Task 3 Evaluation of an Expression

Adrian Jonsson Sjödin

Spring Term 2023

Introduction

The purpose of this task was to implement a program that evaluates a mathematical expression containing variables, and returns the result given an environment for the variables.

Method

The way to solve this task was quite similar to the earlier derivatives task. Like there we started with defining the different allowed expressions and what form a literal will take.

After this we started with implementing the functions given in the skeleton code, these being the `eval/2` functions. Here, like in the previous tasks, we start with considering the base cases of what should happen when we want to evaluate a literal. For the number we simply return the number, while for the variable we use Elixir's `Map` module to retrieve the value bound to the variable. For the quotient we have to do a bit more. We want to evaluate fractions to the lowest common denominator, as well as not have zero as a divisor. To do this we use a conditional `if else` clause and the `Integer` module's `gcd/2` function. The code for this can be seen in code overview 1.

Having done the base cases for evaluation of a literal we now need to implement how to evaluate an expression. This is done by recursively calling the `eval/2` function until it returns the value for the number or variable. When we have that it will go back through the functions and call the corresponding arithmetic functions. This way we will recursively calculate the expression. Furthermore if any of the expressions return undefined during this process, for example because of division by zero, we want the function to immediately return `:undefined` without making any further calculations. To achieve this we add a conditional check before each recursive call to `eval/2`. This can be seen in code overview 2 where we have the code for when we evaluate an addition between two expressions.

Code Overview 1: Evaluation of quotient

```
def eval({:quotient, dividend, divisor}, _environment) do
    if divisor == 0 do :undefined
    else
        gcd = Integer.gcd(dividend, divisor)
        {:quotient, dividend/gcd, divisor/gcd}
    end
end
```

Code Overview 2: Evaluation of addition between two expressions

```
def eval({:add, expression1, expression2}, environment) do
    left = eval(expression1, environment)
    right = eval(expression2, environment)
    if left == :undefined || right == :undefined do
        :undefined
    else
    add(left, right)
    end
end
```

The arithmetic functions `add/2`, `subtract/2`, `multiply/2` and `divide/2` are private functions that takes the number passed along to them and is what does the calculation of the expressions. When writing them it is important that we define them in the correct order and that we have a function for each possible combination that can exist. We must start with operations between two quotients, then number and quotient, and lastly between two numbers. If we have between two numbers first the program will use that to try and calculate everything and it will crash.

Lastly we implemented a `pretty_print/1` function, like in earlier tasks, for easier testing and evaluation of expressions. The code for this is really similar to earlier tasks so we will omit it here.

Result

In code overview we can see the result from a few test and conclude that the program works as intended.

Code Overview 3: Caption here

```
iex(5)> Evaluation.test1
Expression: 2*x + 3 + 1/2
The environment is: %{x: 2}
```

```
Expected: 15/2
Result, pretty format: 15.0/2.0
Result: {:quotient, 15.0, 2.0}
iex(6)> Evaluation.test2
Expression: (2*a + c + 6/8) / (4)
The environment is: %{a: 1, b: 2, c: 3, d: 4}
Expected: 23/16
Result, pretty format: 23.0/16.0
Result: {:quotient, 23.0, 16.0}
iex(7)> Evaluation.test3
Expression: (2) / (0) + x + y
The environment is: %{x: 2, y: 3}
Expected: :undefined
Result, pretty format: Error: invalid expression
Result: :undefined
```

Discussion

Link GitHub.