Lecture 1, Sep 4:

- **Paradigms**: Focus on *functional* programming
- Course covers a comparison between **imperative** and **functional** paradigms with examples.

Key Concepts:

1. **Imperative (Procedural) Programming**:
    - **Central concept**: Modifying state.
        - State includes variable values, memory content, and program location.
    - **Data**: Managed in memory.
    - **Control**: Explicit modification of the state (e.g., assignments, memory allocation).
    - **Example**: Binary trees with an integer value.
        - **Code Pattern**: Traversing the structure to modify data.

```
class Node:
  def __init__(self, data):
    self.left = None
    self.right = None
    self.data = data
def incr(t):
  if t is None:
    return
  t.data += 1
  incr(t.left)
  incr(t.right)
```

Functional Programming:

- **Central concept**: Values of expressions.
    - Variables don't change; data structures are immutable.
- **Data**: Abstract view, focusing on constructors and accessors.
- **Control**: Function calls and recursion are key.
- **Example**: Abstract tree with functional increment.

```
def incr(t):
  if is_empty(t):
    t
  else:
    node(data(t)+1, left(t), right(t))
```

Comparison: Imperative vs Functional

- **Imperative**: Actions like *modify*, *set*, *update*, etc.
- **Functional**: Concepts like *evaluate*, *construct*, *access*.
- **Why Functional Programming?**
    - Reduces common errors related to memory and concurrency.
    - Closer to mathematics, making reasoning easier.

Lecture 2, Sep 9 :

Key Concepts:

1. **Inductive Definitions**:

- A method to define collections (like trees or lists) using base elements and building rules.
- **Base elements**: Objects that don't contain other elements (e.g., leaves in trees).
- **Building rules**: Define how to construct complex objects from simpler ones (e.g., nodes in trees).

2. **Programming Paradigms** (Review):
   - **Procedural/Imperative**:
     - Program executes a sequence of steps, modifying state (variables and memory).
     - Examples: Python, Java, Go, C++, etc.
   - **Functional**:
     - Programs evaluate expressions, where variables represent values instead of memory locations.
     - Examples: Pure functional (Haskell) and languages with functional components (CAML, F#, JavaScript, Python).
3. **Defining Trees**:
   - Trees consist of nodes with children, following a tree-like structure (no cycles).
   - An inductive definition of trees specifies:
     - **Base case**: Defines basic elements (e.g., leaves).
     - **Recursive step**: Defines how to build larger trees from smaller ones.
4. **Examples of Inductive Definitions**:
   - **Integer Lists**:
     - Base case: [] is a list.
     - Building rule: If n is an integer and l is a list, then :: n l is a list.
     - Example: :: 0 (:: 3 (:: 17 [])) is a list.
   - **Binary Trees**:
     - Base case: Leaf n where n is an integer.
     - Building rule: Node n t1 t2 where n is an integer, and t1 and t2 are binary trees.
     - Example: A binary tree can be represented as Node 3 (Node 4 (Leaf 5) (Leaf 6)) (Leaf 7).
5. **Computing with Inductive Objects**:
   - You can manipulate inductive structures (like binary trees) through:
     - **Pattern matching**: Checking the structure (e.g., whether a tree node is a leaf or a node).
     - **Recursion**: Performing operations by applying functions recursively to smaller parts.
   - **Example Function**: Incrementing all elements of a binary tree:

```
def incr(t):
  if t is Leaf n:
    return Leaf (n+1)
  else:
    return Node (n+1) (incr t1) (incr t2)
```

Notation for Inductive Definitions:

- **Premise and conclusion**: Inductive rules can be written as:

assumed objects ---------- new object

Lecture 3, Sep 11:

Binary Trees in Haskell:

- Inductive definitions can be directly represented in Haskell using data types.
- **Binary Tree (BT)**:
  - A binary tree consists of either:
    - A Leaf with an integer value (Leaf n).
    - A Node with an integer value and two child binary trees (Node n t1 t2).

haskell

```
data BT = Leaf Int | Node Int BT BT
```

Increment Function (incr):

- The function incr adds 1 to each integer in the binary tree.
- In Haskell:

haskell

```
incr (Leaf n) = Leaf (n + 1)
incr (Node n t1 t2) = Node (n + 1) (incr t1) (incr t2)
```

- E**valuation**: Haskell evaluates this function by pattern matching:
  - For a Leaf, it adds 1 to the value.
  - For a Node, it recursively applies the function to both subtrees.

 Pattern Matching in Haskell:

- **Pattern matching** is a technique where the structure of the input is checked, and appropriate actions are taken based on its form.
- Example:

```
incr (Node 17 (Leaf 0) (Leaf 1))
-- Result: Node 18 (Leaf 1) (Leaf 2)
```

1. **Example: Cars and Fleets**:
   - Haskell can represent more complex data types like cars and fleets:

haskell

```
data Colour = Blue | Red | Yellow
data Price = Price Int
data Car = Car Colour Price String
data Fleet = Empty | AddCar Car Fleet
```

   - Example of a fleet:

haskell

```
car0 = Car Red (Price 60000) "Lincoln Juggernaut"
car1 = Car Yellow (Price 120000) "BMW Highsnoot"
car2 = Car Blue (Price 10000) "Fiat Roadkill"
fleet = AddCar car0 (AddCar car1 (AddCar car2 Empty))
```

Lecture 4, Sept 16:

1. Lists in Haskell

- Lists are defined as a *type constructor*. For any element type a, [a] is the type of lists with elements of type a.
- Example:

haskell

x ∈ a

l ∈ [a]

  - [] is a list of type [a].
  - (:) x l means x is prepended to list l.
- Example of building lists:

haskell

(:) 5 (3 : 17 : []) == [5, 3, 17]

Haskell shows lists like this for simplicity.

2. Infix vs. Prefix Function Notation

- Haskell allows switching between infix and prefix notation for functions:
  - f x y = x ++ y (prefix)
  - eg = (++) ("a" f "b") "c" (mixed)
  - (:) can be written either way.

3. Pattern Matching with Lists

- Pattern matching is used to check the structure of lists.

haskell

```
f [] = True
f (2 : l) = (length l == 2)
f [1, 2, x] = (x == 17)
f _ = False
```

4. Universal Polymorphism

- Functions like tail work for any list type. Using a type variable a allows them to apply to any element type:

haskell

```
tail :: [a] -> [a]
```

5. Basic Pattern Matching and Recursion for Lists

- Lists can be recursively processed using pattern matching:

haskell

```
listize :: [a] -> [[a]]
listize [] = []
listize (x : l) = [x] : listize l
```

6. Append Function

- The append function adds an element to the end of a list:

haskell

```
append :: [a] -> a -> [a]
append [] y = [y]
append (x : l) y = x : append l y
```

7. Sections in Haskell

- Sections allow partial application by giving an argument "in advance":

haskell

```
(+++) :: Int -> Int -> Int
x +++ y = x + 2 * y
```
Examples of usage:
haskell
```
(1 +++) :: Int -> Int  -- Partially applies 1 to the function
(+++ 1) :: Int -> Int  -- Partially applies 1 to the second argument
```
8. Unnamed (Lambda) Functions

- Lambda expressions are unnamed functions:

haskell
```
\x -> 2*x + 17
```
This is the same as defining a named function f x = 2*x + 17.

9. Convenience Operator: $

- The $ operator allows for avoiding parentheses:

haskell
```
f $ g x  ==  f (g x)
```
10. Guards in Pattern Matching

- Guards enhance pattern matching by adding conditions:

haskell
```
absVal n | n < 0 = -n
absVal n = n
```
Another example:
haskell
```
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x (y : l) | x <= y = x : y : l
insert x (y : l) = y : (insert x l)
```

Lecture 5, Sept 18:

1. map Function

- **Purpose**: The map function applies a given function to every element of a list, creating a new list with the results.
- **Type Signature**:

haskell
map :: (a -> b) -> [a] -> [b]

- **How It Works**:
  - If the list is empty ([]), map returns an empty list.
  - Otherwise, it applies the function f to the first element (x) and recursively maps over the rest of the list (l).
- **Example**:

haskell
map (+1) [1, 2, 3]

Step-by-step:

haskell
= (+1) 1 : map (+1) [2, 3]
= 2 : (+1) 2 : map (+1) [3]
= 2 : 3 : (+1) 3 : []
= [2, 3, 4]

The function (+1) is applied to each element, and the result is [2, 3, 4].

2. filter Function

- **Purpose**: The filter function selects elements from a list that satisfy a given condition (predicate).
- **Type Signature**:

haskell
filter :: (a -> Bool) -> [a] -> [a]

- **How It Works**:
  - If the list is empty ([]), it returns an empty list.
  - Otherwise, it checks the first element (x). If it satisfies the predicate p (i.e., p x = True), it includes x in the result; otherwise, it skips x.
- **Example**:

haskell
filter (<= 3) [4, 3, 0]

Step-by-step:

haskell
= filter (<= 3) [3, 0]
= 3 : filter (<= 3) [0]
= 3 : 0 : filter (<= 3) []
= [3, 0]

Only the elements less than or equal to 3 are included, so the result is [3, 0].

3. all Function

- **Purpose**: The all function checks if all elements in a list satisfy a given condition (predicate). It returns True only if every element in the list makes the predicate True.
- **Type Signature**:

haskell
all :: (a -> Bool) -> [a] -> Bool

- **How It Works**:
  - If the list is empty ([]), all returns True (since all elements in an empty list trivially satisfy any condition).
  - Otherwise, it checks if the first element (x) satisfies the predicate p. If p x = True, it continues checking the rest of the list (l). If any element fails, it returns False.
- **Example**:

haskell
all (<= 3) [0, 3, 2]
Step-by-step:
haskell
= (<= 3) 0 && all (<= 3) [3, 2]
= True && (<= 3) 3 && all (<= 3) [2]
= True && True && (<= 3) 2 && all (<= 3) []
= True && True && True && True
= True
All elements in the list [0, 3, 2] are less than or equal to 3, so the result is True.

Key Takeaways:

- **map** transforms a list by applying a function to each element.
- **filter** extracts elements from a list that satisfy a specific condition.
- **all** checks if all elements in a list meet a given condition.

LECTURE 6 & 7, Sept 25 & Sept 30:

Higher-Order Functions and Currying in Haskell

1. Introduction to Higher-Order Functions

What Are Higher-Order Functions?

- **Definition**: A higher-order function is a function that can take other functions as arguments or return functions as results.
- **Purpose**: They allow for more abstract and concise code by encapsulating common programming patterns.
- **Examples in Haskell**: map, filter, foldr, foldl, zipWith, etc.

Benefits of Using Higher-Order Functions

- **Code Reusability**: Write generic functions that can work with any appropriate function passed as an argument.
- **Abstraction**: Hide complex recursion patterns, making code easier to read and maintain.
- **Functional Composition**: Build complex functions by combining simpler ones.

---

2. Recursion vs. Recursionless Programming

Traditional Recursion

- **Approach**: Functions call themselves with modified parameters until a base case is reached.
- **Example**: Incrementing each element in a list recursively.

haskell
```
add1 :: [Int] -> [Int]
add1 [] = []
add1 (x : xs) = (x + 1) : add1 xs
```

- **Explanation**:
  - **Base Case**: If the list is empty ([]), return an empty list.
  - **Recursive Case**: Add 1 to the head (x) and recursively call add1 on the tail (xs).

Recursionless Programming with Higher-Order Functions

- **Approach**: Use functions like map to abstract away recursion.
- **Example**:

haskell
```
add1' :: [Int] -> [Int]
add1' xs = map (+1) xs
```

- **Explanation**:
  - map applies the function (+1) to each element of the list xs.
  - No explicit recursion is needed; map handles the traversal internally.

Advantages of Recursionless Approach

- **Simplicity**: Less code and clearer intent.
- **Avoid Errors**: Reduces the chance of mistakes in recursion (e.g., missing base cases).
- **Performance**: Built-in functions are often optimized.

---

3. Key Higher-Order Functions in Haskell

3.1. map

- **Type Signature**:

haskell
```
map :: (a -> b) -> [a] -> [b]
```

- **Description**: Applies a function to every element of a list, producing a new list.
- **Example**:

haskell
doubleList :: [Int] -> [Int]
doubleList xs = map (*2) xs

- **Implementation** (for understanding):

haskell
map _ [] = []
map f (x:xs) = f x : map f xs

- **Explanation**:
  - **Base Case**: An empty list returns an empty list.
  - **Recursive Case**: Apply f to the head x, and recursively map over the tail xs.

3.2. filter

- **Type Signature**:

haskell
filter :: (a -> Bool) -> [a] -> [a]

- **Description**: Returns a list of elements that satisfy a predicate.
- **Example**:

haskell
evens :: [Int] -> [Int]
evens xs = filter even xs

- **Implementation**:

haskell
filter _ [] = []
filter p (x:xs)
  | p x   = x : filter p xs
  | otherwise = filter p xs

- **Explanation**:
  - **Predicate p**: A function that returns True or False for an element.
  - **Decision**: Include x in the result if p x is True.

3.3. foldr (Right Fold)

- **Type Signature**:

haskell
foldr :: (a -> b -> b) -> b -> [a] -> b

- **Description**: Reduces a list to a single value by recursively applying a function from the right.
- **Example**:

haskell
sumList :: [Int] -> Int
sumList xs = foldr (+) 0 xs

- **Implementation**:

haskell
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

- **Explanation**:
  - **Base Case**: When the list is empty, return the accumulator z.

    - **Recursive Case**: Apply the function f to the head x and the result of folding over the tail xs.

3.4. foldl (Left Fold)

- **Type Signature**:

haskell
```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

- **Description**: Similar to foldr, but processes the list from the left.
- **Example**:

haskell
```
sumList :: [Int] -> Int
sumList xs = foldl (+) 0 xs
```

- **Implementation**:

haskell
```
foldl _ z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- **Explanation**:
    - The accumulator z is updated by applying f to z and x, then recursively folding over xs.

3.5. zipWith

- **Type Signature**:

haskell
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

- **Description**: Takes two lists and applies a function to corresponding elements, producing a list of results.
- **Example**:

haskell
```
addLists :: [Int] -> [Int] -> [Int]
addLists xs ys = zipWith (+) xs ys
```

- **Implementation**:

haskell
```
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

- **Explanation**:
    - **Base Case**: If either list is empty, return an empty list.
    - **Recursive Case**: Apply f to the heads x and y, then recursively zip over the tails.

3.6. all and any

- **Type Signatures**:

haskell
```
all :: (a -> Bool) -> [a] -> Bool
any :: (a -> Bool) -> [a] -> Bool
```

- **Description**:
    - all returns True if all elements satisfy the predicate.
    - any returns True if any element satisfies the predicate.
- **Example**:

haskell

```
allPositive :: [Int] -> Bool
allPositive xs = all (>0) xs

anyNegative :: [Int] -> Bool
anyNegative xs = any (<0) xs
```

---

4. Currying in Haskell

What Is Currying?

- **Definition**: The process of transforming a function that takes multiple arguments into a series of functions that each take a single argument.
- **Origin**: Named after mathematician Haskell Curry.
- **Syntax in Haskell**: All functions are curried by default.

Function Application

- **Partial Application**: Supplying fewer arguments than a function requires, resulting in a new function that takes the remaining arguments.
- **Example**:

haskell
```
add :: Int -> Int -> Int
add x y = x + y

increment :: Int -> Int
increment = add 1
```

- **Explanation**:
  - add 1 returns a new function that adds 1 to its argument.
  - increment 5 results in 6.

Currying with Higher-Order Functions

- **Using map with Curried Functions**:

haskell
```
addThree :: Int -> Int
addThree = add 3

result :: [Int]
result = map addThree [1, 2, 3]  -- [4, 5, 6]
```

- **Currying and Sections**:
  - **Sections**: Partially apply infix operators by fixing one operand.
  - **Example**:

haskell
```
multiplyByTwo :: Int -> Int
multiplyByTwo = (* 2)

result = map (* 2) [1, 2, 3]  -- [2, 4, 6]
```

---

5. Combining Higher-Order Functions

Example: Dot Product

- **Problem**: Compute the dot product of two vectors.

haskell

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = sum (zipWith (*) xs ys)
```

- **Explanation**:
  - **zipWith (*) xs ys**: Multiplies corresponding elements of xs and ys.
  - **sum**: Adds up the resulting list of products.
- **Detailed Walkthrough**:

haskell
```
xs = [1, 2, 3]
ys = [4, 5, 6]

products = zipWith (*) xs ys  -- [4, 10, 18]
result = sum products         -- 32
```

Example: Implementing map with foldr

- **Implementation**:

haskell
```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\x acc -> f x : acc) []
```

- **Explanation**:
  - **Lambda Function**: \x acc -> f x : acc applies f to x and prepends it to acc.
  - **Base Case**: The accumulator starts as an empty list [].
  - **Process**: Builds the mapped list by folding from the right.

Example: Implementing filter with foldr

- **Implementation**:

haskell
```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
```

- **Explanation**:
  - **Predicate p**: Determines if x should be included.
  - **Decision**: If p x is True, include x; otherwise, skip it.

---

6. Understanding foldr and foldl

6.1. foldr (Right Fold)

- **General Pattern**:

haskell
```
foldr f z [x1, x2, ..., xn] == x1 `f` (x2 `f` ... (xn `f` z)...)
```

- **Visualization**:

go
```
foldr f z [x1, x2, x3]
== x1 `f` (x2 `f` (x3 `f` z))
```

- **Use Cases**:
  - Building up structures (like lists).
  - When the function f is right-associative.

6.2. foldl (Left Fold)

- **General Pattern**:

haskell
```
foldl f z [x1, x2, ..., xn] == (...((z `f` x1) `f` x2) `f`...) `f` xn
```

- **Visualization**:

go
foldl f z [x1, x2, x3]
== ((z `f` x1) `f` x2) `f` x3

- **Use Cases**:
  - Efficient for certain operations due to tail recursion.
  - When the function f is left-associative.

6.3. Differences Between foldr and foldl

- **Order of Application**:
  - foldr processes elements from right to left.
  - foldl processes elements from left to right.
- **Laziness**:
  - foldr can work on infinite lists because it can produce results without traversing the entire list (under certain conditions).
  - foldl is strict and requires traversing the entire list.

6.4. Example with foldl

- **Reversing a List**:

haskell
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

- **Explanation**:
  - The lambda function prepends x to the accumulator acc.
  - Since foldl processes from the left, it effectively reverses the list.

---

7. Practical Examples and Exercises

Exercise 1: Implement map using foldr

- **Solution**:

haskell
map'' :: (a -> b) -> [a] -> [b]
map'' f = foldr (\x acc -> f x : acc) []

Exercise 2: Implement filter using foldr

- **Solution**:

haskell
filter'' :: (a -> Bool) -> [a] -> [a]
filter'' p = foldr (\x acc -> if p x then x : acc else acc) []

Exercise 3: Reverse a List using foldr

- **Solution**:

haskell
reverse'' :: [a] -> [a]
reverse'' = foldr (\x acc -> acc ++ [x]) []

- **Note**: This is less efficient than using foldl due to list concatenation (++).

Exercise 4: Split a List Based on a Predicate

- **Problem**: Separate elements that satisfy a predicate from those that don't.
- **Solution**:

haskell
partition :: (a -> Bool) -> [a] -> ([a], [a])

```
partition p = foldr select ([], [])
  where
    select x (yes, no)
      | p x       = (x : yes, no)
      | otherwise = (yes, x : no)
```

---

8. Conclusion

- **Higher-Order Functions**:
  - Enable concise and expressive code.
  - Promote code reuse and abstraction.
- **Currying**:
  - Fundamental to Haskell's function application.
  - Allows partial application and function composition.
- **Functional Programming Paradigm**:
  - Emphasizes immutability and pure functions.
  - Facilitates reasoning about code and mathematical proofs.

Lecture 8, Oct 2:

1. Tail Recursion: Definition

A recursive function is said to be **tail recursive** if:

- The result of the function is the result of the recursive call.
- There is no need for any further computation after the recursive call returns, making it the last action performed by the function.

In simpler terms, a tail-recursive function has the recursive call as the final operation. No work is left to do after the recursive call returns, making it easy for the compiler or interpreter to optimize the recursion by reusing the current function's stack frame instead of creating new ones.

Example of Tail Recursion:

haskell

```
sum' :: Int -> [Int] -> Int
sum' z []     = z
sum' z (x:xs) = sum' (z + x) xs
```

Here, the recursive call to sum' is the last thing done in the function, so it's tail-recursive.

2. Non-Tail Recursive Function vs Tail Recursive Function

A non-tail recursive function has additional computations that happen after the recursive call, which prevents optimization by reusing the same stack frame.

Non-Tail Recursive Example (Sum):

haskell

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum(xs)
```

In this example, after the recursive call sum(xs) returns, the result still needs to be added to x. This makes it non-tail-recursive.

In contrast, the tail-recursive version:

haskell

```
sum' :: Int -> [Int] -> Int
sum' z []     = z
sum' z (x:xs) = sum' (z + x) xs
```

Here, the recursive call is the final action, with no further computation after it.

3. Understanding Tail Recursive Programs

The key idea behind tail recursion is that the function doesn't need to "remember" anything between recursive calls because there's no work left to do after the recursive step.

For example, in the tail-recursive sum' function:

haskell

```
sum' 0 [1,2,3]
```

This is evaluated as:

haskell

```
sum' 0 [1,2,3]
= sum' (0 + 1) [2,3]
= sum' 1 [2,3]
= sum' (1 + 2) [3]
= sum' 3 [3]
= sum' (3 + 3) []
= sum' 6 []
```

= 6
The key here is that z accumulates the result at each step and is passed forward, so there's nothing left to compute after the recursion returns.

4. General Definition of Tail Recursion

A recursive function is **tail recursive** if:

- The result of the recursive function is immediately returned without any additional computation.

Mathematically, a recursive function f(x) is tail-recursive if:

- f(x) = ... f(y) ... Where the recursive call f(y) is the last thing evaluated before returning.

5. Example: Tail-Recursive Sum

Here's how the tail-recursive sum works step by step:

haskell
```
sum' z [] = z
sum' z (x:xs) = sum' (z + x) xs
```
Running it for [1, 2, 3]:

haskell
```
sum' 0 [1,2,3]
= sum' (0 + 1) [2,3]
= sum' 1 [2,3]
= sum' (1 + 2) [3]
= sum' 3 [3]
= sum' (3 + 3) []
= 6
```
At each step, the result accumulates in z until the list is exhausted, at which point the accumulated result is returned.

6. Tail Recursion and Induction

To prove a tail-recursive function is correct, you can use **induction**. For instance, for a tail-recursive sum, you can prove that the function correctly computes the sum of all elements in a list.

Inductive Proof:

For the function sum' z xs where sum' z xs = z + Σ(xs):

- **Base case**: For an empty list xs = []:

haskell
```
sum' z [] = z = z + Σ []
```
This is correct because the sum of an empty list is 0, and adding 0 to z doesn't change the result.

- **Inductive case**: Suppose for a list xs, sum' z xs correctly computes z + Σ(xs). We want to show that for x:xs, sum' z (x:xs) computes z + Σ(x:xs):

haskell
```
sum' z (x:xs) = sum' (z + x) xs
```
By the induction hypothesis, sum' (z + x) xs = (z + x) + Σ(xs) = z + Σ(x:xs), which completes the proof.

7. Tail Recursive Example: Reverse Function

A classic example of tail recursion is the reverse function. Here’s the tail-recursive version:

haskell
```
rev :: [a] -> [a]
rev xs = rev' [] xs
```

```
rev' :: [a] -> [a] -> [a]
rev' acc []     = acc
rev' acc (x:xs) = rev' (x:acc) xs
```

In this example:

- acc is the accumulator that holds the result as we traverse the list.
- Each recursive call adds an element from the original list to the front of the accumulator.
- This ensures that once the recursion finishes, the reversed list is complete.

8. Loop Emulation in Haskell Using Tail Recursion

You can also emulate loops using tail recursion. For example, a loop that updates multiple variables can be written as a tail-recursive function.

Here's a template for a loop in Haskell:

haskell

```
loopy :: Int -> Int -> Int -> Int
loopy x y z = if condition x y z
              then loopy (updateX x) (updateY y) (updateZ z)
              else result x y z
```

This mimics a while loop that repeatedly updates x, y, and z until a condition is met.

Key Takeaways:

- **Tail recursion** is a form of recursion where the recursive call is the final action in the function.
- **Optimization**: Tail-recursive functions can be optimized by the compiler to reuse the current stack frame, avoiding stack overflow for deep recursion.
- **Accumulators**: Tail-recursive functions often use an accumulator to carry the result forward, avoiding the need for additional computation after the recursive call returns.
- **Proving correctness**: You can use induction to prove the correctness of tail-recursive functions.

Lecture 9, Oct 7:

Understanding and Detailed Notes on Monads, IO, and Maybe from the PDF

1. What is a Monad?

A **monad** is a design pattern used in functional programming, particularly in Haskell, to handle various types of computations in a uniform way. Monads provide a way to chain computations together, allowing you to build complex sequences of actions while maintaining clean and readable code.

A monad is defined by the following two core operations:

- **return**: This takes a value and puts it into a default minimal context (a monadic context).

haskell

return :: a -> m a

Here, a is a value, and m is the monadic context (such as IO, Maybe, or []).

- **bind (>>=)**: This operation takes a monadic value and a function, applies the function to the value inside the monad, and returns a new monadic value.

haskell

(>>=) :: m a -> (a -> m b) -> m b

The >>= operation is crucial for chaining computations in monads.

2. Example of Monad with List

The list monad provides a useful example of how return and >>= work:

- **return x = [x]**: This puts a single value into a list, the minimal context for lists.
- **xs >>= f**: Here, f is a function that takes a single value and returns a list. xs >>= f applies f to every element of the list xs and concatenates the results (i.e., flatMap or bind).

3. IO Monad

Haskell's **IO Monad** is used to handle input and output. Unlike other programming languages where IO can occur anywhere, Haskell separates IO from pure computation using the IO monad. This makes reasoning about effects easier and helps ensure that IO happens only where explicitly allowed.

- **IO a**: A value of type IO a represents an IO action that, when performed, produces a value of type a.
- **putStrLn :: String -> IO ()**: This is a built-in function that takes a string and returns an IO action. The IO action prints the string to the console when executed.
- **getLine :: IO String**: This function reads a line of input from the user.

4. Key Concept: IO Actions Are Not Executed Until Called

Evaluating an IO action (such as putStrLn "Hello") does not immediately perform the IO. Instead, it creates an IO action that can later be run in the appropriate context (like at the top level or in a do block). The actual IO only happens when the program is executed or when the action is run at the ghci prompt.

5. The do Notation

The do notation is syntactic sugar that makes chaining IO actions (or other monadic actions) easier to write and understand. In a do block, each line represents a monadic action.

Example:

haskell

```
f :: String -> IO ()
f greeting = do
  putStrLn greeting
```

```
  str <- getLine
  let output = "Here's the line I read: " ++ str
  putStrLn output
```
In this example:

- putStrLn greeting prints the greeting.
- str <- getLine reads a line from the input and stores it in str.
- The let binding is used to create a normal (non-IO) value output, which is then printed.

6. The return Function in IO

The return function does not perform any IO. Instead, it creates an IO action that, when executed, will simply produce the value you give it. For example:

haskell
```
return 42 :: IO Int
```
This creates an IO action that produces the value 42 but does not perform any actual IO like reading or writing.

7. An Example of an Interactive Calculator in IO

The document provides an example of a simple stack-based calculator. The user can input operators or numbers, and the stack is updated and displayed accordingly.

Operations:

- Numbers are pushed onto the stack.
- + sums the top two elements.
- * multiplies the top two elements.
- / divides the top two elements.
- - negates the top element.
- c clears the stack.

Here's an interaction:

css
```
User Input      Stack
c               []
17              [17]
3               [3,17]
1               [1,3,17]
-               [-1,3,17]
+               [2,17]
*               [34]
```
This shows how you can create complex interactive applications in Haskell using the IO monad.

8. Maybe Monad

The Maybe type is another example of a monad. It represents computations that can fail.

The Maybe type has two constructors:

- **Just a**: Represents a value of type a.
- **Nothing**: Represents a failure or absence of a value.

The monadic operations for Maybe work like this:

- **return x = Just x**: This puts a value into the Maybe context.
- **Nothing >>= f = Nothing**: If the computation has already failed, it stays failed.
- **Just x >>= f = f x**: If there is a value, apply f to the value.

This allows chaining computations that can fail, without needing to manually check for failure at each step.

Example:
haskell
```
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

result :: Maybe Int
result = Just 6 >>= \x -> Just 3 >>= \y -> safeDiv x y
```
This chains two Maybe values and a division that could fail, resulting in Just 2.

Conclusion:

- **Monads** are powerful constructs that allow chaining computations in various contexts, such as IO and Maybe.
- **IO Monad**: Used to handle input and output in a structured and safe way, separating IO from pure computation.
- **Maybe Monad**: Handles computations that can fail, allowing for clean handling of errors without manually checking for failure at each step.
- **do Notation**: Simplifies chaining monadic actions, especially in IO operations.

Lecture 10, Oct 9:

1. The Issue: Uncaught Exceptions

In many programming languages, uncaught exceptions can lead to unexpected program crashes. A typical example is a **divide-by-zero** error, which causes the program to halt:

bash

```
$ some_important_app
***Uncaught exception: divide by zero***
```

Problem: If the program encounters an error such as dividing by zero, it crashes unexpectedly without allowing the programmer to handle the error in a controlled manner.

Solution in Modern Languages: Many modern languages (like **Swift** or **Go**) enforce type-checking mechanisms that make programmers handle potential failures. This ensures that errors are dealt with where they occur, preventing unhandled exceptions from crashing the application.

---

2. The Maybe Type

The **Maybe** type is a way to handle errors or the absence of a value in a type-safe manner. Instead of throwing an exception, Haskell uses **Maybe** to represent computations that might fail.

haskell

```
data Maybe a
  = Nothing
  | Just a
```

- **Just a**: Represents a successful computation with a value of type a.
- **Nothing**: Represents a failure, error, or absence of value.

When using **Maybe**, all consumers of the value must explicitly handle both possible cases: either the value exists (with Just) or it doesn't (with Nothing).

Example:

haskell

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing
safeDivide x y = Just (x `div` y)
```

- safeDivide will return Nothing if the second argument is zero (to avoid division by zero). Otherwise, it will return Just the result of the division.

---

3. Handling Maybe Values Directly Can Be Tedious

When working with **Maybe** values, you often have to write code that manually handles the two cases, which can become cluttered and repetitive.

Example of Direct Handling:

haskell

```
case safeDivide 10 2 of
  Just result -> print result
  Nothing     -> putStrLn "Error: Division by zero!"
```

In more complex applications, handling **Maybe** values directly like this could lead to cumbersome and hard-to-read code.

---

4. Monads to the Rescue (Do Notation for Maybe)

Monads, like **IO** in Haskell, provide a way to chain computations together in a clean and structured way. This can also be applied to the **Maybe** type, which allows for handling computations that might fail in a more readable and elegant manner.
Do Notation allows us to work with **Maybe** values more fluently by chaining operations without having to manually unwrap the **Maybe** values at each step.
Do Notation:
haskell
```
do
  -- line 1
  -- line 2
  -- ...
  -- line n
```
Each line in the **do** block can represent one of the following:

- x <- e where e is an expression of type Maybe a. This line evaluates e, and if the result is Nothing, the entire block results in Nothing. If e is Just v, the value v is assigned to x, and the next line is executed.
- e is an expression of type Maybe a. If the result is Nothing, the entire block results in Nothing. If e is Just v, the next line is executed.
- let x = e assigns the result of e (an expression of any type) to x. This does not involve any **Maybe** value, and it is purely a local binding.

Example:
haskell
```
safeOperations :: Maybe Int
safeOperations = do
  x <- safeDivide 10 2
  y <- safeDivide x 2
  return (x + y)
```
In this example:

- If any safeDivide call results in Nothing, the entire **do** block will evaluate to Nothing.
- If all divisions succeed, the result will be Just (x + y).

---

5. How Do Notation Works with Maybe
The **do notation** provides a structured way to handle a sequence of computations that might fail. Here's how it works step-by-step:

1. **x <- e**:
    - If e is Nothing, the entire **do** block returns Nothing.
    - If e is Just v, the value v is assigned to x, and the remaining lines of the block are executed.
2. **e**:
    - If e is Nothing, the entire **do** block returns Nothing.
    - If e is Just v, the next line is executed.
3. **let x = e**:
    - This simply assigns the result of e to x. It does not affect the **Maybe** monad at all, and no failure is possible at this step.

By using **do notation**, you can avoid manually checking whether each computation returns Nothing, making your code more readable and reducing error-prone boilerplate.

---

6. Conclusion

The **Maybe Monad** allows you to handle computations that might fail in a structured way, without relying on exceptions or manually checking for failures at every step. Using **do notation**, you can chain multiple computations while handling errors gracefully.

This approach ensures that the program either completes successfully with all operations producing valid results, or the entire computation fails as soon as an error is encountered (returning Nothing).
