Program 3: Evaluator with State

Hashim Abdulla

Due Date: 5/4/2025

**LLM Prompts**

(OpenAI's GPT4o and Google's Gemini 2.5 Pro models were also used but both kept generating bad code like I mentioned in the program’s report so I abandoned those attempts and included only successful and relevant results from Claude's 3.7 Sonnet model to keep this documentation nice and concise. Plus, the code was incomplete anyway)

***Prompt 1: Initial Structure***

“I need you to modify this initial Racket code. Here's my starting code that uses Maybe:

1. #lang racket

2. (require data/maybe)

3.

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

5. (if (= y 0)

6. nothing

7. (just (/ x y))))

8.

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

10. (λ (x lst)

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

12.

13. (define (eval expr)

14. (cond

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

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

17. (let( [x (eval (second expr))]

18. [y (eval (third expr))])

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

20. nothing

21. (cond

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

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

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

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

26. ]

27. [else nothing])) ; unknown operation;not an int, not add, sub, mult, or div

28. ; so return nothing

29.

30. (eval '(num 5))

31. (eval '(add (num 5) (mult (num 2) (num 3))))

32. (eval '(sub (num 20) (div (add (mult (num 4) (num 5)) (num 10))(num 6))))

33. (eval '(div (num 5) (sub (num 5) (num 5)))) ;nothing (div by 0)

34.

According to my assignment, I need to:

1. Replace Maybe with Either type (success and failure instead of just and nothing)

2. Add state handling for variables

3. Implement variable operations (define, assign, remove)

4. Create a REPL

Can you first convert this to use Either instead of Maybe?”

Generated code:

1. #lang racket

2.

3. ;; Define our Either type implementation

4. (struct success (value) #:transparent)

5. (struct failure (message) #:transparent)

6.

7. ;; Utility functions for working with Either

8. (define (from-success default either)

9. (if (success? either)

10. (success-value either)

11. default))

12.

13. (define (from-failure default either)

14. (if (failure? either)

15. (failure-message either)

16. default))

17.

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

19. (define in-list?

20. (λ (x lst)

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

22.

23. ;; Safe division that returns Either

24. (define (safe-div x y) ; num num -> Either

25. (if (= y 0)

26. (failure "Division by zero")

27. (success (/ x y))))

28.

29. ;; Basic evaluator for arithmetic operations

30. (define (eval expr)

31. (cond

32. ;; Numeric literal

33. [(equal? (first expr) 'num)

34. (success (second expr))]

35.

36. ;; Arithmetic operations

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

38. (let ([x (eval (second expr))]

39. [y (eval (third expr))])

40. (if (or (failure? x) (failure? y))

41. (if (failure? x) x y) ; Pass failure up

42. (let ([x-val (success-value x)]

43. [y-val (success-value y)])

44. (case (first expr)

45. [(div) (safe-div x-val y-val)]

46. [(add) (success (+ x-val y-val))]

47. [(sub) (success (- x-val y-val))]

48. [(mult) (success (\* x-val y-val))]))))]

49.

50. ;; Unknown operation

51. [else (failure (format "Unknown operation: ~a" (first expr)))]

52. ))

53.

54. ;; Test cases

55. (display "Testing evaluator with Either:\n")

56. (display "(num 5): ")

57. (display (eval '(num 5)))

58. (newline)

59.

60. (display "(add (num 5) (mult (num 2) (num 3))): ")

61. (display (eval '(add (num 5) (mult (num 2) (num 3)))))

62. (newline)

63.

64. (display "(sub (num 20) (div (add (mult (num 4) (num 5)) (num 10))(num 6))): ")

65. (display (eval '(sub (num 20) (div (add (mult (num 4) (num 5)) (num 10))(num 6)))))

66. (newline)

67.

68. (display "(div (num 5) (sub (num 5) (num 5))): ")

69. (display (eval '(div (num 5) (sub (num 5) (num 5)))))

70. (newline)

71.

***Prompt 2: Adding State and Variable Operations***

“Now I need to add state handling and variable operations. So I need to maintain state for variables and implement these operations: 1. (define a) - creates a variable with 'undefined' value 2. (define a expr) - creates with a value from expression 3. (assign a expr) - assigns a new value to an existing variable 4. (id a) - references a variable from state 5. (remove a) - removes a variable from state. I also need to validate variable names - they must start with an alphabetic character and can contain letters, digits, hyphens or underscores. Can you modify the evaluator to handle state and these variable operations? I think we'll need to change the eval function to take and return state, and implement the variable operations.”

Generated code:

1. #lang racket

2.

3. ;; Define our Either type implementation

4. (struct success (value) #:transparent)

5. (struct failure (message) #:transparent)

6.

7. ;; Utility functions for working with Either

8. (define (from-success default either)

9. (if (success? either)

10. (success-value either)

11. default))

12.

13. (define (from-failure default either)

14. (if (failure? either)

15. (failure-message either)

16. default))

17.

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

19. (define in-list?

20. (λ (x lst)

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

22.

23. ;; Safe division that returns Either

24. (define (safe-div x y) ; num num -> Either

25. (if (= y 0)

26. (failure "Division by zero")

27. (success (/ x y))))

28.

29. ;; Check if a string is a valid identifier

30. ;; Must begin with an alphabetic character

31. ;; Followed by 0 or more letters, digits, hyphens (-) or underscores (\_)

32. (define (valid-id? id)

33. (and (string? id)

34. (> (string-length id) 0)

35. (char-alphabetic? (string-ref id 0))

36. (for/and ([c (in-string id)])

37. (or (char-alphabetic? c)

38. (char-numeric? c)

39. (char=? c #\\_)

40. (char=? c #\-)))))

41.

42. ;; Main evaluator function that takes an expression and state

43. (define (eval expr state)

44. (cond

45. ;; Numeric literal

46. [(equal? (first expr) 'num)

47. (values (success (second expr)) state)]

48.

49. ;; Variable reference

50. [(equal? (first expr) 'id)

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

52. (if (hash-has-key? state var-name)

53. (let ([value (hash-ref state var-name)])

54. (if (eq? value 'undefined)

55. (values (failure (format "Variable '~a' is undefined" var-name)) state)

56. (values (success value) state)))

57. (values (failure (format "Variable '~a' not defined" var-name)) state)))]

58.

59. ;; Arithmetic operations

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

61. (let-values ([(x-result state) (eval (second expr) state)])

62. (if (failure? x-result)

63. (values x-result state) ; Pass failure up

64. (let-values ([(y-result state) (eval (third expr) state)])

65. (if (failure? y-result)

66. (values y-result state) ; Pass failure up

67. (let ([x (success-value x-result)]

68. [y (success-value y-result)])

69. (case (first expr)

70. [(div) (values (safe-div x y) state)]

71. [(add) (values (success (+ x y)) state)]

72. [(sub) (values (success (- x y)) state)]

73. [(mult) (values (success (\* x y)) state)]))))))]

74.

75. ;; Variable definition

76. [(equal? (first expr) 'define)

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

78. (if (not (valid-id? (symbol->string var-name)))

79. (values (failure (format "Invalid variable name: '~a'" var-name)) state)

80. (if (hash-has-key? state var-name)

81. (values (failure (format "Variable '~a' already defined" var-name)) state)

82. (if (= (length expr) 2)

83. ;; Just declare with undefined value

84. (values (success 'undefined) (hash-set state var-name 'undefined))

85. ;; Define with initial value

86. (let-values ([(value-result state) (eval (third expr) state)])

87. (if (failure? value-result)

88. (values value-result state) ; Pass failure up

89. (values (success (success-value value-result))

90. (hash-set state var-name (success-value value-result)))))))))]

91.

92. ;; Variable assignment

93. [(equal? (first expr) 'assign)

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

95. (if (not (valid-id? (symbol->string var-name)))

96. (values (failure (format "Invalid variable name: '~a'" var-name)) state)

97. (if (not (hash-has-key? state var-name))

98. (values (failure (format "Cannot assign to undefined variable '~a'" var-name)) state)

99. (let-values ([(value-result state) (eval (third expr) state)])

100. (if (failure? value-result)

101. (values value-result state) ; Pass failure up

102. (values (success (success-value value-result))

103. (hash-set state var-name (success-value value-result))))))))]

104.

105. ;; Remove variable

106. [(equal? (first expr) 'remove)

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

108. (if (not (hash-has-key? state var-name))

109. (begin

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

111. (values (success 'removed) state))

112. (values (success 'removed) (hash-remove state var-name))))]

113.

114. ;; Unknown operation

115. [else (values (failure (format "Unknown operation: ~a" (first expr))) state)]))

116.

117. ;; Test the evaluator with state

118. (define (test-eval-with-state)

119. (display "Testing evaluator with state:\n")

120. (let\* ([empty-state (hash)]

121. [state-with-vars (hash 'a 5 'b 10)])

122.

123. ;; Test variable reference

124. (display "Variable reference (id a):\n")

125. (let-values ([(result state) (eval '(id a) state-with-vars)])

126. (printf "Result: ~a\n" result))

127.

128. ;; Test arithmetic with variables

129. (display "\nArithmetic with variables (add (id a) (id b)):\n")

130. (let-values ([(result state) (eval '(add (id a) (id b)) state-with-vars)])

131. (printf "Result: ~a\n" result))

132.

133. ;; Test variable definition

134. (display "\nVariable definition (define c (add (id a) (id b))):\n")

135. (let-values ([(result new-state) (eval '(define c (add (id a) (id b))) state-with-vars)])

136. (printf "Result: ~a\n" result)

137. (printf "Updated state: ~a\n" new-state))

138.

139. ;; Test assignment

140. (display "\nVariable assignment (assign a (add (id a) (num 1))):\n")

141. (let-values ([(result new-state) (eval '(assign a (add (id a) (num 1))) state-with-vars)])

142. (printf "Result: ~a\n" result)

143. (printf "Updated state: ~a\n" new-state))

144.

145. ;; Test remove

146. (display "\nRemove variable (remove b):\n")

147. (let-values ([(result new-state) (eval '(remove b) state-with-vars)])

148. (printf "Result: ~a\n" result)

149. (printf "Updated state: ~a\n" new-state))

150.

151. ;; Test remove non-existent

152. (display "\nRemove non-existent variable (remove z):\n")

153. (let-values ([(result state) (eval '(remove z) state-with-vars)])

154. (printf "Result: ~a\n" result))))

155.

156. ;; Run tests

157. (test-eval-with-state)

158.

***Prompt 3: REPL Implementation***

“Ok now let's implement the REPL to complete the assignment. The REPL should: 1. Present a prompt for user input 2. Evaluate expressions using our evaluator 3. Print results (success with value or failure with error message) 4. Display the current state 5. Continue until the user enters 'quit' or 'exit'. Can you also replace (test-eval-with-state) with REPL style test expressions as comments that I can run manually?”

Generated code:

1. #lang racket

2. ;; Define our Either type implementation

3. (struct success (value) #:transparent)

4. (struct failure (message) #:transparent)

5.

6. ;; Utility functions for working with Either

7. (define (from-success default either)

8. (if (success? either)

9. (success-value either)

10. default))

11.

12. (define (from-failure default either)

13. (if (failure? either)

14. (failure-message either)

15. default))

16.

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

18. (define in-list?

19. (λ (x lst)

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

21.

22. ;; Safe division that returns Either

23. (define (safe-div x y) ; num num -> Either

24. (if (= y 0)

25. (failure "Division by zero")

26. (success (/ x y))))

27.

28. ;; Check if a string is a valid identifier

29. ;; Must begin with an alphabetic character

30. ;; Followed by 0 or more letters, digits, hyphens (-) or underscores (\_)

31. (define (valid-id? id)

32. (and (string? id)

33. (> (string-length id) 0)

34. (char-alphabetic? (string-ref id 0))

35. (for/and ([c (in-string id)])

36. (or (char-alphabetic? c)

37. (char-numeric? c)

38. (char=? c #\\_)

39. (char=? c #\-)))))

40.

41. ;; Main evaluator function that takes an expression and state

42. (define (eval expr state)

43. (cond

44. ;; Numeric literal

45. [(equal? (first expr) 'num)

46. (values (success (second expr)) state)]

47.

48. ;; Variable reference

49. [(equal? (first expr) 'id)

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

51. (if (hash-has-key? state var-name)

52. (let ([value (hash-ref state var-name)])

53. (if (eq? value 'undefined)

54. (values (failure (format "Variable '~a' is undefined" var-name)) state)

55. (values (success value) state)))

56. (values (failure (format "Variable '~a' not defined" var-name)) state)))]

57.

58. ;; Arithmetic operations

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

60. (let-values ([(x-result state) (eval (second expr) state)])

61. (if (failure? x-result)

62. (values x-result state) ; Pass failure up

63. (let-values ([(y-result state) (eval (third expr) state)])

64. (if (failure? y-result)

65. (values y-result state) ; Pass failure up

66. (let ([x (success-value x-result)]

67. [y (success-value y-result)])

68. (case (first expr)

69. [(div) (values (safe-div x y) state)]

70. [(add) (values (success (+ x y)) state)]

71. [(sub) (values (success (- x y)) state)]

72. [(mult) (values (success (\* x y)) state)]))))))]

73.

74. ;; Variable definition

75. [(equal? (first expr) 'define)

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

77. (if (not (valid-id? (symbol->string var-name)))

78. (values (failure (format "Invalid variable name: '~a'" var-name)) state)

79. (if (hash-has-key? state var-name)

80. (values (failure (format "Variable '~a' already defined" var-name)) state)

81. (if (= (length expr) 2)

82. ;; Just declare with undefined value

83. (values (success 'undefined) (hash-set state var-name 'undefined))

84. ;; Define with initial value

85. (let-values ([(value-result state) (eval (third expr) state)])

86. (if (failure? value-result)

87. (values value-result state) ; Pass failure up

88. (values (success (success-value value-result))

89. (hash-set state var-name (success-value value-result)))))))))]

90.

91. ;; Variable assignment

92. [(equal? (first expr) 'assign)

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

94. (if (not (valid-id? (symbol->string var-name)))

95. (values (failure (format "Invalid variable name: '~a'" var-name)) state)

96. (if (not (hash-has-key? state var-name))

97. (values (failure (format "Cannot assign to undefined variable '~a'" var-name)) state)

98. (let-values ([(value-result state) (eval (third expr) state)])

99. (if (failure? value-result)

100. (values value-result state) ; Pass failure up

101. (values (success (success-value value-result))

102. (hash-set state var-name (success-value value-result))))))))]

103.

104. ;; Remove variable

105. [(equal? (first expr) 'remove)

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

107. (if (not (hash-has-key? state var-name))

108. (begin

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

110. (values (success 'removed) state))

111. (values (success 'removed) (hash-remove state var-name))))]

112.

113. ;; Unknown operation

114. [else (values (failure (format "Unknown operation: ~a" (first expr))) state)]))

115.

116. ;; Function to print the current state

117. (define (print-state state)

118. (printf "Current state:~n")

119. (for ([(key value) (in-hash state)])

120. (printf " ~a = ~a~n" key value)))

121.

122. ;; Main REPL loop

123. (define (repl)

124. (let loop ([state (hash)])

125. (printf "> ")

126. (flush-output) ; Ensure prompt appears immediately

127. (let ([input (read)])

128. (cond

129. [(or (eq? input 'quit) (eq? input 'exit))

130. (printf "Exiting...~n")

131. (void)]

132. [(eof-object? input)

133. (printf "End of input, exiting...~n")

134. (void)]

135. [else

136. (let-values ([(result new-state) (eval input state)])

137. (if (success? result)

138. (printf "Success: ~a~n" (success-value result))

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

140. (print-state new-state)

141. (loop new-state))]))))

142.

143. ;; Start the REPL

144. (repl)

145.

146. ;; Test cases (these would run in the REPL)

147. ;; '(num 5)

148. ;; '(add (num 5) (mult (num 2) (num 3)))

149. ;; '(sub (num 20) (div (add (mult (num 4) (num 5)) (num 10))(num 6)))

150. ;; '(div (num 5) (sub (num 5) (num 5)))

151. ;; '(define a)

152. ;; '(define b (num 10))

153. ;; '(assign a (num 5))

154. ;; '(add (id a) (id b))

155. ;; '(define c (add (id a) (id b)))

156. ;; '(assign a (add (id a) (num 1)))

157. ;; '(remove b)

158. ;; '(remove z)

159.

(I then tested the different expressions and added my name and due date at the top)