Evaluating an expression

Nariman Haidar

February 9, 2023

1 Introduction:

The goal of this assignment is to write a program that can evaluate a set of variable-containing mathematical expressions.

We are going to implement a method evaluation/2,that takes an expression and its context and evaluates it to a literal. The environment is a mapping from variable names to values, and we expect all variables in the expression to have values.

2 Expressions:

First we are going to represent all operations using tuples where the first element is an atom that identifies the operation.

We are going to use the literals either integers, variables or rational numbers. To solve this problem we are going to use data-structure map.

```
@type expr() :: {:add, expr(), expr()}
| {:sub, expr(), expr()}
| {:mul, expr(), expr()}
| {:div, expr(), expr()}
| literal()

@type literal() :: {:num, n()}
  | {:var, a()}
  | {:quot, n(), n()}
```

Then we are going to test the expressions like $2x + 3 + 1/2$, First we are going to define how to add two quotients, then add number to quotient and then add two numbers , then we are going to do this to all of operations subtraction, multiplication and division.

In case division by zero, we are going to implement a function that return undefined without making any further calculations.

Then to test it we are going first to define a function test that is going to follow this Elixir structure like:

```
{:add, {:add, {:mul, {:num, 2}, {:var, :x}}, {:num, 3}}, {:q, 1,2}}

{:sub, {:add, {:mul, {:num, 2}, {:var, :x}}, {:q, 2, 3}}, {:q, 1, 2}}

{:div, {:mul, {:num, 2}, {:var, :x}}, {:add, {:q, 2, 3}, {:q, 1, 2}}}
```

3 Evaluation:

In this section we are going to implement the function eval function. We begin by thinking about all cases that ought to occur while evaluating a literal like in the number case we are going to return the number, but in the variable case to get the value tied to the variable, we use Elixir's Map module. In the quotient case and one of the quotient cases is division by zero, we are going to implement a function that alerting the user when an error has occurred, so that function is going to return undefined. Then, after implementing all cases for literal evaluation, we will develop cases for expression evaluation. This is accomplished by recursively calling the eval function.

```
def evalu({:num, x}, _)
do x
end

def evalu({:var, y}, env)
do Map.get(env, y)
end

def evalu({:add, n, n1}, env)
do add(eval(n, env), evalu(n1, env))
end

def evalu({:sub, n, n1}, env)
do sub(eval(n, env), evalu(n1, env))
end

def evalu({:mul, n, n1}, env)
do mul(eval(n, env), evalu(n1, env))
end

def evalu({:div, n, n1}, env)
do div(eval(n, env), evalu(n1, env))
end

def evalu(_, {:q, n, n1})
do quot(n, n1)
end
```

Here we are going to see some of results of evaluating like:
In the add case: $2x + 2 + 2$ the result is:
$iex(9) > Evalu.test0\{:\texttt{num}, 8\}$

In the sub case: $2x - 2/3 + 1/2$ the result is:
$iex(10) > Evalu.test1\{:\texttt{quot}, 17, 6\}$

In the div case: $2x/3/2 + 1/2$ the result is:
$iex(11) > Evalu.test2\{:\texttt{quot}, 24, 7\}$