Elixir - evaluating an expression

Zak Ora

Spring Term 2023

Introduction

In the first assignment of the course the derivative of an expression was calculated, and even reduced to a simplified form. Furthermore, the second assignment related to a key-value database where a key and value was stored as a tuple in a list. For this project, the two assignments were combined, where an expression containing different variables (such as `2x + y - z`) and assigned values to the variables are evaluated to an actual numerical value with the help of environment (for instance `[{:x, 10}, {:y, 5}, {:z, 2}]`. Thus, the expression $2x + 1$ would be evaluated to 3 with the environment $x = 1$.

1 Environment

The environment for the program was copied from the previous environment project, however the remove and add function were removed since they were not really necessary. Although, for an extended program where one can add and remove variables in real-time, these functions would be added back.

```
defmodule Env do
  def new(lst) do lst end

  def lookup([], _) do nil end
  def lookup([{key, value}|_], key) do value end
  def lookup([_|tail], key) do lookup(tail, key) end
end
```

Here the user has the ability to create a new environment with the `new(list)` line. Moreover, the expression module (where the value is evaluated) uses the lookup functions to search for the variables.

Expression

Since the skeleton code for the expression module was given from the task instructions, the main functions were quite easily made.

```
def eval({:num, n}, _) do {:num, n} end
def eval({:q, n, m}, _) do {:q, n, m} end
def eval({:var, v}, env) do Env.lookup(env, v) end
def eval({:add, n1, n2}, env) do
  add(eval(n1, env), eval(n2, env))
end
:
:
```

The same structure was made for the other arithmetic functions, such as division and multiplication. These were then sent to their respective calculator function, where the eval parameters pattern match with the calculation methods.

```
def add({:num, n1}, {:num, n2}) do n1+n2 end
def add({:num, n}, {:q, a, b}) do
  pprint(reducegcd({:q, ((n*b) + a), b})) end
def add({:q, a, b}, {:num, n}) do
  pprint(reducegcd({:q, (n*b) + a, b})) end
def add({:q, a, b}, {:q, c, d}) do
  pprint(reducegcd({:q, (a*d) + (b*c), b*d})) end
```

The calculations were very easy since it just followed basic maths, such as addition with two fractions or a fraction divided by another fractions. Furthermore, if the output is a fraction, it is reduced with the built-in elixir `Integer.gcd(n, m)` method. Lastly, there were only a few edge cases that needed to be covered, such as division by zero, which returns a `:fail`, or when dividing something by one, which is just the number itself.