Coding a derivative calculator program

Zak Ora

Spring Term 2023

Introduction

This report analyzes and describes the derivative calculator, which was coded in the functional programming language Elixir. Additionally, it simplifies the output for the reader to establish a more user friendly answer. Lastly, the program is an introduction to the language as it contains expressions, functions, backtracking, pattern matching and teaches general Elixir syntax.

Derivative laws

The derivative laws was coded with the pattern matching fundamentals in elixir, using a one to one representation for each law. The pattern matching system reads the expressions and evaluates the literals to its appropriate derivative law, for instance the expression `{:mul, {:num, 2}, {:var, :x}}` would be matched to the derivative law of multiplication due to the atom `:mul`.

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

The initial laws were quite simple to make, however other laws such as sine and the derivative of natural logarithms were a bit more challenging. Mostly, since it would require new implementations rather than just the existing arithmetic. Although this was eventually avoided by utilizing the base code and inputting different expressions. More specifically, divisions were handled with a negative exponential, as shown in the code below:

```
def deriv({:ln, e}, v) do
  {:mul,
      {:exp, e, {:num, -1}},
        deriv(e,v)}
end
```

Here the derivative of $ln(x)$ (which is $1/x$) was handled with the exponent keyword instead, which is exactly how square-roots were processed as well. Lastly, derivatives of trigonometric functions were programmed together with the `:math` keyword and other existing derivative functions, if a composite function should be derived.

```
def deriv({:sin, e}, v) do
  {:mul,
      {:cos, e}, deriv(e,v)}
end
```

Output

The output of the derivative was mainly handled by the `simplify` function, which pattern matched the resulting derivative and simplified it as much as possible, to make the output more user friendly. More specifically, standard operations were handled, such as multiplication with zero and one, simple addition of two numbers, exponents of zero and one, standard sinus and cos values, and much more. Lastly, once simplified, the output was made even more easier to read by the pretty print functions. These would take the simplified derivative and write it in a readable way in the terminal, a few examples can be seen in the code below:

```
def pprint({:exp, e1, e2}) do "(#{pprint(e1)})^(#{pprint(e2)})" end
def pprint({:ln, e}) do "ln(#{pprint(e)})" end
def pprint({:sin, e}) do "sin(#{pprint(e)})" end
def pprint({:cos, e}) do "cos(#{pprint(e)})" end
```

Conclusion

In short, the program can calculate a plethora of derivatives, simplify it to standard form, and give the output in a readable and user-friendly way. Moreover, it thought a lot about how pattern matching can be exploited to remove if-statements, which makes the program much easier and concise to read. It also gave an introduction to how functional programs and structured and how they utilize backtracking.