Derivatives - Report

Adrian Hjert

20/Jan/2023

Introduction

The aim of this task was to begin learning how the programming language elixir works by learning how to use functions and take their derivatives. The aim of this report is to highlight how code was implemented to solve the problems at hand, those being implementing some of the basic derivative rules for; the chain rule; the addition rule; the exponent rule; the logaritmic rule; the root (which is essentially the exponent rule with more steps), and the *sin* rule.

Method and Implementation of Code

The first step was to define the different operations we needto use to make the derivatives work properly and so we can express them in a manner that is easy to read. This is mostly for the reader to quickly see how things should be written. An example of this can be seen below.

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

We can see that the structure follows Normal Polish Notation (NPN) where the operand comes first, followed by the elements to be operated on. Next, is to implement the derivative rules which is relatively straightforward; we define some parameter, i.e. what we want to take the derivative of, then we write how it should be expressed. Below is an example of how the $\frac{1}{x}$ derivative could look like.

```
#-------(1/x)' = -n(1/x^(n+1))----------
def der({:div, e1, e2}, v) do
  {:div, {:sub, {:mul, der(e1, v), e2}, {:mul, e1, der(e2, v)}},
  {:exp, e2, {:num, 2}}}
  end
```

So, the function *der* takes the division (*:div*) of $e1$ and $e2$ and changes its structure to the correct derivative of anything following a $\frac{1}{x}$ structure. For clarity, the code for finding the derivative of *sin* x is shown.

```
#--------(sin(x))' = cos(x)----------
 def der({:sin, e}, v) do
   {:mul, der(e, v), {:cos, e}}
 end
```

So, here, *sin* takes just one expression. The function then derives it according to the correct derivation rules in order to express the result as *cos* x.

Next is to simplify the results and account for some edge cases before finally printing them on the screen in the terminal. The simplifications are done in the following manner:

```
#----------simplifying sin and cos-------
 def simp({:sin, e}) do simp_sin(simp(e)) end
 def simp({:cos, e}) do simp_cos(simp(e)) end
 def simp(e) do e end
```

Which makes use of the edge cases accounted for:

```
#----------edge cases sin and cos--------
 def simp_sin({:num, 0}) do {:num, 0} end
 def simp_sin(e) do {:sin, e} end
 def simp_cos({:num, 0}) do {:num, 1} end
 def simp_cos(e) do {:cos, e} end
```

This essentially looks at the result after the derivative has been taken and attempts to cut out any unnecesary parts of the expression and, well, simlify, for example if we end up with a constant such as 0 in *cos*, we simplify it to 1, instead of having *cos* 0 in our answer which may harm legibility.

Finally, some lines of code were written to print the answer in a nice way. The following example shows the code for nicely printing divisions and logarithmic functions.

```
#------printing ln and divisions--------
    def printStr({:ln, e}) do "ln(#{printStr(e)})" end
    def printStr({:div, e1, e2}) do
    "(#{printStr(e1)}/#{printStr(e2)})" end
```

This essentially takes the expressions and prints them as strings in the terminal so it becomes easier to read, along with the simplifications, otherwise, an output for the derivative of $2x^2$ would look like:

```
{:add, {:mul, {:num, 0}, {:exp, {:var, :x}, {:num, 2}}},
 {:mul, {:mul, {:mul, {:num, 2}, {:exp, {:var, :x}, {:num, 1}}},
 {:num, 1}}, {:num, 2}}}
```

Rater than: $((2 * x) * 2)$ which is not fully simplified but it is much easier to read than the NPN representation we start out with.