UNIT-4

4.1 Basic I/O: Haskell, input and output (I/O) are handled differently compared to imperative languages like Python or Java. Haskell is a pure functional language, so I/O operations are encapsulated in the `IO` type to maintain purity.

4.1.1 Understanding the Essentials of I/O: To understand the essentials of input and output in Haskell, you need to grasp the following foundational concepts:

i. Functional Purity and I/O Type

- Haskell is a purely functional language, meaning functions cannot have side effects.
- Input and output (I/O) are side effects. To manage this while keeping the language pure, Haskell uses the IO type to encapsulate actions that interact with the external world.
- Any function performing I/O must have a return type of IO.

```
main :: IO ()
main = putStrLn "Hello, world!"
```

- `putStrLn "Hello, world!"` is an `IO` action.
- The IO () type means this action performs output but doesn't produce a meaningful value (() is like void in other languages).

ii. Sequencing Actions with do Notation

- The do notation is used to sequence multiple IO actions.
- Each action in the do block runs in order.

```
main :: IO ()
main = do
  putStrLn "What is your favorite number?"
  number <- getLine
  putStrLn ("You entered: " ++ number)
```

- `<-` binds the result of an `IO` action to a variable (e.g., `number`).
- `getLine` reads input as a `String`.

iii. Pure vs. Impure Code

- Haskell separates pure functions (no side effects) from impure actions (I/O).
- Pure functions process data, while I/O is limited to interactions with the environment.

Ex1:
```
square :: Int -> Int
square x = x * x
```
Ex2:
```
main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let num = read input :: Int
  putStrLn ("Square: " ++ show (square num))
```

iv. Laziness in I/O

- Haskell's evaluation is lazy, but I/O actions are executed in the order they appear in the do block.

Ex:
```
main :: IO ()
  main = do
  putStrLn "First action"
  putStrLn "Second action"
```
O/P:
```
First action
Second action
```

v. Common I/O Functions

- **Output**:
  - putStr – prints without a newline
  - putStrLn – prints with a newline.
- **Input**:
  - getLine – reads a single line of input as a String.

- **File Operations**:
  - readFile – reads the contents of a file.
  - writeFile – writes data to a file.

4.1.2 Manipulating I/O Data: Manipulating input and output data in Haskell involves reading data from the user, transforming it with pure functions, and displaying the result. Haskell separates pure and impure code, the process generally follows this structure:

- Read input using I/O actions.
- Transform the data using pure functions.
- Output the results using I/O actions.

i. Reversing User Input reads a line of input, reverses it, and prints the result.

```
main :: IO ()
main = do
  putStrLn "Enter some text:"
  input <- getLine
  let reversed = reverse input  -- Pure function
  putStrLn ("Reversed text: " ++ reversed)
```

ii. Summing a List of Numbers reads a comma-separated list of numbers, computes the sum, and displays it.

```
main :: IO ()
main = do
  putStrLn "Enter a list of numbers separated by commas:"
  input <- getLine
  let numbers = map read (splitBy ',' input) :: [Int]  -- Parse input
    total = sum numbers                    -- Compute sum
  putStrLn ("The sum is: " ++ show total)
```

iii. Transforming Strings converts user input to uppercase.

```
import Data.Char (toUpper)
main :: IO ()
main = do
  putStrLn "Enter a string:"
  input <- getLine
  let uppercased = map toUpper input  -- Pure transformation
  putStrLn ("Uppercased: " ++ uppercased)
```

iv. Filtering Data reads a list of integers and filters out the even numbers.

```
main :: IO ()
main = do
  putStrLn "Enter a space-separated list of integers:"
  input <- getLine
  let numbers = map read (words input) :: [Int]  -- Parse input
    odds = filter odd numbers          -- Filter odd numbers
  putStrLn ("Odd numbers: " ++ unwords (map show odds))
```

v. Using Multiple Outputs performs a series of calculations and displays multiple results.

```
main :: IO ()
main = do
  putStrLn "Enter a number:"
  input <- getLine
  let num = read input :: Int
    square = num * num
    cube = num * num * num
  putStrLn ("Square: " ++ show square)
  putStrLn ("Cube: " ++ show cube)
```

4.1.3 Receiving and Sending I/O with Haskell: Receiving and sending input/output (I/O) in Haskell involves working with the IO monad to interact with the user or external systems.

Receiving Input Haskell provides several ways to receive input, such as reading from the terminal or files.

Reading Input from the Terminal

- **getLine**: Reads a single line of text from the user.
- **getContents**: Reads all input until EOF (useful for piped input).

Ex: getLine

```
main :: IO ()
main = do
  putStrLn "What is your favorite programming language?"
  lang <- getLine
  putStrLn ("You love " ++ lang ++ "!")
```

Ex: getcontents

```
main :: IO ()
main = do
  putStrLn "Type some text and press Ctrl+D (Linux/Mac) or Ctrl+Z (Windows):"
  input <- getContents
  putStrLn "You entered:"
  putStrLn input
```

Sending Output Haskell provides simple functions for output:

- **`putStr`**: Outputs a string without a newline.
- **`putStrLn`**: Outputs a string with a newline.
- **`print`**: Outputs a value after converting it to a string using show

Ex:

```
main :: IO ()
 main = do
  putStr "Hello, "
  putStrLn "world!"
  print 12345
```

O/P: Hello, world! 12345

File I/O: Haskell provides functions to read and write files.

i. Reading a File

- Use `readFile` to read the contents of a file into a `String`.

```
main :: IO ()
main = do
  content <- readFile "example.txt"
  putStrLn "File contents:"
  putStrLn content
```

ii. Writing to a File

- Use `writeFile` to write data to a file.

```
main :: IO ()
main = do
  let content = "Hello, Haskell file!"
  writeFile "output.txt" content
  putStrLn "Data written to output.txt"
```

4.1.4 Interacting with files: Interacting with files in Haskell is straightforward using built-in functions provided in the Prelude and the System.I/O module.g v

i. **Reading from a File** The readFile function reads the contents of a file into a String.

```
main :: IO ()
main = do
  content <- readFile "example.txt"
  putStrLn "File contents:"
  putStrLn content
```

- **`readFile`** takes the file path as input and returns the file's content as a String.
- The file must exist, or the program will throw an error.

ii. **Writing to a File** The writeFile function writes a String to a file. If the file exists, it is overwritten.

```
main :: IO ()
main = do
 let content = "Hello, Haskell!"
   writeFile "output.txt" content
   putStrLn "Data written to output.txt"
```

iii. Appending to a File The appendFile function appends a String to an existing file. If the file doesn't exist, it is created.

```
main :: IO ()
main = do
   let extraContent = "\nAppending this line."
   appendFile "output.txt" extraContent
   putStrLn "Data appended to output.txt"
```

iv. Reading and Writing Binary Files For binary data, you can use Data. ByteString or Data.ByteString.Lazy (for large files).

```
import qualified Data.ByteString as BS
main :: IO ()
main = do
   -- Writing binary data
   BS.writeFile "binary.dat" (BS.pack [72, 101, 108, 108, 111]) -- "Hello"
   -- Reading binary data
   content <- BS.readFile "binary.dat"
   putStrLn ("Binary content: " ++ show content)
```

v. Using System.I/O **for Fine-Grained Control** The System.I/O module provides lower-level control over file operations, such as specifying how a file is opened and closed.

```
import System.IO
main :: IO ()
main = do
   handle <- openFile "example.txt" ReadMode
   content <- hGetContents handle
   putStrLn "File contents:"
   putStrLn content
   hClose handle
```

vi. Checking if a File Exists use the doesFileExist function from the System.Directory module to check for a file's existence.

```
import System.Directory (doesFileExist)
main :: IO ()
main = do
   let filePath = "example.txt"
   exists <- doesFileExist filePath
   if exists
      then putStrLn (filePath ++ " exists!")
      else putStrLn (filePath ++ " does not exist!")
```

4.1.5 Manipulating File content: Manipulating file content in Haskell involves reading data from a file, transforming it, and writing the updated content back to the file or a new file.

i. Reading and Transforming File Content the content of a file, process it using pure functions, and then write the transformed content to another file.

```
import System.IO
import Data.Char (toUpper)

main :: IO ()
main = do
   -- Read file content
   content <- readFile "input.txt"
   -- Transform content
   let uppercased = map toUpper content
```

```
  -- Write to a new file
  writeFile "output.txt" uppercased
  putStrLn "File content converted to uppercase and saved to output.txt"
```

ii. Filtering Lines filter specific lines from a file based on a condition.

```
import System.IO
main :: IO ()
main = do
  putStrLn "Enter a keyword to filter lines:"
  keyword <- getLine
  content <- readFile "input.txt"
  let linesContainingKeyword = unlines (filter (keyword `elem`) (lines content))
  writeFile "filtered.txt" linesContainingKeyword
  putStrLn "Filtered lines saved to filtered.txt"
```

iii. Reversing Lines Reverse the order of lines in a file.

```
main :: IO ()
main = do
  content <- readFile "input.txt"
  let reversedLines = unlines (reverse (lines content))
  writeFile "reversed.txt" reversedLines
  putStrLn "Reversed lines saved to reversed.txt"
```

iv. Counting Words and Lines Count the number of words and lines in a file.

```
main :: IO ()
main = do
  content <- readFile "input.txt"
  let numLines = length (lines content)
      numWords = length (words content)
  putStrLn ("Number of lines: " ++ show numLines)
  putStrLn ("Number of words: " ++ show numWords)
```

4.1.6 Completing File-related Tasks: Completing file-related tasks in Haskell involves combining various file I/O operations such as reading, writing, appending, transforming, and handling exceptions.

i. Copying a File Copy the content of one file to another.

```
import System.IO
main :: IO ()
main = do
  putStrLn "Enter the source file:"
  sourceFile <- getLine
  putStrLn "Enter the destination file:"
  destFile <- getLine
  content <- readFile sourceFile
  writeFile destFile content
  putStrLn $ "File copied from " ++ sourceFile ++ " to " ++ destFile
```

ii. Merging Two Files Read two files and merge their contents into a new file.

```
main :: IO ()
main = do
  putStrLn "Enter the first file:"
  file1 <- getLine
  putStrLn "Enter the second file:"
  file2 <- getLine
  content1 <- readFile file1
  content2 <- readFile file2
  let mergedContent = content1 ++ "\n" ++ content2
  writeFile "merged.txt" mergedContent
  putStrLn "Files merged into merged.txt"
```

iii. Splitting a File Split a file into two files based on a line count.

```
main :: IO ()
main = do
```

```
  putStrLn "Enter the file to split:"
  file <- getLine
  putStrLn "Enter the number of lines for the first split:"
  n <- readLn
  content <- readFile file
  let fileLines = lines content
    (part1, part2) = splitAt n fileLines
  writeFile "part1.txt" (unlines part1)
  writeFile "part2.txt" (unlines part2)
  putStrLn "File split into part1.txt and part2.txt"
```

iv. Deleting Specific Lines Remove lines that match a condition (e.g., lines containing a keyword).

```
import Data.List (isInfixOf)
main :: IO ()
main = do
  putStrLn "Enter the file to modify:"
  file <- getLine
  putStrLn "Enter the keyword to remove lines containing it:"
  keyword <- getLine
  content <- readFile file
  let filteredContent = unlines $ filter (not . isInfixOf keyword) (lines content)
  writeFile file filteredContent
  putStrLn "Lines containing the keyword have been removed."
```

4.2 Handling Errors in Haskell: Error handling in Haskell ensures your program can gracefully handle unexpected scenarios, such as missing files, invalid user input, or runtime exceptions.

i. Basic Error Handling with Maybe and **Either** Haskell's standard types, Maybe and Either, are commonly used for handling errors in pure code.

Ex: Maybe

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing -- Division by zero
safeDivide x y = Just (x `div` y)
main :: IO ()
main = do
  case safeDivide 10 0 of
    Just result -> putStrLn ("Result: " ++ show result)
    Nothing -> putStrLn "Error: Division by zero"
```

Ex: Either

```
safeDivide :: Int -> Int -> Either String Int
safeDivide _ 0 = Left "Error: Division by zero"
safeDivide x y = Right (x `div` y)
main :: IO ()
main = do
  case safeDivide 10 2 of
    Right result -> putStrLn ("Result: " ++ show result)
    Left err -> putStrLn err
```

ii. Exception Handling in I/O For handling runtime exceptions in IO actions, use the Control.Exception module.

```
import System.IO
import Control.Exception
main :: IO ()
main = do
  let file = "nonexistent.txt"
  content <- catch (readFile file)
          (\e -> do
            let err = show (e :: IOError)
            putStrLn $ "Caught exception: " ++ err
            return "")
```

```
    putStrLn content
```

iii. Custom Exceptions Define and handle your own exceptions using the Exception typeclass.

```
import Control.Exception
import Data.Typeable
data MyException = MyException String
  deriving (Show, Typeable)
instance Exception MyException
main :: IO ()
main = do
  let computation = throw (MyException "Something went wrong!")
  catch computation (\(MyException msg) -> putStrLn $ "Caught exception: " ++ msg)
```

iv. Best Practices for Error Handling

1. **Use Pure Error Handling (**Maybe**,** Either**)**:
   - For errors that can be caught at the function level without side effects.
2. **Leverage Exceptions for Unexpected Errors**:
   - For runtime errors like missing files or network failures.
3. **Encapsulate Resources with** bracket:
   - Always clean up resources, like closing file handles or network connections.
4. **Provide Meaningful Error Messages**:
   - Avoid cryptic messages; explain what went wrong.
5. **Use Libraries for Complex Scenarios**:
   - Libraries like safe-exceptions or unliftio provide advanced patterns.

4.2.1 Defining a bug in Haskell: In Haskell a bug refers to an unintended behavior or error in the code due to logic errors, incorrect assumptions, or unforeseen edge cases. Since Haskell emphasizes correctness through its strong static type system and pure functional approach, many bugs can be avoided during compilation.

i. Types of Bugs in Haskell

a. Logical Errors Logical errors occur when the code compiles but does not behave as expected due to flawed logic.

```
Incorrect implementation of factorial
factorial :: Int -> Int
factorial n = if n == 0 then 1 else n + factorial (n - 1)
```

Bug: The `+` operator should be `*`. This results in incorrect computation.

b. Pattern-Matching Failures Pattern-matching failures occur when a function does not handle all possible cases of input.

```
head' :: [a] -> a
head' (x:_) = x
```

Bug: The function does not handle the empty list case. Calling `head' []` will cause a runtime exception: `Non-exhaustive patterns in function`.

c. Infinite Loops Infinite loops occur when a function recurses or evaluates indefinitely.

```
loop :: Int
loop = loop + 1
```

Bug: The recursion has no base case, causing an infinite loop.

d.Laziness-Related Bugs Haskell's lazy evaluation can lead to unintended memory usage or performance issues, such as a space leak**.**

```
sumList :: [Int] -> Int
sumList xs = foldl (\acc x -> acc + x) 0 xs
```

Bug: The use of `foldl` creates a large chain of unevaluated thunks, leading to high memory consumption.

ii.Debugging Bugs in Haskell

a. Use GHC Warnings Enable compiler warnings for catching common issues early:

```
ghc -Wall -Werror MyFile.hs
```

b. Use trace for Debugging The Debug.Trace module allows you to log intermediate results during execution.

```
import Debug.Trace
factorial :: Int -> Int
factorial n = trace ("Calculating factorial for " ++ show n) $        if n == 0 then 1 else n * factorial (n - 1)
```

c. Use GHCi Interactive debugging with GHCi allows you to test functions step by step:

```
$ ghci
```

```
Prelude> :l MyFile.hs
*Main> factorial 5
```

d. Use Profiling GHC profiling tools can help identify performance issues, such as space leaks or excessive laziness:

```
ghc -prof -fprof-auto -rtsopts MyFile.hs
./MyFile +RTS -hc –p
```

iii. Preventing Bugs

- Leverage Haskell's Type System:
  - Use custom types and type constraints to catch logic errors early.
- Write Pure Functions:
  - Minimize side effects to simplify reasoning about code.
- Use Safe Library Functions:
  - Prefer safeHead over head, etc.
- Test Code Thoroughly:
  - Use frameworks like QuickCheck or Hspec for property-based testing.

4.2.2 Understanding the Haskell-Related Errors:

Haskell errors can broadly be categorized into compile-time errors and runtime errors**.** Understanding these errors is crucial to writing robust and bug-free Haskell programs.

i. **Compile-Time Errors** Haskell's type system and strict static analysis catch most errors at compile time. These include:

a. Syntax Errors Occur when the code does not follow the proper syntax.

```
main = putStrLn "Hello World
```

b. Type Errors Haskell enforces type safety, so mismatched or ambiguous types will cause errors.

```
add :: Int -> Int -> Int
add x y = x + y
main = print (add "1" 2)
```

c. Non-Exhaustive Patterns Occurs when pattern matching does not cover all cases.

```
describe :: Bool -> String
describe True = "It's True!"
```

d. Name Resolution Errors Happen when trying to use an undefined variable or function.

```
main = print (myVariable)
```

e. Ambiguous Types Occurs when the compiler cannot infer a specific type.

```
main = print (read "123")
```

ii. Runtime Errors are less common in Haskell because of its strong type system. However, they can still occur:

a. Pattern-Matching Errors Occurs when a function is called with an input that doesn't match any pattern.

```
head' :: [a] -> a
head' (x:_) = x
main = print (head' [])
```

b. Division by Zero Occurs when dividing by zero.

```
main = print (5 `div` 0)
```

c. File Not Found Occurs when trying to read a non-existent file.

```
import System.IO
main = do
  content <- readFile "nonexistent.txt"
  putStrLn content
```

d. Infinite Loops Occurs when a function has no base case or termination condition.

```
main = print (let x = x + 1 in x)
```

iii. Debugging Tools and Techniques

a. Enable Warnings Use `–Wall` to enable all warnings during compilation:

```
ghc -Wall MyFile.hs
```

b. Use trace for Debugging Use the `Debug.Trace` module to print intermediate values.

```
import Debug.Trace
factorial :: Int -> Int
factorial n = trace ("Calculating factorial for " ++ show n) $
  if n == 0 then 1 else n * factorial (n - 1)
```

c. Use GHCi Test individual functions interactively in GHCi:

```
$ ghci MyFile.hs
*Main> factorial 5
```

d. Profiling Identify performance issues using GHC's profiling tools:

```
ghc -prof -fprof-auto -rtsopts MyFile.hs
./MyFile +RTS –p
```

4.2.3 Fixing Haskell Errors Quickly: Fixing Haskell errors efficiently involves leveraging its strong type system, compiler feedback, and debugging tools.

i. Understand the Error Message Haskell's error messages are detailed but can sometimes seem cryptic to beginners.

- **Line number and context**: Look for where the error occurred.
- **Type mismatch**: Read the expected and actual types.
- **Scope issues**: Check for undefined variables or imports.
- **Pattern-matching problems**: Ensure all cases are handled.

ii. Common Haskell Errors and Quick Fixes

a. Syntax Errors Often caused by missing parentheses, brackets, or incorrect indentation.

```
main = putStrLn "Hello World
```

b. Type Mismatch Occurs when a function receives an argument of an unexpected type.

```
add :: Int -> Int -> Int
add x y = x + y
main = print (add "1" 2)
```

c. Undefined Variables or Functions Occurs when a name is used without being defined or imported.

```
main = print (myVariable)
```

d. Non-Exhaustive Pattern Matching Occurs when all cases are not handled in a pattern match.

```
describe :: Bool -> String
describe True = "It's True!"
```

iii. Leverage GHC Options

a. Enable Warnings Compile with `-Wall` to get comprehensive feedback:

```
ghc -Wall MyFile.hs
```

b. Treat Warnings as Errors Use `-Werror` to ensure no warnings go unnoticed:

```
ghc -Wall -Werror MyFile.hs
```

iv. Debugging Tools

a. Use `Debug.Trace` Insert `trace` statements to debug intermediate values.

```
import Debug.Trace
factorial :: Int -> Int
factorial n = trace ("Calculating factorial for " ++ show n) $
  if n == 0 then 1 else n * factorial (n - 1)
```

b. Use GHCi Test functions interactively:

```
$ ghci MyFile.hs
*Main> factorial 5
120
```

c. Profiling Identify performance bottlenecks or memory leaks:

```
ghc -prof -fprof-auto -rtsopts MyFile.hs
./MyFile +RTS –p
```