

AI 1 Documentation

Me: I am working on a project that involves creating a program that can evaluate arithmetic expressions with variables, handle errors, and maintain state. The final version should support operations like add, sub, mult, and div, as well as variable definitions and assignments. I have a basic evaluator that handles arithmetic operations but no state or variables. Can you help me add support for variables?

AI: Sure! First, let's define a helper function to represent the state. The state can be a list of pairs where each pair is (id . value). Here's a starting point:

```
(define (initial-state) '())
```

This creates an empty state. Next, we'll need a function to look up a variable in the state. Try implementing lookup-var that takes an id and the state and returns the value if the variable exists, or an error otherwise.

Me: Got it. Here's my attempt:

```
[code submitted]
```

AI: Good start! Now, let's handle undefined variables. Modify lookup-var to return an error if the variable exists but is 'undefined. Use equal? to check the value.

Me: Like this?

```
[code submitted]
```

AI: Perfect! Now, let's build update-var to add or modify a variable in the state. It should return a new state with the updated binding.

Me: Ok:

```
[code submitted]
```

AI: Correct! Now, let's integrate this into the eval function. Start by adding a case for 'id expressions that calls lookup-var.

Me: I added this to eval:

```
[code submitted]
```

But how do I handle define and assign?

AI: For define, check if the variable exists first. If not, add it to the state. For assign, ensure the variable exists before updating. Try implementing define first.

Me: Here's my define case:

[code submitted]

AI: Almost! Add support for optional expressions in define, like (define x (add 1 2)). Use let-values to evaluate the expression if provided.

Me: Updated:

[code submitted]

AI: Great! Now add the assign case similarly, but ensure the variable exists first.

Me: Done!

[code submitted]

AI: Excellent! Now, test your code with these inputs:

Basic Arithmetic Tests

racket

- > (add (num 5) (num 3)) ; Should return 8
- > (sub (num 10) (num 4)) ; Should return 6
- > (mult (num 2) (num 7)) ; Should return 14
- > (div (num 15) (num 3)) ; Should return 5
- > (div (num 5) (num 0)) ; Should fail with division by zero
- > (add (num 1) (mult (num 2) (num 3))) ; Nested operations

Variable Definition Tests

racket

- > (define x) ; Define x as undefined
- > (define y (num 10)) ; Define y with value 10
- > (define invalid-1 (num 5)) ; Should fail (starts with number)
- > (define _var (num 3)) ; Should fail (starts with underscore)
- > (define x (num 5)) ; Should fail (redefinition)

Variable Usage Tests

racket

- > (id x) ; Should fail (undefined variable)
- > (id y) ; Should return 10
- > (add (id y) (num 5)) ; Should return 15
- > (id undefined-var) ; Should fail (nonexistent variable)

Assignment Tests

racket

- > (assign x (num 20)) ; Should fail (x is undefined)
- > (assign y (num 20)) ; Should succeed
- > (assign y (add (id y) (num 1))) ; Increment operation
- > (assign z (num 5)) ; Should fail (z not defined)

Remove Operation Tests

racket

- > (remove y) ; Should remove y
- > (id y) ; Should now fail
- > (remove non-existent) ; Should show "ignoring" message
- > (define y (num 100)) ; Should work again after removal

Complex Expression Tests

racket

- > (define a (num 5))
- > (define b (mult (id a) (num 2)))
- > (assign a (add (id a) (num 1)))
- > (div (id b) (sub (id a) (num 3))) ; Should return $10/(6-3) = 3.333$

Edge Case Tests

racket

- > (define valid-name (num 1)) ; Hyphen in name

> (define a (div (num 5) (num 0))) ; Division by zero in definition

> (assign a (id undefined-var)) ; Assign undefined variable

> (add (num 1) (id no-such-var)) ; Using undefined var in expression