# Haskell: Symbolic Differentiation

NAME: PROGRAMME:

## Some practicalities

- You have two hours to solve this assignment **individually**.
- You may only consult the Haskell slides (on-line or in paper form, along with any hand-written notes) and any exercise that has been made available for this course on Toledo. You may also consult the manuals that are mentioned on Toledo, and Hoogle as well.
- In the folder **1516_Graded/Haskell_Thursday** on Toledo you can find the files `SymbolicDifferentiation.hs` and `SymbolicDifferentiationTest.hs`. The hand-in module can be found in the same folder.
  - **Download and open** the file `SymbolicDifferentiation.hs`. This file already contains a template for your solution.
  - Fill in your name, studentnumber and programme at the top of the file first. For example:

    ```
    -- Jan Jansen
    -- r0123456
    -- master cw
    ```

  - You may also import **additional functions and types**.
  - For every exercise there are a number of predefined functions, along with a corresponding type signature. The type signature must not be changed. Replace in the function body `undefined` with your implementation. Naturally, it is always permitted to write additional (helper) functions.
  - You can test your solution with `SymbolicDifferentiationTest.hs`. You do so by running the following command in the directory containing both `.hs` files:

    ```
    runhaskell SuggestionsTest.hs
    ```

    **Note** Having all tests succeed does not mean that your program is entirely correct, or that you will earn the maximum number of points!
  - Hand in the file `SymbolicDifferentiation.hs` via Toledo after two hours, or whenever you have finished the assignment.

# Symbolic Differentiation

Many functions have simple rules to compute their derivative. You use these rules implicitly whenever you compute derivatives by hand on a piece of paper. For example, the derivative of the polynomial $x^2 + x + 1$ is computed as follows:

$$\begin{aligned}
\frac{d(x^2+x+1)}{dx} &= \frac{d(x^2)}{dx} + \frac{dx}{dx} + \frac{d1}{dx} && \text{(the derivative of a sum is the sum of the derivatives)} \\
&= \frac{d(x^2)}{dx} + 1 + 0 && \text{(the derivative of } x \text{ is 1, the derivative of 1 is 0)} \\
&= 2x + 1 + 0 && \text{(the derivative of } x^n \text{ is } nx^{n-1}\text{)} \\
&= 2x + 1
\end{aligned}$$

In this assignment, we program the computer to obtain derivatives following the same procedure.

## Part 1: Representing functions

We limit ourselves to functions from the reals to the reals, consisting of (real) constants, a variable (that may occur multiple times), addition, multiplication, exponentiation with a constant exponent and the natural logarithm.

These functions are represented by the data type `Function`:

```
data Function
  = Const Double
    -- ^ constant
  | X
    -- ^ variable
  | Function :+: Function
    -- ^ addition
  | Function :*: Function
    -- ^ multiplication
  | Function :^: Double
    -- ^ power with constant exponent
  | Ln Function
    -- ^ natural logarithm
  deriving (Show, Eq)
```

Note that this definition uses data type operator-constructors (`:+:`, `:*:`, `:^:`). Just like normal constructors you can pattern match on these constructors, for example:

```
isPlus :: Function -> Bool
isPlus (_ :+: _) = True
isPlus _         = False
```

| Operation | Domain | Result given $x = a$ | Derivative |
|---|---|---|---|
| `Const c` | $x \in \mathbb{R}$ | $c$ | $0$ |
| `X` | $x \in \mathbb{R}$ | $a$ | $1$ |
| `f :+: g` | $x \in \mathbb{R}$ | $f(a) + g(a)$ | $\frac{d}{dx}f(x) + \frac{d}{dx}g(x)$ |
| `f :*: g` | $x \in \mathbb{R}$ | $f(a)g(a)$ | $\left(\frac{d}{dx}f(x)\right)g(x) + f(x)\frac{d}{dx}g(x)$ |
| `f :^: c` | $\begin{cases} x \in \mathbb{R}_0 & \text{if } c \leq 0 \\ x \in \mathbb{R} & \text{if } c > 0 \end{cases}$ | $(f(a))^c$ | $c(f(x))^{c-1}\frac{d}{dx}f(x)$ |
| `Ln f` | $x \in \mathbb{R}_0^+$ | $\ln(f(a))$ | $\frac{1}{f(x)} \cdot \frac{d}{dx}f(x)$ |

Table 1: The domains, results, and derivatives of the operations.

The function `isPlus` is `True` when its argument is an addition (`:+:`) and `False` otherwise. It checks this by pattern matching on its argument.

**Exercise 1** Complete the defintion of the `Num`-instance for `Function`. Fill in the `undefined`s in the instance such that for example the following expressions become possible:

```
ghci> X + 1
X :+: Const 1.0
ghci> (-X) * (Ln 1)
(Const (-1.0) :*: X) :*: Ln (Const 1.0)
```

***Remark*** *Leave the* `undefined`*s as-is if you cannot solve this exercise. Pay attention when writing the examples in this case, since using* `+` *or* `*` *instead of* `:+:` *or* `:*:` *causes errors!.*

## Part 2: Evaluating functions

A value `f` of type `Function` can be evaluated by substituting a value $a \in \mathbb{R}$ for every `X` in `f` and doing the obvious thing for the other operations (e.g. use an addition for `f :+: g`, see Table 1 for reference).

Not all functions are defined for every real number. In particular, exponentiation ($x^r$) is not defined if $r \leq 0$ and $x = 0$, and the natural logarithm $\ln(x)$ is not not defined for $x > 0$.

Table 1 lists the domain and expected result for every operation (here we use $\mathbb{R}_0$ to denote $\mathbb{R}\backslash\{0\}$, and $\mathbb{R}_0^+$ to denote all strictly positive reals).

**Exercise 2** Complete the function `evaluate :: Function -> Double -> Maybe Double` such that `evaluate f a` gives the expected result corresponding to Table 1. Return `Nothing` if $a$ falls outside the domain, otherwise return `Just`. Use the `Maybe`-monad and `do`-notation.

***Hint:*** *`:info Double` prints a list of all typeclasses that are instantiated by* `Double`*. Use the methods in these typeclasses to implement* `evaluate`*.*

```
ghci> evaluate X 1
Just 1.0
ghci> evaluate (1 :+: (2 :*: X):^:2) 1
```

```
Just 5.0
ghci> evaluate (Ln (X:^:2 :+: (-4))) 2.0
Nothing
ghci> evaluate (Ln (X :+: 1)) 0
Just 0.0
```

## Part 3: Derivatives

Now, apply the rules for computing derivates to the `Function`-data type. The derivation rules for all supported operations can be found in the last column of Table 1. Pay attention to the use of the chain rule in the last two entries.

**Opdracht 3** Complete the function `derivative :: Function -> Function`, such that `derivative f` is a new `Function`, representing the derivative of `f`.

Some examples of `derivative` in action:

```
ghci> derivative (X :+: 1)
Const 1.0 :+: Const 0.0
ghci> derivative (X:^:2 :*: 1)
((Const 2.0 :*: X :^: 1.0) :*: Const 1.0) :*: Const 1.0 :+: X :^: 2.0 :*: Const 0.0
ghci> derivative (Ln (X:^:2))
(X :^: 2.0) :^: (-1.0) :*: ((Const 2.0 :*: X :^: 1.0) :*: Const 1.0)
ghci> evaluate (derivative (Ln (X:^:2)))) 2
Just 1.0
```

## Part 4: Pretty Printing

The output of the `Show`-instance of `Function` leaves much to be desired in terms of aestetics. Therefore, we define a function `pretty` that produces `String`s that are much more visually pleasing.

**Opdracht 4** Complete the function `pretty' :: Int -> Function -> String` for `Const c`, `X`, `:+:`, `:*:`, `:^:` and `Ln`. Write these constructors as `c`, `x`, `+`, `*`,`^` and `ln`. Always insert spaces around `+` and `*`, but not around `^` or `ln`.

The function `pretty'` has the type `Int -> Function -> String`. The first argument (of type `Int`) indicates the priority of the operator of the expression in which the second argument (of type `Function`) occurs. The priorities of the operators are listed in Table 2. Only insert parentheses around a function when it is neccessary to display this function correctly. In other words, only insert parentheses when the function occurs as the operand of another operator with a higher priority.[1]

For instance, in the next expression we put parentheses around (1 + 2), because otherwise the expression would be (incorrectly) interpreted as 1 + (2 * 3) ((1 + 2) is an operand of the `*`-operator):

[1]Note that `Ln` has the highes priority. This implies that we will never insert parentheses around `Ln`, for example: `pretty (Ln (Ln 1)) == "lnln(1.0)"`

| Operator | priority |
|---|---|
| `Ln` | 4 |
| `Const`, `X` | 3 |
| `:^:` | 2 |
| `:*:` | 1 |
| `:+:` | 0 |

Table 2: The operators and their priorities.

```
ghci> pretty ((1 :+: 2) :*: 3)
"(1.0 + 2.0) * 3.0"
```

But we won't use any parentheses in the next expression:

```
ghci> pretty ((1 :*: 2) :*: 3)
"1.0 * 2.0 * 3.0"
```

A more complex example of `pretty`:

```
ghci> pretty ((Ln (1 :+: X:^:2)) :*: X)
"ln(1.0 + x^2.0) * x"
```

The desired behaviour for `pretty'` in particular is, for example:

```
ghci> pretty' 3 (Const 1)
"1.0"
ghci> pretty' 4 (Const 1)
"(1.0)"
```

## Part 5: User Interface

Now we write a command line program `evaluateIO :: Function -> IO ()` that receives a `Function` as an argument and then performs the following steps:

1. The program `pretty`-prints the function it received as its argument.
2. The program asks the user to enter a number.
3. The program evaluates the function for this number and prints the result.
4. The program `pretty`-prints the derivative of the function.
5. The program asks the user to enter another number.
6. The program evaluates the derivative and prints the result.

The following is an example of the expected behaviour:

```
ghci> evaluateIO (X:^:2)
Please enter a value for x to evaluate the function f(x) on, where:
  f(x) = x^2.0
x = 2                                    <--- "2" is user-input
  f(x) = Just 4.0
The derivative of this function is:
  df/dx = 2.0 * x^1.0 * 1.0
Enter a value for x to evaluate df/dx on:
x = 2                                    <--- "2" is user-input
  df/dx | (x = 2.0) = Just 4.0
```

**Opdracht 5** Implement the program `evaluateIO :: Function -> IO ()` as it has been described above. If you have not yet finished exercise 4 (`pretty`), then use `show` from the `Show`-typeclass instead. If you have not yet finished exercise 3 (`derivative`), reuse the function instead of the derivative.