# Functions as First-class Citizens and Higher-order Functions

# Introduction to Functional Programming in Haskell

## 1 Pure Functions

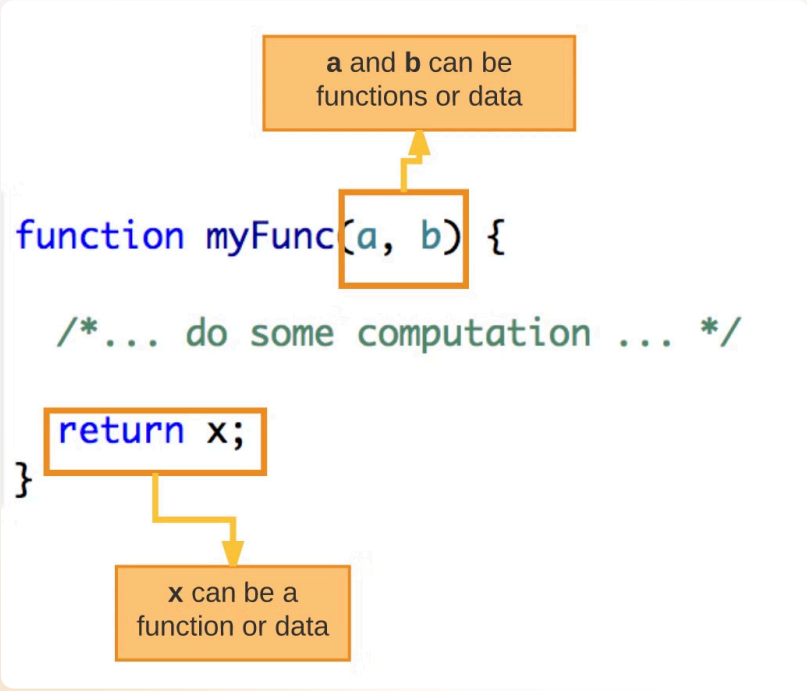
Functions in Haskell are pure, meaning they always produce the same output for a given input.

## 2 Immutable Data

Data is immutable in Haskell, ensuring that values cannot be modified after they are created.

## 3 Recursion

Haskell primarily uses recursion to iterate over data structures.



# Functions as Values: Definition and Use

## Defining Functions

In Haskell, functions are defined using the arrow operator (->) separating the input type from the output type.

## Using Functions

Functions are invoked by placing the arguments in parentheses after the function name.

# Assigning Functions to Variables

1

## Assignment

Functions can be assigned to variables using the "=" operator.

```
-- Define a function that adds two numbers

add :: Int -> Int -> Int

add x y = x + y
```

2

## Invocation

The variable can then be used to invoke the function.

```
-- Assign the function to a variable

addFunction = add

-- Use the variable to invoke the function

result = addFunction 5 3 -- result is 8
```

# Passing Functions as Arguments

- Higher-order functions can accept other functions as input.
- This allows for flexible and reusable code, as functions can be passed as arguments to various higher-order functions.
- Functions that are passed as arguments are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned as values.
- A function that takes one or more functions as arguments or returns a function as its result is known as a higher-order function. This is a fundamental concept in functional programming that promotes code reuse and abstraction.

**Benefits**:

- Passing functions as arguments enables:
- **Custom behavior**: Users can define their functions to customize the behavior of higher-order functions.
- **Reduction of code duplication**: Common patterns can be encapsulated into higher-order functions, reducing redundancy.

-- Define a higher-order function that takes a function as an argument

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

-- Define a simple function to increment a number

**increment :: Int -> Int increment x = x + 1**

-- Use applyTwice with the increment function

**result = applyTwice increment 5 -- result is 7**

# Returning Functions from Functions

### Nested Functions

Functions can be defined within other functions, creating nested structures.

### Returning Functions

The outer function can return the inner function as its result.

# Nested functions

Nested functions are defined within the scope of another function. They can access variables from the outer function's scope.

The **where** keyword allows you to define local functions within a function's body. These functions are only accessible within the parent function's scope.

For example, consider a function **calculateAverage** that takes a list of integers and returns the average. The **sumList** function is nested within the **calculateAverage** function, and is only accessible within the scope of the outer function.

```
calculateAverage :: [Int] -> Float
calculateAverage numbers = sumList numbers / fromIntegral (length numbers)
  where
    sumList [] = 0
    sumList (x:xs) = x + sumList xs
```

# Returning Functions from Functions

Nested functions in Haskell allow you to define functions within other functions. This structure helps organize code and create modular functions that are reusable within specific scopes. The inner function, which is defined within the outer function, has access to the outer function's variables and parameters.

For example, consider this function that creates a function that adds a constant value to its input:

```
addConstant :: Int -> (Int -> Int)
addConstant n = \x -> x + n
```

This function, `addConstant`, takes an integer `n` as input and returns a new function. This returned function takes an integer `x` as input and returns `x + n`.

The outer function can return the inner function as its result. This enables you to dynamically create functions based on specific conditions or inputs within the outer function. The returned inner function then captures the state or parameters of the outer function at the time it was returned.

Here's how to use `addConstant`:

```
addFive = addConstant 5
result = addFive 3 -- result is 8
```

In this example, `addFive` is a function that adds 5 to its input. It was created by calling `addConstant` with the argument 5. We then call `addFive` with the argument 3, which returns 8.

# Higher-order Functions: Map, Reduce, and Filter

1

## Map

Applies a function to each element of a list.

2

## Reduce

Combines elements of a list into a single value.

3

## Filter

Creates a new list containing only elements that meet a specific condition.

# Applying Map, Reduce, and Filter in Haskell

### Map Example

map (+1) [1, 2, 3] will output [2, 3, 4]

### Reduce Example

foldl (+) 0 [1, 2, 3] will output 6

### Filter Example

filter even [1, 2, 3, 4, 5] will output [2, 4]

# Creating and Using Custom Higher-order Functions

## 1 Function Definition

Define the higher-order function, accepting a function as an argument.

-- Define a higher-order function that applies a function to each element of a list

**applyToList :: (a -> b) -> [a] -> [b]**

**applyToList f xs = [f x | x <- xs]**

## 2 Function Call

Pass a specific function as an argument to the higher-order function.

-- Define a function to double a number

**double :: Int -> Int**

**double x = x * 2**

-- Use applyToList with the double function

**result = applyToList double [1, 2, 3, 4, 5]**

-- result is [2, 4, 6, 8, 10]

# Implementing Custom Higher-order Functions

By leveraging higher-order functions, you can abstract common patterns and create reusable components, leading to more concise and expressive code. Let's explore some examples of implementing custom higher-order functions in Haskell.

## Situations to Use Custom Higher-order Functions

1. **Encapsulating Repetitive Logic:**
   - When you find yourself writing similar logic multiple times, HOFs allow you to encapsulate this behavior in one place.
   - **Example:** Filtering elements in a list based on different conditions can be abstracted into a single HOF.
2. **Customizing Behavior:**
   - When you need to apply a specific operation to a collection of data, HOFs can allow for customization without altering the underlying logic.
   - **Example:** Passing different functions to a mapping function to apply various transformations.
3. **Composing Functions:**
   - HOFs enable you to compose multiple functions together, creating more complex behavior while maintaining readability.
   - **Example:** Combining functions for data processing pipelines, such as filtering, mapping, and reducing data.