Result Exercise

1. Basic File Reader

Rust

```rust
use std::fs;

// Create a function that reads a file and returns:
// - Ok(content) if file exists and is readable
// - Err with a descriptive message otherwise
fn read_file_contents(path: &str) -> Result<String, String> {
    todo!()
}

// Example test cases:
// let exists = read_file_contents("Cargo.toml");
// let missing = read_file_contents("nonexistent.txt");

```

2. Number Parser

Rust

```rust
// Write a function that parses a string into a positive even integer
// Return Ok(number) if valid, Err with explanation if invalid
fn parse_even_number(s: &str) -> Result<u32, String> {
    todo!()
}

// Example test cases:
// assert_eq!(parse_even_number("42"), Ok(42));
// assert_eq!(parse_even_number("43"), Err("Number must be even"));
// assert_eq!(parse_even_number("abc"), Err("Invalid number format"));

```

3. User Validation

Rust

```rust
struct User {
    username: String,
    age: u8,
}

// Create a function that validates user data:
// - Username must be 3-20 alphanumeric chars
// - Age must be 13-120
// Return Ok(User) if valid, Err with all validation errors if invalid
fn validate_user(username: String, age: u8) -> Result<User, Vec<String>> {
    todo!()
}

// Example test case:
// let valid = validate_user("john_doe".to_string(), 25);
// let invalid = validate_user("x".to_string(), 250);

```

4. Result Combinators

Rust

```rust
// Implement a function that:
// 1. Takes two Results (Result<i32>, Result<i32>)
// 2. If both are Ok, returns Ok(sum)
// 3. If either is Err, returns first Err encountered
// 4. If both are Err, returns first Err
fn sum_results(a: Result<i32, String>, b: Result<i32, String>) -> Result<i32, String> {
    todo!()
}

// Example test cases:
// assert_eq!(sum_results(Ok(1), Ok(2)), Ok(3));
// assert_eq!(sum_results(Ok(1), Err("error".to_string())), Err("error".to_string()));

```

5. Error Type Conversion

```rust
use std::num::ParseIntError;

// Create a function that:
// 1. Parses two strings to i32
// 2. Divides them (a/b)
// 3. Handles all possible errors (parse, division)
// 4. Returns a custom error type
#[derive(Debug)]
enum MathError {
    ParseError(ParseIntError),
    DivisionByZero,
}

fn divide_strings(a: &str, b: &str) -> Result<f64, MathError> {
    todo!()
}

// Example test cases:
// assert_eq!(divide_strings("4", "2"), Ok(2.0));
// assert_eq!(divide_strings("a", "2"), Err(MathError::ParseError(...)));
// assert_eq!(divide_strings("4", "0"), Err(MathError::DivisionByZero));

```