Programming in Haskell – Project Assignment

UNIZG FER, 2016/2017

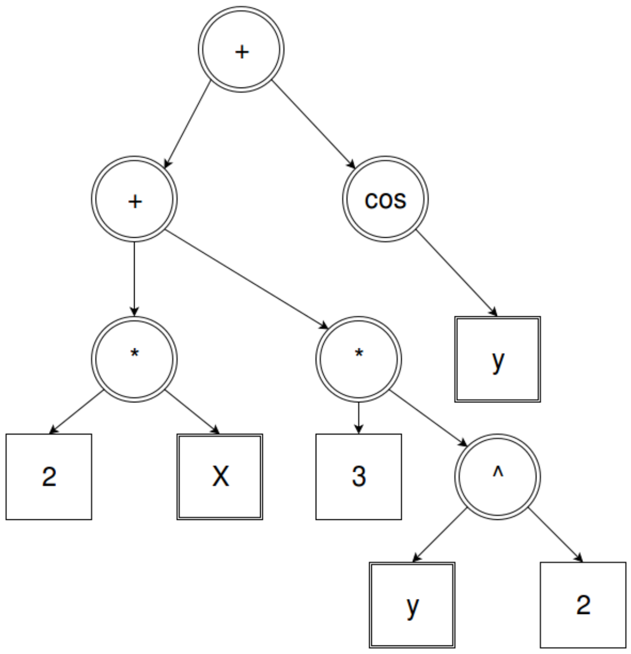
Handed out: December 22 2016. Due: January 17 2016

1 Introduction

Your task is to implement a simple symbolic computation framework, a baby version of Wolfram Alpha or MATLAB. The idea is that you can write an arithmetic expression such as: `2 * x + 3 * y^2 + cos y` and then do transformations on this expression such as evaluation at a given point, getting a partial derivative, adding it with another expression and so on. How you implement this functionality is entirely up to you, however, we will provide hints of the solution in this document as well as defining features your implementation must support.

2 Guidelines

You will begin by defining your own tree-like data structure representing an arithmetic expression. For example, the expression `2 * x + 3 * y^2 + cos y` can be represented as:



First thing for you to do is to think about how would define this data structure. Notice that some nodes contain two children (operators), some nodes contain only one child (functions) and some nodes contain no children (variables and constants).

If you've defined your data structure correctly, you should have no problem implementing all the basic functionalities. Evaluating the expression is simple, first you substitute the variables x and y with actual numbers, and then you recurse down the structure applying the operations as you go until you reach the constants when you return the result.

Getting the partial derivative of the expression is done recursively using the well known Chain rule. Notice that you must somehow check whether an expression is constant when doing partial derivation.

That's the core part of the problem, for getting there you need to somehow construct the tree when given an expression written in infix notation. This can be done easily in two steps.

1. Convert the expression from infix into postfix or prefix notation.
2. Use the converted expression to build the tree

You are already familiar with the Postfix notation used in the RPN calculator task from the last homework. Constructing an expression tree from prefix or postfix notation is then straightforward. We suggested using RPN because there exists a simple algorithm for infix to RPN conversion.

When implementing your code, think about optimizations that can be made to an expression. For example, you can remove the node that contains the multiplication of anything with a zero and replace it with a zero. You can then remove nodes that contain addition with zero and replace them with a single node containing the other part of the addition.

Finally, define your code to be an instance of the `Show` class. This will allow for pretty printing of your expression. Because we humans are accustomed to the infix notation, make sure your code prints out expressions as you would write them in.

3 Requirements

For this project, the only boundary is your creativity. Once you have the base code done, you can easily extend it to support a wide variety of features. However, for the purpose of grading you are expected to provide at least the following functionalities:

1. We expect your code to at least support the basic arithmetic operators ($+, -, *, /, \hat{}$) and the basic functions (sin, cos, exp, log).
2. Your code should work for an arbitrary number of variables.
3. Expressions can be evaluated for arbitrary values of variables in the expression. Expressions can be derived by any variable present in the expression.
4. The results are printed out in infix notation. Regardless of how you implement the main algorithm, the results must be printed out in human friendly form.

This means that you can write code like this (function names and signatures can of course be different):

```
testTree    = createExpression "2 * ( x ^ 2 ) + sin ( y + x )"
> 2 * ( x ^ 2 ) + sin ( y + x )
derivTree   = deriveBy testTree "x"
> 2 * ( 2 * x ) + cos ( y + x ) + 0
derivTree'  = testTree `deriveBy` "y"
> 0 + cos ( y + x ) + 0
subsTree    = substitute "x" testTree 0
> 2 * ( 0 ^ 2 ) + sin ( y + 0 )
evaluate subsTree
> error "Tree still contains variables, substitute them before evaluation"
subsTree'   = substitute "y" subsTree $ pi / 2
> 2 * ( 0 ^ 2 ) + sin ( 1.5707963267948966 + 0 )
evaluate subsTree'
> 1
```

Handle errors appropriately. You do not have to follow the sketched model, as long as your solution has at least the listed features, is coherent and well-documented.

3.1 Simplifications

1. You can expect that the input will always be properly tokenized so you can separate them using the `words` function. This means that the input `x*2 + 3* (y +1)` is invalid while `x * 2 + 3 * ( y + 1 )` is valid.
2. The power function (ˆ) will always have a number as it's second argument. That is, expressions like `( x + 1 ) ^ ( x + 2 )` are invalid while `( x + 1 ) ^ 3.5` is valid.
3. Variables will always be a single character (though not necessarily x, y, z)
4. You don't need to employ arithmetic simplifications. First focus on making your code working properly and if you have the time, find some things that can be optimized out.

4 Submission

The project is due for Tuesday, January 17th, in the last week of classes, at a time and place yet to be determined, in the evening hours. You will demonstrate your project to the teaching assistants on your own laptop. Your submission will be graded depending on the percentage of implemented functionality, robustness, code quality and the "cool factor" of your code. You need at least 15 points (out of 30) to pass the course.

Consider this an early Christmas present.
You're welcome.