Derivatives

Emil Göransson

Spring Term 2023

Introduction

This report will cover the implementation of a program that given a mathematical expression written in an AST (Abstract Syntax tree) form will return both a non simplified derived expression, and the same derived expression simplified. This will only cover real numbers.

Supported operations and expected returns

The following is supported: Addition, subtraction, exponential functions, square-root, division, the natural logarithm and sinus. Using the following atoms they are all defined from these snippets of code.

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

  @type expr() :: literal()   | {:add, expr(), expr()}
    | {:mul, expr(), expr()} | {:exp, expr(), literal()}
    | {:sqrt, expr()}        | {:div, expr(), expr()}
    | {:ln, expr()}          | {:sin, expr()}
    | {:cos, expr()}         | {:sub, expr(), expr()}
```

Here we can tell how AST works, the thing you wish to be integrated is entered using the syntax above. For example we write 2*x + 1 as

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

Rules of Derivatives and their implementation

In order to correctly perform derivation we need to implement some rules and keep some things in mind, the rules used in this program will be the standard rules of derivation, the Chain- rule, the Product rule and the Quotient rule.

As deriving single numbers, single variables, addition, multiplication and exponential functions were all done by the professor I will not cover those topics since the code was not written by me. The operations I will be focusing on is: subtraction, ln, division, square-root and sinus.

Subtraction

Subtraction is very similar to how addition works and is implemented in the following way:

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

The Natural Logarithm

Ln(x) is also very simple and is implemented using the :div - tag, we take the inner derivative of x and make it the numerator, and just put x as the denominator.

```
def deriv({:ln, e}, v) do
   {:div, deriv(e,v), e}
 end
```

Fractions

This is the most tricky of the two, needing a entirely new atom (:div) and also the implementation of the Quotient rule. The snippet used to perform the derivation is the following, although it is barely readable. It is however just the quotient rule written in our AST notation with some help of recursion.

```
   def deriv({:div, e1, e2}, v) do
   {:div, {:sub, {:mul, deriv(e1, v), e2}, {:mul, e1, deriv(e2, v)}},
   {:exp, e2, {:num, 2}}}
 end
```

Square root

Since the rules of derivations are known the implementation is quite simple It uses the :div tag to get a fraction and takes the inner derivative of e as the Numerator.

```
    def deriv({:sqrt, e}, v) do
   {:div, deriv(e, v), {:mul, {:num,2},{:sqrt, e}}}
 end
```

Sinus

A very simple function, works by changing the sin to a cos (new atom) and taking the inner derivative and multiplying it with the cosinus.

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

Pretty print

Since the AST formation used is barley readable we use a function PPrint to convert the messy AST to a real numerical synax that we are used it. E.g instead of writing :cos , :var, :x we get cosx!

Since we have a lot of different atoms and operations we need quite a bit of functions but since they all look about the same, I will only show a couple. Once you've seen two of them you've pretty much seen them all. The PPrint covering sin and div looks like the following:

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

    def pprint({:sin, e}) do
     "sin(#{pprint(e)})"
    end
```

An issue with PPrint is that it just doesn't fix all the problems with readability. It does make it more simple however if you were to perform a derivation of a long and complex expression you will get a lot of unnecessary characters (e.g 0*1 instead of nothing) To solve this we use the function simplify.

Simplify

Simplify is a function that contains a lot of known rules in math. These are simple ones like that 1+1=2 and that 5/1 = 5 but also more complex ones such as ones covering the rules of adding two ln functions. Simplify works by simplifying each expression one by one. E.g if a division between two expression is called

```
simplify({:div, e1, e2})
```

both e1 and e2 will by the help of recursion each get simplified one by one, simplify(e1), simplify(e2) The results from those two calls are then used to call a new simplified function for division in particular (works the same for every other operator)

```
simplify_div(simplify(e1), simplify(e2))
```

simplify_div is multiple functions that is able to perform a bunch of shortening operations on the e1 and e2. One example of this is if e1 and e2 are two numbers both being of the :num tag. We can then just calculate the division and return that calculated number as a single :num atom.

```
def simplify_div({:num, n1}, {:num, n2}) do
  {:num, n1 / n2}
end
```

Every other operator works in the same manner and has a collection of rules that is used to shorten the expression though recursion