# 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 |

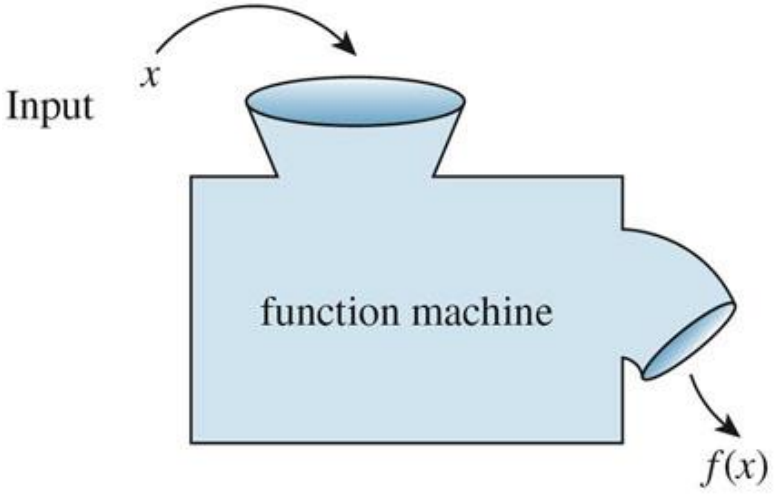
# 3.0: Introduction to Functional Programming

# 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




| f(x)=x*x | |
|---|---|
| 1 | 1 |
| 2 | 4 |
| 3 | 9 |

$$f(x) = x^2$$

$$f(x) = \{x \times x | x \in Naturals\}$$

$$f(\bar{x}) = \bar{x}^{\mathsf{T}} \cdot \bar{x}$$

- Maps Inputs To Outputs
- Computationally agnostic

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:

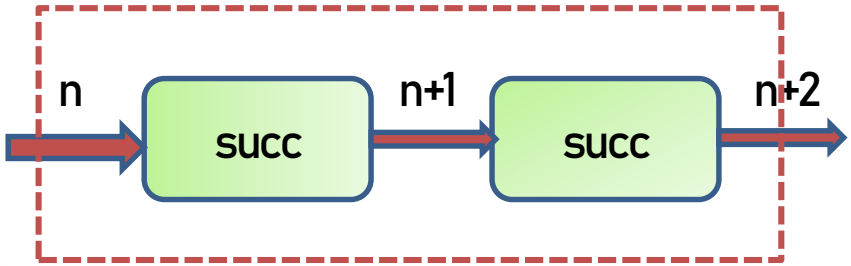
{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, … }

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*

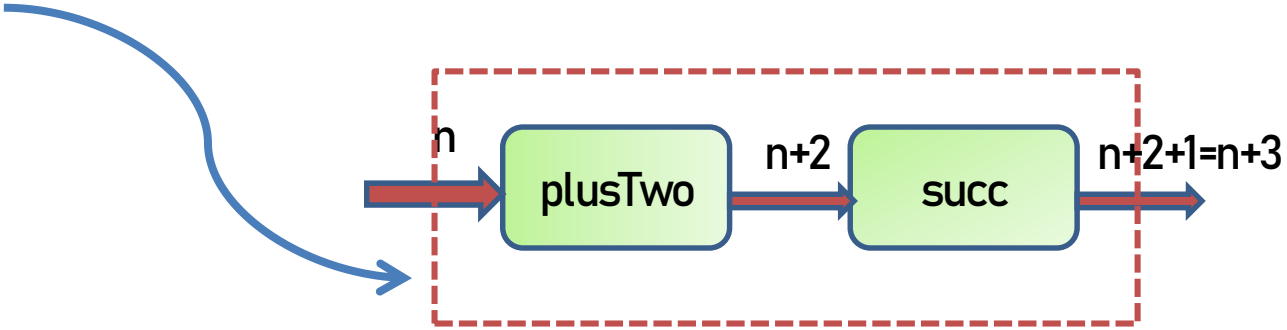
succ 0=1
succ 1=2
succ 2=3

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

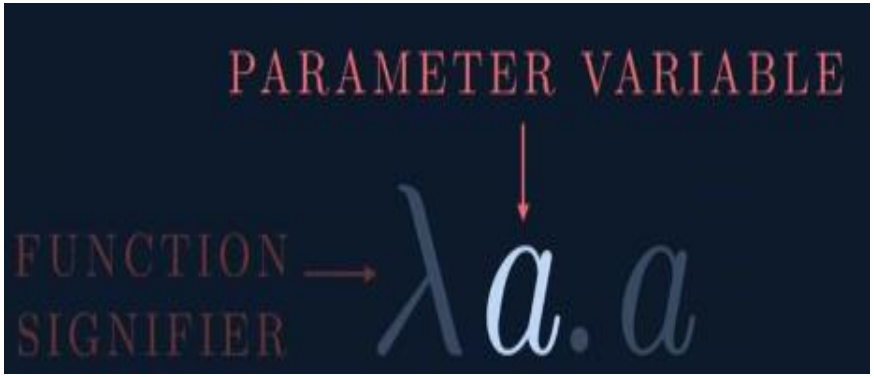
4. Functional programming is based on Lambda Calculus

**Definition:**

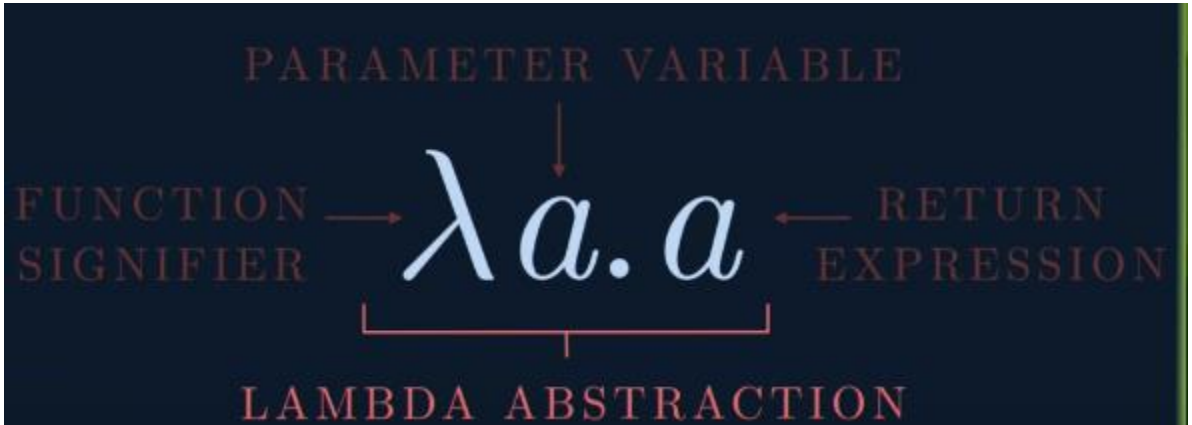
**Lambda calculus** (also written as **λ-calculus**) is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.

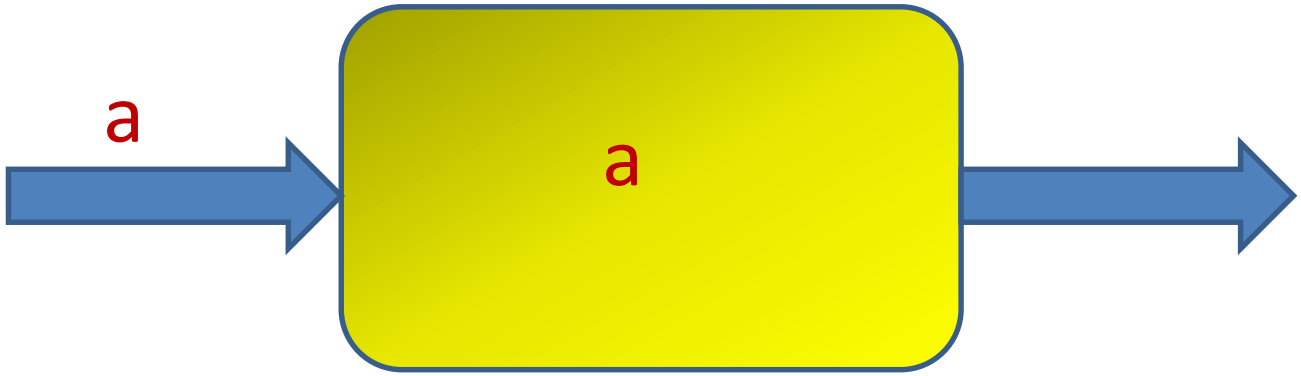
# INTRODUCTION TO LAMBDA CALCULUS



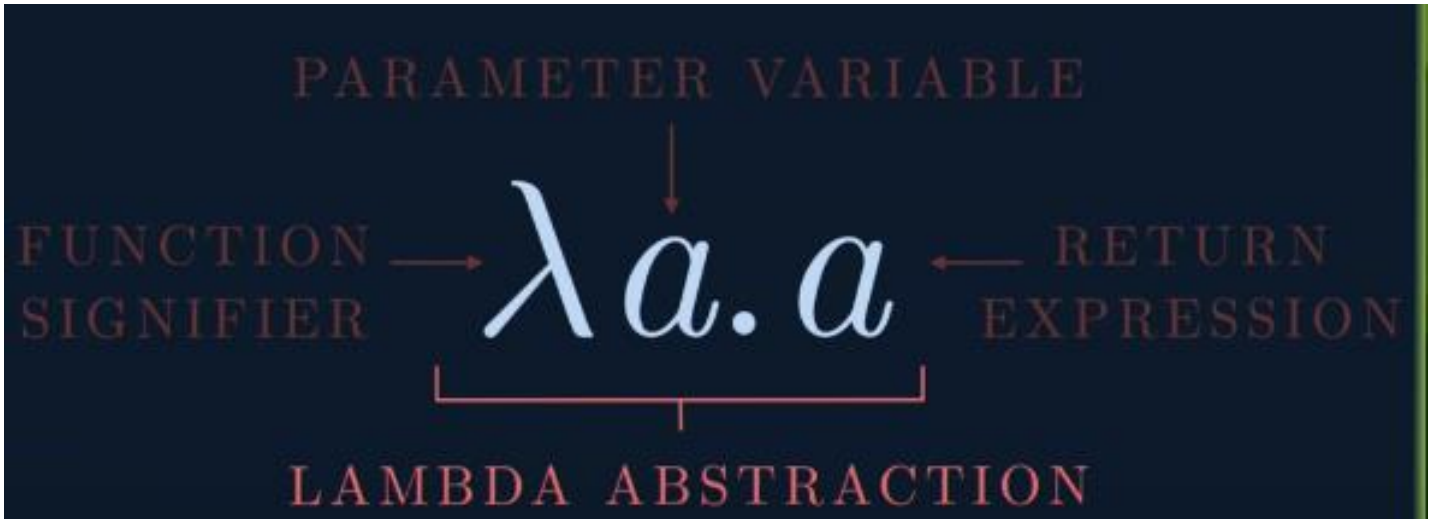




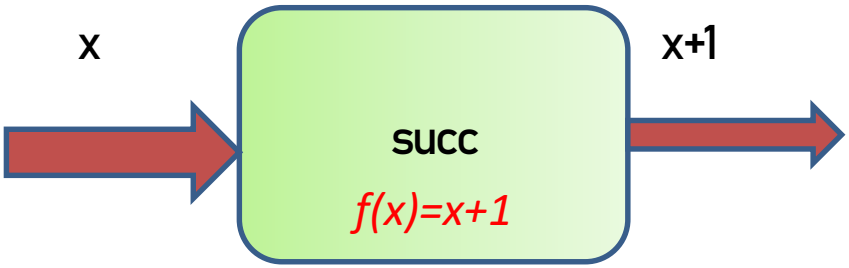





$\lambda a.a$



# FUNCTIONS USING LAMBDA CALCULUS-SUCC FUNCTION



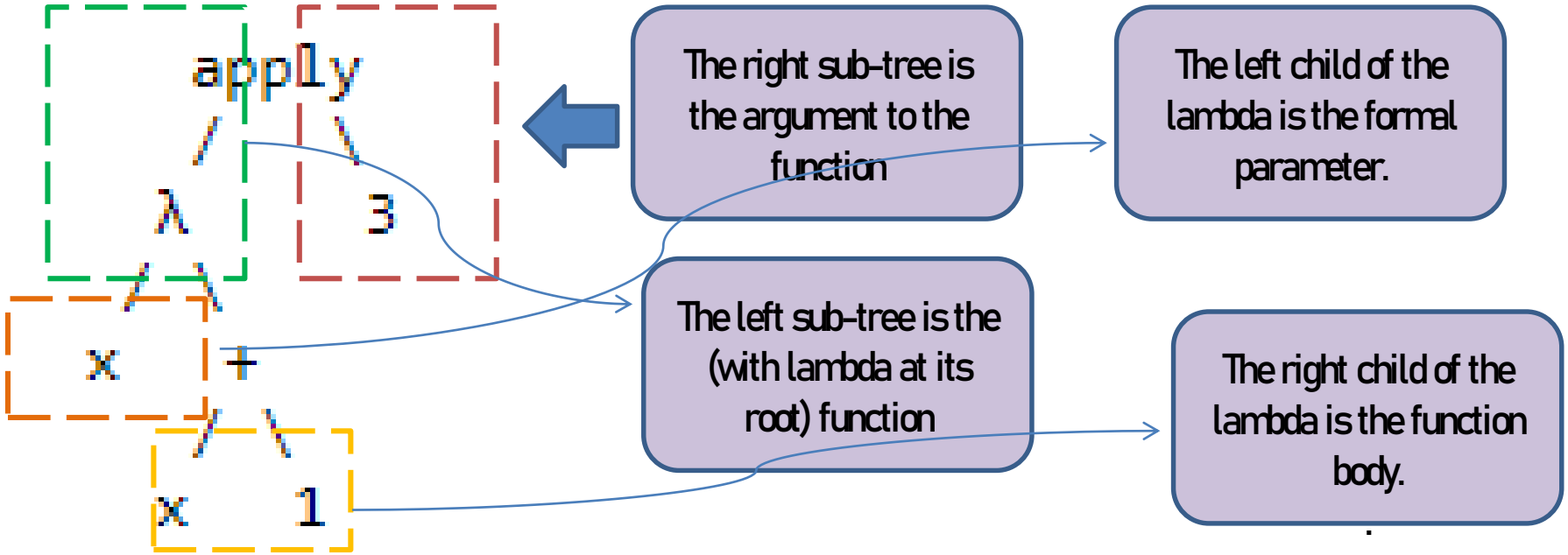

$f(x)=x+1$

$\lambda\ x.\ x+1$

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

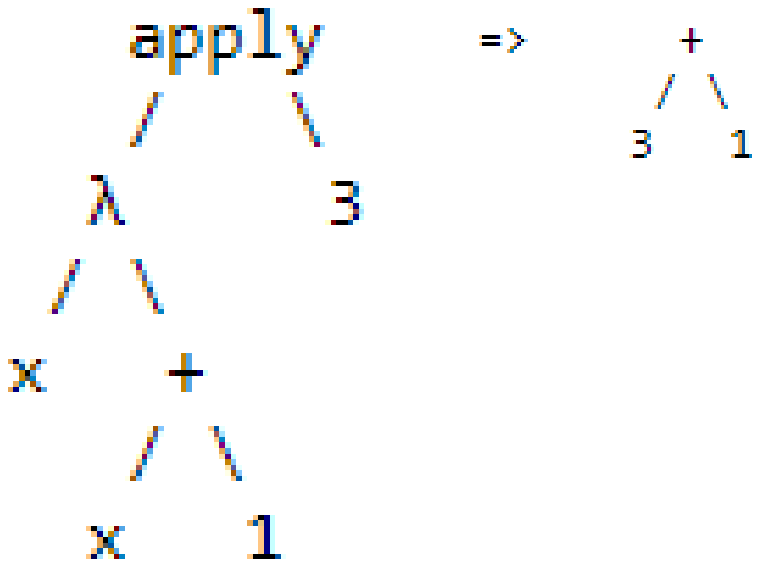
$(\lambda x.x+1)3 \Rightarrow 3+1 \Rightarrow 4$

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



There is only one apply node in our example;

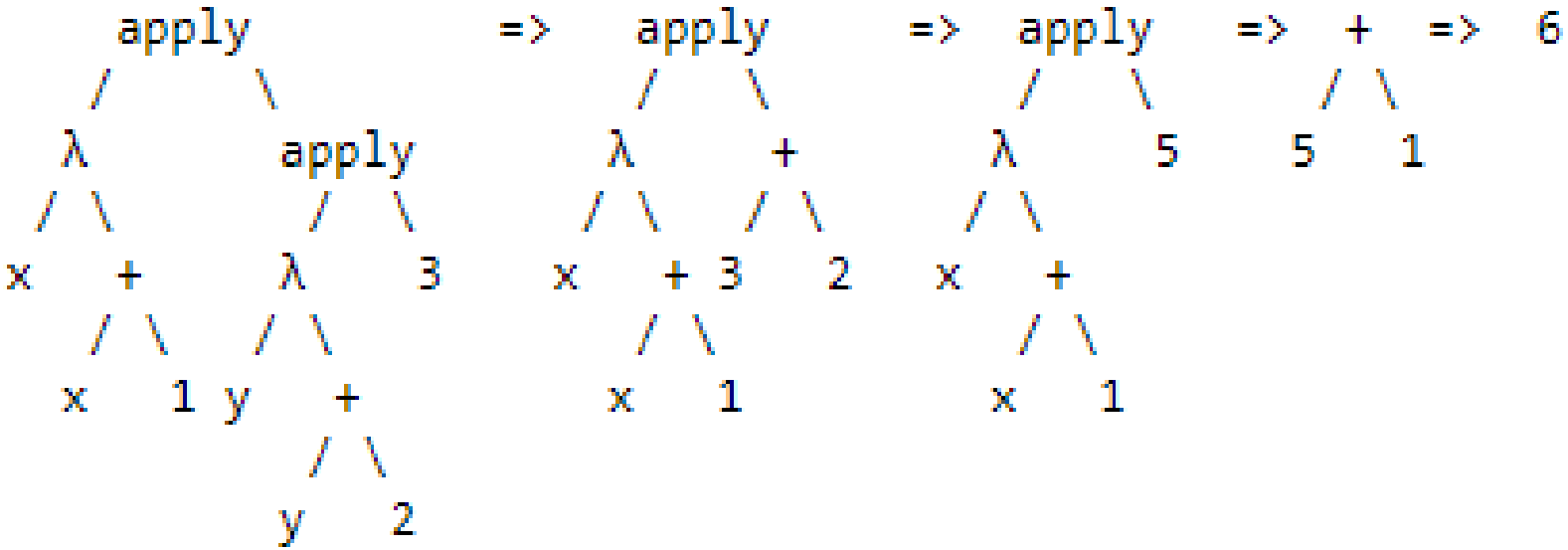
- The argument is 3
- The function is λx.x+1;
- The formal parameter is x
- The function body is x+1. Here's the rewriting step:



# 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



# 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.

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

(λx.x)(λy.yx) ←

- ✓ 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.

# 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"

# 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

# 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).

Clear screen command CNTL+L
Exit GHCI command CNTL+D
Exit Prelude :n
https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-invocation.html

## 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

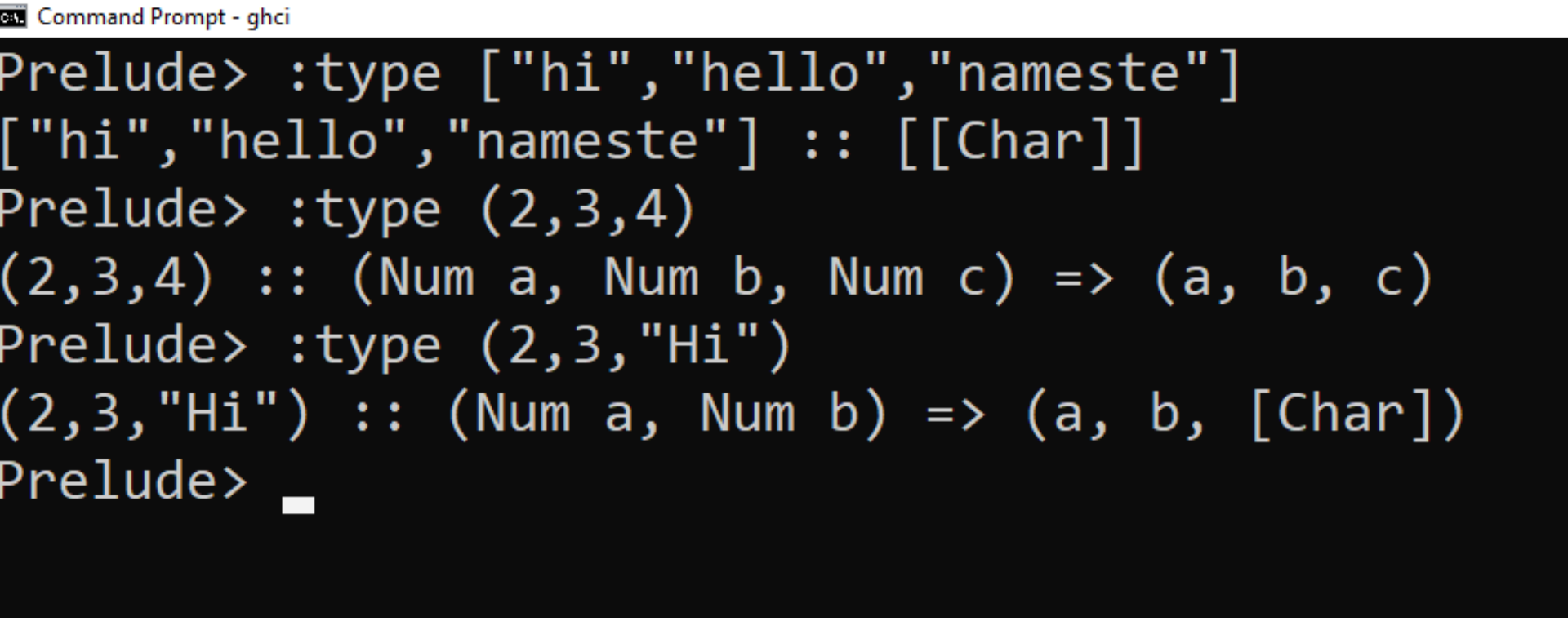
- **EQ**
  - Type class is an interface which provides the functionality to test the equality of an expression.
- **Num and Fractional**
  - This type class is used for numeric operations. Types such as Int, Integer, Float, and Double come under this Type class.
- **Integral**
  - sub-class of the Num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

# DATA TYPES

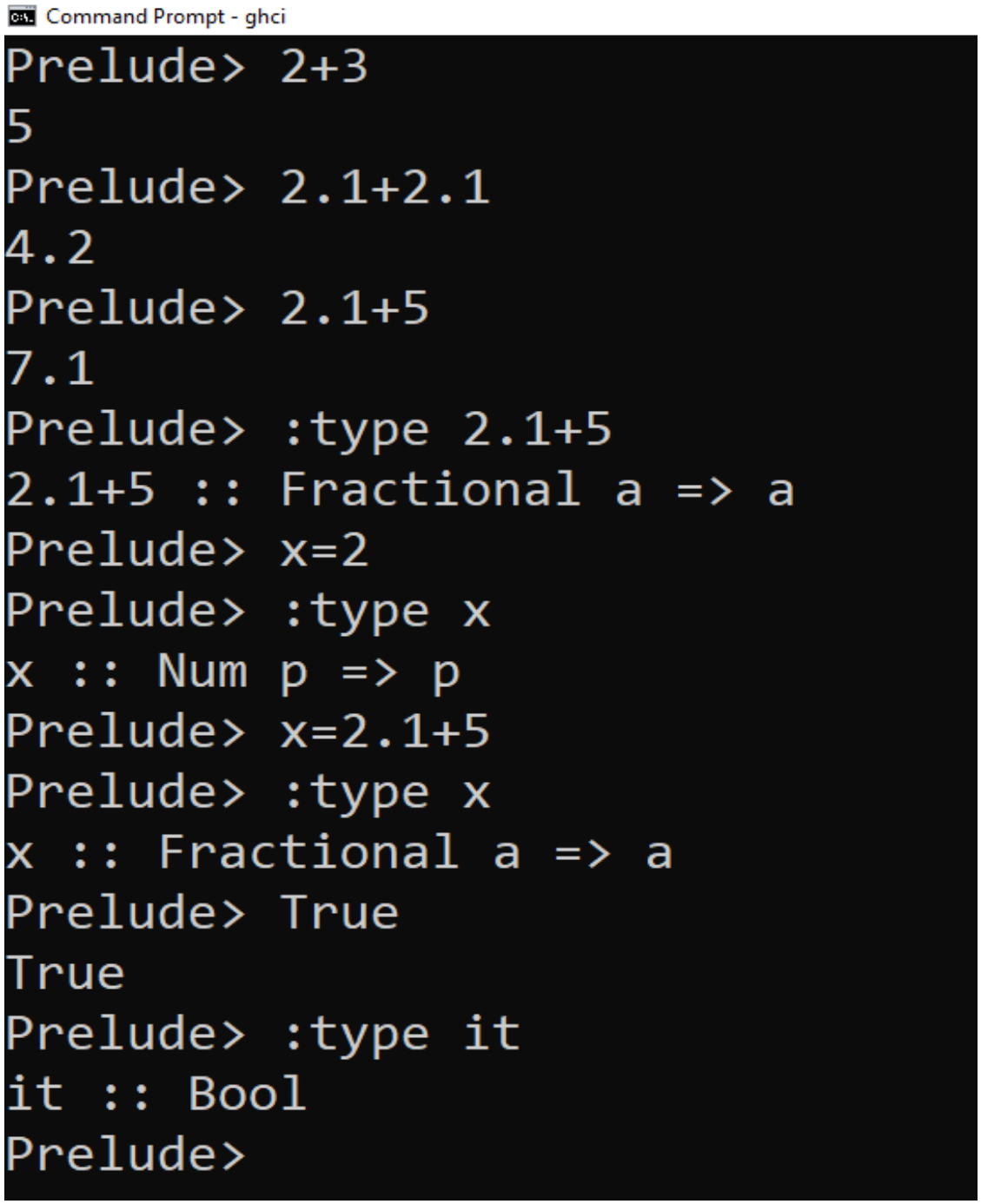
Command Prompt - ghci

```
C:\Users\Mrinmoyee>ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 2
2
Prelude> :type 2
2 :: Num p => p
Prelude> :type 2.1
2.1 :: Fractional p => p
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
Prelude> :type "Hello"
"Hello" :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]
Prelude> :type [2.1,3.1,4.1]
[2.1,3.1,4.1] :: Fractional a => [a]
Prelude>
```

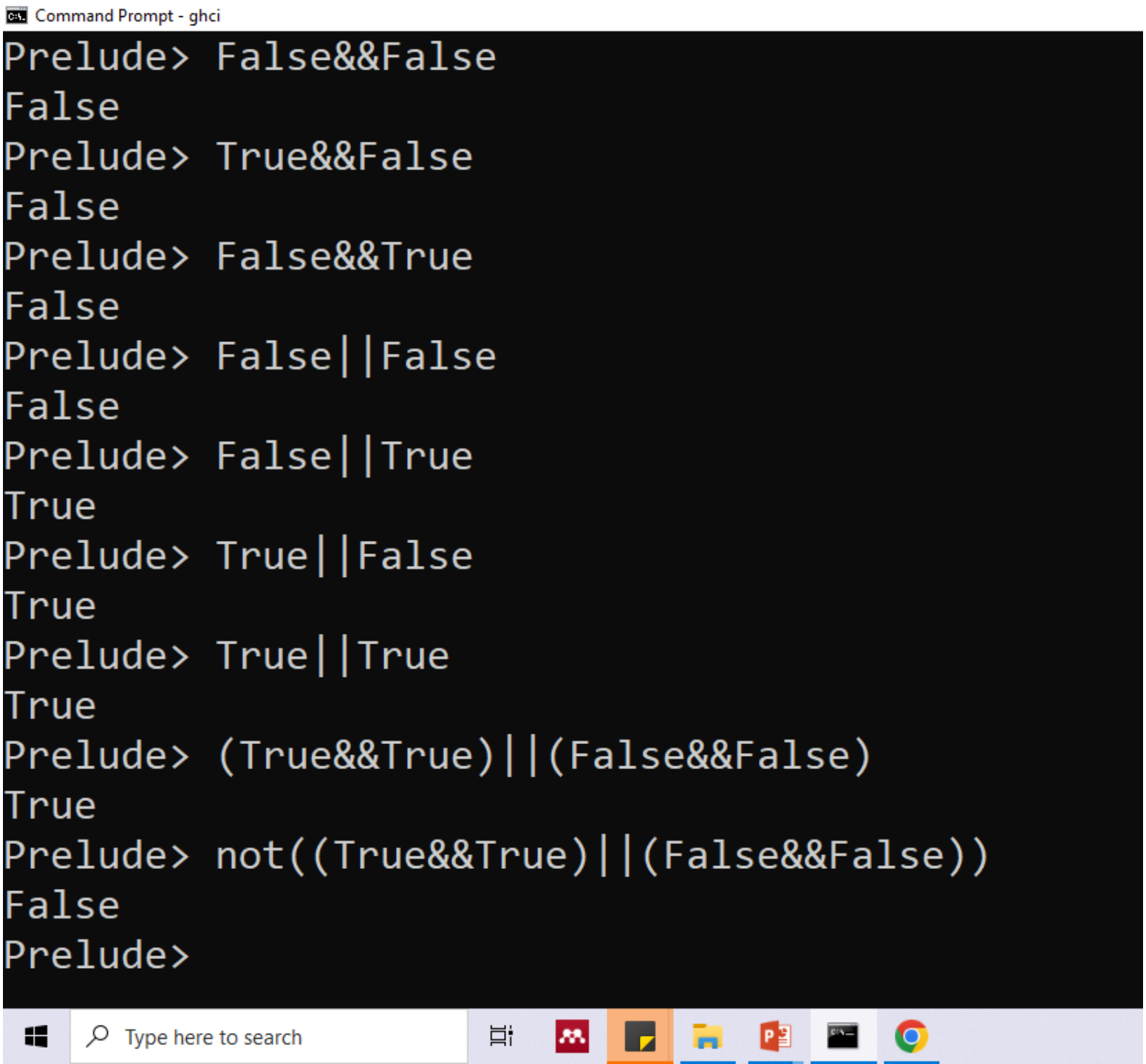
## DATA TYPES


Command Prompt - ghci
Prelude> :type ["hi","hello","nameste"]
["hi","hello","nameste"] :: [[Char]]
Prelude> :type (2,3,4)
(2,3,4) :: (Num a, Num b, Num c) => (a, b, c)
Prelude> :type (2,3,"Hi")
(2,3,"Hi") :: (Num a, Num b) => (a, b, [Char])
Prelude>

## ARITHMETIC OPERATORS



## LOGICAL OPERATORS



# COMPARATIVE OPERATORS

Command Prompt - ghci

```
Prelude> 2==2
True
Prelude> 2==3
False
Prelude> not(2==3)
True
Prelude> not(not(2==3))&&True
False
Prelude> not(not(2==3))&&True+2

<interactive>:36:17: error:
    * No instance for (Num Bool) arising from a use of `+'
    * In the second argument of `(&&)', namely `True + 2'
      In the expression: not (not (2 == 3)) && True + 2
      In an equation for `it': it = not (not (2 == 3)) && True + 2
Prelude>
```

Command Prompt - ghci

```
Prelude> 2/=2
False
Prelude> 2/=3
True
Prelude> 2<3
True
Prelude> 2>3
False
Prelude> not(2>3)
True
Prelude> 2^3
8
Prelude> 2**3
8.0
Prelude> 2^4
16
Prelude> 2**4
16.0
Prelude>
```

Command Prompt - ghci

```
Prelude> "Hi"=="Hi"
True
Prelude> "Hi"=="hi"
False
Prelude> not("Hi"=="hi")&&True
True
Prelude> not(not("Hi"=="hi")&&True)||False
False
Prelude> "Hello world"=="Hello world"
True
Prelude>
```

# LIST

- Much like shopping lists in the real world, lists in Haskell are very useful.
- It's the most used data structure
- In Haskell, lists are a **homogenous** data structure.
- It stores several elements of the same type.
- That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

> ❖ A **list** in Haskell can be written using square brackets with commas separating the list's individual values.
>
> ❖ Has only homogeneous values

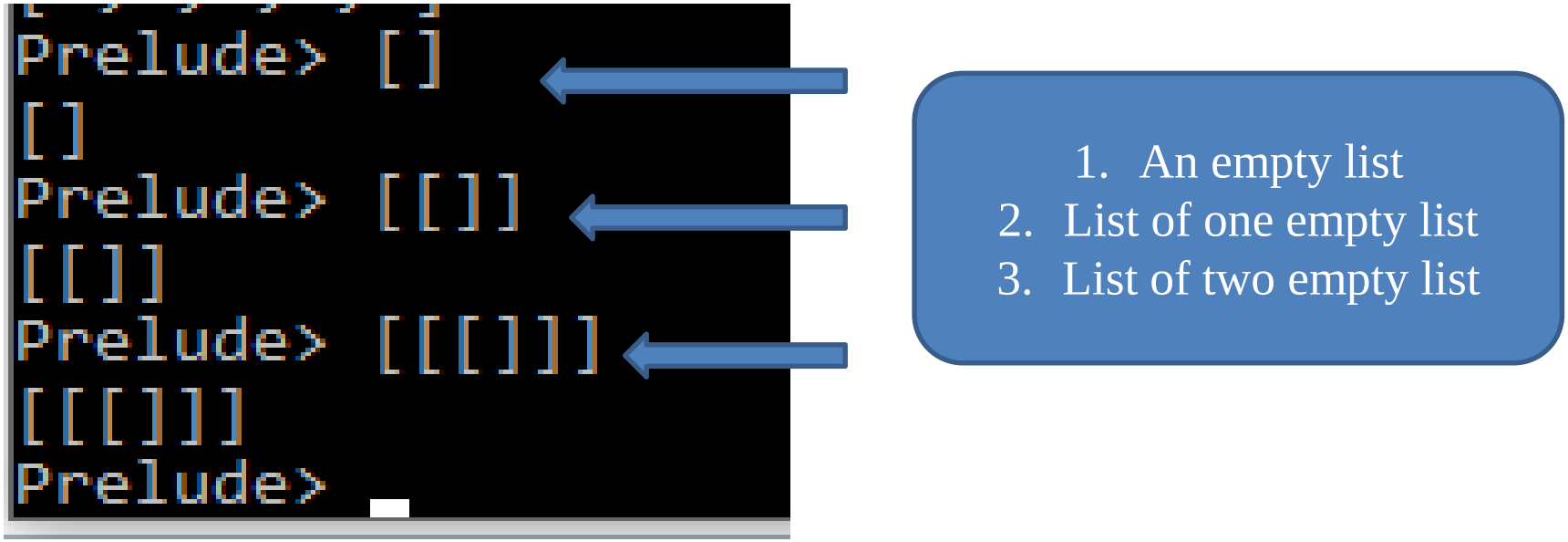
```
Prelude> let list1=[1,2,3,4]
Prelude> print list1
[1,2,3,4]
Prelude> let list2=[1,2,3,'a','b']
```

```
Prelude> [1,2,3,4]++[5,6,7,8]
[1,2,3,4,5,6,7,8]
Prelude> "hello"++ " "++"world"
"hello world"
Prelude> ['w','o']++['o','t']
"woot"
Prelude>
```

- Internally, Haskell has to walk through the whole list on the left side of ++
- That's not a problem when dealing with lists that aren't too big
- But putting something at the end of a list that's fifty million entries long is going to take a while
- However, putting something at the beginning of a list using the **:** operator (also called the cons operator) is instantaneous.

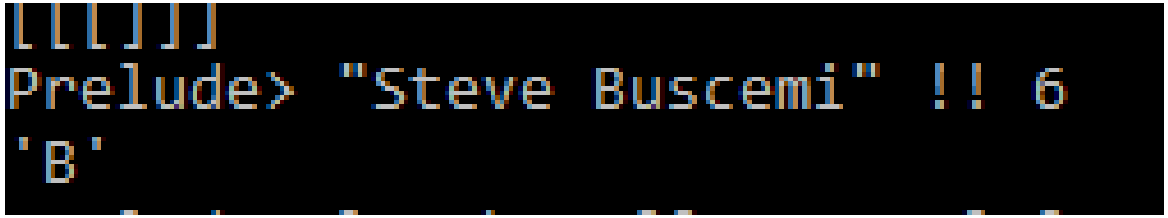
```
Prelude> 'A':" Small Cat"
"A Small Cat"
Prelude> 5:[1,2,3,4]
[5,1,2,3,4]
Prelude>
```

# LIST ....Continues....




1. An empty list
2. List of one empty list
3. List of two empty list

- If you want to get an element out of a list by index, use **!!**. The indices start at 0
- But if you try to get the sixth element from a list that only has four elements, you'll get an error so be careful!

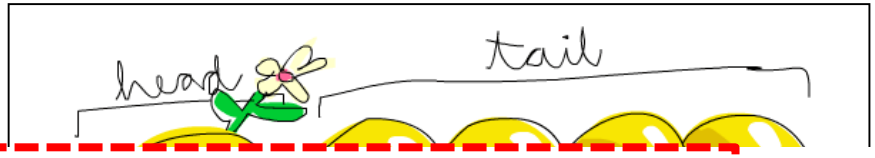


# LIST ....Continues....

Lists can also contain lists. They can also contain lists that contain lists that contain lists …

```
[[[]]]
Prelude> "Steve Buscemi" !! 6
'B'
Prelude> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
Prelude> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b !! 2
[1,2,2,3,4]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude>
```

# LIST ....Continues....




**head** takes a list and returns its head. The head of a list is basically its first element.

```
ghci> head [5,4,3,2,1]
5
```

**tail** takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

**last** takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

**init** takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

# LIST ....Continues....

`length` takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

`null` checks if a list is empty. If it is, it returns `True`, otherwise it returns `False`. Use this function instead of `xs == []` (if you have a list called `xs`)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` takes number and a list. It extracts that many elements from the beginning of the list. Watch.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
```

# LIST ....Continues....

`drop` works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` takes a list of stuff that can be put in some kind of order and returns the biggest element.

`minimum` returns the smallest.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

`sum` takes a list of numbers and returns their sum.

`product` takes a list of numbers and returns their product.

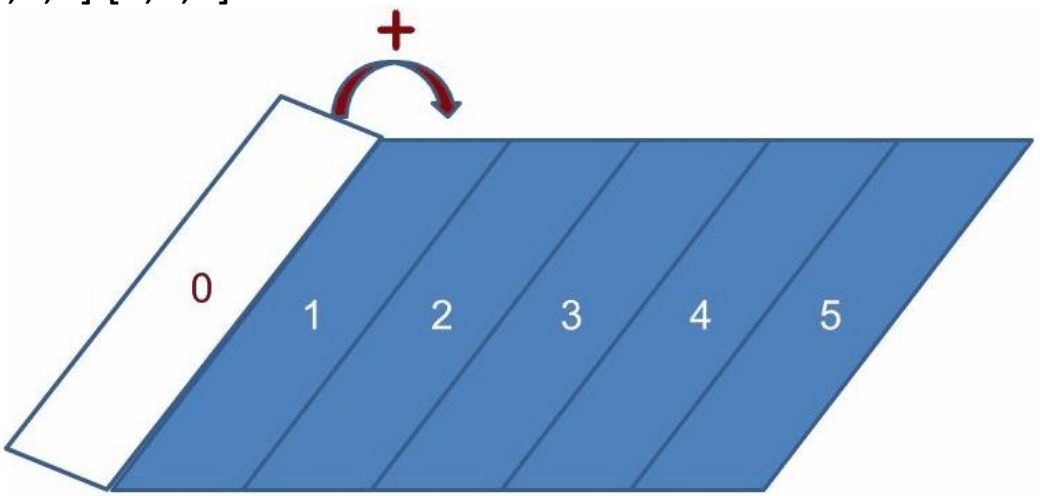
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

# LIST ....Continues....

`elem` takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

zipWith (*) [1,2,3] [2,3,4]




## What if we want a list of all numbers between 1 and 200?

# TUPLES...

- Are used when you know exactly how many values you want to combine
- Its type depends on how many components it has and the types of the components.
- They are denoted with parentheses and their components are separated by commas.
- Unlike a list, a tuple can contain a combination of several types.
  (represent someone's name and age in Haskell),**("Christopher", "Walken", 55)**
- Use tuples when you know in advance how many components some piece of data should have. There are singleton lists, there's no such thing as a singleton tuple.
- Like lists, tuples can be compared with each other if their components can be compared
- Only you can't compare two tuples of different sizes, whereas you can compare two lists of different sizes.

```
Prelude> fst (8,9)
8
Prelude> fst (8,"wow")
8
Prelude> snd (8,9)
9
Prelude> snd (8,"wow")
"wow"
Prelude> zip [1,2,3,4,5] [6,6,6,6,6]
[(1,6),(2,6),(3,6),(4,6),(5,6)]
Prelude> zip [1,2,3,4,5] [6,6,6,6]
[(1,6),(2,6),(3,6),(4,6)]
Prelude> zip [1,2,3,4] [6,6,6,6,6]
[(1,6),(2,6),(3,6),(4,6)]
Prelude> zip [1..5] ["a","b","c","d","e"]
[(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")]
Prelude> zip [a..f] ["a","b","c","d","e"]

<interactive>:25:6: error: Variable not in scope: a

<interactive>:25:9: error: Variable not in scope: f
Prelude> zip ['a'..'f'] ["a","b","c","d","e"]
[('a',"a"),('b',"b"),('c',"c"),('d',"d"),('e',"e")]
Prelude>
```

# 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: 2023-24

# 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).

Clear screen command CNTL+L
Exit GHCI command CNTL+D
Exit Prelude :n
https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/ghci-invocation.html

## 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

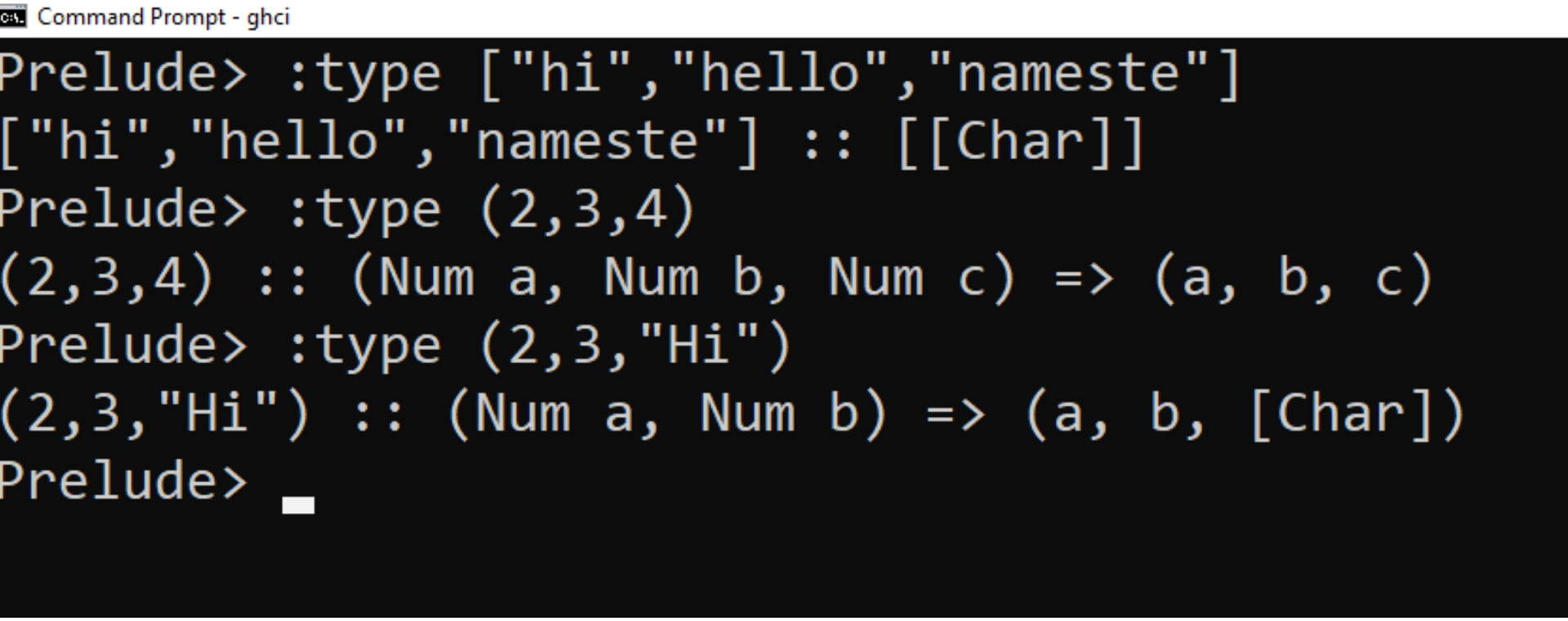
- **EQ**
  - Type class is an interface which provides the functionality to test the equality of an expression.
- **Num and Fractional**
  - This type class is used for numeric operations. Types such as Int, Integer, Float, and Double come under this Type class.
- **Integral**
  - sub-class of the Num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

# DATA TYPES

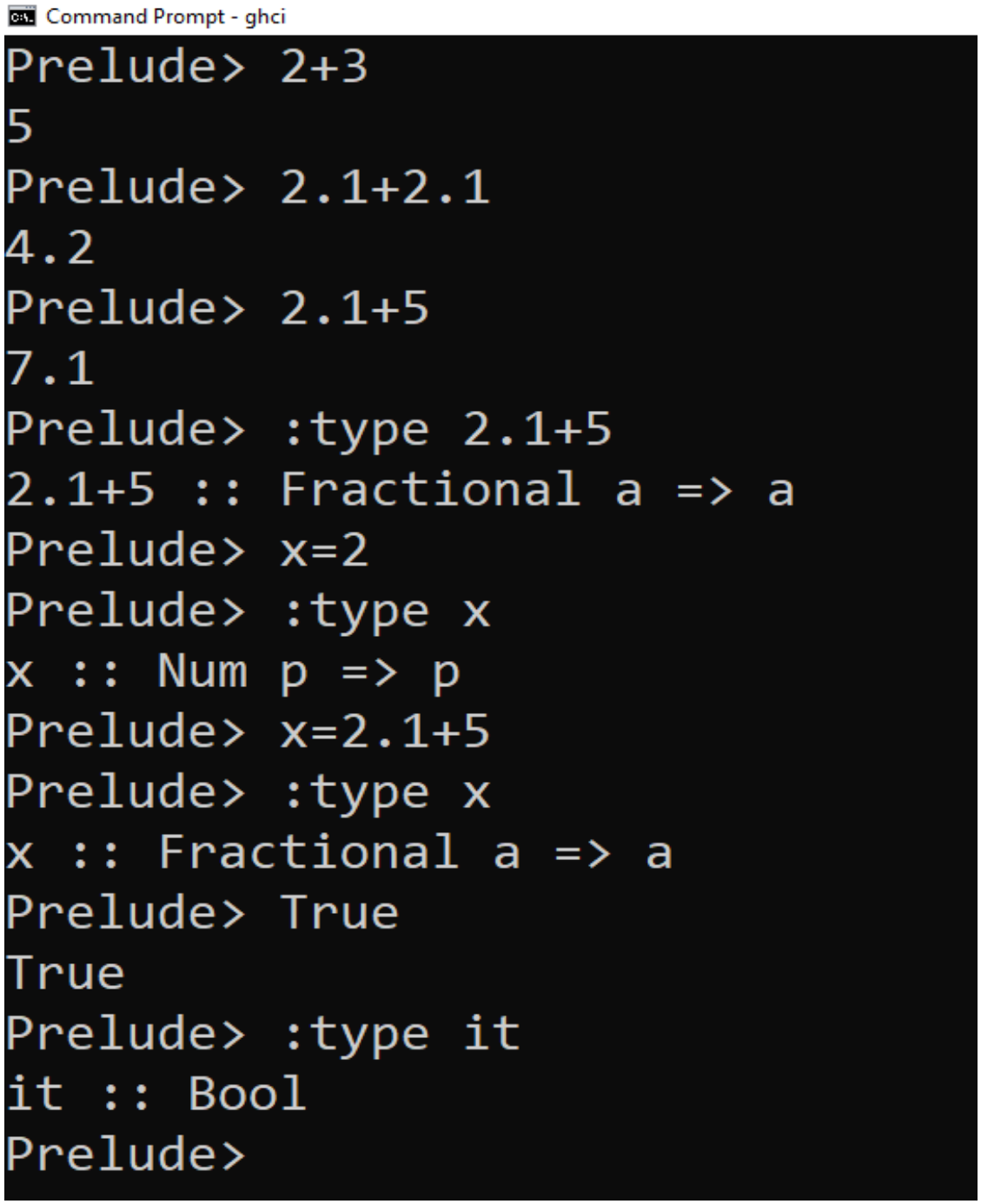
Command Prompt - ghci

```
C:\Users\Mrinmoyee>ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 2
2
Prelude> :type 2
2 :: Num p => p
Prelude> :type 2.1
2.1 :: Fractional p => p
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
Prelude> :type "Hello"
"Hello" :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: Num a => [a]
Prelude> :type [2.1,3.1,4.1]
[2.1,3.1,4.1] :: Fractional a => [a]
Prelude>
```

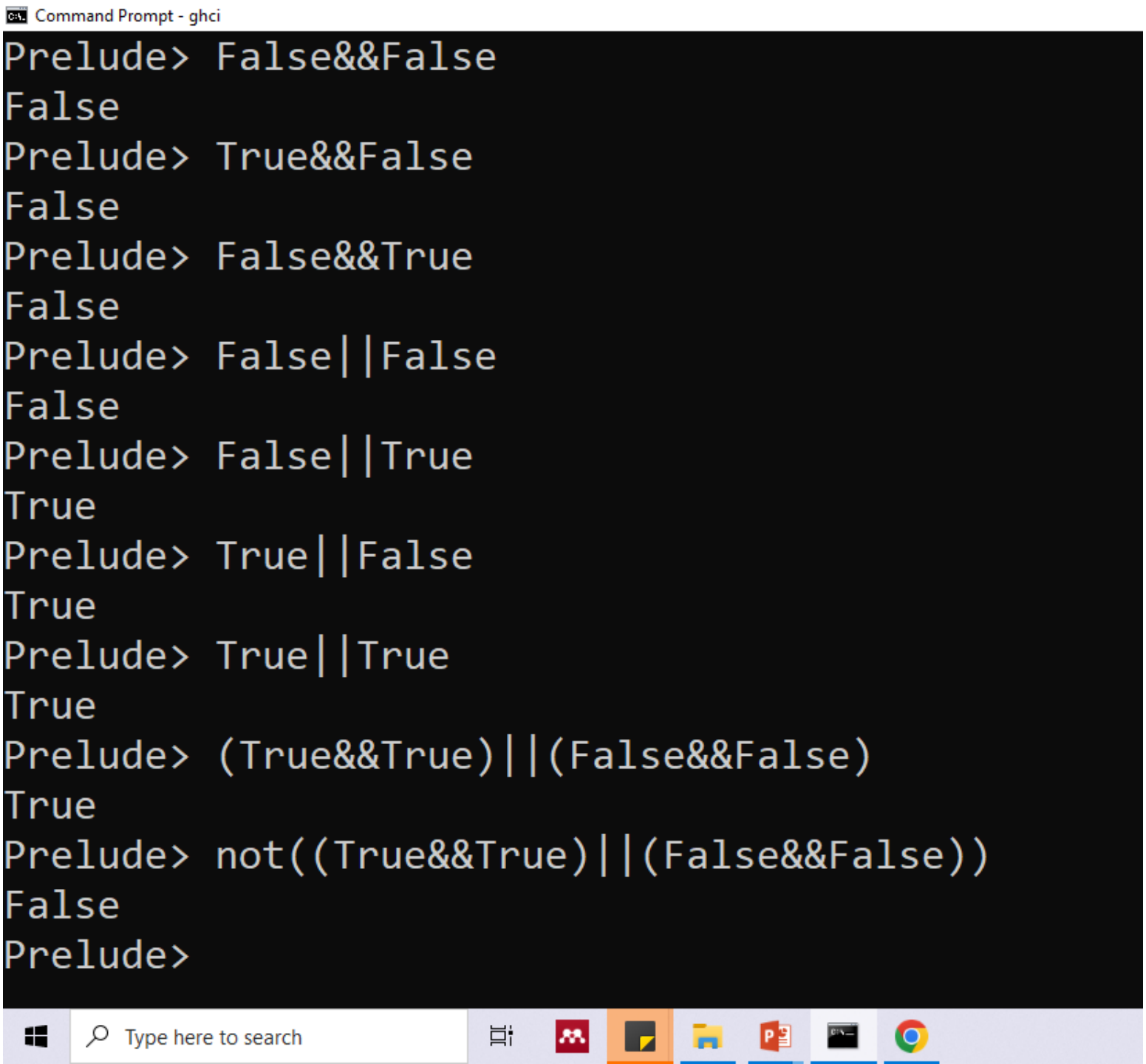
## DATA TYPES


Command Prompt - ghci
Prelude> :type ["hi","hello","nameste"]
["hi","hello","nameste"] :: [[Char]]
Prelude> :type (2,3,4)
(2,3,4) :: (Num a, Num b, Num c) => (a, b, c)
Prelude> :type (2,3,"Hi")
(2,3,"Hi") :: (Num a, Num b) => (a, b, [Char])
Prelude>

## ARITHMETIC OPERATORS



## LOGICAL OPERATORS



# COMPARATIVE OPERATORS

Command Prompt - ghci

```
Prelude> 2==2
True
Prelude> 2==3
False
Prelude> not(2==3)
True
Prelude> not(not(2==3))&&True
False
Prelude> not(not(2==3))&&True+2

<interactive>:36:17: error:
    * No instance for (Num Bool) arising from a use of `+'
    * In the second argument of `(&&)', namely `True + 2'
      In the expression: not (not (2 == 3)) && True + 2
      In an equation for `it': it = not (not (2 == 3)) && True + 2
Prelude>
```

Command Prompt - ghci

```
Prelude> 2/=2
False
Prelude> 2/=3
True
Prelude> 2<3
True
Prelude> 2>3
False
Prelude> not(2>3)
True
Prelude> 2^3
8
Prelude> 2**3
8.0
Prelude> 2^4
16
Prelude> 2**4
16.0
Prelude>
```

Command Prompt - ghci

```
Prelude> "Hi"=="Hi"
True
Prelude> "Hi"=="hi"
False
Prelude> not("Hi"=="hi")&&True
True
Prelude> not(not("Hi"=="hi")&&True)||False
False
Prelude> "Hello world"=="Hello world"
True
Prelude>
```

# LIST

- Much like shopping lists in the real world, lists in Haskell are very useful.
- It's the most used data structure
- In Haskell, lists are a **homogenous** data structure.
- It stores several elements of the same type.
- That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.

> ❖ A **list** in Haskell can be written using square brackets with commas separating the list's individual values.
>
> ❖ Has only homogeneous values

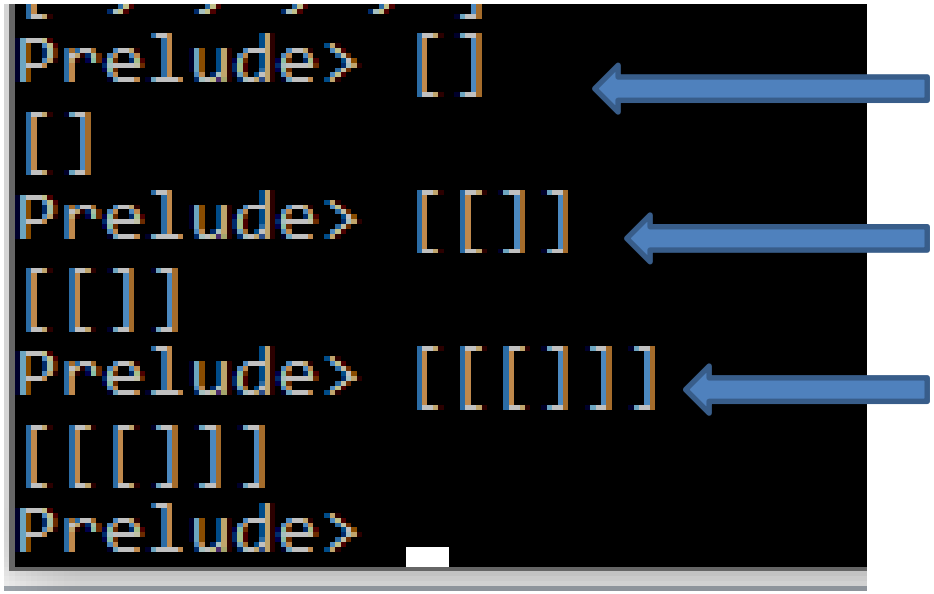
```
Prelude> let list1=[1,2,3,4]
Prelude> print list1
[1,2,3,4]
Prelude> let list2=[1,2,3,'a','b']
```

```
Prelude> [1,2,3,4]++[5,6,7,8]
[1,2,3,4,5,6,7,8]
Prelude> "hello"++ " "++"world"
"hello world"
Prelude> ['w','o']++['o','t']
"woot"
Prelude>
```

- Internally, Haskell has to walk through the whole list on the left side of ++
- That's not a problem when dealing with lists that aren't too big
- But putting something at the end of a list that's fifty million entries long is going to take a while
- However, putting something at the beginning of a list using the **:** operator (also called the cons operator) is instantaneous.

```
Prelude> 'A':" Small Cat"
"A Small Cat"
Prelude> 5:[1,2,3,4]
[5,1,2,3,4]
Prelude>
```

# LIST ....Continues....




1. An empty list
2. List of one empty list
3. List of two empty list

- If you want to get an element out of a list by index, use **!!**. The indices start at 0
- But if you try to get the sixth element from a list that only has four elements, you'll get an error so be careful!

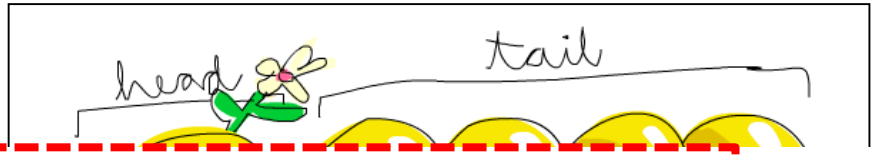
```
[[[]]]
Prelude> "Steve Buscemi" !! 6
'B'
```

# LIST ....Continues....

Lists can also contain lists. They can also contain lists that contain lists that contain lists …

```
[[[]]]
Prelude> "Steve Buscemi" !! 6
'B'
Prelude> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
Prelude> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b !! 2
[1,2,2,3,4]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude>
```

# LIST ....Continues....




**head** takes a list and returns its head. The head of a list is basically its first element.

```
ghci> head [5,4,3,2,1]
5
```

**tail** takes a list and returns its tail. In other words, it chops off a list's head.

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

**last** takes a list and returns its last element.

```
ghci> last [5,4,3,2,1]
1
```

**init** takes a list and returns everything except its last element.

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

# LIST ....Continues....

`length` takes a list and returns its length, obviously.

```
ghci> length [5,4,3,2,1]
5
```

`null` checks if a list is empty. If it is, it returns `True`, otherwise it returns `False`. Use this function instead of `xs == []` (if you have a list called `xs`)

```
ghci> null [1,2,3]
False
ghci> null []
True
```

`reverse` reverses a list.

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

`take` takes number and a list. It extracts that many elements from the beginning of the list. Watch.

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
```

# POST IAT-1 EXAM

# LIST ....Continues....

`drop` works in a similar way, only it drops the number of elements from the beginning of a list.

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

`maximum` takes a list of stuff that can be put in some kind of order and returns the biggest element.

`minimum` returns the smallest.

```
ghci> minimum [8,4,2,1,5,6]
1
ghci> maximum [1,9,2,3,4]
9
```

`sum` takes a list of numbers and returns their sum.

`product` takes a list of numbers and returns their product.

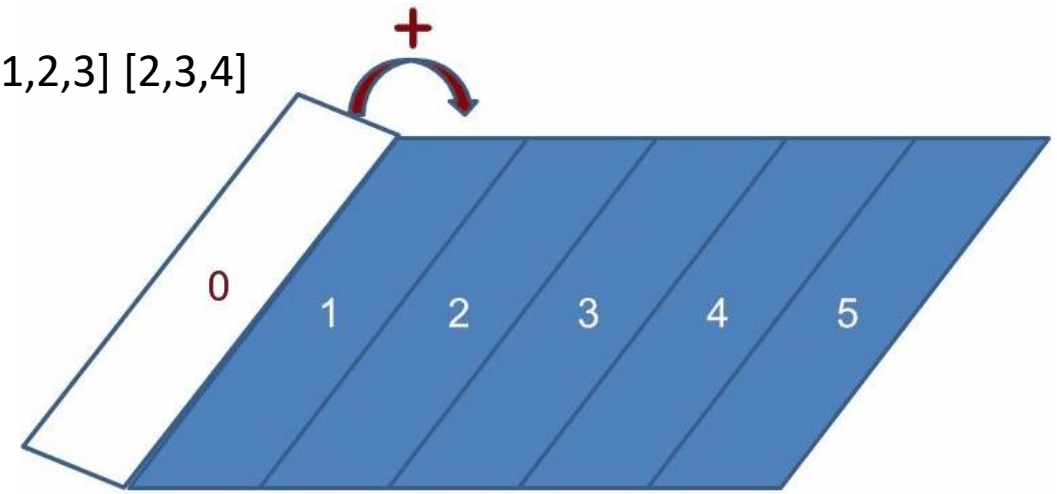
```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

# LIST ....Continues....

`elem` takes a thing and a list of things and tells us if that thing is an element of the list. It's usually called as an infix function because it's easier to read that way.

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

zipWith (*) [1,2,3] [2,3,4]



# LIST ....Continues....(Ranges)

Command Prompt - ghci

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
Prelude> ['A'..'Z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
Prelude> ['a'..'p']
"abcdefghijklmnop"
Prelude> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [3,6..30]
[3,6,9,12,15,18,21,24,27,30]
Prelude> [5,10..50]
[5,10,15,20,25,30,35,40,45,50]
Prelude> [20,19..1]
[20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1]
Prelude> [100,90..1]
[100,90,80,70,60,50,40,30,20,10]
Prelude> [0.1,0.3..1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
Prelude> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
Prelude> take 12 (cycle "LOL ")
"LOL LOL LOL "
Prelude> take 12 (cycle "LOL")
"LOLLOLLOLLOL"
Prelude> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
Prelude> replicate 3 10
[10,10,10]
Prelude>
```

# LIST ....Comprehension (Ranges)

- A basic comprehension for a set that contains the first ten even natural numbers is

$$S = \{2 \cdot x \mid x \in \mathbb{N},\ x \leq 10\}$$

- The part before the pipe is called the output function
- **x** is the variable
- **N** is the input set and **x <= 10** is the predicate.
- That means that the set contains the doubles of all natural numbers that satisfy the predicate.

```
Prelude> [x*2|x<-[1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [x*2|x<-[1..10],x*2>=12]
[12,14,16,18,20]
Prelude> [x*3|x<-[1..10],x*2>=12]
[18,21,24,27,30]
Prelude> [x*3|x<-[1..10],x*2>=9]
[15,18,21,24,27,30]
Prelude>
```

# LIST ....Comprehension (Ranges)

- How about if we wanted all numbers from 50 to 100 whose remainder when divided with the number 7 is 3?

- If we wanted all numbers from 10 to 20 that are not 13, 15 or 19

- If we have two lists, **[2,5,10]** and **[8,10,11]** and we want to get the products of all the possible combinations between numbers in those lists

```
Prelude> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
Prelude> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
Prelude> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

```
Prelude> [x*1|x<-[1..20]]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> [x*2|x<-[1..20]]
[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40]
Prelude> [x*2|x<-[1..20], x/=10]
[2,4,6,8,10,12,14,16,18,22,24,26,28,30,32,34,36,38,40]
Prelude> [x*2|x<-[1..20], x/=13]
[2,4,6,8,10,12,14,16,18,20,22,24,28,30,32,34,36,38,40]
Prelude> [x^2|x<-[1..20]]
[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225,256,289,324,361,400]
Prelude> [2^x|x<-[1..20]]
[2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536,131072,262144,524288,1048576]
```

# TUPLES…

- Are used when you know exactly how many values you want to combine
- Its type depends on how many components it has and the types of the components.
- They are denoted with parentheses and their components are separated by commas.

- Unlike a list, a tuple can contain a combination of several types.
  (represent someone's name and age in Haskell),**("Christopher", "Walken", 55)**

- Use tuples when you know in advance how many components some piece of data should have. There are singleton lists, there's no such thing as a singleton tuple.

- Like lists, tuples can be compared with each other if their components can be compared

- Only you can't compare two tuples of different sizes, whereas you can compare two lists of different sizes.

```
Prelude> fst (8,9)
8
Prelude> fst (8,"wow")
8
Prelude> snd (8,9)
9
Prelude> snd (8,"wow")
"wow"
```

# Tuple

- Like Lists, Tuples are used to store the values or group the values together.
- List in Haskell store same type of data inside it.
- Tuple can store different type of data inside it.

- Ex. of tuple

("String", Boolean, numbers)

- Functions used for tuples

fst (1,2,"a")

snd(1,"a",2)

**Ex.**

```
main = do
  print("Tuple in Haskell")
  let tuple1 =  (100, 200)
  let tuple2  =  (10.5, "hello")
  let tuple3  =  ("hello", "world")
  let tuple4  =  ("i am first", "i am second")
  let tuple5  =  ("i am string", "i am second
                    string")
  let tuple6 =  (260, 270)
  print(fst(tuple1))
  print(snd(tuple2))
```

# Decision Making (If..else)

- if<Condition> then <True-Value> else <False-Value>

```
main = do
  let var = 23
  if var == 0
     then putStrLn "Number is zero"
  else if var `rem` 2 == 0
     then putStrLn "Number is Even"
  else putStrLn "Number is Odd"
```

# Evaluation Order

- **Expressions:**
- Haskell has no statements, only expression.
- Pure functional programming languages don't have any statements—no assignments, no jumps
- All computations are performed by evaluating expressions

- Ex:

3+4 --> 7

3+4*5 is equivalent to 3 + (4*5) --> 23

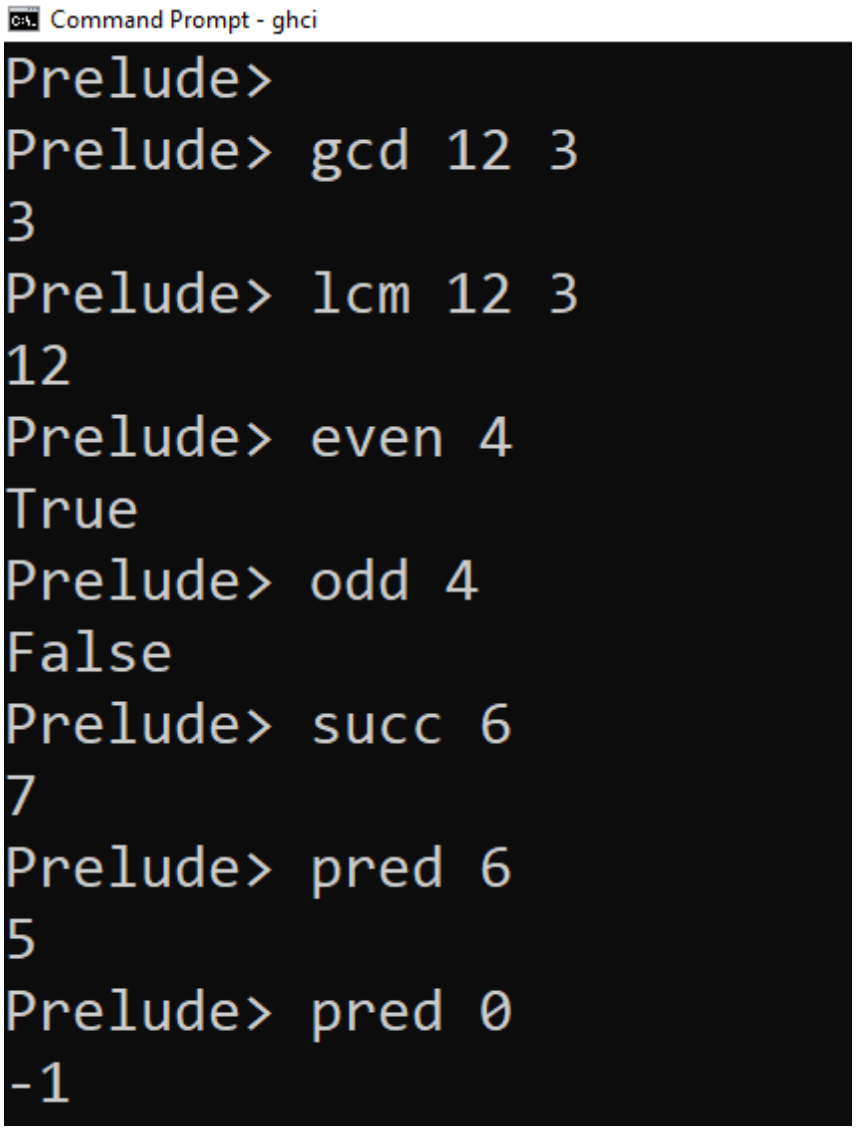
(3+4)*5 is equivalent to 7*5 --> 35

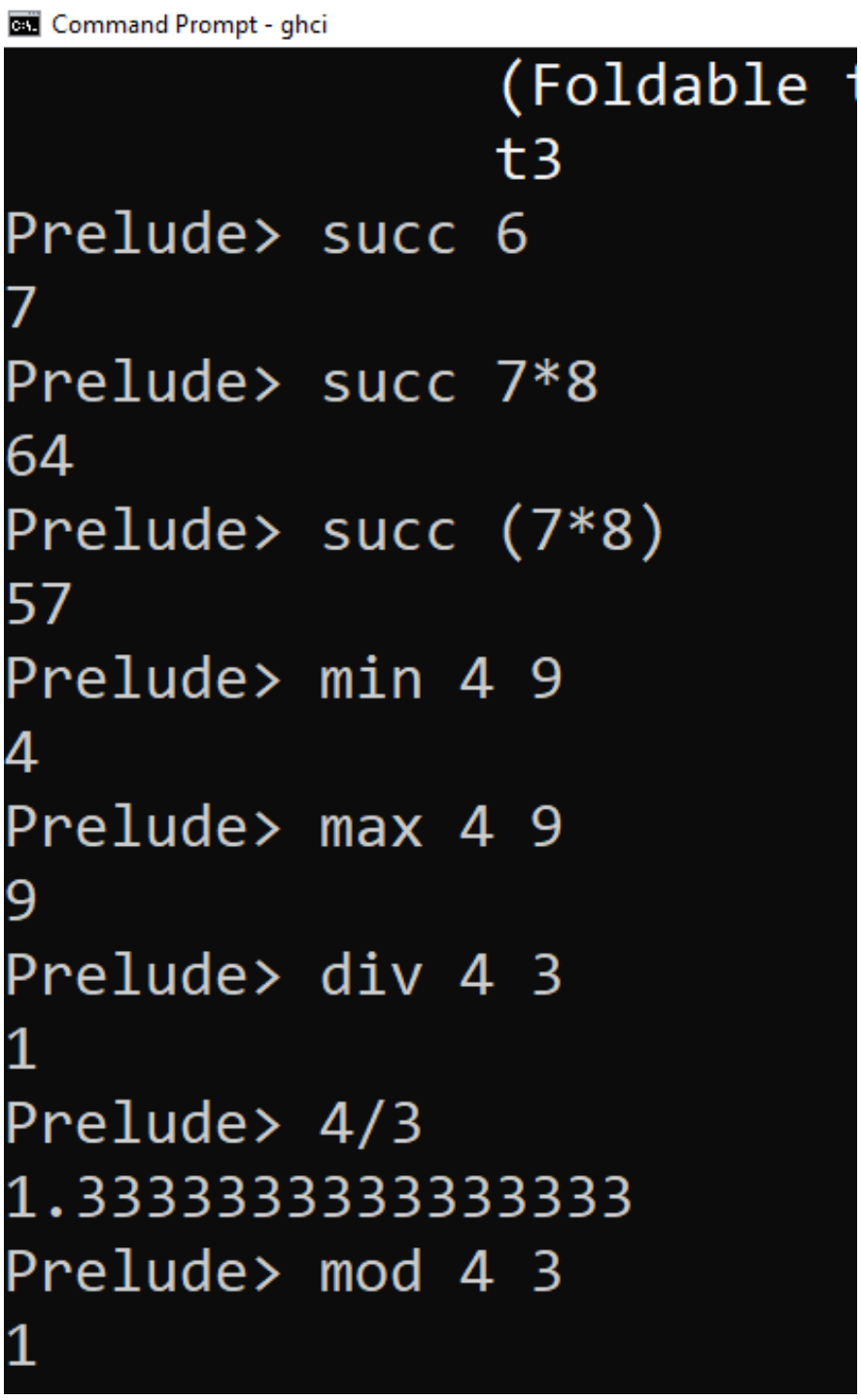
```
Prelude> sum ([3,7])
10
Prelude> sum ([3,7]++[5])
15
Prelude> 2
2
Prelude> 3+4
7
Prelude> 3+4*5
23
Prelude> (3+4)*5
35
```

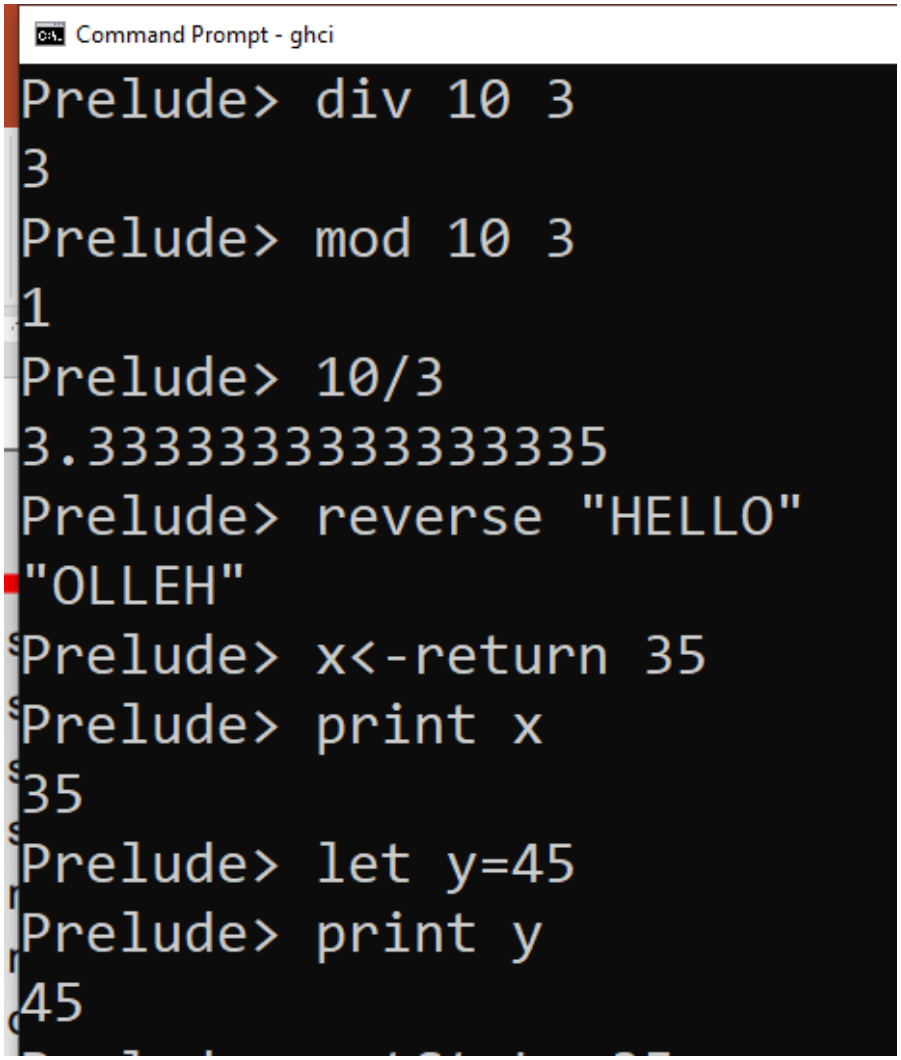
# Built-in Functions

- succ 6
- succ 7 * 8
- succ (7 * 8)
- min 4 9
- max 4 9
- div 4 3
- div 3 4
- mod 3 2
- 4/3

- reverse “hello”
- return True
- x <- return 35
- print x
- let y = 35
- print y
- putStrLn "hello“


Command Prompt - ghci
Prelude>
Prelude> gcd 12 3
3
Prelude> lcm 12 3
12
Prelude> even 4
True
Prelude> odd 4
False
Prelude> succ 6
7
Prelude> pred 6
5
Prelude> pred 0
-1


Command Prompt - ghci
(Foldable t
t3
Prelude> succ 6
7
Prelude> succ 7*8
64
Prelude> succ (7*8)
57
Prelude> min 4 9
4
Prelude> max 4 9
9
Prelude> div 4 3
1
Prelude> 4/3
1.3333333333333333
Prelude> mod 4 3
1


Command Prompt - ghci
Prelude> div 10 3
3
Prelude> mod 10 3
1
Prelude> 10/3
3.3333333333333335
Prelude> reverse "HELLO"
"OLLEH"
Prelude> x<-return 35
Prelude> print x
35
Prelude> let y=45
Prelude> print y
45

THANK YOU

# 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: 2023-24

# Functions

- Functions play a major role in Haskell, as it is a functional programming language.
- Function declaration consists of the function name and its argument list along with its output. Function definition is where you actually define a function. **Examples**:

**Ex1:**

```
add :: Integer -> Integer -> Integer      --function declaration
add x y =  x + y                          --function definition
main = do
  putStrLn "The addition of the two numbers is:"
  print(add 2 5)                          --calling a function
```

Lambda abstraction: `\x -> \y -> x+y` or `\x y -> x+y`

Curried function: `(add 2) 5`

**Ex2:**

```
add :: (Num a) => a -> a -> a
add x y = x+y
main = do
   print(add 2 4)
```

Output = 6

**or**

```
add :: (Floating a) => a -> a -> a
add x y = x+y
```

Output = 6.0

# Program 1: Hello world

helloworld.hs - Notepad

File Edit Format View Help

```
{-Name:Mrinmoyee
File:helloworld.hs
Description: Hello world program
-}

main::IO()
main=putStrLn "Hello World"
```

Pay Attention to multiline comment

Pay Attention to multiline comment

Main is the top level function. By lambda calculus the function is defined first

```
6
*Main> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello World
*Main>
```

Desktop OneDrive

# Program 2: Addition Operation

addn.hs - Notepad

File Edit Format View Help

```
{-Name: Mrinmoyee
  Date: 16/9/2020
-}

main::IO()
main=do
     let var1=6
     let var2=7
     putStrLn "The addition is"
     print(var1+var2)
```

Use space and not tab

addwithinput.hs - Notepad

File Edit Format View Help

```
main:: IO()
main = do
 putStrLn "Insert the first value: "
 one <- getLine
 putStrLn "Insert the second value: "
 two <- getLine
 putStrLn "The result is:"
 print ((read one) + (read two))
```

getLine produces a string, you'll need to convert it to the correct type. We can use read to get a number from it

getLine returns a String in the IO monad and Strings cannot be added.
readLn has polymorphic return type, and the compiler infers that the return type is Integer (in the IO monad) so you can add them.

```
addwithinput.hs:2:9: error:
    Unexpected do block in f
        do putStrLn "Insert
    You could write it with
    Or perhaps you meant to
  |
2 |  main = do putStrLn "Ins

  |         ^^^^^^^^^^^^^^^^
Failed, no modules loaded.
Prelude> :l addwithinput
[1 of 1] Compiling Main
hs, interpreted )
Ok, one module loaded.
*Main> main
Insert the first value:
12
Insert the second value:
13
The result is:
25
*Main>
```

# Functions

**Ex3:**

```
inc :: Integer ->  Integer        --function declaration
inc x =  x + 1                    --function definition
main = do
  putStrLn "The incremented value is:"
  print(inc 5)                     --calling a function
```

Lambda abstraction: \x-> x+1

# Examples of Functions

- **Ex4**:

```
double :: Int -> Int
double n = 2*n
main = do
   putStrLn "The double of the number is:"
   print(double 5)    --calling a function
```

Lambda abstraction:

```
\n-> 2*n
```

- **Ex5**:

```
DoubleUs :: Int -> Int -> Int
DoubleUs x y = 2*x + 2*y
main = print(DoubleUs  4 5)
```

```
\x y-> 2*x + 2*y
```

- **Ex6**:

```
mysum1  :: Int -> Int
mysum1  n = sum [1..n]
main = print(mysum1   10)
```

```
\n -> sum [1..n]
```

```
doubleme1::Int->Int
doubleme1 x=x*2

main=print(doubleme1 10)
```

```
    * No instance for (Show (Int -> Int)) arising from a use of `print'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
*Main> doubleme1 3
6
*Main> :l doubleme1
[1 of 1] Compiling Main             ( doubleme1.hs, interpreted )
Ok, one module loaded.
*Main> main
20
```

```
doubleus::Int->Int->Int
doubleus x y=x*2+y*2

main=print(doubleus 10 20)
```

Command Prompt - ghci

```
'
        (maybe you haven't applied a function to enough arguments?)
    * In a stmt of an interactive GHCi command: print it
*Main> doubleme1 3
6
*Main> :l doubleme1
[1 of 1] Compiling Main             ( doubleme1.hs, interpreted )
Ok, one module loaded.
*Main> main
20
*Main> :l doubleus
[1 of 1] Compiling Main             ( doubleus.hs, interpreted )
Ok, one module loaded.
*Main> doubleus 10 20
60
*Main> :l doubleus
[1 of 1] Compiling Main             ( doubleus.hs, interpreted )
Ok, one module loaded.
*Main> main
60
```

# Examples of Functions

- **Ex7**:

```
sayTwice :: String -> String
sayTwice s = s ++ s
-- or sayTwice s = s ++ " " ++ s
main = print (sayTwice "hello")
```

Lambda abstraction:

```
\s-> s ++ s
```

- **Ex8**:

```
nand :: Bool -> Bool -> Bool
nand False False = True
nand False True  = True
nand True  False = True
nand True  True  = False
main = do
   print(nand True True)          --calling a function
```

# Pattern Matching in Recursive Function

- In recursion the function is applied inside its own definition.
- Use of Pattern Matching in récursive function to match one or more expressions
- **Ex2**:

```
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )
main = do
    putStrLn "The factorial of 5 is:"
    print (fact 4)
```

Output = 24

- **Ex3**:

```
power :: Float -> Int -> Float
power x 0 = 1.0
power x n =  x * (power x (n-1))
main = print (power 2 3)
```

Output = 8.0

# Higher Order Function Ex6

```
divideByTen :: (Floating a) => a -> a
divideByTen x = (/10) x

main = do
    print(divideByTen 400)
```

Output:

40.0

# Higher Order Function Ex7

```
compareWithHundred :: (Num a, Ord a) => a -> Ordering
compareWithHundred x = compare 100 x

main = do
   print(compareWithHundred 2)
   print(compareWithHundred 102)
```

Output:

GT

LT

The function returns "LT" if the first argument is less than the second one, "EQ" if the arguments are equal, and "GT" if the first argument is greater than the second one.

# Higher Order Function Ex8

```
apply :: (a -> a) -> a -> a
apply f x = f x
main = do
    print(apply (+5) 7)
```

Output = 12

---

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)

main = do
    print(applyTwice (+5) 7)
```

Output: 17

# 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: 2023-24

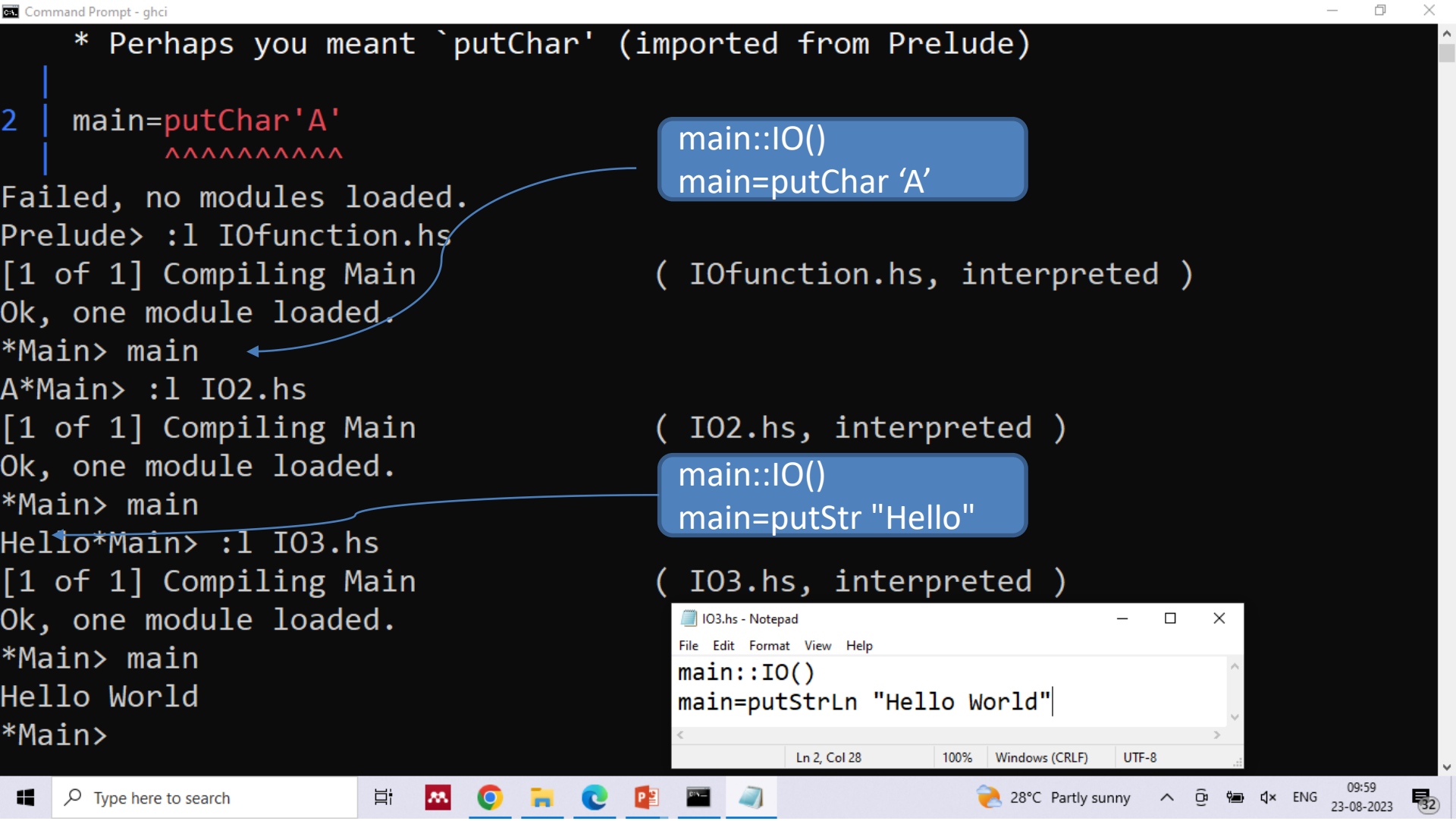
# IO Actions

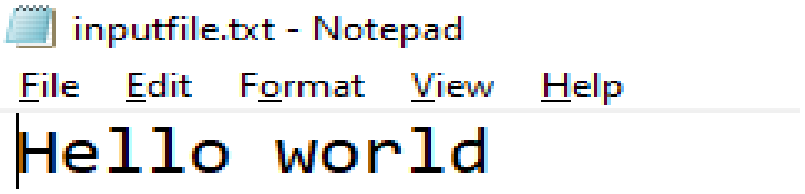
- **Input Actions:**
- These read from standard input (cin):
- **getChar** :: IO Char -- gets 1 char
- **getLine** :: IO String -- strips \n
- **getContents** :: IO String
- -- returns ALL the rest of cin!
- This reads the entire contents of a file!
- **readFile** :: FilePath -> IO String

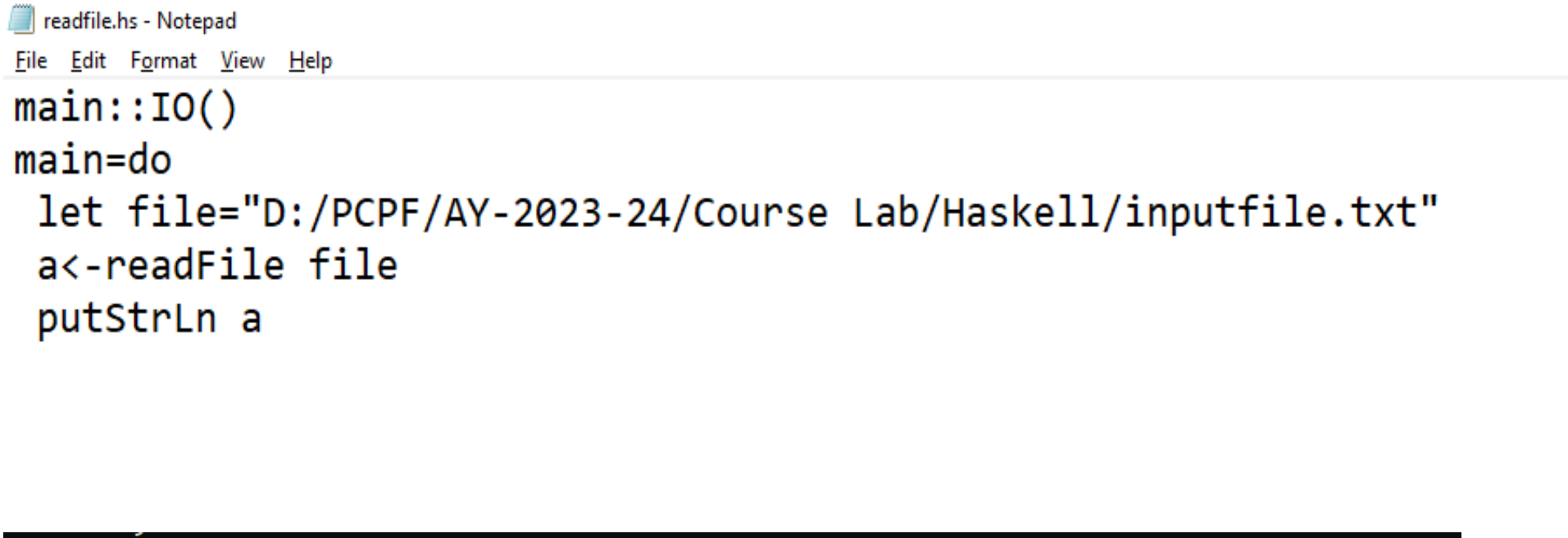
```
main = do
  let file = "D:\abc.txt"
  a <- readFile file
  putStrLn a
```

- **Output Actions:**
- These write to standard output (cout)
- **putChar** :: Char -> IO ()
- **putStr** :: String -> IO ()
- **putStrLn** :: String -> IO () -- adds \n
- These write to (or create) output files:
- type FilePath = String
- **writeFile** :: FilePath -> String -> IO()
- **appendFile**:: FilePath -> String -> IO()

```
The first IO program is
main :: IO ()
main = putStrLn “Hello World”
```


Command Prompt - ghci
* Perhaps you meant `putChar' (imported from Prelude)
2 | main=putChar'A'
^^^^^^^^^^
Failed, no modules loaded.
Prelude> :l IOfunction.hs
[1 of 1] Compiling Main ( IOfunction.hs, interpreted )
Ok, one module loaded.
*Main> main
A*Main> :l IO2.hs
[1 of 1] Compiling Main ( IO2.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello*Main> :l IO3.hs
[1 of 1] Compiling Main ( IO3.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello World
*Main>
main::IO()
main=putChar 'A'
main::IO()
main=putStr "Hello"
IO3.hs - Notepad
File Edit Format View Help
main::IO()
main=putStrLn "Hello World"
Ln 2, Col 28
100%
Windows (CRLF)
UTF-8
Type here to search
28°C Partly sunny
ENG
09:59
23-08-2023




```
Prelude> :l readfile.hs
[1 of 1] Compiling Main             ( readfile.hs, interpreted )
Ok, one module loaded.
*Main> main
Hello world
*Main> _
```

# IO Program Ex-3

```
getInt p = do
{
print p;
line <- getLine;
return (read line :: Int)
}
```

```
getGreater p = do   {
 print p;
a <- getInt "First Num" ;
b <- getInt "Second Num" ;
print "greater num is ";
if a>=b then do
{   return (a)   }
else do
{    return (b)  }
}
main = do {
y <- getGreater "Program to find
greater num";
print y }
```