System Programing Assignment #1

2022320039 컴퓨터학과 정진욱

Problem 1. Minimum Grade

Code Explanation:

1. Input:

The above code is processing input.

- Receives all input at once and splits based on space characters.
- Call next whenever necessary and retrieve the value through unwrap().
- Additional parsing is performed when processing with numbers is required.

```
let mut input: String = String::new();
stdin().read_to_string(&mut input).unwrap();
let mut input = input.split_ascii_whitespace();

 ... input.next().unwrap()
 ... input.next().unwrap().parse::<usize>().unwrap();
```

2. Letter to Grade Mapping (`letter_to_grade`):

This function converts a letter grade (e.g., A+, B0) into its corresponding GPA score based on the conversion table provided in the problem statement.

3. Minimum Letter Grade Calculation (`min_letter`):

This function takes a GPA value and returns the corresponding letter grade based on the thresholds defined in the problem.

4. Main Function (`main`):

This function performs the core logic of the program:

- It first reads the input and extracts the number of subjects `n` and the minimum GPA required `x`.
- To satisfy “If the GPA for this semester "exceeds" the minimum GPA criteria, it is considered to have met the criteria.” condition, I just add 0.01 to `x`.
- It then iterates through all subjects except one, converting their letter grades to GPA scores using `letter_to_grade()`, and adds their contribution to the total GPA sum.
- After processing all but the last subject, the program calculates the minimum GPA required for the remaining subject using the equation

  $$g = \frac{(x \times \text{total credits}) - \text{sum of GPA}}{\text{credits of remaining subject}}$$

  Finally, it prints the minimum letter grade required using the `min_letter()` function.

Problem 2. Stack Calculator

Code Explanation:

1. Constants and Setup:

- **LIMIT**: This defines the maximum absolute value allowed for the stack operations. Any result exceeding this value will result in an error.

2. Command Implementations:

Each command (e.g., `POP`, `INV`, `ADD`, etc.) is implemented as a function that takes the stack as input and performs the operation. The functions return a boolean indicating whether the operation was successful or not. If an operation fails, it returns `false`.

3. Main Logic (`main`):

This is the core part of the program that reads the input, processes each command, and handles multiple executions for each calculator description.

```
fn main() {
    let mut input: String = String::new();
    stdin().read_to_string(&mut input).unwrap(); // Read entire input as a string.
    let mut input = input.lines(); // Split input into lines.

    loop {
        let mut program: Vec<&str> = Vec::new();

        // Read program instructions until "END" is encountered.
        loop {
            match input.next().unwrap() {
                "END" => break,
                "QUIT" => return, // Stop if "QUIT" is encountered.
                "" => continue, // Skip empty lines.
                op => program.push(op), // Add operations to the program.
            }
        }

                // Number of executions.
        let n: i64 = input.next().unwrap().parse::<i64>().unwrap();

        // Execute the program `n` times.
        for _ in 0..n {
            let mut stack: Vec<i64> = Vec::new(); // Initialize stack.
            // First input value.
            let v : i64 = input.next().unwrap().parse::<i64>().unwrap();
            stack.push(v);

            // Process each operation in the program.
            for op in &program {
                if op.starts_with("NUM") {num(op[4..].parse::<i64>().unwrap(), &mut st
ack);}
                else if !(match *op {
                    "POP" => pop(&mut stack),
                    "INV" => inv(&mut stack),
                    "DUP" => dup(&mut stack),
                    "SWP" => swp(&mut stack),
                    "ADD" => add(&mut stack),
                    "SUB" => sub(&mut stack),
                    "MUL" => mul(&mut stack),
                    "DIV" => div(&mut stack),
                    "MOD" => modulo(&mut stack),
                    _ => false,
                }) {stack.clear(); break;} // Clear stack if error occurs.
            }

            // Output result or "ERROR" if stack is not valid.
            if stack.len() == 1 {println!("{}", stack.pop().unwrap())}
            else {println!("ERROR");}
        }
        println!(); // Print blank line after each calculator description.
    }
}
```

- **Program Execution**: The input is read, and the program instructions are stored until "END" is encountered. Then, for each input value (`v`), the program is executed on the stack.
- **Error Handling**: If any operation fails (e.g., division by zero, insufficient elements in the stack), the program clears the stack and moves to the next input value, printing "ERROR".
- **Output**: If the program successfully completes and exactly one value remains in the stack, that value is printed. Otherwise, "ERROR" is printed.

Problem 3. Omok

Problem Analysis:

The game of **Omok (Gomoku)** is played on a 19×19 board, where players alternately place black and white stones. The game is won by the player who places exactly five consecutive stones of their color in any direction (horizontal, vertical, or diagonal). However, if more than five consecutive stones are placed, it does not count as a win. The program must determine if there is a winner, and if so, which color wins.

Code Explanation:

1. Position Validation (`valid_pos`):

This function checks if a given position is valid on the 19×19 board (i.e., within the boundaries).

```
fn valid_pos(x: i16, y: i16) -> bool {
    (0..19).contains(&x) && (0..19).contains(&y)
}
```

- It returns `true` if the given coordinates `(x, y)` are within the valid range of the board (0 to 18), and `false` otherwise.

2. Main Function (`main`):

The core logic for reading the input and checking for five consecutive stones is implemented in the `main` function.

```
fn main() {
    let mut input = String::new();
    stdin().read_to_string(&mut input).unwrap();  // Read the entire input as a strin
g.
    let mut input = input.split_ascii_whitespace().flat_map(str::parse::<i32>);
    // Split input by whitespace and convert to integers.

    let mut board = [[0; 19]; 19];  // Initialize the 19x19 board with zeros.

    // Populate the board with the input values.
    for i in 0..19 {
        for j in 0..19 {
            board[i][j] = input.next().unwrap();
        }
    }

    // Directions for horizontal, vertical, right diagonal, and left diagonal checks.
    let directions:[(i16, i16); 4] = [(0, 1), (1, 0), (1, 1), (-1, 1)];

    // Iterate over the entire board.
    for i in 0..19 {
        for j in 0..19 {
            if board[i][j] != 0 {  // If the position is not empty.
                for &(dx, dy) in &directions {  // Check in all four directions.
                    let mut five: bool = true;

                    // Check if there are exactly 5 consecutive stones in the current
direction.
                    for k in 1..5 {
                        let (nx, ny) = (i as i16 + k * dx, j as i16 + k * dy);
                        // Calculate next position.
                        // If the next position is invalid or the stone is not the sam
e, break the loop.
                        if  !valid_pos(nx, ny) || board[i][j] != board[nx as usize][ny
as usize] {
                            five = false;
                            break;
                        }
                    }

                    // If 5 consecutive stones are found.
                    if five {
                        let (px, py) = (i as i16 - dx, j as i16 - dy);
                        // Check if there's a preceding stone (to avoid 6 in a row).
                        if valid_pos(px, py) && board[i][j] == board[px as usize][py a
s usize]
                        { continue; }

                        let (nx, ny) = (i as i16 + 5 * dx, j as i16 + 5 * dy);
                        // Check if there's a stone after the 5th one.
                        if valid_pos(nx, ny) && board[i][j] == board[nx as usize][ny a
s usize]
                        { continue; }

                        // Print the winner (1 for black, 2 for white) and the positio
n of the winning stone.
                        println!("{}", board[i][j]);
                        println!("{} {}", i + 1, j + 1);
                        return;
                    }
                }
            }
        }
    }

    // If no winner is found, print 0.
    println!("0");
}
```

Key Steps in the Code:

1. **Input Parsing**: The board is read as a single block of input, and the 19×19 grid is populated.
2. **Direction Setup**: There are four directions to check for consecutive stones:
    - Right `(0, 1)`,
    - Down `(1, 0)`,
    - Down-right diagonal `(1, 1)`,
    - Up-right diagonal `(-1, 1)`.
3. **Checking for Five Stones**: For each stone on the board, the program checks if there are exactly five consecutive stones in any of the four directions.
    - When five consecutive stones are found, the program checks both the next position (i.e., the sixth stone) and the position before the starting point (i.e., moving in the opposite direction) to ensure that no additional stones extend the sequence. This ensures that exactly five consecutive stones, as having six or more consecutive stones does not count as a valid win.
    - If a valid winning sequence is found, the program prints the winning color and the coordinates of the leftmost or topmost stone in the sequence.
4. **Output**: If no winner is found after checking the entire board, the program prints `0`.