# Homework 11: Derivation of arithmetic expressions in Scala

*Stephen Wagstaff*
*CS 431*
*November 24, 2018*

---

## Derivation of arithmetic expressions

For this homework, you will use Scala to implement the functions of homework 8 to take derivatives of arithmetic expressions, print the results, and simplify them.

You will define Scala classes to represent the following ML data types for arithmetic expressions.

```
datatype exp = Const of int
             | Var of string
             | Plus of exp * exp
             | Times of exp * exp
             | Pow of exp * int;
```

You should define an *abstract class* called `Exp` and some case classes such as `Const` and `Plus` to represent each of the data constructors. For example, the variable `e` as defined below

```
val e = Times (Times (Var("x"), Var("y")), Plus (Var("x"), Const(3)))
```

represents the expression `(x × y) × (x + 3)`. The variable `e1` as defined below

```
val e1 = Pow (Var("x"), 4)
```

represents the expression `x4`.

The following are some rules for derivations.

dc/dx = 0 -> where `c` is a constant
dx/dx = 1
dy/dx = 0 -> where y != x
d(u+v)/dx = du/dx + dv/dx
d(u×v)/dx = (du/dx) * v + u * (dv/dx)
d(un)/dx = n * un−1 * (du/dx)

*For the following methods, you should use def instead of val.*

1. Implement a method toString: string in Exp class to convert this expression to its string representation. For example, `e.toString` should return the string `"((x * y) * (x + 3))"` and `print(e1)` should return the string `"(x^4)"`

   ```
   override def toString: String ={
       this match{
         case Const(value) => value.toString
         case Var(value) => value
         case Plus(value1, value2) => "(" + value1 + " + " + value2 + ")"
         case Times(value1, value2) => "(" + value1 + " * " + value2 + ")"
         case Pow(base, power) => "(" + base + "^" + power + ")"
       }
   }
   ```

2. Implement a method `deriv: (x: String)Exp` in `Exp` class to derive this expression `u` against the input string `x` and return the derivative.
   *Note that the second parameter of the method* ***deriv*** *is the string of some variable name.*

   For example, `e.deriv("x").toString` should return `"((((1 * y) + (x * 0)) * (x + 3)) + ((x * y) * (1 + 0)))"` while `e1.deriv("x").toString` should return `"((4 * (x^3)) * 1)"`

   ```
   def deriv(dx: String): Exp ={
       this match{
         case Const(_) => Const(0)
         case Var (variable) => if (variable.equals(dx)) Const(1) else Const(0)
   ```

```
      case Plus (exp1, exp2) => Plus(exp1.deriv(dx), exp2.deriv(dx))
      case Times (exp1, exp2) => Plus(Times(exp1.deriv(dx), exp2), Times(exp1, exp2.deriv(dx)))
      case Pow (exp1, int) => Times(Times(Const(int), Pow(exp1, int-1)), exp1.deriv(dx))
    }
}
```

3. Implement a method `simplify: Exp` in `Exp` class to simplify this expression as much as possible.
   For example, `e.deriv("x").simplify.toString` should return `"((y * (x + 3)) + (x * y))"` while `e1.deriv("x").simpli` should return `"(4 * (x^3))"`
   Also, if `val e2 = Pow (Plus (Var("x"), Const(0)), 2)`, then `e2.toString` should return `"((x + 0)^2)"` while `e2.simplify.toString` should return `"(x^2)"`.

   *Hint: for this question, you may want to define a helper method* ***simp*** *to simplify obvious expressions.* ***simp(e × 0)*** *= 0,* ***simp(e × 1)*** *= e,* ***simp(e + 0)*** *= e, etc. The method simplify should call* ***simp*** *after recursively calls itself on components of* ***plus****,* ***times****, and* ***pow*** *expressions.*

```
protected def attemptRootSimplification: Exp ={
    this match {
    case Times(Const(1), x) => x
    case Times(x, Const(1)) => x
    case Times(Const(0), _) => Const(0)
    case Times(_, Const(0)) => Const(0)
    case Plus(Const(0), x) => x
    case Plus(x, Const(0)) => x
    case Pow(x, 1) => x
    case Pow(_, 0) => Const(1)
    case _ => this
    }
}

def simplify: Exp ={
    this match{
    case Const(x) => Const(x)
    case Var(x) => Var(x)
    case Times(exp1, exp2) => Times(exp1.simplify, exp2.simplify).attemptRootSimplification
    case Plus(exp1, exp2) => Plus(exp1.simplify, exp2.simplify).attemptRootSimplification
    case Pow(exp, int) => Pow(exp.simplify,int).attemptRootSimplification
    }
}
```