Elixir Derivative Function

Tomás Belmar da Costa

Spring Term 2022

Introduction

For this assignment we had to make an elixir function `deriv(e, v)` that takes in an expression (e) and finds its derivative with respect to v.

Setting up

I started by splitting the concept of an expression into two sub-parts. Literals, and expressions. A literal deals with either a number or a variable. These are represented by tuples:
`{:num, 5}`
`{:var, :x}`

An expression is a mathematical combination of literals using operators such as (but not limited to) adding, multiplying, exponentiating, and more. This would also be represented by tuples as follows:
`{:mul, x, y}` (an example of multiplication of literals, or expressions, x and y)

Derivation Functions

The simplest cases for the derivatives of an expression were expressions that were either just a constant or just a variable. If the expression was just a number, the derivative is 0. If the expression was the variable we are deriving with respect to, the derivative is 1. If the expression was another variable, we can treat it as a constant and therefore the derivative is 0.

```elixir
# Constant
def deriv({:num, _}, _) do
  {:num, 0}
end

# Itself
```

```
def deriv({:var, x}, x) do
  {:num, 1}
end

# Itself with respect to another variable
def deriv({:var, _}, _) do
  {:num, 0}
end
```

The next challenge would be returning the derivative of a sum. Since the derivative of a sum is simply the sum of the two derivatives, I used recursion to sum the results of the two individual derivatives of expressions e1 and e2.

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

The derivative of a product ($e_1 * e_2$) follows the rule: $e_1 * deriv(e_2) + e_2 * deriv(e_1)$. This was once again simple to achieve with recursion.

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

When deriving x^n (with respect to x), the formula for the derivative is $n * x^{n-1}$. Using the function signature to ensure we only receive the inputs of a variable and a number, the derivative was acheived like so:

```
def deriv({:pow, {:var, x}, {:num, n}}, x) do
  {:mul, n, {:pow, x, (n - 1)}}
end
```

For the natural logarithm, while the derivative of the natural logarithm of x is $1/x$, it's possible that a constant may be inputted. Therefore, I included both a base case where a constant is inputted ($ln(n)$), which will return 0, and the case where the variable is inputted ($ln(x)$).

```
def deriv({:ln, {:num, _}}, _) do
  {:num, 0}
end

def deriv({:ln, {:var, x}}, x) do
  {:pow, :x, -1}
end
```

The derivative of $\sqrt{x}$ is $1/(2\sqrt{x})$. Again accounting for constants being inputted, this is easily achieved.

```
def deriv({:sqrt, {:num, _}}, _) do
  {:num, 0}
end

def deriv({:sqrt, {:var, x}}, x) do
  {:pow, {:mul, 2, {:sqrt, x}}, -1}
end
```

Finally the only function remaining is the sine function. Since the derivative of $sin(x)$ is $cos(x)$ we simply need to return that.

```
def deriv({:sin, {:num, _}}, _) do
  {:num, 0}
end

def deriv({:sin, {:var, x}}, x) do
  {:cos, x}
end
```

Simplifying

Even though the derivative functions above all work, and now the derivative for even complex expressions can be found, they return an ugly result.

For example, let's find the derivative of $2x+x^4$. This is expressed in Elixir as
`{:add, {:mul, {:num, 2}, {:var, :x}}, {:exp, {:var, :x}, {:num, 4}}}`

Elixir then returns the following:
`{:add, {:add, {:mul, {:num, 0}, {:var, :x}}, {:mul, {:num, 1}, {:num, 2}}}, {:mul, 4, {:pow, :x, 3}}}}`

`While this is technically correct, this function can be simplified in order to be more readable. We can delete expressions that are multiplied by zero, remove zero when it's simply added to other expressions, remove one when it's multiplied by other expressions, combine like terms, and perform arithmetic operations on numbers.`

`First off, to delete expressions that are multiplied by zero, accounting for the fact that 0 can come from both sides, we do the following.`

```
def simplify_mul({:num, 0}, _) do
  {:num, 0}
end

def simplify_mul(_, {:num, 0}) do
  {:num, 0}
end
```

Similarly, for adding to 0, we can simply remove the 0 from the expression.

```
  def simplify_add({:num, 0}, e2) do
    e2
  end

  def simplify_add(e1, {:num, 0}) do
    e1
  end
```

Following in these footsteps, we can make a similar pattern for multiplying by 1.

```
  def simplify_mul({:num, 1}, e2) do
    e2
  end

  def simplify_mul(e1, {:num, 1}) do
    e1
  end
```

Next comes combining like terms. This involves adding two expressions that are the same variable, and converting that to a multiplication by 2.

```
  def simplify_add({:var, x}, {:var, x}) do
    {:mul, {:num, 2}, {:var, x}}
  end
```

Finally, performing arithmetic operations on numbers involves detecting two numbers and using the built in Elixir arithmetic operators to simplify them into one number.

```
  def simplify_add({:num, n1}, {:num, n2}) do
    {:num, n1 + n2}
  end

  def simplify_mul({:num, n1}, {:num, n2}) do
    {:num, n1 * n2}
  end
```