1. **Start by explain the code that was given to me - Claude**

(require data/maybe)

**This imports Racket's Maybe module, which handles computations that might fail.**

(define (safe-div x y) ; num num -> maybe num

(if (= y 0)

nothing

(just (/ x y))))

**This is a division function that returns nothing if dividing by zero, otherwise returns the result wrapped in just.**

(define in-list? ; item list -> bool

(λ (x lst)

(not (false? (member x lst)))))

**A helper function to check if an item is in a list.**

(define (eval expr)

(cond

[(equal? (first expr) 'num) (just (second expr))]

[(in-list? (first expr) '(div add sub mult))

(let(

[x (eval (second expr))]

[y (eval (third expr))])

(if (or (nothing? x) (nothing? y))

nothing

(cond

[(equal? (first expr) 'div) (safe-div (from-just 1 x) (from-just 1 y))]

[(equal? (first expr) 'add) (just(+ (from-just 1 x) (from-just 1 y)))]

[(equal? (first expr) 'sub) (just (- (from-just 1 x) (from-just 1 y)))]

[else (just(\* (from-just 1 x) (from-just 1 y)))])))]

[else nothing]))

**This is the main evaluator function that:**

* **Handles numeric literals like (num 5)**
* **Evaluates arithmetic expressions with add, sub, mult, and div operations**
* **Returns nothing for unknown operations**
* **Each operation returns a result wrapped in just or nothing for errors**

1. **Please break down what it is asking me to do and make sure to give me a clear checklist of the items so I can keep track of it. (uploaded program doc) - GPT**

You will:

* **Modify an existing numeric expression evaluator** to use a Result type (success or failure) instead of Maybe (just or #<nothing>).
* **Add variable handling** using:  
  + define: for declaring variables.
  + assign: for assigning new values.
  + remove: for deleting variables.
* **Track a state** (a mapping of variable names to their values) throughout the evaluation.
* **Create a REPL loop** that:  
  + Prompts the user for input.
  + Evaluates the expression.
  + Prints the result and the current state.
  + Continues until the user chooses to quit.

### **📘 Features Your Interpreter Should Support**

* **Basic math operations**: e.g., (add (num 2) (num 3))
* **Variables**:  
  + Declare: (define a) or (define a expr)
  + Assign: (assign a expr)
  + Use in expressions: (id a)
  + Remove: (remove a)
* **State Management**:  
  + Represent the environment using an immutable structure like a list or a hash.
  + Pass the state into your evaluator and return the updated state.

1. **I have to switch from a result type instead of a maybe evaluator, the basic operations will be the same, need to create a state tracker and repl loop. - Claude**

#lang racket

;; CS 441 Program 3 - Expression evaluation with environment

;; This program implements an interpreter for evaluating expressions with variables

;; It converts the Maybe-based evaluator to an Either/Result-based one and adds state management

;; Define success and failure constructors and accessors for our Either type

(struct success (value) #:transparent)

(struct failure (message) #:transparent)

;; Helper functions similar to from-just but for our Either type

(define (from-success default result)

(if (success? result)

(success-value result)

default))

(define (from-failure default result)

(if (failure? result)

(failure-message result)

default))

;; Helper function to check if an item is in a list

(define in-list?

(λ (x lst)

(not (false? (member x lst)))))

;; Safe division that handles division by zero

(define (safe-div x y)

(if (= y 0)

(failure "Division by zero")

(success (/ x y))))

;; Variable name validation

(define (valid-id? id)

(and (symbol? id)

(regexp-match? #rx"^[a-zA-Z][a-zA-Z0-9\_-]\*$" (symbol->string id))))

;; Environment operations

(define (make-env) '())

(define (lookup-var id env)

(let ([pair (assoc id env)])

(if pair

(success (cdr pair))

(failure (format "Variable '~a' not defined" id)))))

(define (get-var-value id env)

(let ([result (lookup-var id env)])

(if (success? result)

(let ([value (success-value result)])

(if (eq? value 'undefined)

(failure (format "Variable '~a' is undefined" id))

(success value)))

result)))

(define (define-var id value env)

(if (assoc id env)

(failure (format "Cannot define '~a': already defined" id))

(success (cons (cons id value) env))))

(define (assign-var id value env)

(if (assoc id env)

(success (map (lambda (pair)

(if (eq? (car pair) id)

(cons id value)

pair))

env))

(failure (format "Cannot assign to '~a': not defined" id))))

(define (remove-var id env)

(if (assoc id env)

(success (filter (lambda (pair) (not (eq? (car pair) id))) env))

(failure (format "Cannot remove '~a': not defined" id))))

;; Main evaluation function with state

(define (eval expr env)

(cond

;; Number literal

[(equal? (first expr) 'num)

(success (cons (second expr) env))]

;; Variable reference

[(equal? (first expr) 'id)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (get-var-value var-name env)])

(if (success? result)

(success (cons (success-value result) env))

result))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable definition

[(equal? (first expr) 'define)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(cond

;; Define with no initial value

[(= (length expr) 2)

(let ([result (define-var var-name 'undefined env)])

(if (success? result)

(success (cons 'ok (success-value result)))

(failure (cons (failure-message result) env))))]

;; Define with initial value

[(= (length expr) 3)

(let\* ([init-result (eval (third expr) env)]

[init-success? (success? init-result)])

(if init-success?

(let\* ([value (car (success-value init-result))]

[new-env (cdr (success-value init-result))]

[def-result (define-var var-name value new-env)])

(if (success? def-result)

(success (cons 'ok (success-value def-result)))

(failure (cons (failure-message def-result) new-env))))

(failure (failure-message init-result))))]

[else (failure (cons "Invalid define syntax" env))])

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable assignment

[(equal? (first expr) 'assign)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let\* ([eval-result (eval (third expr) env)]

[eval-success? (success? eval-result)])

(if eval-success?

(let\* ([value (car (success-value eval-result))]

[new-env (cdr (success-value eval-result))]

[assign-result (assign-var var-name value new-env)])

(if (success? assign-result)

(success (cons 'ok (success-value assign-result)))

(failure (cons (failure-message assign-result) new-env))))

(failure (failure-message eval-result))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable removal

[(equal? (first expr) 'remove)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (remove-var var-name env)])

(if (success? result)

(success (cons 'ok (success-value result)))

(begin

(printf "Error: remove ~a: variable not defined, ignoring~n" var-name)

(success (cons 'ok env)))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Arithmetic operations

[(in-list? (first expr) '(div add sub mult))

(let\* ([x-result (eval (second expr) env)]

[x-success? (success? x-result)])

(if x-success?

(let\* ([x-value (car (success-value x-result))]

[mid-env (cdr (success-value x-result))]

[y-result (eval (third expr) mid-env)]

[y-success? (success? y-result)])

(if y-success?

(let\* ([y-value (car (success-value y-result))]

[final-env (cdr (success-value y-result))]

[op-result (cond

[(equal? (first expr) 'div) (safe-div x-value y-value)]

[(equal? (first expr) 'add) (success (+ x-value y-value))]

[(equal? (first expr) 'sub) (success (- x-value y-value))]

[else (success (\* x-value y-value))])])

(if (success? op-result)

(success (cons (success-value op-result) final-env))

(failure (cons (failure-message op-result) final-env))))

(failure (failure-message y-result))))

(failure (failure-message x-result))))]

;; Unknown operation

[else (failure (cons "Unknown operation" env))]))

;; REPL (Read-Eval-Print Loop)

(define (repl)

(let loop ([env (make-env)])

(printf "> ")

(let ([input (read)])

(cond

[(eq? input 'quit) (printf "Goodbye!~n") (void)]

[(eq? input 'env)

(printf "Current environment:~n")

(for-each (lambda (pair)

(printf " ~a: ~a~n" (car pair) (cdr pair)))

env)

(loop env)]

[else

(let ([result (eval input env)])

(if (success? result)

(let ([value (car (success-value result))]

[new-env (cdr (success-value result))])

(printf "Success: ~a~n" value)

(loop new-env))

(begin

(printf "Failure: ~a~n" (failure-message result))

(loop env))))]))))

;; Run the REPL

(repl)

1. **There was an error when I was trying to define a new variable what happened here, why can I not defined another variable - claude**

#lang racket

;; CS 441 Program 3 - Expression evaluation with environment

;; This program implements an interpreter for evaluating expressions with variables

;; It converts the Maybe-based evaluator to an Either/Result-based one and adds state management

;; Define success and failure constructors and accessors for our Either type

(struct success (value) #:transparent)

(struct failure (message) #:transparent)

;; Helper functions similar to from-just but for our Either type

(define (from-success default result)

(if (success? result)

(success-value result)

default))

(define (from-failure default result)

(if (failure? result)

(failure-message result)

default))

;; Helper function to check if an item is in a list

(define in-list?

(λ (x lst)

(not (false? (member x lst)))))

;; Safe division that handles division by zero

(define (safe-div x y)

(if (= y 0)

(failure "Division by zero")

(success (/ x y))))

;; Variable name validation

(define (valid-id? id)

(and (symbol? id)

(regexp-match? #rx"^[a-zA-Z][a-zA-Z0-9\_-]\*$" (symbol->string id))))

;; Environment operations

(define (make-env) '())

(define (lookup-var id env)

(let ([pair (assoc id env)])

(if pair

(success (cdr pair))

(failure (format "Variable '~a' not defined" id)))))

(define (get-var-value id env)

(let ([result (lookup-var id env)])

(if (success? result)

(let ([value (success-value result)])

(if (eq? value 'undefined)

(failure (format "Variable '~a' is undefined" id))

(success value)))

result)))

(define (define-var id value env)

(if (assoc id env)

(failure (format "Cannot define '~a': already defined" id))

(success (cons (cons id value) env))))

(define (assign-var id value env)

(if (assoc id env)

(success (map (lambda (pair)

(if (eq? (car pair) id)

(cons id value)

pair))

env))

(failure (format "Cannot assign to '~a': not defined" id))))

(define (remove-var id env)

(if (assoc id env)

(success (filter (lambda (pair) (not (eq? (car pair) id))) env))

(failure (format "Cannot remove '~a': not defined" id))))

;; Main evaluation function with state

(define (eval expr env)

(cond

;; Handle non-list expressions

[(not (list? expr))

(failure (cons (format "Invalid expression: ~a is not a list" expr) env))]

;; Handle empty expressions

[(empty? expr)

(failure (cons "Empty expression" env))]

;; Number literal

[(equal? (first expr) 'num)

(if (>= (length expr) 2)

(success (cons (second expr) env))

(failure (cons "Invalid number expression: missing value" env)))]

;; Variable reference

[(equal? (first expr) 'id)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (get-var-value var-name env)])

(if (success? result)

(success (cons (success-value result) env))

result))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable definition

[(equal? (first expr) 'define)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(cond

;; Define with no initial value

[(= (length expr) 2)

(let ([result (define-var var-name 'undefined env)])

(if (success? result)

(success (cons 'ok (success-value result)))

(failure (cons (failure-message result) env))))]

;; Define with initial value

[(= (length expr) 3)

(let\* ([init-result (eval (third expr) env)]

[init-success? (success? init-result)])

(if init-success?

(let\* ([value (car (success-value init-result))]

[new-env (cdr (success-value init-result))]

[def-result (define-var var-name value new-env)])

(if (success? def-result)

(success (cons 'ok (success-value def-result)))

(failure (cons (failure-message def-result) new-env))))

(failure (failure-message init-result))))]

[else (failure (cons "Invalid define syntax" env))])

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable assignment

[(equal? (first expr) 'assign)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let\* ([eval-result (eval (third expr) env)]

[eval-success? (success? eval-result)])

(if eval-success?

(let\* ([value (car (success-value eval-result))]

[new-env (cdr (success-value eval-result))]

[assign-result (assign-var var-name value new-env)])

(if (success? assign-result)

(success (cons 'ok (success-value assign-result)))

(failure (cons (failure-message assign-result) new-env))))

(failure (failure-message eval-result))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Variable removal

[(equal? (first expr) 'remove)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (remove-var var-name env)])

(if (success? result)

(success (cons 'ok (success-value result)))

(begin

(printf "Error: remove ~a: variable not defined, ignoring~n" var-name)

(success (cons 'ok env)))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))]

;; Arithmetic operations

[(in-list? (first expr) '(div add sub mult))

(let\* ([x-result (eval (second expr) env)]

[x-success? (success? x-result)])

(if x-success?

(let\* ([x-value (car (success-value x-result))]

[mid-env (cdr (success-value x-result))]

[y-result (eval (third expr) mid-env)]

[y-success? (success? y-result)])

(if y-success?

(let\* ([y-value (car (success-value y-result))]

[final-env (cdr (success-value y-result))]

[op-result (cond

[(equal? (first expr) 'div) (safe-div x-value y-value)]

[(equal? (first expr) 'add) (success (+ x-value y-value))]

[(equal? (first expr) 'sub) (success (- x-value y-value))]

[else (success (\* x-value y-value))])])

(if (success? op-result)

(success (cons (success-value op-result) final-env))

(failure (cons (failure-message op-result) final-env))))

(failure (failure-message y-result))))

(failure (failure-message x-result))))]

;; Unknown operation

[else (failure (cons "Unknown operation" env))]))

;; REPL (Read-Eval-Print Loop)

(define (repl)

(let loop ([env (make-env)])

(printf "> ")

(let ([input (read)])

(cond

[(eq? input 'quit) (printf "Goodbye!~n") (void)]

[(eq? input 'env)

(printf "Current environment:~n")

(for-each (lambda (pair)

(printf " ~a: ~a~n" (car pair) (cdr pair)))

env)

(loop env)]

[else

(let ([result (eval input env)])

(if (success? result)

(let ([value (car (success-value result))]

[new-env (cdr (success-value result))])

(printf "Success: ~a~n" value)

(loop new-env))

(begin

(printf "Failure: ~a~n" (failure-message result))

(loop env))))]))))

;; Run the REPL

(repl)

1. **Okay I need to update the Repl to make it look better and more readable - claude**

#lang racket

;; CS 441 Program 3 - Expression evaluation with environment

;; This program implements an interpreter for evaluating expressions with variables

;; It converts the Maybe-based evaluator to an Either/Result-based one and adds state management

;; Define success and failure constructors and accessors for our Either type

(struct success (value) #:transparent)

(struct failure (message) #:transparent)

;; Helper functions similar to from-just but for our Either type

(define (from-success default result)

(if (success? result)

(success-value result)

default))

(define (from-failure default result)

(if (failure? result)

(failure-message result)

default))

;; Helper function to check if an item is in a list

(define (in-list? x lst)

(not (false? (member x lst))))

;; Safe division that handles division by zero

(define (safe-div x y)

(if (= y 0)

(failure "Division by zero")

(success (/ x y))))

;; Variable name validation

(define (valid-id? id)

(and (symbol? id)

(regexp-match? #rx"^[a-zA-Z][a-zA-Z0-9\_-]\*$" (symbol->string id))))

;; Environment operations

(define (make-env) '())

(define (lookup-var id env)

(let ([pair (assoc id env)])

(if pair

(success (cdr pair))

(failure (format "Variable '~a' not defined" id)))))

(define (get-var-value id env)

(let ([result (lookup-var id env)])

(if (success? result)

(let ([value (success-value result)])

(if (eq? value 'undefined)

(failure (format "Variable '~a' is undefined" id))

(success value)))

result)))

(define (define-var id value env)

(if (assoc id env)

(failure (format "Cannot define '~a': already defined" id))

(success (cons (cons id value) env))))

(define (assign-var id value env)

(if (assoc id env)

(success (map (lambda (pair)

(if (eq? (car pair) id)

(cons id value)

pair))

env))

(failure (format "Cannot assign to '~a': not defined" id))))

(define (remove-var id env)

(if (assoc id env)

(success (filter (lambda (pair) (not (eq? (car pair) id))) env))

(failure (format "Cannot remove '~a': not defined" id))))

;; Check if an expression has the correct number of arguments

(define (check-arg-count expr expected-count)

(if (= (length expr) expected-count)

(success #t)

(failure (format "Operation '~a' expects ~a arguments, got ~a"

(first expr)

(- expected-count 1)

(- (length expr) 1)))))

;; Main evaluation function with state

(define (eval expr env)

(cond

;; Handle empty or non-list expressions

[(not (list? expr))

(failure (cons "Invalid expression: not a list" env))]

[(empty? expr)

(failure (cons "Invalid expression: empty list" env))]

;; Number literal

[(equal? (first expr) 'num)

(let ([arg-check (check-arg-count expr 2)])

(if (success? arg-check)

(if (number? (second expr))

(success (cons (second expr) env))

(failure (cons "Number expected after 'num'" env)))

(failure (cons (failure-message arg-check) env))))]

;; Variable reference

[(equal? (first expr) 'id)

(let ([arg-check (check-arg-count expr 2)])

(if (success? arg-check)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (get-var-value var-name env)])

(if (success? result)

(success (cons (success-value result) env))

(failure (cons (failure-message result) env))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))

(failure (cons (failure-message arg-check) env))))]

;; Variable definition

[(equal? (first expr) 'define)

(if (or (= (length expr) 2) (= (length expr) 3))

(let ([var-name (second expr)])

(if (valid-id? var-name)

(cond

;; Define with no initial value

[(= (length expr) 2)

(let ([result (define-var var-name 'undefined env)])

(if (success? result)

(success (cons (format "Defined variable '~a' as undefined" var-name)

(success-value result)))

(failure (cons (failure-message result) env))))]

;; Define with initial value

[(= (length expr) 3)

(let\* ([init-result (eval (third expr) env)]

[init-success? (success? init-result)])

(if init-success?

(let\* ([value (car (success-value init-result))]

[new-env (cdr (success-value init-result))]

[def-result (define-var var-name value new-env)])

(if (success? def-result)

(success (cons (format "Defined variable '~a' with value: ~a" var-name value)

(success-value def-result)))

(failure (cons (failure-message def-result) new-env))))

(failure (cons (failure-message init-result) env))))])

(failure (cons (format "Invalid variable name: ~a" var-name) env))))

(failure (cons "Invalid define syntax: expected (define id) or (define id expr)" env)))]

;; Variable assignment

[(equal? (first expr) 'assign)

(let ([arg-check (check-arg-count expr 3)])

(if (success? arg-check)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let\* ([eval-result (eval (third expr) env)]

[eval-success? (success? eval-result)])

(if eval-success?

(let\* ([value (car (success-value eval-result))]

[new-env (cdr (success-value eval-result))]

[assign-result (assign-var var-name value new-env)])

(if (success? assign-result)

(success (cons (format "Assigned variable '~a' to value: ~a" var-name value)

(success-value assign-result)))

(failure (cons (failure-message assign-result) new-env))))

(failure (cons (failure-message eval-result) env))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))

(failure (cons (failure-message arg-check) env))))]

;; Variable removal

[(equal? (first expr) 'remove)

(let ([arg-check (check-arg-count expr 2)])

(if (success? arg-check)

(let ([var-name (second expr)])

(if (valid-id? var-name)

(let ([result (remove-var var-name env)])

(if (success? result)

(success (cons (format "Removed variable '~a'" var-name)

(success-value result)))

(begin

(printf "Error: remove ~a: variable not defined, ignoring~n" var-name)

(success (cons (format "Attempted to remove undefined variable '~a'" var-name) env)))))

(failure (cons (format "Invalid variable name: ~a" var-name) env))))

(failure (cons (failure-message arg-check) env))))]

;; Arithmetic operations

[(in-list? (first expr) '(div add sub mult))

(let ([arg-check (check-arg-count expr 3)])

(if (success? arg-check)

(let\* ([x-result (eval (second expr) env)]

[x-success? (success? x-result)])

(if x-success?

(let\* ([x-value (car (success-value x-result))]

[mid-env (cdr (success-value x-result))]

[y-result (eval (third expr) mid-env)]

[y-success? (success? y-result)])

(if y-success?

(let\* ([y-value (car (success-value y-result))]

[final-env (cdr (success-value y-result))]

[op-result (cond

[(equal? (first expr) 'div)

(if (= y-value 0)

(failure "Division by zero")

(success (/ x-value y-value)))]

[(equal? (first expr) 'add) (success (+ x-value y-value))]

[(equal? (first expr) 'sub) (success (- x-value y-value))]

[else (success (\* x-value y-value))])])

(if (success? op-result)

(success (cons (success-value op-result) final-env))

(failure (cons (failure-message op-result) final-env))))

(failure (cons (failure-message y-result) env))))

(failure (cons (failure-message x-result) env))))

(failure (cons (failure-message arg-check) env))))]

;; Unknown operation

[else (failure (cons (format "Unknown operation: ~a" (first expr)) env))]))

;; Pretty print environment

(define (print-env env)

(printf "Current environment:~n")

(if (empty? env)

(printf " <empty>~n")

(for-each (lambda (pair)

(printf " ~a: ~a~n" (car pair) (cdr pair)))

(sort env (lambda (a b) (string<? (symbol->string (car a)) (symbol->string (car b))))))))

;; REPL (Read-Eval-Print Loop)

(define (repl)

(printf "Expression Evaluator REPL~n")

(printf "Enter expressions to evaluate, 'env' to show environment, or 'quit' to exit~n")

(let loop ([env (make-env)])

(printf "> ")

(flush-output) ;; Ensure prompt appears before reading input

(let ([input (read)])

(cond

[(eq? input 'quit) (printf "Goodbye!~n") (void)]

[(eq? input 'env)

(print-env env)

(loop env)]

[(eq? input 'help)

(printf "Available commands:~n")

(printf " quit - Exit the REPL~n")

(printf " env - Show current environment~n")

(printf " help - Show this help message~n")

(printf "Expressions:~n")

(printf " (num n) - Numeric literal~n")

(printf " (id name) - Variable reference~n")

(printf " (define id) - Define variable without value~n")

(printf " (define id expr) - Define variable with value~n")

(printf " (assign id expr) - Assign value to variable~n")

(printf " (remove id) - Remove variable~n")

(printf " (add expr expr) - Addition~n")

(printf " (sub expr expr) - Subtraction~n")

(printf " (mult expr expr) - Multiplication~n")

(printf " (div expr expr) - Division~n")

(loop env)]

[else

(let ([result (eval input env)])

(if (success? result)

(let ([value (car (success-value result))]

[new-env (cdr (success-value result))])

(printf "Result: ~a~n" value)

(loop new-env))

(begin

(printf "Error: ~a~n" (car (failure-message result)))

(loop env))))]))))

;; Run the REPL

(repl)

1. **I need you to review the code that you gave and let me know if it follows the details for the assignments - deepseek**

I've reviewed the code you provided, and it does implement most of the requirements mentioned in the specification. Here's an analysis of how it meets each requirement:

1. **Result Type Instead of Maybe**:
   * ✅ The code defines success and failure structs to represent the Result type
   * ✅ All evaluation functions return either a success or failure
2. **Variable Handling**:
   * ✅ define: Implemented with (define id) and (define id expr) syntax
   * ✅ assign: Implemented with (assign id expr) syntax
   * ✅ remove: Implemented with (remove id) syntax
   * ✅ Variables can be used in expressions with (id var-name)
3. **State Management**:
   * ✅ Uses an association list to track variable bindings
   * ✅ Environment is passed through evaluation and updated appropriately
   * ✅ The state is immutable - operations return new environments rather than mutating
4. **REPL Features**:
   * ✅ Has a working REPL that prompts for input
   * ✅ Evaluates expressions and prints results
   * ✅ Shows current environment with env command
   * ✅ Continues until user enters quit
   * ✅ Provides helpful error messages
5. **Additional Features**:
   * ✅ Includes input validation for variable names
   * ✅ Handles division by zero
   * ✅ Provides a help command
   * ✅ Maintains environment between commands

Overall, this implementation meets all the core requirements and includes some nice additional features like the help system and robust error handling. The use of immutable state and the Result type pattern is well implemented.