Learn You a Physics for Great Good!

>>> WORK IN PROGRESS <<<

Calculus / Syntax trees

[src: Calculus/SyntaxTree.lhs] Previous: Plotting graphs Table of contents Next: Vectors

```haskell
module Calculus.SyntaxTree where
```

```haskell
import           Calculus.FunExpr
import           Calculus.DifferentialCalc
import           Calculus.IntegralCalc
```

A fun and useful way to visualize expressions is to model them as trees. In our case we want to model $FunExpr$ where the nodes and leaves will be our constructors.

In order to do this will import two packages, one for constructing trees and one for pretty printing them.

```haskell
import           Data.Tree         as T
import           Data.Tree.Pretty  as P
```

Now we can construct the function that takes a FunExpr and builds a tree from it. Every node is a string representation of the constructor and a list of its sub trees (branches).

```haskell
makeTree :: FunExpr -> Tree String
makeTree (e1 :+ e2)  = Node "+"    [makeTree e1, makeTree e2]
makeTree (e1 :- e2)  = Node "-"    [makeTree e1, makeTree e2]
makeTree (e1 :* e2)  = Node "*"    [makeTree e1, makeTree e2]
makeTree (e1 :/ e2)  = Node "Div"  [makeTree e1, makeTree e2]
makeTree (e1 :^ e2)  = Node "**"   [makeTree e1, makeTree e2]
makeTree (e1 :. e2)  = Node "o"    [makeTree e1, makeTree e2]
makeTree (D e)       = Node "d/dx" [makeTree e]
makeTree (Delta r e) = Node "Δ"    [makeTree (Const r), makeTree e]
```

```haskell
makeTree (I e)       = Node "I"    [makeTree e]
makeTree Id          = Node "Id"   []
makeTree Exp         = Node "Exp"  []
makeTree Log         = Node "Log"  []
makeTree Sin         = Node "Sin"  []
makeTree Cos         = Node "Cos"  []
makeTree Asin        = Node "Asin" []
makeTree Acos        = Node "Acos" []
makeTree (Const num) = Node (show num) [] --(show (floor num)) [] -- | Note the use of
  floor
```

Now we construct trees from our expressions but we still need to print them out. For this we'll use the function `drawVerticalTree` which does exactly what its name suggests. We can then construct a function to draw expressions.

```haskell
printExpr :: FunExpr -> IO ()
printExpr = putStrLn . drawVerticalTree . makeTree
```

Now let's construct a mildly complicated expression

```haskell
e = Delta 3 (Delta (negate 5) (I Acos) :. (Acos :* Exp))
```

And print it out.

```
ghci > printExpr e
          Δ
          |
  -----------
 /           \
 3           o
             |
       ----------
      /          \
      Δ          *
      |          |
     ----       -----
    /    \     /     \
   -5    I   Acos    Exp
         |
        Acos
```

Pretty prints the steps taken when canonifying an expression

```haskell
prettyCan :: FunExpr -> IO ()
prettyCan e =
  let t  = makeTree e
      e' = canonify e
      t' = makeTree e'
    in if t == t' then putStrLn $ drawVerticalTree t
                 else do
                     putStrLn $ drawVerticalTree t
                     prettyCan e'
```

Pretty prints syntactic checking of equality

```haskell
prettyEqual :: FunExpr -> FunExpr -> IO Bool
prettyEqual e1 e2 = if e1 == e2 then
  do
    putStrLn "It's equal!"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    return True
  else do
    putStrLn "Not equal -> Simplifying"
    putStrLn $ drawVerticalForest [makeTree e1, makeTree e2]
    let c1 = canonify e1
        c2 = canonify e2
      in if c1 == e1 && c2 == e2 then putStrLn "Can't simplify no more"
                                   >> return False
                              else prettyEqual c1 c2
```

Syntactic checking of equality

```haskell
equal :: FunExpr -> FunExpr -> Bool
equal e1 e2 = (e1 == e2) ||
                 (let c1 = canonify e1
                      c2 = canonify e2
                    in (not (e1 == c1 && c2 == e2) && equal c1 c2))
```

Parse an expression as a Tree of Strings

Of course this is all bit too verbose, but I’m keeping it that way until every case is covered, Calculus is a bit of a black box for me right now

```haskell
canonify :: FunExpr -> FunExpr
```

Addition

```haskell
canonify (e :+ Const 0)        = canonify e
canonify (Const 0 :+ e)        = canonify e
canonify (Const x :+ Const y) = Const (x + y)
canonify (e1 :+ e2)            = canonify e1 :+ canonify e2
```

Subtraction

```haskell
canonify (e :- Const 0)        = canonify e
canonify (Const a :- Const b) = Const (a - b)
canonify (e1 :- e2)            = canonify e1 :- canonify e2
```

Multiplication

```haskell
canonify (_ :* Const 0)        = Const 0
canonify (Const 0 :* _)        = Const 0
canonify (e :* Const 1)        = canonify e
canonify (Const 1 :* e)        = canonify e
canonify (Const a :* Const b) = Const (a * b)
canonify (e1 :* e2)            = canonify e1 :* canonify e2
```

Division

```haskell
canonify (Const a :/ Const b) = Const (a / b)
canonify (e1 :/ e2)            = canonify e1 :/ canonify e2
```

Delta

```haskell
canonify (Delta r e)             = Delta r $ canonify e
```

Derivatives

```haskell
canonify (D e)                 = derive e
```

Composition

```haskell
canonify (e1 :. e2)            = canonify e1 :. canonify e2
```

Catch all

```haskell
canonify e                     = e
```

“Proofs”

```haskell
syntacticProofOfComForMultiplication :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForMultiplication e1 e2 = prettyEqual (e1 :* e2) (e2 :* e1)


syntacticProofOfAssocForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :* e3))
                                                              ((e1 :* e2) :* e3)


syntacticProofOfDistForMultiplication :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfDistForMultiplication e1 e2 e3 = prettyEqual (e1 :* (e2 :+ e3))
                                                              ((e1 :* e2) :+ (e1 :*
  e3))


{- syntacticProofOfIdentityForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfIdentityForMultiplication e = -}
{-   putStrLn "[*] Checking right identity" >> -}
{-     prettyEqual e (1 :* e) >> -}
{-       putStrLn "[*] Checking left identity" >> -}
{-         prettyEqual e (e :* 1) -}


{- syntacticProofOfPropertyOf0ForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOf0ForMultiplication e = -}
{-   prettyEqual (e :* 0) 0 -}


-- | Fails since default implementation of negate x for Num is 0 - x
{- syntacticProofOfPropertyOfNegationForMultiplication :: FunExpr -> IO Bool -}
{- syntacticProofOfPropertyOfNegationForMultiplication e = -}
{-   prettyEqual (Const (-1) :* e)  (negate e) -}


syntacticProofOfComForAddition :: FunExpr -> FunExpr -> IO Bool
syntacticProofOfComForAddition e1 e2 = prettyEqual (e1 :+ e2) (e2 :+ e1)


syntacticProofOfAssocForAddition :: FunExpr -> FunExpr -> FunExpr -> IO Bool
syntacticProofOfAssocForAddition e1 e2 e3 = prettyEqual (e1 :+ (e2 :+ e3))
                                                        ((e1 :+ e2) :+ e3)


test :: FunExpr -> FunExpr -> IO Bool
test b c = prettyEqual b (a :* c)
```

```haskell
  where
    a = b :/ c
```

```haskell
syntacticProofOfIdentityForAddition :: FunExpr -> IO Bool
syntacticProofOfIdentityForAddition e = putStrLn "[*] Checking right identity" >>
  prettyEqual e (0 :+ e) >>
    putStrLn "[*] Checking left identity" >>
      prettyEqual e (e :+ 0)
```

Dummy expressions

```haskell
e1 = Const 1
e2 = Const 2
e3 = Const 3
e4 = (Const 1 :+ Const 2) :* (Const 3 :+ Const 4)
e5 = (Const 1 :+ Const 2) :* (Const 4 :+ Const 3)
e6 = Const 2 :+ Const 3 :* Const 8 :* Const 19
```