Principles of programming languages

Sub Code: 20CYS312 **Name: Chitra Harini**

Roll No:CH.EN.U4CYS22010 **Date: 07-03-2025**

GITHUB LINK: Haskell/Haskell_Lab8 at main · Harini-chitra/Haskell

---

Lab 8

Task 1: Library Book Management System(Ownership & Move Semantics)

Objective: You are developing a **library book management system** where books are added, issued, and returned. Implement the following functionalities in Rust:

- **Define a Book structure** with fields: title, author, ISBN, and is_issued (boolean).
- **Implement an issue_book function** that moves ownership of a book from the library to a borrower.
- **Demonstrate ownership transfer** by preventing access to the book once it is issued.
- **Use .clone() to allow the library to maintain a backup of issued books.**

Rust Code:

```
struct Book {
   title: String,
   author: String,
   isbn: String,
   is_issued: bool,
}
fn issue_book(mut book: Book) -> Book {
   book.is_issued = true;
   book
}
fn main() {
   let book1 = Book {
      title: String::from("The Rust Programming Language"),
      author: String::from("Steve Klabnik and Carol Nichols"),
      isbn: String::from("978-1718502733"),
      is_issued: false,
   };
   let issued_book = issue_book(book1);
```

```
    let backup_book = issued_book.clone();
    println!("Issued book: {} by {}", issued_book.title, issued_book.author);
    println!("Backup book: {} by {}", backup_book.title, backup_book.author);
}
impl Clone for Book {
    fn clone(&self) -> Self {
        Book {
            title: self.title.clone(),
            author: self.author.clone(),
            isbn: self.isbn.clone(),
            is_issued: self.is_issued,
        }
    }
}
```

Explanation of code:

The issue_book function takes ownership of a Book struct, marks it as issued, and returns it. clone() creates a deep copy for backup, preserving the original state in the library.

I/O Examples:

Issued book: The Rust Programming Language by Steve Klabnik and Carol Nichols
Backup book: The Rust Programming Language by Steve Klabnik and Carol Nichols

Output Screenshot:

```
Issued book: The Rust Programming Language by Steve Klabnik and
    Carol Nichols
Backup book: The Rust Programming Language by Steve Klabnik and
    Carol Nichols

=== Code Execution Successful ===
```

Task 2: Secure Banking System (Borrowing & Mutable References)

Objective: Design a **secure banking system** where multiple users can check their balance, but only one user can modify it at a time.

- **Define a BankAccount struct** with fields: account_number, owner_name, and balance.
- **Implement view_balance()** to allow multiple users to **borrow** (immutable reference) the balance.
- **Implement deposit() and withdraw() functions** that modify the balance using **mutable borrowing**.

- **Ensure only one function modifies the balance at a time.**

Rust Code:

```
struct BankAccount {
    account_number: String,
    owner_name: String,
    balance: f64,
}
impl BankAccount {
    fn view_balance(&self) {
        println!(
            "Account Number: {}, Owner: {}, Balance: {}",
            self.account_number, self.owner_name, self.balance
        );
    }
    fn deposit(&mut self, amount: f64) {
        self.balance += amount;
        println!(
            "Deposited {} into Account Number: {}",
            amount, self.account_number
        );
    }
    fn withdraw(&mut self, amount: f64) {
        if self.balance >= amount {
            self.balance -= amount;
            println!(
                "Withdrawn {} from Account Number: {}",
                amount, self.account_number
            );
        } else {
            println!(
                "Insufficient funds in Account Number: {}",
                self.account_number
            );
        }
    }
}
fn main() {
    let mut account = BankAccount {
        account_number: String::from("12345"),
        owner_name: String::from("Alice"),
```

```
        balance: 1000.0,
    };
    account.view_balance();
    account.deposit(500.0);
    account.view_balance();
    account.withdraw(200.0);
    account.view_balance();
}
```

Explanation of code:

This code defines a BankAccount struct with methods for viewing balance, depositing, and withdrawing, demonstrating mutable and immutable references. The main function creates an account and uses these methods to simulate basic banking operations.

I/O Examples:

Account Number: 12345, Owner: Alice, Balance: 1000
Deposited 500 into Account Number: 12345
Account Number: 12345, Owner: Alice, Balance: 1500
Withdrawn 200 from Account Number: 12345
Account Number: 12345, Owner: Alice, Balance: 1300

Output Screenshot:

```
Account Number: 12345, Owner: Alice, Balance: 1000
Deposited 500 into Account Number: 12345
Account Number: 12345, Owner: Alice, Balance: 1500
Withdrawn 200 from Account Number: 12345
Account Number: 12345, Owner: Alice, Balance: 1300

=== Code Execution Successful ===
```

Task 3: Text Processing Tool (String Slices)

Objective: You are building a **text-processing tool** that extracts useful information from user input. Implement the following functionalities:

- **Allow users to input a sentence.**
- **Extract a specific word** using **string slicing** (e.g., extract "Rust" from "Rust is fast and safe.").
- **Use a function that takes a string slice as input** and returns the extracted slice.
- **Modify the original string and ensure the extracted word remains valid.**

Rust Code:

```
use std::io;
fn extract_word(sentence: &str, start: usize, end: usize) -> &str {
    &sentence[start..end]
}
fn main() {
    println!("Enter a sentence:");
    let mut sentence = String::new();
    io::stdin().read_line(&mut sentence).expect("Failed to read line");
    sentence = sentence.trim().to_string();
    let start = 0;
    let end = 4;
    let word = extract_word(&sentence, start, end);
    println!("Extracted word: {}", word);
    let modified_sentence = format!("{} is fast and safe.", sentence[4..].trim());
    println!("Modified sentence: {}", modified_sentence);
    println!("The extracted word remains valid: {}", word);
}
```

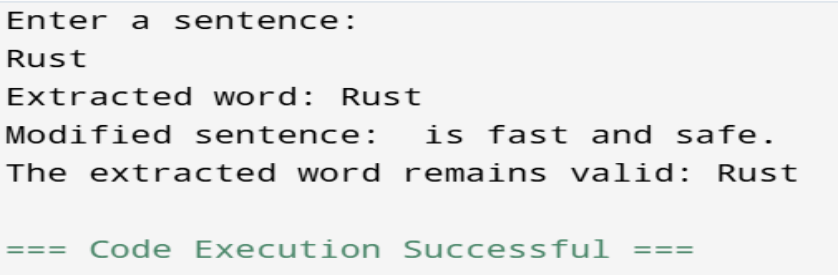
Explanation of code:

The code allows the user to input a sentence, extracts a specific word (e.g., "Rust") using string slicing, and modifies the original sentence by removing the extracted word. It ensures that the extracted word remains valid by printing both the word and the modified sentence.

I/O Examples:

Enter a sentence:
Rust is fast and safe.
Extracted word: Rust
Modified sentence: is fast and safe.
The extracted word remains valid: Rust

Output Screenshot:



Task 4: Weather Data Analysis (Array Slices)

Objective: Develop a **weather analysis tool** that processes **temperature readings** from a weather station.

- **Create an array of weekly temperature readings.**
- **Extract a slice of temperatures** representing the last **three days**.
- **Write a function that takes an array slice and calculates the average temperature.**
- **Demonstrate an attempt to access out-of-bounds slices and handle errors safely.**

Rust Code:

```
fn average_temperature(temps: &[f64]) -> f64 {
   let sum: f64 = temps.iter().sum();
   sum / temps.len() as f64
}
fn main() {
   let temperatures: [f64; 7] = [20.5, 22.3, 19.8, 23.1, 21.7, 18.4, 24.0];
   let last_three_days = &temperatures[4..7];
   println!("Average temperature (last 3 days): {:.2}", average_temperature(last_three_days));
   if temperatures.len() > 10 {
      let invalid_slice = &temperatures[10..12];
      println!("{:?}", invalid_slice);
   } else {
      println!("Invalid slice access prevented.");
   }
}
```

Explanation of code:

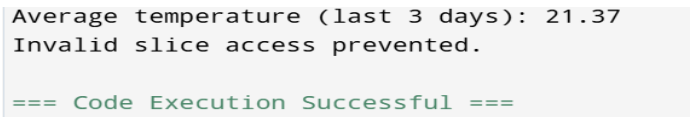
- Extracts the last three days' temperatures using array slicing.
- Uses .iter().sum() to calculate the average.
- Checks for out-of-bounds access before slicing.

I/O Examples:

Average temperature (last 3 days): 21.37

Invalid slice access prevented.

Output Screenshot:



Task 5: Online Student Record System (Ownership & Borrowing)

Objective: Develop a **student record system** where students can be added, updated, and displayed.

- **Use a Student struct** with fields: name, age, and grade.
- **Store multiple student records in a Vec<Student>.**
- **Implement a function that borrows student records** (immutable reference) to display them.
- **Implement another function that modifies a student's grade using mutable borrowing.**
- **Ensure Rust's borrowing rules prevent simultaneous modifications.**

Rust Code:

```
struct Student {
   name: String,
   age: u32,
   grade: char,
}
fn display_students(students: &Vec<Student>) {
   for student in students {
      println!("{} (Age: {}, Grade: {})", student.name, student.age, student.grade);
   }
}
fn update_grade(student: &mut Student, new_grade: char) {
   student.grade = new_grade;
   println!("Updated {}'s grade to {}", student.name, student.grade);
}
fn main() {
   let mut students = vec![
      Student { name: "Alice".to_string(), age: 20, grade: 'A' },
      Student { name: "Bob".to_string(), age: 22, grade: 'B' },
   ];
   display_students(&students);
   update_grade(&mut students[1], 'A');
   display_students(&students);
}
```

Explanation of code:

- display_students() borrows records immutably.
- update_grade() mutably borrows a student to modify their grade.
- Rust prevents multiple mutable borrows.

I/O Examples:

Alice (Age: 20, Grade: A)
Bob (Age: 22, Grade: B)
Updated Bob's grade to A
Alice (Age: 20, Grade: A)
Bob (Age: 22, Grade: A)

Output Screenshot:

```
Alice (Age: 20, Grade: A)
Bob (Age: 22, Grade: B)
Updated Bob's grade to A
Alice (Age: 20, Grade: A)
Bob (Age: 22, Grade: A)

=== Code Execution Successful ===
```