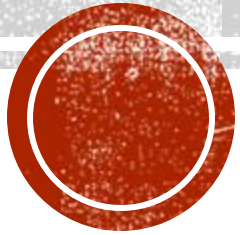
FUNCTIONAL PROGRAMMING



TYPE DECLARATIONS

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

```
type Pos = (Int,Int)
```

we can define:

```
origin    :: Pos
origin     = (0,0)
```

```
left        :: Pos -> Pos
left (x,y) = (x-1,y)
```


Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult       :: Pair Int -> Int
mult (m,n) = m*n
```

```
copy       :: a -> Pair a
copy x      = (x,x)
```


Type declarations can be nested:

```
type Pos   = (Int,Int)

type Trans = Pos -> Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

DATA DECLARATIONS

A completely new type can be defined by specifying its values using a data declaration.

```
data Bool = False | True
```

Bool is a new type, with two new values False and True.

Note:

- The two values False and True are called the constructors for the type Bool.
- Type and constructor names must begin with an upper-case letter.

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float
           | Rect Float Float
```

we can define:

```
area              :: Shape -> Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```


RECURSIVE TYPES

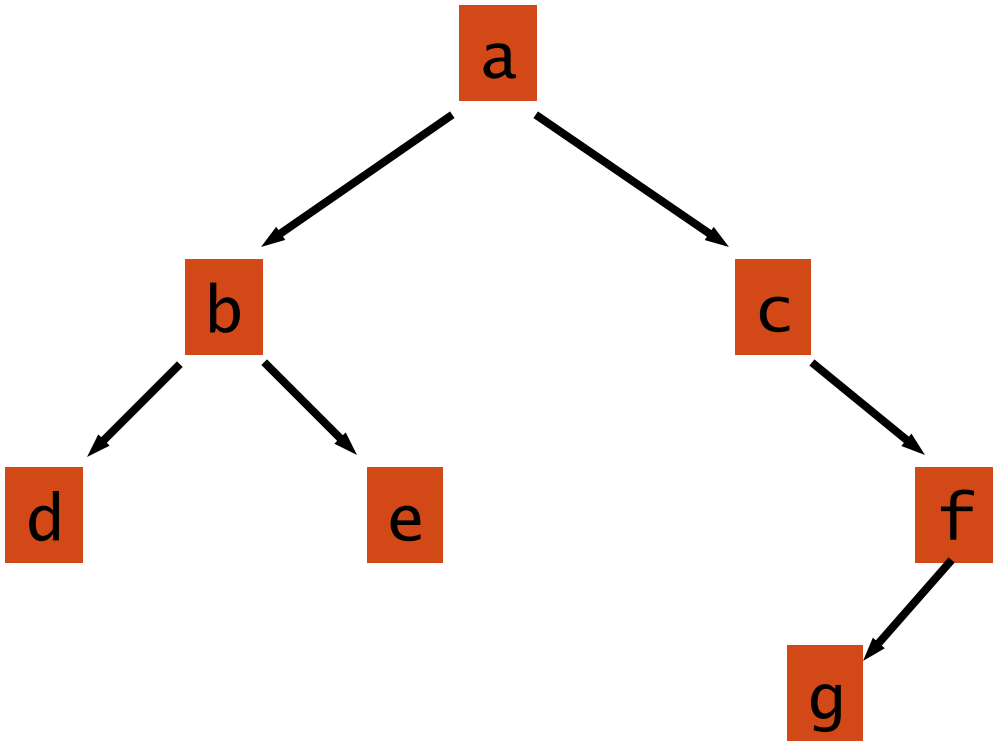
In Haskell, new types can be declared in terms of themselves. That is, types can be recursive.

```
data Expr = Num Integer
          | Add Expr Expr
          | Mul Expr Expr
```

Expression	Haskell representation
2	`Num 2`
2+2	`Add (Num 2) (Num 2)`
(1+2)*3	`Mul (Add (Num 1) (Num 2)) (Num 3)`
1+2*3	`Add (Num 1) (Mul (Num 2) (Num 3))`

BINARY TREES

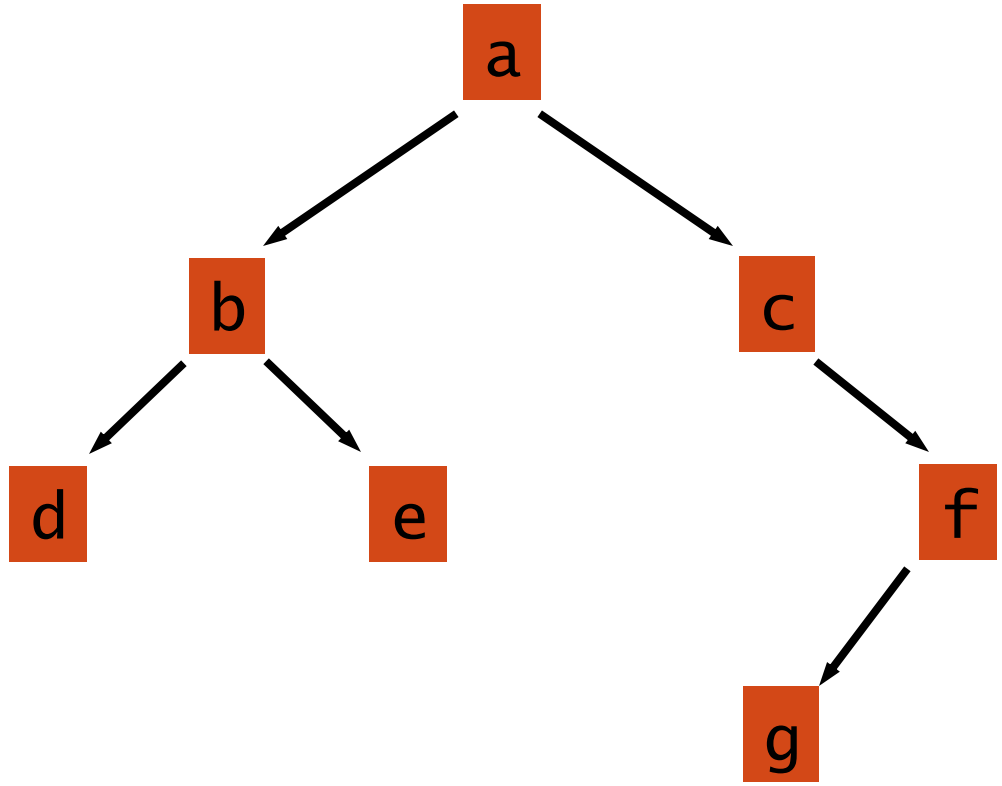
In computing, it is often useful to store data in a two-way branching structure or binary tree.



Using recursion, a suitable new type to represent such binary trees can be declared by:

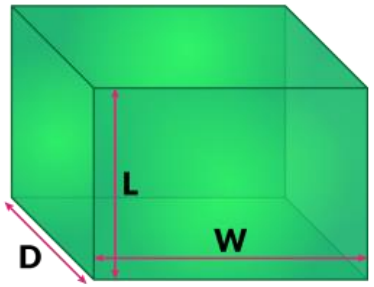
```
data Tree a = Empty | Branch a (Tree a) (Tree a)
              deriving (Show, Eq)
```

```
tree1 = Branch 'a' (Branch 'b'
(Branch 'd' Empty Empty)

(Branch 'e' Empty Empty))
                    (Branch 'c'
Empty

(Branch 'f' (Branch 'g' Empty
Empty)

Empty))
```



PRACTICE

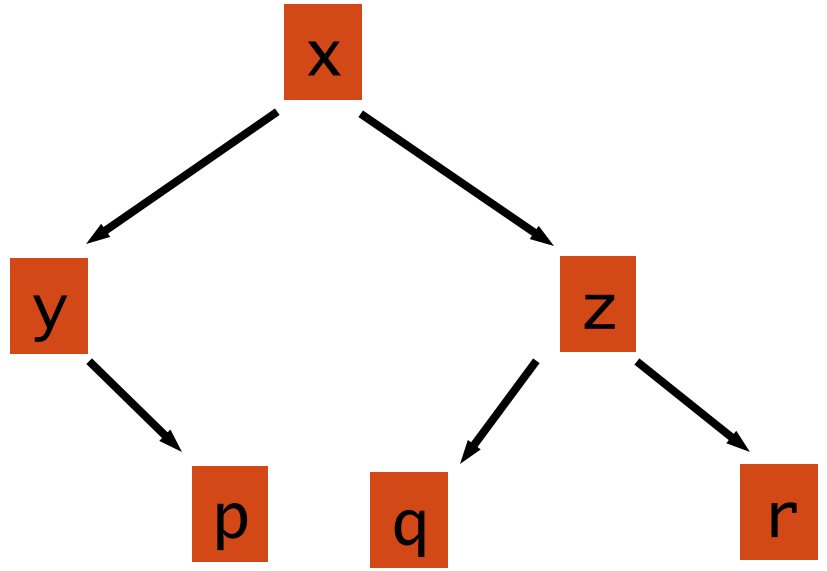
(1) Write a program which declares a **data type named Box** in Haskell for finding the surface area and volume of a box.



Surface Area of a Box = 2(L × W) + 2(L × D) + 2(W × D)

Volume of a Box = L × W × D

(2) Write a program in Haskell which calculates the following expression:
(1+2)/(9*2)+56*3.4
Using **recursive data type** named Expr.



(3) Write a program in Haskell which represents the tree in above figure.

ASSIGNMENT-2 (DEADLINE 10.12.2019)

(1) Write a program in Haskell which calculates the sum of all odd numbers in a given range.

(2) Write a program in Haskell to input electricity unit charge and calculate the total electricity bill according to the given condition:
For first 50 units Rs. 0.50/unit
For next 100 units Rs. 0.75/unit
For next 100 units Rs. 1.20/unit
For unit above 250 Rs. 1.50/unit
An additional surcharge of 20% is added to the bill.