Evaluating an expression

Axel Månson Lokrantz

Spring Term 2023

Introduction

The task for this assignment was to evaluate mathematical expressions containing variables. For simplicity, the expressions are limited to addition, subtraction, division and multiplication. The literals that will be used are either integers, `{:num, n}`, variables `{:var, v}` or rational numbers represented as quotients `{:q, q1, q2}`. The program was written in elixir, together with Axel Lystem at KTH.

The task

An environment

The function, `eval` takes an expression and an environment `env` to evaluate the expression to a literal. The environment is setup as a list where a given binding can be described as a tuple, `[{{:var, x}, {:num, n}}]`. The program is expected to have values for all variables in the expression, meaning there will be no free variables. The functionality of our environment is to either create a new environment with a given set of bindings or find a binding with a given value.

The evaluation

First and for most, we wanted to rewrite the expression with the given bindings. For an operation like addition the following functions were used.

```
def eval(:undefined, _) do :error end
def eval({:num, n}, _) do {:num, n} end
def eval({:var, v}, env) do env({:var, v}, env) end
def eval({:q, q1, q2}, _) do reduce({:q, q1, q2}) end

def eval({:add, e1, e2}, env) do
  eval(add(eval(e1, env), eval(e2, env)), env) end
```

Both operands are evaluated as either integers, variables or quotients. If either or both of the operands are represented as a quotient we call a function `reduce` to, if possible, simplify the expression. The same is true for the answer, which we might also be able to simplify, this is why the `eval` function is called a third time.

```
def reduce({:q, _, 0}) do :error end
def reduce({:q, 0, _}) do {:num, 0} end
def reduce({:q, q1, q2}) when rem(q1, q2) == 0 do
{:num, div(q1,q2)} end
def reduce({:q, q1, q2}) do
  gcd = Integer.gcd(q1, q2)
  {:q, div(q1, gcd), div(q2,gcd)}
end
```

A quotient which is divided by zero will result in an `:error`. If the numerator is 0, the function will return 0 no matter the denominator. If the remainder is 0, then we perform a division to simplify the quotient to an integer. If none of the above is true we will calculate the greatest common divider between the operands and divide the numerator and denominator by their gcd. The operation will always succeed, even if the expression is non reducible, since their gcd then will be equal to 1.

```
def add({:num, n1}, {:num, n2}) do {:num, n1 + n2} end
def add({:q, q1, q2}, {:q, q3, q2}) do {:q, (q1 + q3), q2} end
def add({:q, q1, q2}, {:num, n}) do {:q, (n * q2) + q1, q2} end
def add({:num, n}, {:q, q1, q2}) do {:q, (n * q2) + q1, q2} end
def add({:q, q1, q2}, {:q, q3, q4}) do
  {:q, (q1 * q4) + (q3 * q2), (q2 * q4)} end
```

To be able to add fractions we had to implement the appropriate rules of fraction addition which can be seen above. The rest of the mathematical operations such as subtraction, division, and multiplication followed a similar procedure. To see the rest of the operations follow the link to the GitHub repository.

Examples

The below expressions use the same environment where x equals 3, y equals 4 and z equals 5.

```
e = Eval.env([x: 3, y: 4, z: 5])

Eval.eval({:sub, {:add, {:mul, {:num, 3}, {:var, :x}},
{:num, 0}}, {:div, {:var, :y}, {:var, :z}}}, e)
```

```
Output expression 1:  {:q, 41, 5}

Eval.eval({:mul, {:add, {:mul, {:var, :y}, {:num, 0}},
{:div, {:num, 0}, {:var, :z}}}, {:div, {:var, :x}, {:num, 3}}}, e)

Output expression 2:  {:num, 0}

Eval.eval({:div, {:var, :y}, {:num, 0}}, e)

Output expression 3:  :error
```