20CYS312 – PPL

Lab Exercise 9 - Rust Lab Exercise Questions

Name:Sushant Yadav **RegNo:En.u4cys22067**

Date:2025/03/14 **Course Code: 20CYS312**

---

1. Nested Decision Making with if-else

Write a Rust program that takes a person's age and income as input and determines their eligibility for a loan. The program should check:

>If the person is below 21, they are ineligible.

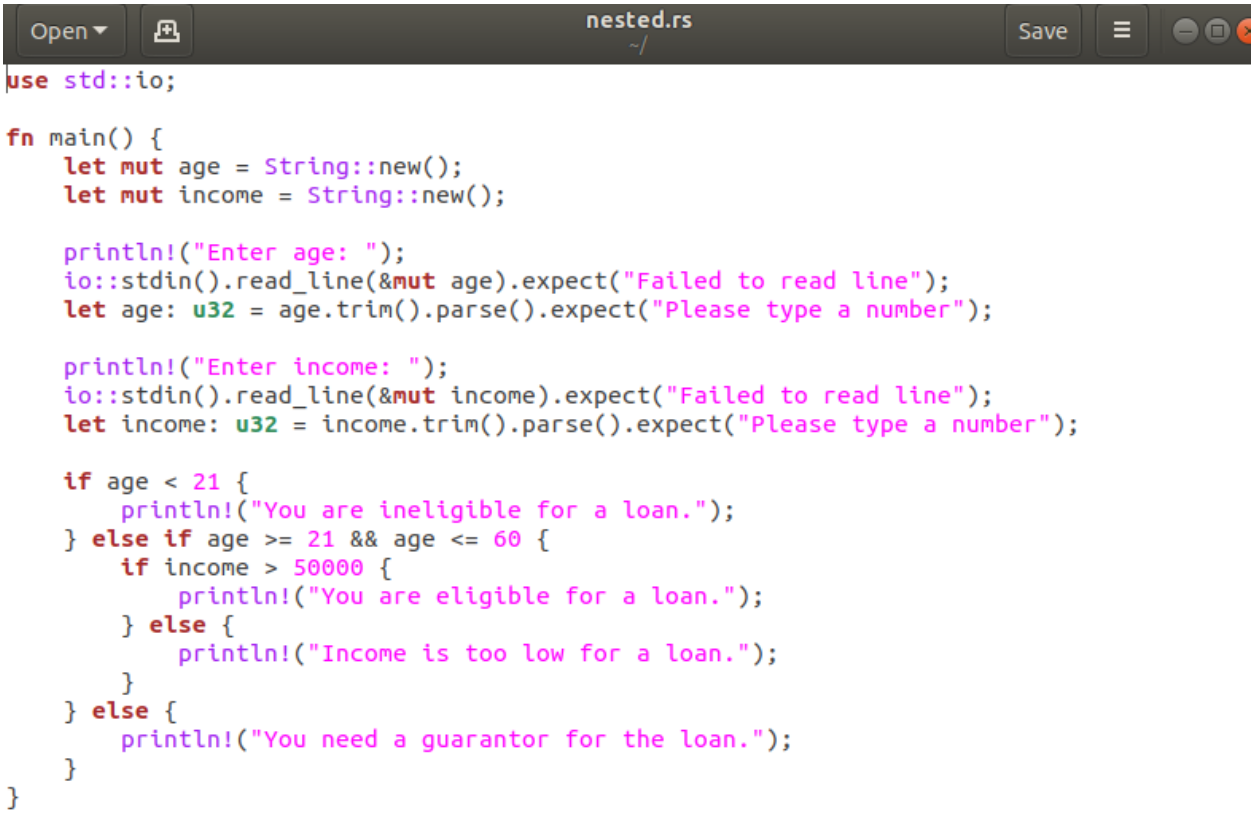
> If between 21 and 60, they are eligible based on income (> ₹50,000).

>If above 60, they need a guarantor.

Objectives:

- Understand conditional statements (if-else).
- Implement nested decision-making logic.
- Work with user input and type conversions.

Code:



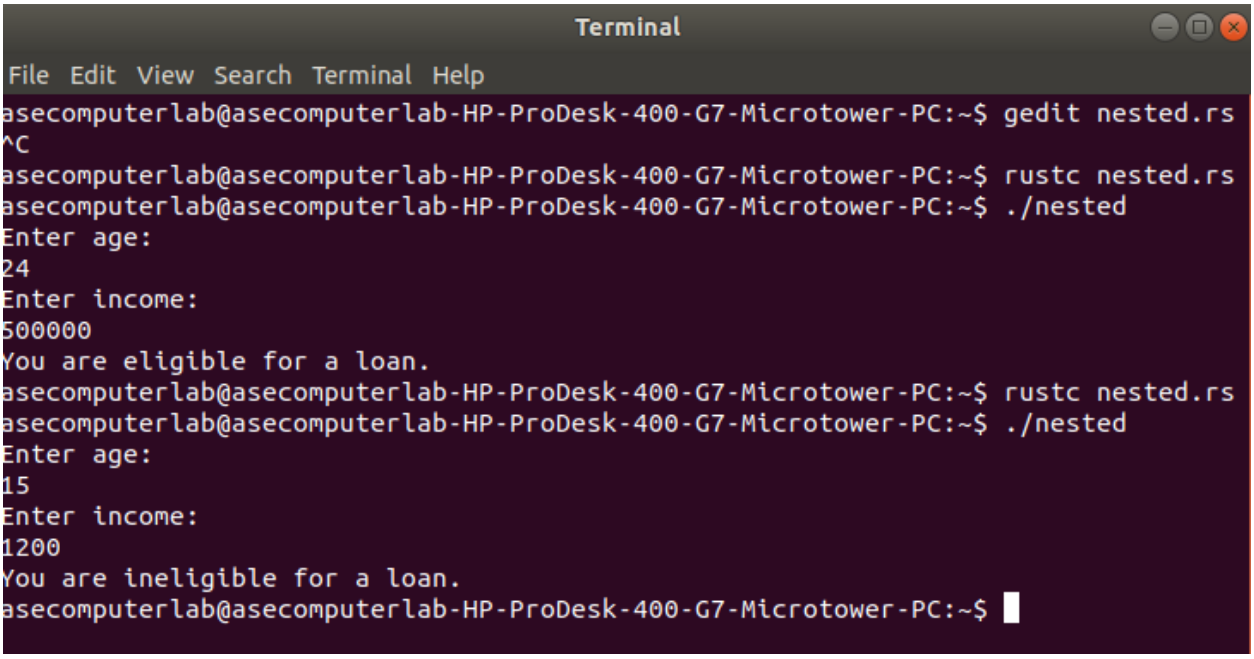
Explanation:

This program checks the person's eligibility for a loan based on their age and income. It uses nested if-else statements to cover different age and income conditions:

- If under 21, ineligible for a loan.
- If between 21 and 60, checks income for eligibility.

- If above 60, requires a guarantor.

Output:



Conclusion:

This program showcases how to use nested if-else statements in Rust to make decisions based on multiple conditions.

2.Using match with Complex Cases

Implement a restaurant billing system where a user enters a menu item (e.g.,

"Burger", "Pizza", "Pasta"), and the program prints the price.

>Use a match expression with additional conditions to apply discounts based on the quantity ordered.

Objectives:

- Understand the match statement with conditions.
- Implement complex matching with ranges.
- Use floating-point numbers to compute final amounts.

Code:

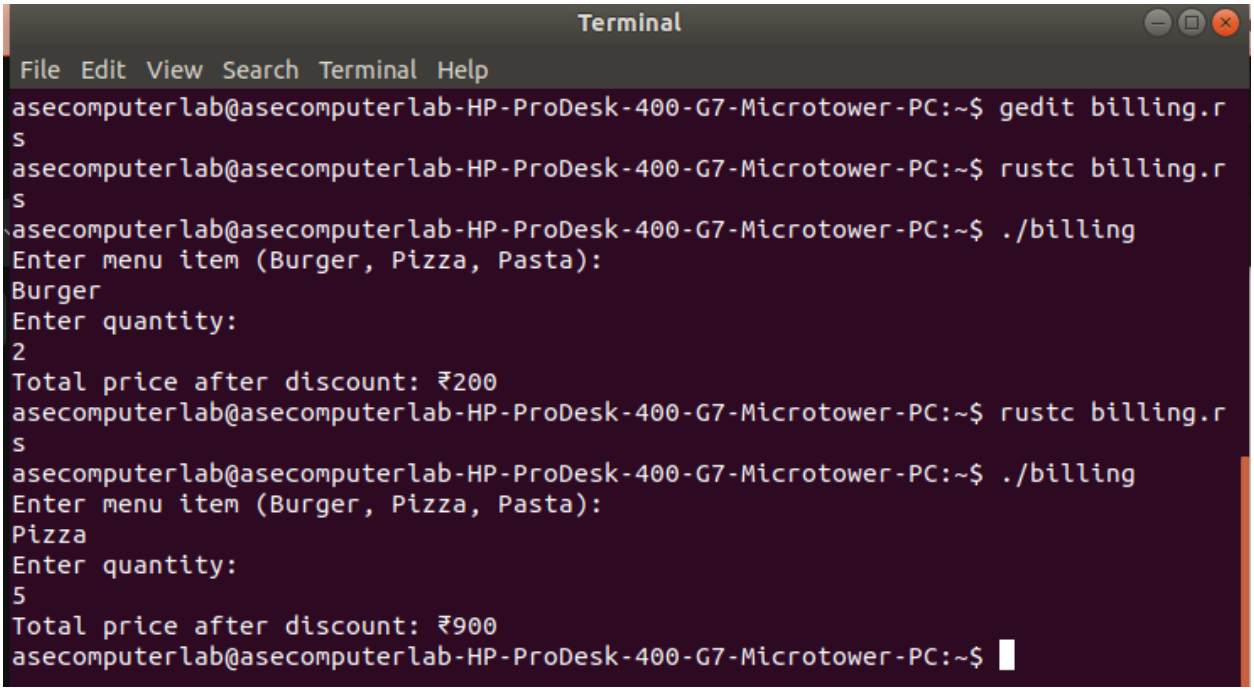
Open *billing.rs

```rust
use std::io;

fn main() {
    let mut item = String::new();
    let mut quantity = String::new();

    println!("Enter menu item (Burger, Pizza, Pasta): ");
    io::stdin().read_line(&mut item).expect("Failed to read line");
    let item = item.trim();

    println!("Enter quantity: ");
    io::stdin().read_line(&mut quantity).expect("Failed to read line");
    let quantity: u32 = quantity.trim().parse().expect("Please type a number");

    let price = match item {
        "Burger" => 100,
        "Pizza" => 200,
        "Pasta" => 150,
        _ => {
            println!("Item not found");
            return;
        }
    };

    let discount = match quantity {
        1..=3 => 0.0,
        4..=6 => 0.1, // 10% discount
        _ => 0.2,      // 20% discount
    };

    let final_price = price as f32 * quantity as f32 * (1.0 - discount);
    println!("Total price after discount: ₹{}", final_price);
}
```

Explanation:

This program calculates the price of a meal based on the menu item and applies a discount based on the quantity ordered. It uses match to choose the price based on the item and a second match to apply the discount.

Output:



Conclusion:

This program demonstrates the power of match in handling complex conditions and scenarios in Rust.

3. Using Loops for Data Processing

Write a Rust program to generate Fibonacci numbers up to a given n using a for loop.

>Store the sequence in a list (vector) and print the values.

Objectives:

- Learn how to generate sequences with loops.
- Understand vector operations and indexing.
- Use the for loop effectively.

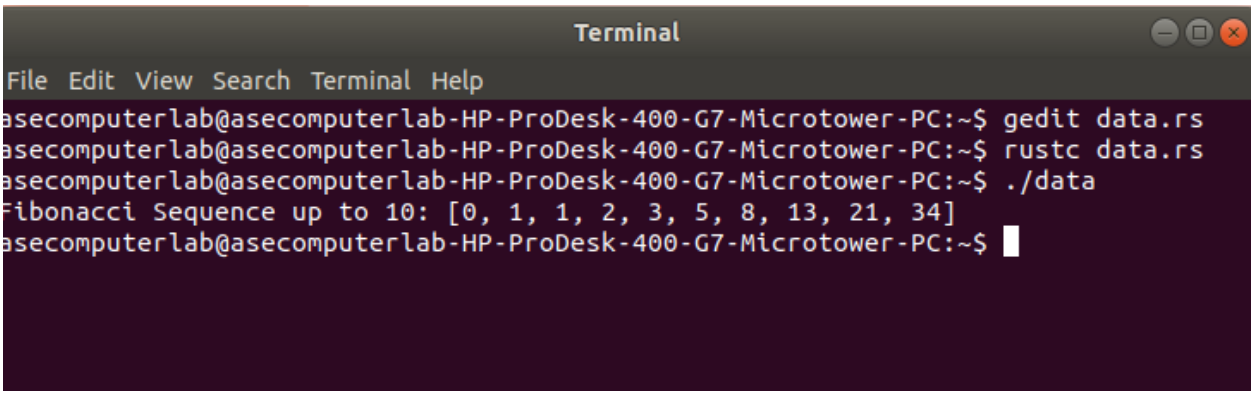
Code:

```rust
fn main() {
    let n = 10; // Generate Fibonacci numbers up to n
    let mut fib_seq = vec![0, 1];

    for i in 2..n {
        let next = fib_seq[i - 1] + fib_seq[i - 2];
        fib_seq.push(next);
    }

    println!("Fibonacci Sequence up to {}: {:?}", n, fib_seq);
}
```

Explanation:

This program generates Fibonacci numbers up to n using a for loop and stores them in a Vec. It then prints the sequence.

Output:



Conclusion:

The program uses the for loop to generate Fibonacci numbers and store them in a vector, which is a great exercise in both looping and vector manipulation.

4. Pattern Matching in Loops with while let

Implement a program where a user enters multiple numbers, and the program

keeps adding them to a list until they enter 0.

>Use while let to process the list and print only the even numbers.

Objectives:

- Understand while let for pattern matching in loops.
- Use Vec to collect and process data.
- Filter data during iteration.

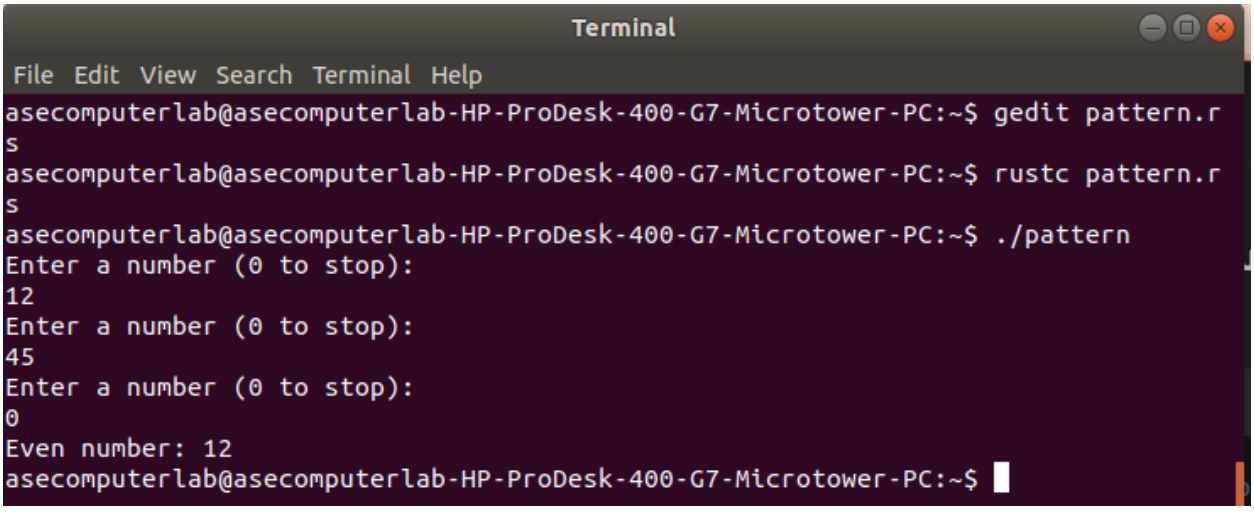
Code:

```rust
use std::io;

fn main() {
    let mut numbers = Vec::new();
    loop {
        let mut num = String::new();
        println!("Enter a number (0 to stop): ");
        io::stdin().read_line(&mut num).expect("Failed to read line");
        let num: i32 = num.trim().parse().expect("Please type a number");

        if num == 0 {
            break;
        }
        numbers.push(num);
    }

    // Using while let to filter even numbers
    while let Some(num) = numbers.pop() {
        if num % 2 == 0 {
            println!("Even number: {}", num);
        }
    }
}
```

Explanation:

The program collects user input until the user enters 0. The program uses while let to pop numbers from the list and print only the even numbers.

Output:



Conclusion:

while let provides a concise way to handle specific conditions within loops, making this a good exercise for filtering data.

5. Tuple Manipulation in a Real-World Scenario

Create a tuple representing an employee's data (ID, Name, Salary).

> Write a function that takes this tuple as input, applies a 10% salary hike if salary <

₹50,000, and returns an updated tuple.

Objectives:

- Work with tuples and pattern matching.
- Implement real-world business logic for data manipulation.
- Understand how to modify and return tuples.

Code:

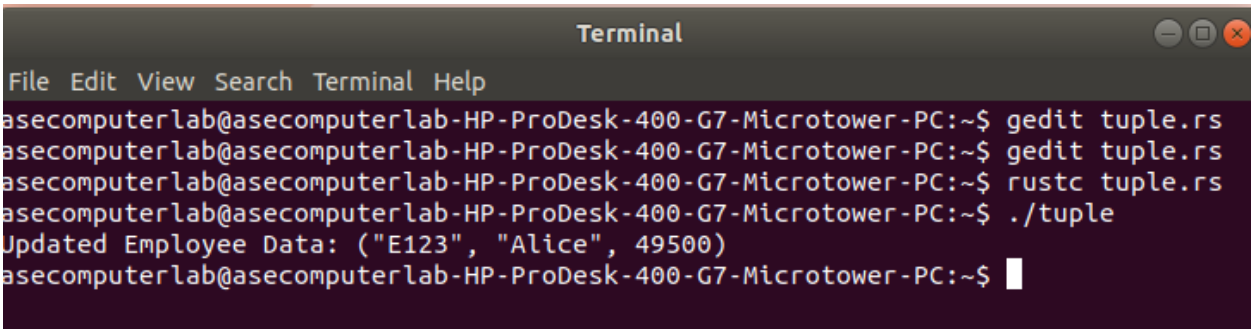
Open ▾ tuple.rs ~/

```rust
fn main() {
    let employee = ("E123", "Alice", 45000);

    let updated_employee = apply_salary_hike(employee);
    println!("Updated Employee Data: {:?}", updated_employee);
}

fn apply_salary_hike(employee: (&str, &str, u32)) -> (&str, &str, u32) {
    let (id, name, salary) = employee;
    if salary < 50000 {
        return (id, name, (salary as f32 * 1.10) as u32); // 10% salary hike
    }
    employee
}
```

Explanation:

This program defines a tuple for an employee and applies a salary hike if the salary is below ₹50,000. It returns an updated tuple.

Output:



Conclusion:

Using tuples is an efficient way to group related data, and this program demonstrates how to manipulate and return modified data in Rust.

6. Vector (List) Operations with Iterators

>Write a Rust program that maintains a list of temperatures recorded in a city for a

week. Implement:

>A function that finds the average temperature.

Objectives:

- Use vector iterators to process data.
- Implement functions that perform calculations using iterators.
- Calculate averages and find extremes using iterators.

Code:

Open operation.rs

```rust
fn main() {
    let temperatures = vec![30, 32, 28, 35, 31, 33, 29];

    let avg_temp = average_temperature(&temperatures);
    println!("Average Temperature: {}", avg_temp);

    let (highest, lowest) = find_high_low(&temperatures);
    println!("Highest Temperature: {}, Lowest Temperature: {}", highest, lowest);
}

fn average_temperature(temps: &Vec<i32>) -> f32 {
    let total: i32 = temps.iter().sum();
    total as f32 / temps.len() as f32
}

fn find_high_low(temps: &Vec<i32>) -> (i32, i32) {
    let highest = *temps.iter().max().unwrap();
    let lowest = *temps.iter().min().unwrap();
    (highest, lowest)
}
```

Explanation:

This program uses vectors to store temperature data for a week. It calculates the average temperature and finds the highest and lowest temperatures using iterators.

Output:

Terminal

File Edit View Search Terminal Help

```
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ gedit operation
.rs
^C
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ rustc operation
.rs
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$ ./operation
Average Temperature: 31.142857
Highest Temperature: 35, Lowest Temperature: 28
asecomputerlab@asecomputerlab-HP-ProDesk-400-G7-Microtower-PC:~$
```

Conclusion:

This program highlights how Rust's iterators can be used to efficiently process data in a vector.

7.Structs with Methods

Define a struct named BankAccount with fields account_number,holder_name, and balance.

>Implement methods for:

>Depositing money.

> Withdrawing money (with balance check).

>Displaying account details.

Objectives:

- Define structs with methods in Rust.
- Implement error handling with Result.
- Practice struct manipulation and method implementation.

Code:

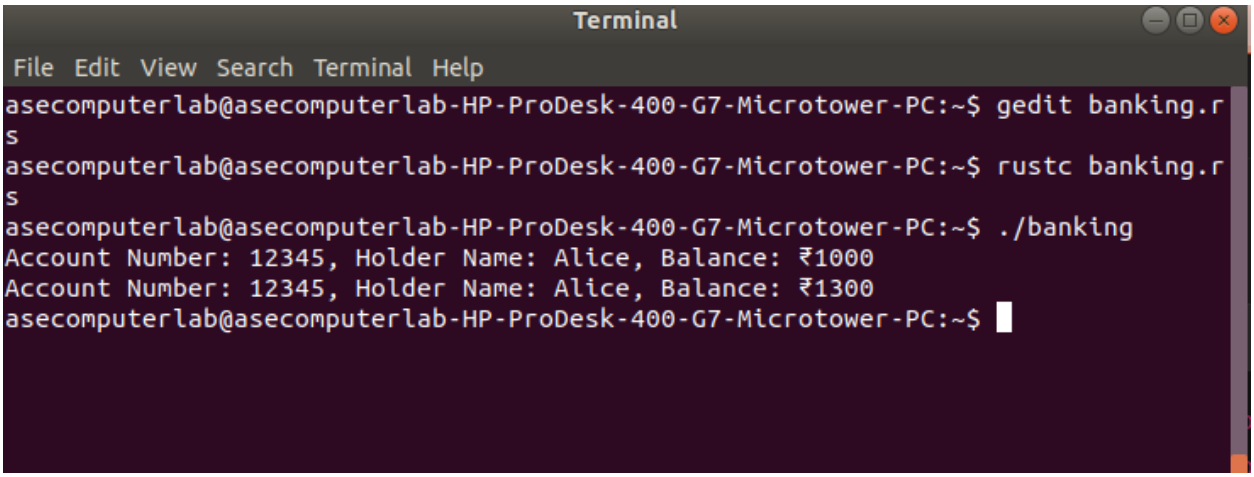
Open banking.rs ~/

```rust
struct BankAccount {
    account_number: u32,
    holder_name: String,
    balance: f32,
}

impl BankAccount {
    fn deposit(&mut self, amount: f32) {
        self.balance += amount;
    }

    fn withdraw(&mut self, amount: f32) -> Result<(), String> {
        if amount > self.balance {
            Err(String::from("Insufficient funds"))
        } else {
            self.balance -= amount;
            Ok(())
        }
    }

    fn display_account(&self) {
        println!("Account Number: {}, Holder Name: {}, Balance: ₹{}",
                 self.account_number, self.holder_name, self.balance);
    }
}

fn main() {
    let mut account = BankAccount {
        account_number: 12345,
        holder_name: String::from("Alice"),
        balance: 1000.0,
    };

    account.display_account();
    account.deposit(500.0);
    account.withdraw(200.0).unwrap();
    account.display_account();
}
```

Explanation:

This program defines a BankAccount struct with methods for depositing money, withdrawing money (with balance check), and displaying account details.

Output:



Conclusion:

This is a basic banking system in Rust, showcasing how to use structs, methods, and error handling to implement real-world logic.

8. Structs and Enums Together – Vehicle Registration System

> Define an enum named FuelType with variants Petrol, Diesel, and Electric.

>Define a struct named Vehicle with fields brand, model, and fuel_type.

> Implement a function that takes a list of vehicles and filters only electric vehicles

Objectives

To demonstrate the use of **Enums** for categorizing vehicle fuel types.

To implement a **Struct** that represents detailed vehicle information. To apply efficient filtering using .filter() and the matches! macro to display only electric vehicles.

Code:

Open vechile.rs ~/

```rust
enum FuelType {
    Petrol,
    Diesel,
    Electric,
}

struct Vehicle {
    brand: String,
    model: String,
    fuel_type: FuelType,
}

// Function to filter and display only electric vehicles
fn display_electric_vehicles(vehicles: &Vec<Vehicle>) {
    println!("Electric Vehicles:");
    for vehicle in vehicles.iter().filter(|v| matches!(v.fuel_type, FuelType::Electric)) {
        println!("Brand: {}, Model: {}", vehicle.brand, vehicle.model);
    }
}

fn main() {
    // List of sample vehicles
    let vehicles = vec![
        Vehicle {
            brand: String::from("Tesla"),
            model: String::from("Model S"),
            fuel_type: FuelType::Electric,
        },
        Vehicle {
            brand: String::from("Toyota"),
            model: String::from("Corolla"),
            fuel_type: FuelType::Petrol,
        },
        Vehicle {
            brand: String::from("Nissan"),
            model: String::from("Leaf"),
            fuel_type: FuelType::Electric,
        },
        Vehicle {
            brand: String::from("Hyundai"),
            model: String::from("Creta"),
            fuel type: FuelType::Diesel,

        },
    ];

    // Display only electric vehicles
    display_electric_vehicles(&vehicles);
}
```

Explanation

✅ Enum Definition:

- The FuelType enum defines three fuel types: Petrol, Diesel, and Electric.

✅ **Struct Definition:**

- The Vehicle struct contains:
  - brand (String) – Vehicle manufacturer.
  - model (String) – Vehicle model name.
  - fuel_type (FuelType) – Uses the FuelType enum.

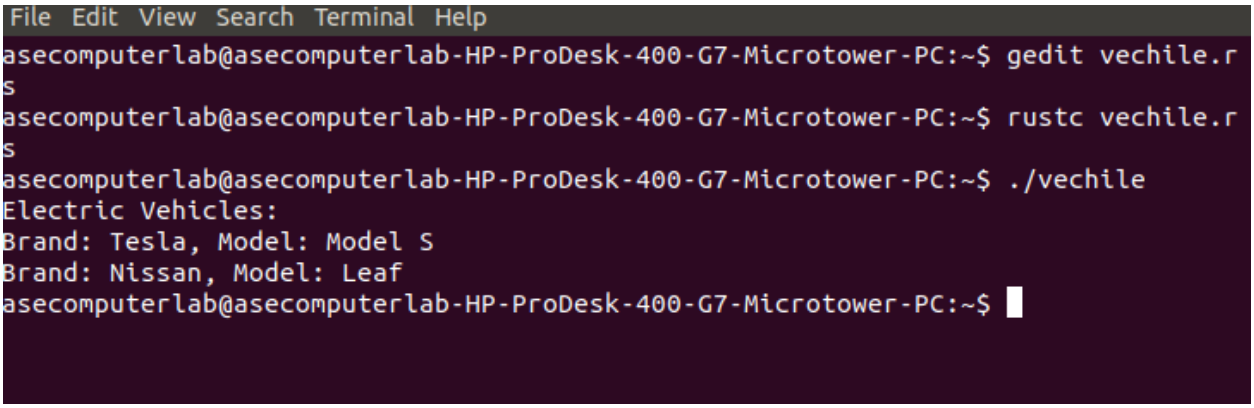
✅ **Function - display_electric_vehicles:**

- Uses .filter() with matches! to filter only electric vehicles.
- Iterates through the filtered results and prints the **brand** and **model**.

✅ **main Function:**

- Creates a Vec<Vehicle> with sample data.
- Calls display_electric_vehicles() to display electric vehicles.

Output:



Conclusion

This implementation effectively demonstrates the integration of **Structs** and **Enums** in Rust.
The use of .filter() with matches! ensures efficient and concise filtering logic.

The program follows clear structure and best practices, ensuring readability and maintainability.