ID1019: Evaluation of an expression

Alexander Lundqvist

Spring Term 2023

Introduction

The task in this assignment is to create a program that can evaluate a small set of mathematical expressions that contains variables. The only public interface the program should have is an evaluation/2 method which takes an expression and an environment as its only parameters.

The environment is more or less a key-value based data structure that holds the value for each variable, also known as their binding. It can be implemented in several ways but it needs enough functionality to allow for creating a new environment, adding variables and their bindings as well as finding a binding for a given variable.

Method

The instructions provided a clear description of how the syntax should be constructed (which was very similar to the derivative task). We also got a basic skeleton for the evaluation functions that we could build upon. Furthermore, the instructions mentions that we have to implement an environment.

The arithmetic operations are implemented with the most common scenarios in mind such as operations between numbers, variables and fractions. This is done with method overloading and Elixir's pattern matching.

The environment was easily implemented with the Map module, as it already is a key-value structure that supports creation, insertion, update and removal. It is done implicitly by writing the program in a way that it expects to receive a Map, but the actual environment will be defined in a separate Test module that will call the evaluation program functions.

Result

In this section I will show some of the code and briefly talk about how the program works. I will also include some of the printouts from the test module to show that the evaluation method(s) works as intended.

The following code segment is an example of how the evaluation function works. The method is called as mentioned earlier with an expression and the environment. Since one of the scenarios that we want to account for is division by zero, we want a way to indicate to the user that an error has happened. We do this by implementing a guard case where we evaluate the components of the initial expression recursively and if at any point we get `:undefined`, then we return `:undefined` to the initial evaluation call.

```
def evaluate({:mul, expression1, expression2}, environment) do
   evaluated1 = evaluate(expression1, environment)
   evaluated2 = evaluate(expression2, environment)
   if evaluated1 == :undefined || evaluated2 == :undefined do
     :undefined
   else
     multiply(evaluated1, evaluated2)
end
```

In this code segment we can see how the program actually interacts with the environment. The evaluate function for {`:var, variable`} can be seen as the entry point to the environment. Here we utilize the Map module to retrieve the bound value for the variable and return it as a proper {`:num, number`} that the program can handle.

```
def evaluate({:num, number}, _environment) do
   {:num, number}
end
def evaluate({:var, variable}, environment) do
   {:num, Map.get(environment, variable)}
end
def evaluate({:quot, numerator, denominator}, _environment) do
  simplify({:quot, numerator, denominator})
end
```

In the code above we also had a call to a function called `simplify/1`. That function is described in the following segment. My early attempts to reduce the fractions was not too successful as I only managed to reduce the fraction from the first recursive call. My solution was to instead create an entire function dedicated to this task and call it everytime a function would return a {`:quot, numerator, denominator`}. It also gave me a chance to try Elixir's `cond do` statement (which I have learned works as `else if`). We do simple checks for zero numerators and denominator as well as reducing

by the greatest common denominator or just returning a number if it is evenly divisible.

```
defp simplify({:quot, numerator, denominator}) do
    cond do
      numerator == 0 ->
        {:num, 0}
      denominator == 0 ->
        :undefined
      rem(numerator, denominator) == 0 ->
        {:num, trunc(numerator/denominator)}
      true ->
        gcd = Integer.gcd(numerator, denominator)
        {:quot, trunc(numerator/gcd), trunc(denominator/gcd)}
    end
end
```

Lastly we have the testing of the evaluation. The following segments are but a few of the tests. In the first one we can see that it handles simple operations on fractions and reduces them accordingly.

```
------ Testing fraction multiplication -----
Expression: (3/5) * (5/8)
Expected: 3/8
Result: (3/8)

--------- Testing fraction division --------
Expression: (3/5) / (5/8)
Expected: 24/25
Result: (24/25)
```

I also implemented a test for expressions containing variables. Here we can see the initial expression, the binding for `x` and the result of the evaluation. I omitted the expected result here since the expression in this function changed quite often during testing.

```
iex(19)> Test.testExpression()
------- Testing expression evaluation ------
Expression: 2 * x + (8/14) + (3/7)
x = 2
Result: 5
:ok
```

Lastly I implemented a function to test for errors. I only really check for division by zero but if I add some more atoms I could make the error handling a little bit more extensive and descriptive. The test shows that the

program does not crash in the event of invalid input and just tells the user that the expression was invalid.

```
iex(20)> Test.testError()
--- Testing zero division multiplication ---
Expression: 2x + 3 + 1/0
x = 2
Result: Error: invalid expression
```

The entire code is available to read at this github repository.

Discussion

This section won't be long since the task was quite simple. I only had to follow the instructions and no benchmark nor comparison was asked for. The only thing I can think of worth discussing would be the test cases for the program but I already wrote about them in the result section. I can however talk about my experience with this assignment.

This task was quite difficult initially since I felt that the video lectures did not connect to the subject. I had to read much online to even understand what the task was. I did however get a clue from the Canvas discussion about using Map for environment. The actual development was very frustrating because I got a lot of errors and the Elixir shell did not output any meaningful information and thus it was hard to find where the actual issues were located. Fortunately I received some help from a course mate with finding most of the hidden errors.