

Definition : **Functional programming** is a programming paradigm — a style of building the structure and elements of computer programs — that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data

Advantages of functional programming

- Functional programming is based on mathematical functions
- Easier to determine inputs
- Easier to determine outputs
- Easier to demonstrate prove that you have a correct program
- Easier to test programs that are too difficult to prove
- Examples of functional programming paradigm
- Haskell, Lisp, Python, Erlang, Racket, F#,

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.
- Functions are first class and can be higher order
- It allows us to handle functions as if they were normal data types
- Functions can either accept another function as argument or can return a function themselves
- Some of the popular functional programming languages include: Lisp, Python, Erlang, Haskell, Clojure, etc.

BUILDING UP PROGRAMs

How do we build up these programs……

Assume that

- We have some built in functions and values
- Use these to build more complex functions

- Example
- We have

Whole Numbers

Set of whole numbers:

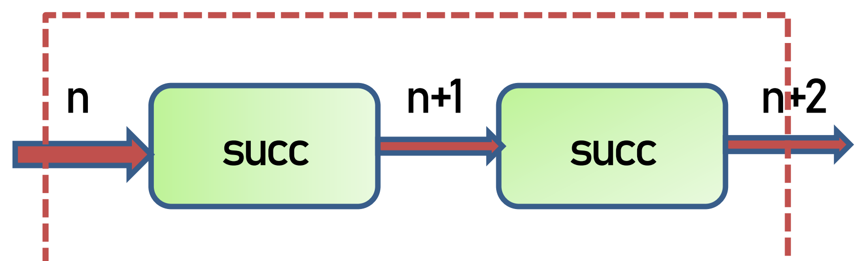
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, … }

Successor function, *succ*

succ 0=1
succ 1=2
succ 2=3

We can **compose succ** twice to built a new function

plusTwo n =succ(succ n)



Whole Numbers

Set of whole numbers:

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ... }

Successor function, *succ*

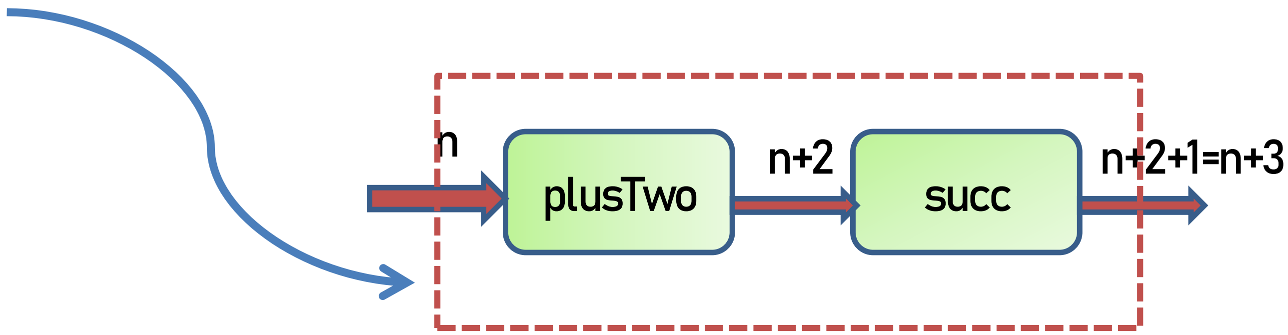
succ 0=1
succ 1=2
succ 2=3

function, plusTwo n

succ(succ n)

We can **compose succ** twice to built a new function

plusThree n =succ(plusTwo)



We can combine functions to form new compositions/function

CORE PROGRAMMING CHARACTERISTICS/CONCEPTS

1. PURE Functions: Functions are pure in Functional Programming
   - A function called multiple times with the same arguments will always return the same value. Always.

2. Functions are first class and can be higher order
   - It allows you to handle functions as if they were normal data types
   - Functions can either accept another function as argument or can return a function themselves

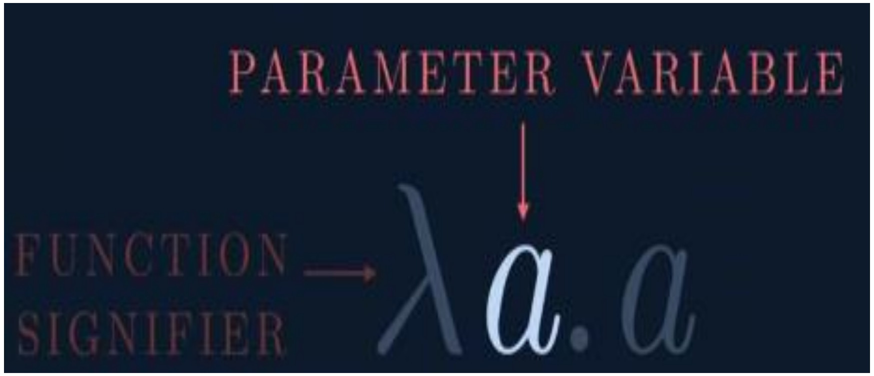
3. Variables are Immutable
   - You can't modify a variable after it has been initialized
   - You can create new variables, but you can't modify the existing variables

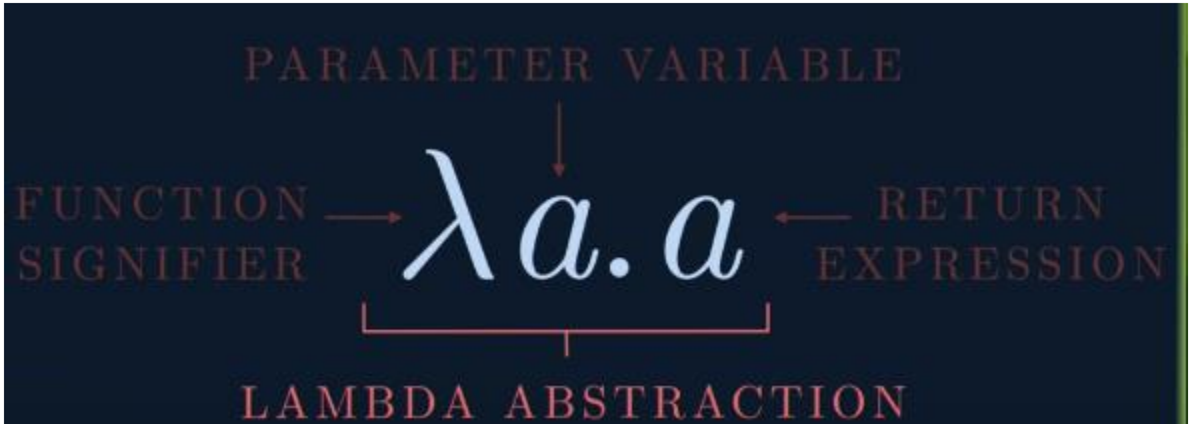
4. Functional programming is based on Lambda Calculus

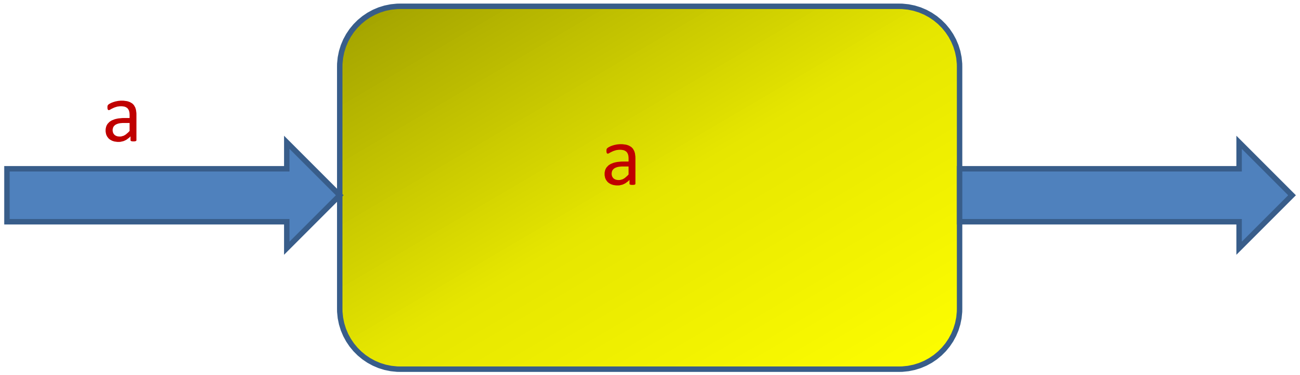
Definition:

Lambda calculus (also written as **λ-calculus**) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

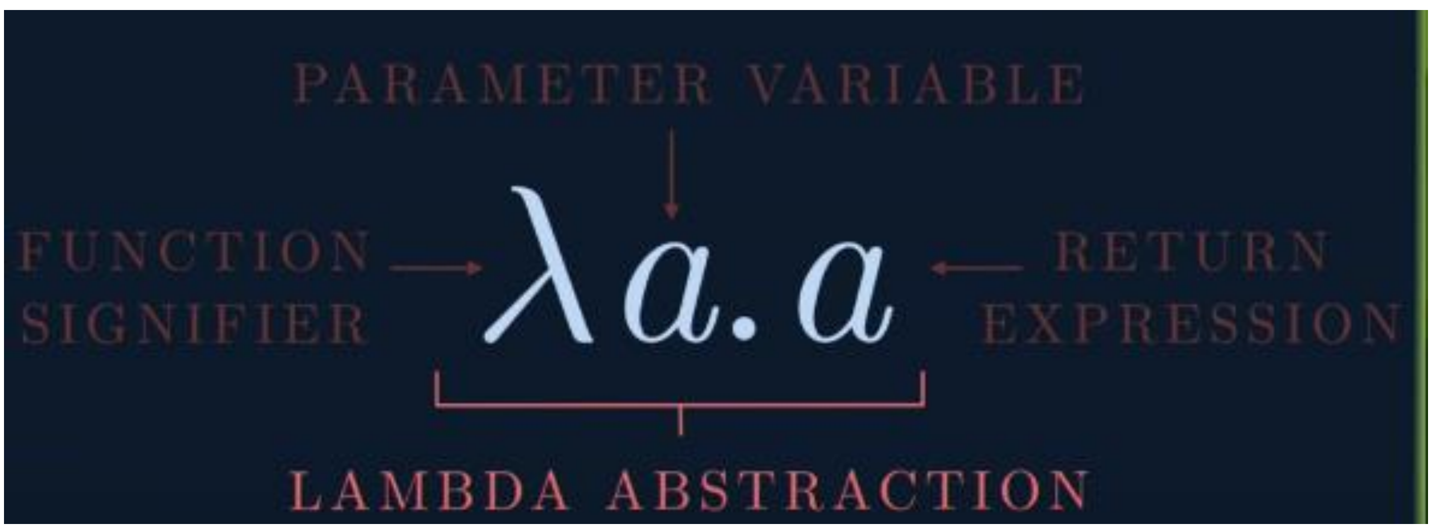
INTRODUCTION TO LAMBDA CALCULUS



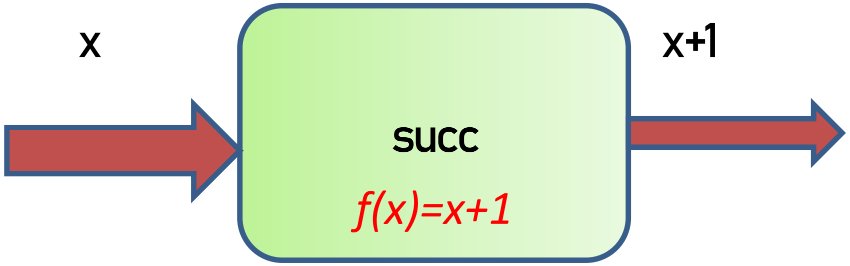



a
a

λa.a


PARAMETER VARIABLE
FUNCTION SIGNIFIER
λa.a
RETURN EXPRESSION
LAMBDA ABSTRACTION

FUNCTIONS USING LAMBDA CALCULUS-SUCC FUNCTION



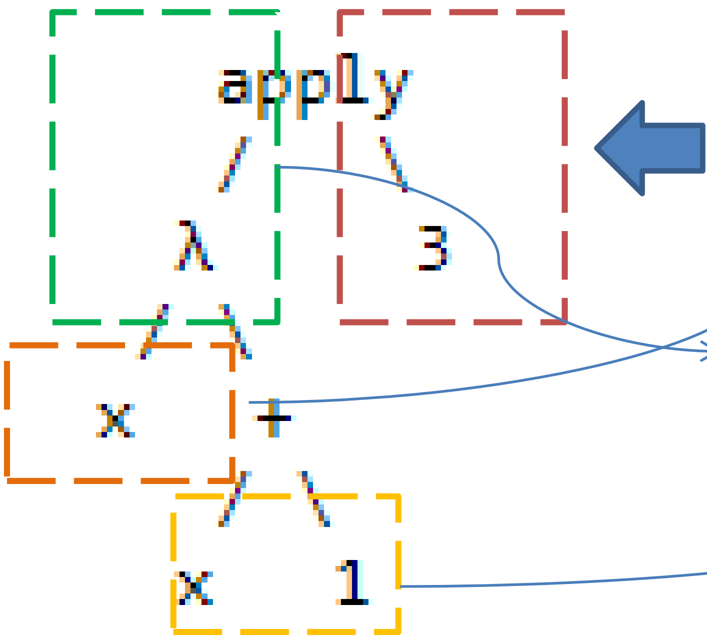
$f(x)=x+1$

$\lambda x.\ x+1$

$(\lambda x.x+1)3$

$(\lambda x.x+1)3 \Rightarrow 3+1 \Rightarrow 4$

abstract-syntax tree (where λ is the abstraction operator, and apply is the application operator)



The right sub-tree is the argument to the function

The left child of the lambda is the formal parameter.

The left sub-tree is the (with lambda at its root) function

The right child of the lambda is the function body.

There is only one apply node in our example;

- The argument is 3
- The function is λx.x+1;
- The formal parameter is x
- The function body is x+1. Here's the rewriting step:

```
      apply     =>      +
      /   \            / \
     λ     3          3   1
    / \
   x   +
      / \
     x   1
```

FUNCTIONS USING LAMBDA CALCULUS-PLUSONE, PLUSTWO

$$(\lambda x.x+1)((\lambda y.y+2)3)$$

- ✓ The first lambda expression defines the "plus-one" function
- ✓ The argument to that function is itself an application
- ✓ which applies the "plus-two" function to the value 3

```
    apply         =>   apply     =>  apply   =>  +  =>  6
   /     \             /   \         /   \      / \
  λ       apply       λ     +       λ     5    5   1
 / \       /  \      / \   / \     / \
x   +     λ    3    x   + 3   2   x   +
   / \   / \           / \           / \
  x   1 y   +         x   1         x   1
           / \
          y   2
```

FREE and BOUND VARIABLES

In an expression, each appearance of a variable is either "free" (to λ) or "bound" (to a λ).

- ✓ **Bound Variable**: a variable that is associated with some lambda.
- ✓ **Free Variable**: a var that is *not* associated with any lambda.

(λx.xy)

- ✓ The variable x is bound and y is free.

(λx.x)(λy.yx)

- ✓ The x in the body of the first expression from the left is bound to the first λ.
- ✓ The y in the body of the second expression is bound to the second λ and the x is free

It is very important to notice that the x in the second expression is totally independent of the x in the first expression.

CHARACTERISTICS OF LAMBDA CALCULUS

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.

CHARACTERISTICS OF LAMBDA CALCULUS

Function Application

- 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.

- Evaluating Lambda Calculus:

- Ex1: (+ (* 5 6) (* 8 3))
- Here, we can't start with '+' because it only operates on numbers. There are two reducible expressions: (* 5 6) and (* 8 3).

- We can reduce either one first. For example −

(+ (* 5 6) (* 8 3))

(+ 30 (* 8 3))

(+ 30 24)

= 54

DATA TYPES

- Bool
- Char
- Int
- Float
- Double
- List
- Tuple
- Function

- In Haskell all computations are done via the evaluation of expressions
- Examples of expressions include atomic values (built-in) such as
  - the integer 5,
  - the character 'a', and
  - the function \x -> x+1, as well as structured values such as
  - the list [1,2,3] and
  - the pair ('b',4).

Types (set of Values)

- Bool
- Char
- Int (64 bit)
- Integer (Superset of Int)
- Float
- Double
- List
- Tuple
- Function

Examples

- :type True
- :type “hi”
- :type 5
- :type 5.34
- :type (True, False)

Type Class

- **EQ**
  - Type class is an interface which provides the functionality to test the equality of an expression.
- **Num and Fractional**
  - This type class is used for numeric operations. Types such as Int, Integer, Float, and Double come under this Type class.
- **Integral**
  - sub-class of the Num Type Class.
  - Int and Integer are the types under this Type class.
- **Floating**
  - sub-class of the Num Type Class.
  - Float and Double come under this type class.

ARITHMETIC AND LOGICAL OPERATORS

- 2 + 3
- 2 – 3
- 2 * 3
- **2 * (-3)**
- 2 / 3
- it (result)
- 50 * 100 - 4999
- 50 * (100 – 4999)
- 40*100-3000+50/5
- (40*100-3000+50)/5
- 2 + “hi”

- False
- True
- True && False
- True && True
- False || True
- not False
- not (True && False)
- not (True || False)

```
Prelude> :t "a"
Prelude> '\97'
'a'
Prelude> '\67' 'C'
Prelude> :t "mrinmoyee.in"
Prelude> [1,2,3,4,5]
Prelude> (1,1,'a')
```

COMPARATIVE OPERATORS

- 2 == 3
- 2 == 0
- 2 /= 2
- 2 /= 0
- 2 < 3
- 2 > 3
- 2 ^ 3
- not (2 < 3)
- “hi” = = “hi”
- “hi” = = “Hi”

INBUILT FUNCTIONs

1. succ 6
2. succ (succ 5)
3. min 5 6
4. max 5 6
5. max 101 101
6. succ 9 + max 5 4 + 1
7. (max 5 4)+(succ 9)+1
8. (succ 9) + (max 5 4) + 1
9. We wanted to get the successor of the product of numbers 9 and 10. we couldn't write **succ 9 * 10** because that would get the successor of 9, which would then be multiplied by 10
10. succ 9*10
11. succ (9*10)
12. **div 92 10**
13. div 3 4
14. div 4 3
15. 4/3
16. mod 7 5
17. mod 3 1
18. mod 7 2
19. reverse "hello"
20. x=45
21. print x
22. return True
23. return False
24. x <- return 35
25. print x
26. putStrLn "hello"