Evaluating an expression

Björn Formgren

Spring 2023

Introduction

The third assignment in ID1019 consisted of setting up an environment for variable binding and a syntax for writing mathematical expressions. The goal was to have the ability to evaluate an expression to a literal.

An expression

The expression was represented the same way as in the "Taking the derivative" - assignment. A literal could either be a number, a variable or a rational number. An expression could be either a literal or any operation and so on.

```
@type literal() :: {:num, number()}
| {:var, atom()}
| {:q, number(), number()}
@type expr() :: literal()
| {:add, expr(), expr()}
| {:mul, expr(), expr()}
| {:sub, expr(), expr()}
| {:div, expr(), expr()}
```

An environment

The task assignment talked about implementing two functions, one that would return a new environment with a given set of bindings and one that would lookup the binding given a variable name. Since the Map module now was available for the students to use, the functions were directly taken from the module. An environment was created using the Map syntax:

```
env = %{x: 10, y: 15, z: 20}
```

In order to lookup a value given a variable name one could simply use the syntax

```
env[:x] # x = 10
```

Evaluate

There were a few base cases for the evaluation functions. If a variable could be found in the environment then it could simply be evaluated to that value. If a number was given to the evaluate function then the number would of course be returned. If a rational number was entered then another function divi was called upon the numerator and the denominator. The divi function took care of the special cases of integer division and reduced a rational number as far as possible. This was done by using the greatest common divisor function found in the Integer module. It was also made sure that it was not possible to divide by zero.

```
def divi({:num, n1},{:num, n2}) do
  case rem(n1,n2) do
  0 -> {:num, div(n1,n2)}
  _ -> {:q, div(n1, Integer.gcd(n1,n2)),
        div(n2, Integer.gcd(n1,n2))}
  end
end
```

All the different operations had to take in to account if a rational number was involved, this resulted in a few new functions resolving this. For example if two rational numbers were to be multiplied we did cross-multiplication and then reduced the result using the divi function. The add and sub operations needed to make sure they could handle rational numbers as well. The different cases were handled mathematically and divi was used once again to reduce the resulting rational numbers. Bellow are some example expressions and their results in Elixir's interactive shell.

```
expr1 = {:add, {:add, {:mul, {:num, 3}, {:var, :x}},
    {:div, {:num,7},{:num, 2}}},{:mul, {:div, {:var, :y},
    {:num, 3}},{:num, 5}}}
# Result
{:q, 87, 2}

expr2 = {:add, {:div, {:num, 3},
        {:num, 4}}, {:div, {:num, 3}, {:num, 4}}}
# Result
{:q, 3, 2}

expr3 = {:div, {:add, {:num, 10}, {:var, :x}}, {:num, 5}}
# Result
{:num, 3}
```