HASKEL

```haskell
greet :: String -> String
-- greet :: [Char] -> [Char]
greet name = "Hello " ++ name

howLong :: [a] -> Integer
howLong [] = 0
howLong (x:xs) = 1 + howLong xs

-- UNION TYPES
-- 'Food' => Type Constructor
-- 'Fruit', 'Dairy', ... => Data Constructors
data Food = Fruit | Dairy | Fishb | MetaData

-- PRODUCT TYPES
data Point2D a = Point2D a a

manhattanDistance
    (Point2D x0 y0)
    (Point2D x1 y1) = (x1 - x0) + (y1 - y0)

p1 = Point2D 0 0
p2 = Point2D 1 1

data Point2D' a = Point2D' {pointX, pointY :: a}

p3 = Point2D' 1 2
-- It gives us two 'selector functions':
-- pointX p3 -> 1
-- pointY p3 -> 2

showJustX (Point2D' x1 y1) = pointX (Point2D' x1
y1)

getJustX Point2D' {pointX = x} = x


-- RECURSIVE TYPES

data Tree a = Leaf a | Branch (Tree a) (Tree a)

myTree = Branch (Leaf "a")
                (Branch (Leaf "b") (Leaf "c"))

-- SYNONYM TYPES

type MyString = [Char]

name :: MyString
name = "Yuliya"

lst :: [Integer]
lst = (1 : (2 : (3 : [])))

-- Partial functions
```

```haskell
add1 = (1 +)
double = (2 *)

myList = [1,2,3,4,5]

comp = add1 . double
-- map (add1 . double) myList => [3,5,7,9,11]
-- functions are applied from RIGHT to LEFT:
-- map (double . add1) myList => [4,6,8,10,12]

-- LET for local scope variables
cylinderArea r h =
  let sideArea = 2 * pi * r * h
      topArea = pi * r ^ 2
   in sideArea + 2 * topArea

-- sum $ map (\x -> double x) myList
-- sum (map (\x -> double x) myList)

-- '@' DESTRUCTURING in pattern matching
-- it keeps the reference to the whole list,
-- the head, and the rest.

showMyList lst@(x : xs) =
    "Head: " ++ show x
    ++ " Rest: " ++ show xs
    ++ " List: " ++ show lst


infList = [1 ..]
finiteList = [1, 3 .. 15]
take5 = take 5 infList

-- List comprehension
evenNums = [x * 2 | x <- [0 ..]]

-- a^2 + b^2 = c^2
pytagorianTriples = [(a, b, c)
    | c <- [1 ..],
    b <- [1 .. c],
    a <- [1 .. b],
    a ^ 2 + b ^ 2 == c ^ 2]


-- ZIP : takes in parallel one element from each
of two lists
-- and makes a pair out of them
-- zip [1, 2, 3, 4, 5] "ciao"
-- [(1,'c'),(2,'i'),(3,'a'),(4,'o')]

-- BOOLEAN GUARD
calcBMI weight height
    | bmi <= underweight = "Underweight"
    | bmi < normal = "Normal"
    | bmi < overweight = "Overweight"
    | otherwise = "Obese"
    where
        bmi = weight / height ^ 2
        -- using pattern matching for assignment
        (underweight, normal, overweight) =
(18.5, 25.0, 30.0)


initials firstName lastName = [f] ++ " " ++ [l]
    where
        (f) = head firstName
        (l) = head lastName

initials' firstName lastName = [f] ++ " " ++ [l]
    where
        (f : _) = firstName
        (l : _) = lastName

-- IF
ifExample x = [1, if x > 2 then 999 else 0]

-- "CASE"

head' :: [a] -> a
head' [] = error "Empty list"
head' (x : _ ) = x

head'' lst = case lst of
    [] -> error "No head"
    (x : xs) -> x

-- Forces evaluation of x
-- seq x y -> y
```

```haskell
-- myFunc :: Num a => a -> a -> a

-- class Equal a where
--     (==) :: a -> a -> Bool
--     x /= y = not (x == y)

data Tree a = Empty | Leaf a | Branch (Tree a)
(Tree a) deriving (Eq, Show)

-- instance (Eq a) => Eq (Tree a) where
--     Leaf a == Leaf b = a == b
--     Branch l1 r1 == Branch l2 r2 = l1 == l2
&& r1 == r2
--     _ == _ = False

-- SUBCLASSES can be defined as:
-- class Eq a => Ord a where
--     (<), (<=), (>), (>=) :: a -> a -> Bool
--     min, max :: a -> a -> a

-- For custom SHOW create an instance
-- instance (Show a) => Show (Tree a) where
--     show (Leaf a) = show a
```

```
--     show (Branch l r) = "<" ++ show l ++ "|"
++ show r ++ ">"

-- FOLDABLE
-- FOLD <binary function> <accumulator> <data>
treeFoldr f z Empty = z
treeFoldr f z (Leaf x) = f x z
treeFoldr f z (Branch l r) = treeFoldr f
(treeFoldr f z r) l

instance Foldable Tree where
    foldr :: (a -> b -> b) -> b -> Tree a -> b
    foldr = treeFoldr

-- `foldl` can be expressed in terms of `foldr`
-- as `foldr` can work on infinite lists, while
-- `foldl` cannot.
foldl f a bs = foldr (\b g x -> g(f x b)) id bs
a

-- instance Foldable Maybe where
--     foldr _ z Nothing = z
--     foldr f z (Just x) = f x z

data Result a = Err | Ok a deriving (Eq, Ord,
Show)

safediv :: Int -> Int -> Result Int
safediv n m =
    if m == 0
        then Err
        else Ok (n `div` m)

-- FUNCTOR
instance Functor Tree where
    fmap :: (a -> b) -> Tree a -> Tree b
    fmap _ Empty = Empty
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch l r) = Branch (fmap f l)
(fmap f r)

instance Functor Result where
    fmap :: (a -> b) -> Result a -> Result b
    fmap _ Err = Err
    fmap f (Ok x) = Ok (f x)
-- we can call fmap with an infix notation as "f
<$> t"

-- APPLICATIVE
instance Applicative Result where
    (<*>) :: Result (a -> b) -> Result a ->
Result b
    pure x = Ok x -- wraps an argument in our
structure
    _ <*> Err = Err
    Err <*> _ = Err
    (Ok f) <*> (Ok x) = Ok (f x)
    -- (Ok f) <*> x = f <$> x

-- instance Applicative Tree where
--     (<*>) :: Tree (a -> b) -> Tree a -> Tree
b
--     pure x = Leaf x
--     _ <*> Empty = Empty
--     Empty <*> _ = Empty
--     (Leaf f) <*> (Leaf x) = Leaf (f x)
--     (Leaf f) <*> (Branch l r) = Branch (Leaf
f <*> l) (Leaf f <*> r)
--     (Branch f g) <*> (Leaf x) = Branch (f <*>
Leaf x) (g <*> Leaf x)
--     (Branch f g) <*> (Branch l r) = Branch (f
<*> l) (g <*> r)

-- Tree concatenation
tconc Empty t = t
tconc t Empty = t
tconc t1 t2 = Branch t1 t2

tconcat t = foldr tconc Empty t
tconcmap f t = tconcat (fmap f t)

-- for lists concatMap f l = concat (map f l)

instance Applicative Tree where
    pure x = Leaf x
    (<*>) :: Tree (a -> b) -> Tree a -> Tree b
    fs <*> xs = tconcmap (\f -> fmap f xs) fs


-- instance Applicative [] where
--     pure x = [x]
--     fs <*> xs = concatMap (\f -> map f xs) fs

{--
APPLICATIVE RULES:
pure id <*> v = v                             --
Identity
pure f <*> pure x = pure (f x)                --
Homomorphism
u <*> pure y = pure ($ y) <*> u               --
Interchange
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) --
Composition
--}
```

```
data Result a = Ok a | Err deriving (Eq, Show)

data Expr = Val Int | Div Expr Expr deriving
(Eq, Show)

eval :: Expr -> Int
eval (Val n) = n
eval (Div x y) = eval x `div` eval y

ex1 = Div (Val 4) (Val  2)
ex2 = Div (Val 4) (Val  0)

safeDiv :: Int -> Int -> Result Int
safeDiv n m =
    if m == 0
        then Err
        else Ok (n `div` m)

eval' :: Expr -> Result Int
eval' (Val n) = Ok n
eval' (Div x y) =
    case eval' x of
        Err -> Err
        Ok x -> case eval' y of
            Err -> Err
            Ok y -> safeDiv x y
-- eval' (Div x y) = eval x `safeDiv` eval y

bind :: Result Int -> (Int -> Result Int) ->
Result Int
m `bind` f = case m of
    Err -> Err
    Ok x -> f x

eval'' :: Expr -> Result Int
eval'' (Val n) = Ok n
eval'' (Div x y) =
    eval'' x `bind` (\n -> eval'' y `bind` \m ->
safeDiv n m)

mEval :: Expr -> Result Int
mEval (Val n) = Ok n
mEval (Div x y) = do
    n <- mEval x
    m <- mEval y
    safeDiv n m

instance Functor Result where
    fmap f (Ok x) = Ok (f x)
    fmap _ Err = Err

instance Applicative Result where
    pure = Ok
    _ <*> Err = Err
    Err <*> _ = Err
    Ok f <*> Ok x = fmap f (Ok x)

instance Monad Result where
    return = pure
    Err >>= _ = Err
    Ok x >>= f = f x

{--
1. LEFT IDENTITY: `return x >>= f` == `f x`
```

```
2. RIGHT IDENTITY: `m >>= return` == `m`
3. ASSOCIATIVITY: `(m >>= f) >>= g` ==
                  `m >>= (\x -> f x >>= g)
--}

----------------

square x = x ^ 2
addOne x = x + 1

x = addOne (square 2)

data NumberWithLogs = NumberWithLogs
    {
    number :: Int,
    logs :: [String]
    }
    deriving (Eq, Show)

square1 :: Int -> NumberWithLogs
square1 x = NumberWithLogs (x^2) ["Squared " ++
show x ++ " to get " ++ show (x^2)]

square2 :: NumberWithLogs -> NumberWithLogs
square2 x = NumberWithLogs (number x ^ 2) $ logs
x ++ ["Squared " ++ show (number x) ++ " to get
" ++ show (number x ^ 2)]

addOne1 :: Int -> NumberWithLogs
addOne1 x = NumberWithLogs (x + 1) ["Added 1 to
" ++ show x ++ " to get " ++ show (x + 1)]

addOne2 :: NumberWithLogs -> NumberWithLogs
addOne2 x = NumberWithLogs (number x + 1) (logs
x ++ ["Added 1 to " ++ show (number x) ++ " to
get " ++ show (number x + 1)])


wrapWithLogs :: Int -> NumberWithLogs
wrapWithLogs x = NumberWithLogs x []

runWithLogs :: NumberWithLogs -> (Int ->
NumberWithLogs) -> NumberWithLogs
runWithLogs input transform =
    let newNumberWithLogs = transform (number
input)
        in NumberWithLogs
            (number newNumberWithLogs)
            (logs input ++ logs
newNumberWithLogs)

a = wrapWithLogs 2
b = runWithLogs a square1
c = b `runWithLogs` addOne1
```