Derivative Programming II

Sushil KC

Spring Term 2023

Introduction

In this assignment the student were assign to with functions as symbolic expressions and compute the derivative of functions. We were tasked to learn how we can represent functions using the data structures that we have in the programming language for this course called Elixir.

Method

We will represent a function with an AST abstract syntax tree. We represent all numbers using tuples with atom num and the number value. We can know that this tuple is a number just by looking at the atom that identifies our number. We also represent variables like x in a tuple where the variable is an atom that identifies the variable. If numbers and variables are our literals, we have the following definition.

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

Expressions

For the expressions we will also represent them using tuples where the first element in the tuple is the expression itself so that we can identify the operation and two more elements for the numbers that the operation is done. Now we can identify a number, a variable and add, multiplication arithmetic. We can write some simple expression in elixir for example the expression

$$5x^2 + 2$$

will look like this.

```
{:add, {:mul, {:exp, {:var, :x}, {:num, 2}}, {:num, 5}}, {:num, 2}}
```

Results/discussion

When we derive an expression there are some clear set of rules on how we can do that. We will define those rules in elixir that the program drives a function automatically. The set of rules are that the derivative of a variable is 1 and the derivative of a number is 0. If the expression is addition then the derivative is the sum of the derivatives of the terms. For the product there is a general rule the says that the derivative of two terms is dx (f $\times$ g) dx(f) $\times$ (g) + (f) $\times$ dx(g). This rule can be applied for every product and by using the rule we can get derivatives of every product. In the code below you can see the implementation of the product rule in elixir. For example the derivative of 5x will be 5 and using the product rule for derivative we will get 0 * 5 + 5*1 which is the answer that will be printed out. More specifically this

```
{:add, {:mul, {:num, 0}, {:var, :x}}, {:mul, {:num, 5}, {:num, 1}}}
```

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

As you can see if we have a product between two terms then the derivative will be the derivative of the first term multiplied by the second term and added by the first term multiplied by the derivative of the second term. We also do derivative for other types of expressions like ln, square root of x and sin x. For these we use the chain rule to derivative these expression. We the inner function derivative times the derivative of the outer function. As you can see in the code below when we take the derivative of square root of an element we first the derivative of the elements and divide it by the derivative of the outer element the square root it self.

```
  //sinus
  def deriv({:sin, e1}, v) do
    {:mul,
    deriv(e1, v),
    {:cos, e1}}end
  //squre root
  def deriv({:sqrt, e1}, v) do
    {:div,
    deriv(e1, v),
    {:mul, {:num, 2}, {:sqrt, e1}}
  }
  end
```

This expression

$$2x^2 + 3x + 5$$

can be represented in elixir like

```
{:add,{:add, {:mul, {:num, 2}, {:mul, {:var, :x}, {:var, :x}} },
                {:mul, {:num, 3}, {:var, :x}}}, {:num, 5}}
```

The output of the expression will be.

```
{:add,{:add, {:add, {:mul, {:num, 0}, {:mul, {:var, :x}, {:var, :x}}},
{:mul, {:num, 2},  {:add, {:mul, {:num, 1}, {:var, :x}}, {:mul,
{:var, :x}, {:num, 1}}}}}}, {:add, {:mul, {:num, 0}, {:var, :x}},
{:mul, {:num, 3}, {:num, 1}}}}}, {:num, 0}}
```

As you can see the results of our derivation might be correct but they are very hard to read. They contain multiplications with zero, addition with constant values etc. All of those could be removed if we simplified the results. To simplify the results we can make different cases where we check if an expression matches our case then we return something. Cases can be for example when we multiply something with 0 then we just return 0 or when we elevate something with 0 then we just return 1. There are such cases that apply to every expression.

```
def simplify_mul({:num, 0}, _) do {:num, 0} end
def simplify_exp(_, {:num, 0}) do {:num, 1} end
```

With the simplified version we get results that are much more readable and you can see below.

```
 expression: ((2 * x * x + 3 * x) + 5)
 expressions derivative: (((0 * x * x + 2 *
                        (1 * x + x * 1)) + (0 * x + 3 * 1)) + 0)
 simplified: (2 * (x + x) + 3)
```