List 3

wtorek, 4 stycznia 2022 21:27

Ex. 4

Ex. 4

$$\text{let } x = y \text{ in } z$$

$$(\lambda x \to z)\, y$$

$$\text{let } x_1 = y_1;\ x_2 = y_2;\ \dots;\ x_n = y_n \text{ in } z \;==\; (\lambda x_1 x_2 \dots x_n \to z)\, y_1 y_2 \dots y_n$$

Ex. 7

```haskell
data Point = Point Float Float
data Shape = Circle Point Float | Rectangle Point Point

surf :: Shape -> Float
surf (Circle _ r) = pi * r ^ 2
surf (Rectangle (Point x y) (Point x2 y2))= abs(x2-x) * abs(y2-y)
```

Ex. 8

```haskell
data Vector3D a = Vector3D a a a -- deriving (Show)

add :: (Num a) => Vector3D a -> Vector3D a -> Vector3D a
add (Vector3D x1 x2 x3) (Vector3D y1 y2 y3) = Vector3D (x1+y1) (x2+y2) (x3+y3)

scalarProduct :: (Num a) => Vector3D a -> Vector3D a -> a
scalarProduct (Vector3D x1 x2 x3) (Vector3D y1 y2 y3) = x1*y1+x2*y2+x3*y3

mult :: (Num a) => a -> Vector3D a -> Vector3D a
mult s (Vector3D x1 x2 x3) = Vector3D (s*x1) (s*x2) (s*x3)


instance Show a => Show (Vector3D a) where
  show (Vector3D x1 x2 x3) = "V = ["++ show x1++", "++ show x2++", "++ show x3++"]"
```

Ex. 10

```haskell
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a) -- deriving (Show)
```

functor

```haskell
instance Functor Tree where
  fmap = treeMap

treeMap :: (a -> b) -> (Tree a) -> (Tree b)
treeMap _ Empty = Empty
treeMap f (Leaf a) = Leaf (f a)
treeMap f (Node left a right) = Node (treeMap f left) (f a) (treeMap f right)
```

foldable

```haskell
instance Foldable Tree where
    foldr = treeFoldr
    foldl = treeFoldl

treeFoldr :: (a -> b -> b) -> b -> Tree a -> b
treeFoldr f z Empty = z
treeFoldr f z (Leaf a) = f a z
```

```haskell
instance Foldable Tree where
    foldr = treeFoldr
    foldl = treeFoldl

treeFoldr :: (a -> b -> b) -> b -> Tree a -> b
treeFoldr f z Empty = z
treeFoldr f z (Leaf a) = f a z
treeFoldr f z (Node left a right) = treeFoldr f (f a (treeFoldr f z right)) left


treeFoldl :: (a -> b -> a) -> a -> Tree b -> a
treeFoldl f z Empty = z
treeFoldl f z (Leaf b) = f z b
treeFoldl f z (Node left b right) = treeFoldl f ( f (treeFoldl f z left) b) right
```

is in tree

```haskell
isInTree :: (Eq a) => a -> Tree a -> Bool
isInTree x = treeFoldr (\y acc -> (y == x || acc)) False
```

counts

```haskell
countNodes :: Tree a -> Int
countNodes tr = treeFoldr (+) 0 $ treeMap (\x -> 1) tr


countLeaves :: Tree a -> Int
countLeaves (Leaf _) = 1
countLeaves (Node left a right) = (countLeaves left) + (countLeaves right)
```

height

```haskell
treeHeightR :: Tree a -> Int
treeHeightR Empty = 0
treeHeightR (Leaf a) = 1
treeHeightR (Node left _ right) = 1 + max (treeHeightR left) (treeHeightR right)
```

Ex. 11

```haskell
data Rose_tree a = Empty | Leaf a | Node a [Rose_tree a] deriving
(Show, Eq)
```

functor

```haskell
instance Functor Rose_tree where
  fmap _ Empty = Empty
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node x xs) = Node (f x) (map (fmap f) xs)
```

foldable

```haskell
instance Foldable Rose_tree where
  foldl _ z Empty = z
  foldl f z (Leaf x) = f z x
  foldl f z (Node x xs) = f temp_z x where temp_z = foldl (foldl
   f) z xs

  foldr _ z Empty = z
  foldr f z (Leaf x) = f x z
  foldr f z (Node x xs) = f x temp_z where temp_z = foldr (\x y
  -> (foldr f y x)) z xs
```

count

```haskell
count_el :: Rose_tree a -> Int

count_el t = foldl (\x y -> (x+1)) 0 t
```