Praneesh R V
CB.SC.U4CYS23036

Principles of Programming Languages

Qn1,

Paradigm 1
Concurrent Programming – Rust

Introduction to Rust:
Rust is a systems programming language created by Mozilla that focuses on memory safety, high performance, and fearless concurrency. Unlike C and C++, Rust prevents memory errors without using a garbage collector, making it ideal for multithreaded applications.

Note:
Rust ensures thread safety using ownership, borrowing, and lifetimes. These rules enforce safety at compile time, preventing data races before the application runs.

Key Rust Language Features:
- Ownership & Borrowing – prevents unsafe shared mutable access across threads
- Send & Sync traits – guarantees when data is safe to move or share across threads
- Channels – thread-safe message passing pattern inspired by CSP
- async/await – lightweight asynchronous concurrency
- Arc/Mutex – safe shared-state synchronization when needed

Short codes:

Thread Spawning + Message Passing:

```
use std::thread;
use std::sync::mpsc;

fn main() {
let (tx, rx) = mpsc::channel();
thread::spawn(move || {
tx.send("Hello from thread").unwrap();
});
println!("{}", rx.recv().unwrap());
}
```

Async Concurrency:

```
async fn fetch() { println!("Fetching data..."); }
#[tokio::main]
async fn main() {
let t1 = fetch();
let t2 = fetch();
futures::join!(t1, t2);
}
```

Real-World Rust Concurrency Usage:

- Actix and Axum web servers — async performance rivaling Go
- Solana and Near blockchain clients — safe high-throughput execution
- Game engines and simulations — parallel computation without data races
- Operating systems (Redox OS + Linux components) — reliable multithreading

Paradigm 2
Logic Programming – CHR (Constraint Handling Rules)

Introduction to CHR:
CHR is a declarative logic programming language used for writing constraint solvers. Instead of specifying step-by-step instructions, the programmer defines rules, and the system determines the solution by applying and resolving constraints until consistency is reached.

Note:
CHR is commonly embedded in Prolog, Haskell, and Java. It is widely used when rule-based reasoning, deduction, and automated inference are required.

Key CHR Language Features:
- Constraints – represent relationships between variables
- Rules – transformation logic for constraints
- Simplification – replace complex constraints with simpler ones
- Propagation – infer new constraints from existing ones
- Declarative semantics – programmer specifies what the solution must satisfy, not how to compute it

Short codes:

CHR Example – Simplification + Propagation:

```
:- use_module(library(chr)).
:- chr_constraint leq/2.

leq(X,Y), leq(Y,X) <=> X = Y. % Simplification rule
leq(X,Y), leq(Y,Z) ==> leq(X,Z). % Propagation rule
```

Real-World CHR Usage:
- Scheduling and timetabling systems
- Type inference and compiler reasoning engines

• Expert systems and AI decision-making
• Access-control and policy reasoning in cybersecurity

Comparison Between Concurrent-Rust and Logic-CHR

Feature / Concept — Rust (Concurrent Programming) — CHR (Logic Programming)

Purpose — Enable high-performance parallel execution without data races — Automate reasoning and constraint solving based on logic rules
Core Ideas — Ownership, threads, async, message passing, synchronization — Constraints, rules, propagation, simplification
Execution Style — Imperative, step-by-step processing — Declarative, solver infers solution
Where Logic Lives — Inside threads, tasks, and concurrency control code — In rules and constraint expressions
Error Prevention — Compile-time guarantees enforcing thread safety — Logical consistency and rule semantics
Typical Use — Web servers, OS, blockchain, simulations, HPC — Scheduling, compilers, expert systems, AI reasoning
Main Benefit — Safe and efficient concurrency — High-level reasoning without explicit algorithms

---

Qn2,

Square of a number

```
square :: Num a => a -> a
square x = x * x
```

```haskell
main :: IO ()
main = do
  let result = square 7
  putStrLn $ "Square of 7: " ++ show result
```

```
> ghc square_of_a_number.hs
[1 of 2] Compiling Main             ( square_of_a_number.hs, square_of_a_number.o ) [Source file changed]
[2 of 2] Linking square_of_a_number [Objects changed]
> ./square_of_a_number
Square of 7: 49

~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ >
```

Qn3, length of list

```haskell
len :: [a] -> Int
len [] = 0
len (_:xs) = 1 + len xs

main :: IO ()
main = do
  putStrLn "enter the list([1,2,3,4]):"
  input <- getLine
  let list = read input :: [Int]
  putStrLn $ "len of the list is: " ++ show (len list)
```

```
>
> ./length_of_list
enter the list([1,2,3,4]):
[1,2,3,4,5,6,7,8]
len of the list is: 8

~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ >
```

Qn4, check if list is empty

```
main :: IO ()
main = do
 putStrLn "enter list:"
 xs <- readLn :: IO [Int]
 print (null xs)
```

```
> ./empty_list
enter list:
[]
True
> ./empty_list
enter list:
[1,2,3]
False
```

Qn5, area of cicle

```
main :: IO ()
main = do
 putStrLn "radius:"
 r <- readLn :: IO Double
 print (pi * r * r)
```

```
> ./area_of_circle
radius:
7
153.93804002589985

~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ >
```

Qn 6,Factorial using accumulator

```
main :: IO ()
main = do
 let factorial n = product [1..n]
 print (factorial 7)
```

```
❯ ghc factorial.hs
[1 of 2] Compiling Main             ( factorial.hs, factorial.o ) [Source file changed]
[2 of 2] Linking factorial [Objects changed]
❯ ./factorial
5040
```

Qn 7, Check if string is palindrome

```
isPalindromeStr :: String -> Bool
isPalindromeStr s = s == reverse s
main :: IO ()
main = do
 putStrLn "enter"
 line <- getLine
 putStrLn $ "the string is a palindrome: " ++ show
(isPalindromeStr line)
```

```
❯ ghc palindrome.hs
[1 of 2] Compiling Main             ( palindrome.hs, palindrome.o )
[2 of 2] Linking palindrome
❯ ./palindrome
enter
ahah
the string is a palindrome: False
❯ ./palindrome
enter
ahaha
the string is a palindrome: True
```

Qn 8, reverse a string

```
reverseStr :: String -> String
reverseStr [] = []
reverseStr (x:xs) = reverseStr xs ++ [x]
main :: IO ()
main = do
```

```
putStrLn "enter"
line <- getLine
putStrLn $ "the reverse is: " ++ show (reverse line)
```

```
> ghc reverse_a_string.hs
[1 of 2] Compiling Main             ( reverse_a_string.hs, reverse_a_string.o ) [Source file changed]
[2 of 2] Linking reverse_a_string [Objects changed]
> ./reverse_a_string
enter
praneesh
the reverse is: "hseenarp"

~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ >
```

Qn 9, The library function last selects the last element of a non-empty list; for example,
last [1,2,3,4,5] = 5. Show how the function last could be defined in terms of the other
library functions. Give two possible function definitions.

```
cLast :: [a] -> a
cLast [x] = x
cLast (_:xs) = cLast xs
cLast2 :: [a] -> a
cLast2 xs = head (reverse xs)
main :: IO ()
main = do
   putStrLn"enter"
   inp <- getLine
   let cLast0 = read inp :: [Int]
   putStrLn $ "answer: " ++ show (cLast cLast0)
```

```
●❯ ghc qn9.hs
  [1 of 2] Compiling Main             ( qn9.hs, qn9.o ) [Source file changed]
  [2 of 2] Linking qn9 [Objects changed]
●❯ ./qn9
  enter
  [1,67,5,8,34,5,78,10]
  answer: 10

○ ~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ ❯
```

Qn 10, The library function init removes the last element from a non-empty list;
for example, init [1,2,3,4,5] = [1,2,3,4]. Show how init could similarly be defined in two
different ways

```haskell
cInit :: [a] -> [a]
cInit [_] = []
cInit (x:xs) = x : cInit xs
cInit2 :: [a] -> [a]
cInit2 xs = reverse (tail (reverse xs))
main :: IO ()
main = do
   putStrLn"enter"
   inp <- getLine
   let cLast0 = read inp :: [Int]
   putStrLn $ "answer: " ++ show (cInit cLast0)
```

```
●❯ ghc qn10.hs
  [1 of 2] Compiling Main             ( qn10.hs, qn10.o )
  [2 of 2] Linking qn10
●❯ ./qn10
  enter
  [1,2,3,6,8,10]
  answer: [1,2,3,6,8]

○ ~/Praneesh/Academics/Sem6/POPL/Lab1 main* ⇡ ❯
```

Qn 11,
Find the second largest element (no sorting).
secondLargest :: Ord a => [a] -> a

```
secondLargest :: Ord a => [a] -> a
secondLargest (x:y:xs) = findSecond xs (max x y) (min x y)
   where
       findSecond [] max1 max2 = max2
       findSecond (z:zs) max1 max2
           | z > max1                = findSecond zs z max1
           | z > max2 && z < max1    = findSecond zs max1 z
           | otherwise               = findSecond zs max1 max2

main :: IO ()
main = do
   print (secondLargest [75,24,66,10,10,89])
```

```
> ghc qn11.hs
[1 of 2] Compiling Main             ( qn11.hs, qn11.o ) [Source file changed]
[2 of 2] Linking qn11 [Objects changed]
> ./qn11
75
```

Qn 12,
Check whether a list is a palindrome (recursive or reverse)
isPalindrome :: Eq a => [a] -> Bool

```
isPalindromeRev :: Eq a => [a] -> Bool
isPalindromeRev xs = xs == reverse xs

isPalindromeRec :: Eq a => [a] -> Bool
isPalindromeRec [] = True
isPalindromeRec [_] = True
```

```haskell
isPalindromeRec (x:xs) = x == last xs && isPalindromeRec (init
xs)

main :: IO ()
main = do
   print (isPalindromeRev [1,2,10])
   print (isPalindromeRev [1,2,1])
   print (isPalindromeRec [10,90,56,7])
```

```
● > ghc qn12.hs
  [1 of 2] Compiling Main             ( qn12.hs, qn12.o ) [Source file changed]
  [2 of 2] Linking qn12 [Objects changed]
● > ./qn12
  False
  True
  False
```