Derivatives

Lucas Larsson

Spring Term 2022

Introduction

This report is a brief summary of the Derivatives assignment implementation and result.

Representation

The representation of the functions in this assignment follows the suggested representation in the lectures given by the Examiner Hohan. That is the functions are represented using variables and constants using `atoms`.

A lot of consideration goes into deciding how one would model a representation of the functions, the teacher already goes though this in the lecture so i am not going to explain a lot here, The main idea is that we need a representation that is identifiable (can differentiate functions), and that we can preform operations on.

The idea is pretty simple, basically one can represent almost any function only using addition and multiplication operations. The functions will be a bit on the longer side syntax wise, But it is still readable and with this choice of representation it possible to do operations as pattern matching to be able to differentiate different types of functions since different functions have different derivation rules.

```
Example:

@type literal() :: {:num, number()} | {:var, atom()}

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

The above inserted code basically means that a literal can either be a number or a variable/constant as mentioned before. Atoms `:num, :var` get assigned to be number, atom respectively.

Next step the `expr()` that is build by the first representation, get used to build the `:add,:mul` functions.

Later on the above stated expression will be added to depending on the function to be derived.

For an example to be able to implement a function that derive a natural logarithm function `ln(x)`, Two expressions were needed `:ln` and `:div`

```
  {:ln, expr()}
| {:div, expr(), expr()}
```

`:ln` function has one argument as indicated above with a single `expr()` and the division function takes in 2 arguments as indicated. The division function is implemented first because it is needed to impleminted the `:ln` finction since the deviate of `ln(x)` is $\frac{1}{x}$.

Implementation

After the representation is decided and out of the way, I started with the implementation, first step is considering the base cases, Then moving to the more advanced rules.

```
def deriv({:num, _ }, _ ) do {:num, 0} end
def deriv({:var, v}, v) do {:num, 1} end
# anything besides " v " because it is mentioned in the function before
def deriv({:var, _}, _) do {:num, 0} end

def deriv({:add, e1, e2}, v) do
  {:add, deriv(e1, v), deriv(e2, v)}
end

def deriv({:mul, e1, e2}, v) do
  {:add,
    {:mul, deriv(e1, v), e2 },
    {:mul, e1, deriv(e2, v)}
  }
end

  def deriv({:div, e1, e2 }, v) do
  {:div,
      {:add,
          {:mul, deriv(e1, v), e2},
          {:mul,
            {:mul, e1, deriv(e2, v)},
            {:num, -1}}},
```

```
        {:exp, e2, {:num, 2}}
    }
end
```

The above shown code shows the result of the implementation.

- First `deriv` function is in case one tries to get the derivative of a number the result should be `0`.
- Second `deriv` function indicates that the derivative of variable is `1`.
- And the third indicates that the derivative of a variable with respect to another variable is `0`. With the base cases out of the way the other cases are handled following the general derivatives formulas.

$$\frac{d}{dx}(g(x) + f(x)) = g'(x) + f'(x)$$

$$\frac{d}{dx}(f(x) * g(x)) = g'(x) * f(x) + g(x) * f'(x)$$

$$\frac{d}{dx}\left(\frac{f(x)}{g(x)}\right) = \frac{g'(x) * f(x) - g(x) * f'(x)}{g(x)^2}$$

Following the above stated rules of derivation the functions are implemented with addition of an extra steps of multiplying the second argument in the numerator since only subtraction operation is not implemented.

Simplification

This section is mainly to simplify the expressions computed according to the previous two sections.

```
{:div,

    {:add,

        {:mul, {:num, 0}, {:var, :x}},
        {:mul,

            {:mul, {:num, 5}, {:num, 1}},
            {:num, -1}}},

    {:exp, {:var, :x}, {:num, 2}}}
```

the expression shown above is correct but not piratically readable at first glance.

So for better readability a series of simplification methods are created(se below).

```
    # calls other operation specific simplify method
    def simplify({:div, e1, e2}) do
        simplify_div(
            simplify(e1),
            simplify(e2)
        )
    end

    # takes operation specific base case and calculations
    def simplify_div(e1, {:num, 1}) do {:num, e1} end
    def simplify_div({:num, 0}, _) do {:num, 0} end
    def simplify_div({:num, n1},{:num, n2}) do {:num, n1/n2} end

    def pprint({:div, e1, e2})  do
        "(#{pprint(e1)})/(#{pprint(e2)})"
    end
```

The methods `simplify()`, `simplifydiv()` and `pprint()` change the output from the code shown in the beginning of this section to what is shown below with the input of function $\frac{5}{x}$

```
Expression: (5)/(x)
Derivative: ((0 * x + 5 * 1 * -1))/((x)^(2))
Simplified: (-5)/((x)^(2))
Calculated: -5.0
```

A test method is also implemented to validate the output of the `deriv()` functions

```
    def testDiv() do

        e = {:div, {:num, 5}, {:var, :x}}
        d = deriv(e, :x)
        c = calc(d, :x, 1)
        IO.inspect(d)
        IO.inspect(c)
        IO.write("Expression: #{pprint(e)}\n")
        IO.write("Derivative: #{pprint(d)}\n")
        IO.write("Simplified: #{pprint(simplify(d))}\n")
        IO.write("Calculated: #{pprint(simplify(c))}\n")

        :ok

    end
```

Finally a `calc()` function is implemented to get the result of the computation for a specific `x` value.

The report only takes in the `div()` function, but the rest of the functions are implemented in the same way.

```
def calc({:div, e1, e2}, v, n) do
    {:div, calc(e1, v, n), calc(e2, v, n)}
end

def calc({:num, n}, _, _) do {:num, n}  end
def calc({:var, v}, v, n) do {:num, n}  end
def calc({:var, v}, _, _) do {:var, v}  end
```