Middle East Technical University Department of Computer Engineering

CENG 242

Programming Language Concepts
Spring 2023
Programming Exam 3

Due: 2 May 2023 23:55
Submission: **via ODTUClass**

1 Problem Scenario

In this programming exam, we will be building a function "calculator" for some basic mathematical functions using types and typeclasses in Haskell. We will represent these mathematical functions using custom data types, which we will later make instances of several typeclasses to extend their functionality. This programming exam is made up of three parts, each of which is explained below.

1.1 General Specifications

- The signatures of the functions, their explanations and typeclass specifications are given in the following section. Read them carefully.
- Make sure that your implementations comply with the function signatures.
- You may define helper function(s) as needed. You may also define your own data types and typeclasses, or make our types instances of Haskell's other typeclasses.
- Whenever you need to output a `Double` value (which you will only need for the `Evaluable` typeclass), you must use the following function to round it to 2 decimal places:

```
getRounded :: Double -> Double
getRounded x = read s :: Double
               where s = printf "%.2f" x
```

- Data types and typeclasses that we will be using are explained below:

1.1.1 Data Types

We will use a total of five custom data types for this homework. Four of these will represent the basic types of functions that we will be using, and the fifth will group them into one so that we may write formulas containing all of them. Note that all our mathematical functions and formulas for this homework will be univariate, i.e. they will only contain one variable.

– The `Power` type represents a mathematical power function (x^n). It only contains one `Integer`, which shows the order of the power term. Note that we only deal with non-negative integer powers, i.e. we won't have negative or fractional powers. Its definition is:

```
data Power  = Power Integer
```

– Our second data type is the `Polynomial`. Polynomials are sums of power functions, each one multiplied with an integer coefficient ($c_0x^0 + c_1x^1 + c_2x^2 + ...$, note that some c_i's may be zero, in which case the corresponding power would be missing from the polynomial); we will use a list of coefficient-power pairs to represent each one. It is definition is:

```
data Polynomial = Polynomial [(Integer, Power)]
```

– Next we have `Exponential`. We will only use exponentials with the base e, so we only need to keep the exponent (which may be any `Polynomial` value):

```
data Exponential = Exponential Polynomial
```

– Finally, we have the `Trigonometric` type to represent our trigonometric functions, which is defined similarly to the `Exponential` type, but has two value constructors. We will only use *sin* and *cos*. Inside the *sin* and *cos* we may have any `Polynomial` value.

```
data Trigonometric = Sin Polynomial | Cos Polynomial
```

– Our final data type, `Term`, unifies all our previous types and their combinations. It represents all kinds of mathematical functions that we will be dealing with in this homework. A `Term` can be a constant integer, a power term, an exponential term or a trigonometric term. A power term is made up of a `Power` and an integer coefficient, e.g. $3x^2$. Exponential and Trigonometric terms have an integer coefficient and a multiplicative `Power` (this term will come in handy when dealing with the derivatives), e.g. $2xcos(x+3)$, $4x^3e^{(x^2+2x+1)}$. Its definition is below:

```
data Term = Const Integer
          | Pw Integer Power
          | Exp Integer Power Exponential
          | Trig Integer Power Trigonometric
```

- Here are some examples that show how each mathematical function would be represented by our `Term` data type:
  * x: `Pw 1 (Power 1)`
  * $4x^3$: `Pw 4 (Power 3)`
  * e^{x^2+1}: `Exp 1 (Exponential (Polynomial [(1, Power 2), (1, Power 0)]))`
  * $2xcos(x)$: `Trig 2 (Power 1) (Trigonometric (Polynomial [(1, Power 1)]))`

2 Part I: Built-in Typeclasses (35 points)

One thing you may have noticed is that we do not have any derived instances for the built-in typeclasses, such as `Show`, in our data definitions. Even though Haskell can derive instances for such typeclasses automatically based on the definition of the associated type, in many cases the default behaviour of such instances is not what we want. In such cases, we can write the instance definitions ourselves instead of having Haskell automatically generate them for us. This is what we will be doing in this section.

The skeletons for the instance definitions are already available in the `PE3.hs` file. You are required the implement the necessary functions given to complete these.

2.1 Show

The `Show` typeclass is a very commonly used typeclasses that denotes types that can be converted into strings. It is also what determines how values are displayed in `ghci`. In our previous programming exams, we had derived that type automatically. We could still do that here, but it would be very difficult to read our mathematical formulas with such a notation. Therefore, we will define the string conversion ourselves, so that they are easier to read and more similar to mathematical notation. Make sure you follow the following specifications for how each of our data types should look (also, be careful not introduce extraneous whitespace where it is not specified):

- `Power`s should look like so: `x^n` where n is the power that the variable is being raised to. Do not write out the power part if it is not necessary, i.e. if it is 0 or 1. Here are some examples:
  - x^0: `1`
  - x^1: `x`
  - x^2: `x^2`

- `Polynomials` are shown as sums of the power terms that they contain. Coefficients are written directly next to the power part on the left side, without any parentheses. Make sure to surround the plus signs with one space on either side. Here are some examples:
  - 12: `12`
  - $7x$: `7x`
  - $x^2 + x + 1$: `x^2 + x + 1`
  - $3x^3 + 2x^2 + 5x + 4$: `3x^3 + 2x^2 + 5x + 4`
- `Exponentials` are shown as `e^polynomial`. The polynomial part should be surrounded with parentheses only if it contains more than one part, or a power higher than 1. Here are some examples:
  - e^3: `e^3`
  - e^{2x}: `e^2x`
  - e^{x^2}: `e^(x^2)`
  - e^{5x+1}: `e^(5x + 1)`
  - e^{x^2+4x+3}: `e^(x^2 + 4x + 3)`
- `Trigonometric` functions are displayed as `sin(polynomial)` and `cos(polynomial)`, respectively. Similarly to the exponentials, parentheses are only necessary if there is more than one part, or a power higher than 1 in the polynomial. Here are some examples:
  - $sin(3)$: `sin3`
  - $cos(2x)$: `cos2x`
  - $sin(x^2)$: `sin(x^2)`
  - $cos(5x + 1)$: `cos(5x + 1)`
  - $sin(x^2 + 4x + 3)$: `sin(x^2 + 4x + 3)`
- `Terms` are shown as the sum of the parts that they are made up of. `Consts` are shown as just their values, `Power` terms should be shown in the same way that they were in the `Polynomials`. Exponential and trigonometric terms should first have their coefficient, then the power (without parentheses), and then the actual exponential/trigonometric part. Here are some examples:
  - $sin(x) + cos(x)$: `sinx + cosx`
  - $2x + 2x^2 + e^x$: `2x + 2x^2 + e^x`
  - $12x^3e^{x^2+2x+1}$: `12x^3e^(x^2 + 2x + 1)`
  - $x^3sin3x + 3xcos(x + 2) + e^{x^2+2x}$: `x^3sin3x + 3xcos(x + 2) + e^(x^2 + 2x)`

2.2 Eq

The `Eq` typeclass is for equality comparisons, which defines the function `(==)`. We will only need to make Power an instance of `Eq`. To check for equality, simply compare the powers of the terms (Note that Haskell could have defined that automatically for us, but we'll make it by hand anyway). Here's an example:

```
ghci> Power 2 == Power 2
True

ghci> Power 1 == Power 3
False
```

2.3 Ord

`Ord` is for orderable types. Some built-in functions, like `sort`, require that their arguments contain values that are instances of `Ord`. Like `Eq`, we will only do this for `Power`. They should be ordered by how high the power is, like so:

```
ghci> Power 2 > Power 3
False

ghci> Power 1 <= Power 4
True

ghci> Power 5 < Power 7
True

ghci> Power 3 >= Power 3
True
```

3 Part II: Custom Typeclasses (50 points)

So far, we have defined our data types and made them instances of some of the built-in typeclasses of Haskell. Now, we will go one step further and define our own typeclasses. Once we define a typeclass, we can use it as a type constraint in type signatures of functions just like the built-in typeclasses. If we later want to use these functions with other types, we would just need to make these types instances of our typeclass, after which we can use values of that type as arguments to our functions.

3.1 Evaluable

Evaluable is our own typeclass for things that we can be evaluated, by building Haskell functions from them. It only requires one function to be defined to make a type an instance of it. That function is called `function`, which accepts one `Evaluable` argument and returns a function that takes one number and returns the result of evaluating the `Evaluable` using this number. It is defined in the files as:

```
class Evaluable a where
    function :: a -> (Integer -> Double)
```

You will be making all five of our types members of `Evaluable`, as given in the `PE3.hs` file.

HINT: Pay attention to the type signature of the `function` function. Even though we deal with integer coefficients, powers and arguments, the resulting values are `Double`s. You may need to adjust the type of your result before you can return it. You may find Haskell's built-in `fromInteger` function useful for such purposes.

Here are some examples of using this function:

```
ghci> function (Power 2) 3
9.0

ghci> f = function (Pw 3 (Power 3))
ghci> f 2
24.0
```

3.2 Differentiable

Differentiable is our second custom typeclass, which we will use to represents things that we can take derivatives of. Like `Evaluable`, it only requires one function, namely `derivative`, which takes one argument of a type that is `Differentiable`, and returns a list of `Term`s that result from taking the derivative of the argument.

```
class Differentiable a where
    derivative :: a -> [Term]
```

You will be making all five of our types members of `Differentiable`, as given in the `PE3.hs` file. Note that when taking the derivatives of exponential and trigonometric functions, you may need to multiply them with polynomials, since they may have polynomials inside. In such cases, you will need to expand the multiplication into separate terms. For example, the drivative of:

```
e^(x^2 + 3x + 5)
```

is:

```
2xe^(x^2 + 3x + 5) + 3e^(x^2 + 3x + 5)
```

Here are some example usages:

```
ghci> derivative (Power 3)
[3x^2]

ghci> derivative (Polynomial [(4, Power 5), (2, Power 3), (5, Power 1)])
[20x^4,6x^2,5]

ghci> derivative (Exponential (Polynomial [(1, Power 2), (1, Power 1)]))
[2xe^(x^2 + x),e^(x^2 + x)]

ghci> derivative (Sin (Polynomial [(1, Power 1), (1, Power 0)]))
[cos(x + 1)]

ghci> derivative (Exp 3 (Exponential (Polynomial [(3, Power 1)])))
[9e^3x]
```

4 Part III: Instances for Parameterised Types (15 points)

In Haskell, list is a parameterised type, which means we can apply a type to it and then make it an instance of our typeclasses. (Which means we can make lists of `things`'s, where `thing` is another type, members of our typeclasses.) We could do this for all of our types (`[Power]`, `[Exponential]` etc.), but for this homework we will only define such instances for lists of `Terms` (i.e. `[Term]`). This will allow us to evaluate and differentiate many combinations of our functions.

4.1 Evaluable

You should make `[Term]` a member of the `Evaluable` typeclass. A list of terms represent a sum of the terms they contain (so the value of the list is the sum of the values of all the elements in the list). An example use:

```
ghci> f = function [Trig 2 (Power 0) (Sin (Polynomial [(1, Power 0)])), Const 2]
    ghci> f 1
    3.68
```

4.2 Differentiable

Finally, you will make `[Term]` a member of the `Differentiable` typeclass. Since lists of terms represent the sum of the terms, it is sufficient for you to take the derivatives of each term separately. Do note, however, that some terms may result in multiple terms when differentiated, and some other terms may disappear completely. Make sure you handle such cases correctly. Here is an example use:

```
ghci> derivative [Trig 2 (Power 0) (Sin (Polynomial [(1, Power 0)])), Pw 2 (Power 3)]
[6x^2]
```

5 Regulations

1. **Implementation and Submission:** The template file named "PE3.hs" is available in the Virtual Programming Lab (VPL) activity called "PE3" on OdtuClass. At this point, you have two options:

   - You can download the template file, complete the implementation and test it with the given sample I/O (and also your own test cases) on your local machine. Then submit the same file through this activity.
   - You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

   If you work on your own machine, make sure that your implementation can be compiled and tested in the VPL environment after you submit it.
   There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating.** People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

3. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

- **Important Note:** The given sample I/O's are only to ease your debugging process and NOT official. Furthermore, it is not guaranteed that they cover all the cases of required functions. As a programmer, it is your responsibility to consider such extreme cases for the functions. Your implementations will be evaluated by the official test cases to determine your ***actual*** grade after the deadline.