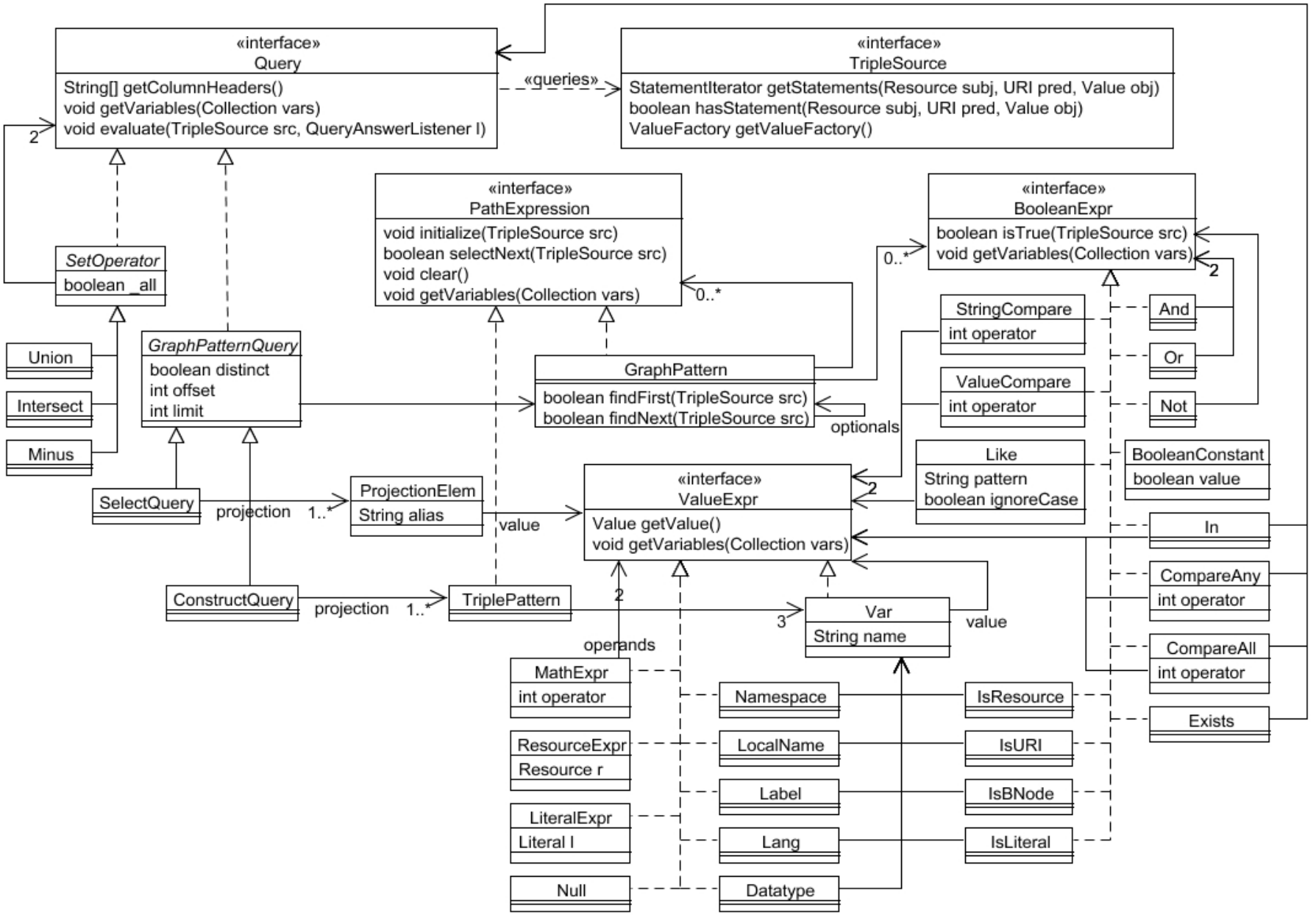
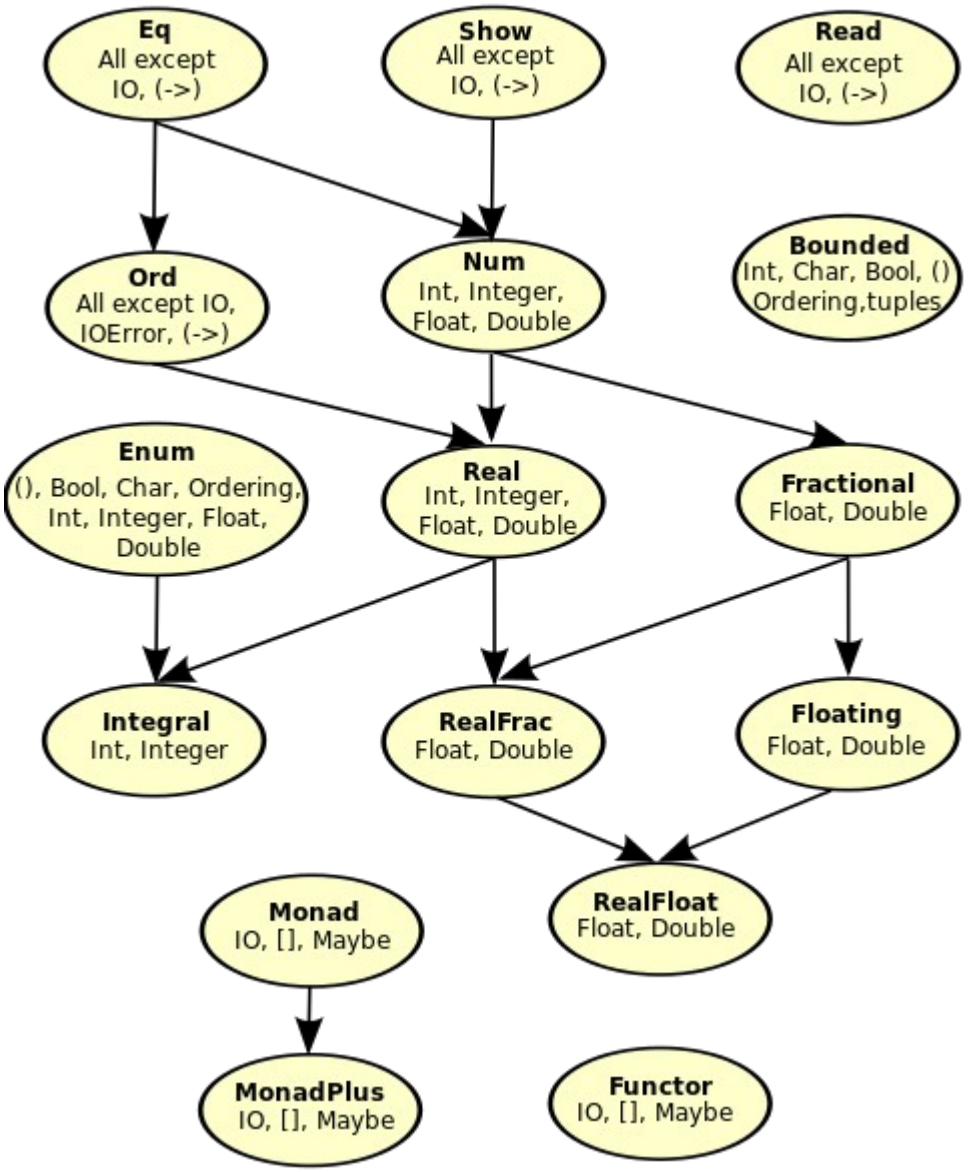
# Типы


«interface»
Query
String[] getColumnHeaders()
void getVariables(Collection vars)
void evaluate(TripleSource src, QueryAnswerListener l)
«queries»
«interface»
TripleSource
StatementIterator getStatements(Resource subj, URI pred, Value obj)
boolean hasStatement(Resource subj, URI pred, Value obj)
ValueFactory getValueFactory()
2
SetOperator
boolean _all
«interface»
PathExpression
void initialize(TripleSource src)
boolean selectNext(TripleSource src)
void clear()
void getVariables(Collection vars)
0..*
«interface»
BooleanExpr
boolean isTrue(TripleSource src)
void getVariables(Collection vars)
0..*
2
Union
Intersect
Minus
GraphPatternQuery
boolean distinct
int offset
int limit
GraphPattern
boolean findFirst(TripleSource src)
boolean findNext(TripleSource src)
optionals
StringCompare
int operator
ValueCompare
int operator
And
Or
Not
SelectQuery
projection
1..*
ProjectionElem
String alias
value
«interface»
ValueExpr
Value getValue()
void getVariables(Collection vars)
2
Like
String pattern
boolean ignoreCase
BooleanConstant
boolean value
In
CompareAny
int operator
CompareAll
int operator
Exists
ConstructQuery
projection
1..*
TriplePattern
2
3
Var
String name
value
operands
MathExpr
int operator
ResourceExpr
Resource r
LiteralExpr
Literal l
Null
Namespace
LocalName
Label
Lang
Datatype
IsResource
IsURI
IsBNode
IsLiteral

# Типы


Eq
All except
IO, (->)
Show
All except
IO, (->)
Read
All except
IO, (->)
Ord
All except IO,
IOError, (->)
Num
Int, Integer,
Float, Double
Bounded
Int, Char, Bool, ()
Ordering,tuples
Enum
(), Bool, Char, Ordering,
Int, Integer, Float,
Double
Real
Int, Integer,
Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

# Типы

```
data Anniversary =
  Birthday String Int Int Int
  | Wedding String String Int Int Int
  | Death String Int Int Int

> Birthday "someone" 2012 11 7
Birthday "someone" 2012 11 7
> let today = Birthday "someone" 2012 11 7
> today
Birthday "someone" 2012 11 7
> :t today
today :: Anniversary
```

# Типы

```
> :t Birthday
Birthday :: String -> Int -> Int -> Int ->
Anniversary

> :t Death
Birthday :: String -> Int -> Int -> Int ->
Anniversary
```

# Типы

```
kurtCobain :: Anniversary
kurtCobain = Birthday "Kurt Cobain" 1967 2 20

kurtWedding :: Anniversary
kurtWedding = Wedding "Kurt Cobain" "Courtney
Love" 1990 1 12
```

# Типы

```
anniversaries = [
   kurtCobain,
   kurtWedding,
   Death "Kurt Cobain" 1994 4 5
   ]
```

# Типы

```
type AnniversaryBook = [Anniversary]
```

```
showDate :: Int -> Int -> Int -> String
showDate y m d = show y ++ "." ++ show m ++ "."
++ show d
```

# Типы

```
showAnniversary :: Anniversary -> String

showAnniversary (Birthday name year month day) =
   name ++ " born " ++ showDate year month day

showAnniversary (Wedding name1 name2 year month day) =
   name1 ++ " married " ++ name2 ++ " on " ++ showDate
   year month day

showAnniversary (Death name year month day) =
   name ++ " dead in " ++ showDate year month day
```

# Типы

```
who :: Anniversary -> String
who (Birthday him _ _ _) = him
who (Wedding him _ _ _ _) = him
who (Death him _ _ _) = him


map who anniversaries
```

# Типы

```
showAnniversaries :: AnniversaryBook -> [String]
showAnniversaries = map showAnniversary

["Kurt Cobain born 1967-2-20","Kurt Cobain married
Courtney Love on 1990-1-12","Kurt Cobain dead 1994-4-5"]
```

1) Kurt Cobain born 1967-2-20
2) Kurt Cobain married Courtney Love on 1990-1-12
3) Kurt Cobain dead 1994-4-5

?

# Типы

```
> anniversaries
    No instance for (Show Anniversary)
      arising from a use of `print'
    Possible fix: add an instance declaration for (Show Anniversary)
    In a stmt of an interactive GHCi command: print it
```

```
data Anniversary =
  Birthday String Int Int Int
  | Wedding String String Int Int Int
  | Death String Int Int Int
  deriving (Show)
```

# Всё человечно м.б.

```
data Point = Pt {pointx, pointy :: Float}

> let myPoint = Pt {pointx = 42.0, pointy = 13.666}
```

json?

```
pointx :: Point -> Float
pointy :: Point -> Float


absPoint :: Point -> Float
absPoint p = sqrt (pointx p * pointx p + pointy p * pointy p)


absPoint (Pt {pointx = x, pointy = y}) = sqrt (x*x + y*y)
```

# Параметры типов

```
data Maybe a = Nothing | Just a
> Just "str"
Just "str"
> :t Just "str"
Just "str" :: Maybe [Char]
> Just 42
Just 42
> :t Just 42
Just 42 :: (Num t) => Maybe t
> :t Nothing
Nothing :: Maybe a
> Just 42 :: Maybe Double
Just 42.0
> Just 42 : [Nothing] ?
> Just 42 : [Just "str", Nothing] ?
```

привет, полиморфизм!
дженерики

# Рекурсивные структуры

```
data List a = Nil | Cons a (List a) deriving (Show, Read, Eq, Ord)

> 3 `Cons` (4 `Cons` (5 `Cons` Nil))
Cons 3 (Cons 4 (Cons 5 Nil))
```

# Рекурсивные структуры

```
infixr 5 :-:
data List a = Nil | a :-: (List a) deriving (Show, Read, Eq, Ord)

> let a = 3 :-: 4 :-: 5 :-: Nil
> 100 :-: a
(:-:) 100 ((:-:) 3 ((:-:) 4 ((:-:) 5 Nil)))
```

слонёнок

# Рекурсивные структуры

```
infixr 5  .++
(.++) :: List a -> List a -> List a
Nil .++ ys = ys
(x :-: xs) .++ ys = x :-: (xs .++ ys)

> let a = 3 :-: 4 :-: 5 :-: Nil
> let b = 6 :-: 7 :-: Nil
> a .++ b
(:-:) 3 ((:-:) 4 ((:-:) 5 ((:-:) 6 ((:-:) 7 Nil))))
```

# Рекурсивные структуры

```
data Tree a =
    EmptyTree
    | Node a (Tree a) (Tree a)
    deriving (Show, Read, Eq)
```

# Рекурсивные структуры

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

# W/o foldr

```
list2tree :: (Ord a) => [a] -> Tree a
list2tree = l2t EmptyTree
  where
    l2t acc [] = acc
    l2t acc (head:tail) = l2t (treeInsert head acc) tail

list2tree [12, 1, 6, 4, 90, 9]
```

# tree2list

```
tree2list :: (Ord a) => Tree a -> [a]
tree2list EmptyTree = ?
tree2list (Node val left right) = ?
```

# tree2list

```
tree2list :: (Ord a) => Tree a -> [a]
tree2list EmptyTree = []
tree2list (Node val left right) =
    (tree2list left) ++ (val : tree2list right)


treeSort :: (Ord a) => [a] -> [a]
treeSort = tree2list . list2tree

> treeSort [12, 1, 6, 4, 90, 9]
```

# tree2list

```
tree2list :: (Ord a) => Tree a -> [a]
tree2list EmptyTree = []
tree2list (Node val left right) = ?
```

```
treeSortDesc :: (Ord a) => [a] -> [a]
treeSortDesc = tree2list . list2tree
```

```
> treeSort [12, 1, 6, 4, 90, 9]
```

# Рекурсивные структуры

Кол-во вершин дерева (== кол-во чисел в нём)

treeNum :: Tree a -> Int

# Рекурсивные структуры

```
treeNum :: Tree a -> Int
treeNum EmptyTree = 0
treeNum (Node val left right) = 1 + (treeNum left) +
(treeNum right)
```

# Рекурсивные структуры

Дерево можно кодировать наборами из нулей и единиц. Рассмотрим, например, укладку дерева на плоскости. Начиная с какой либо вершины, будем двигаться по ребрам дерева, сворачивая в каждой вершине на ближайшее справа ребро и поворачивая назад в концевых вершинах дерева. Проходя по некоторому ребру, записываем 0 при движении по ребру в первый раз и 1 при движении по ребру второй раз (в обратном направлении). Если m — число рёбер дерева, то через 2m шагов мы вернемся в исходную вершину, пройдя по каждому ребру дважды. Полученная при этом последовательность из 0 и 1 (код дерева) длины 2m позволяет однозначно восстанавливать не только само дерево D, но и его укладку на плоскости.

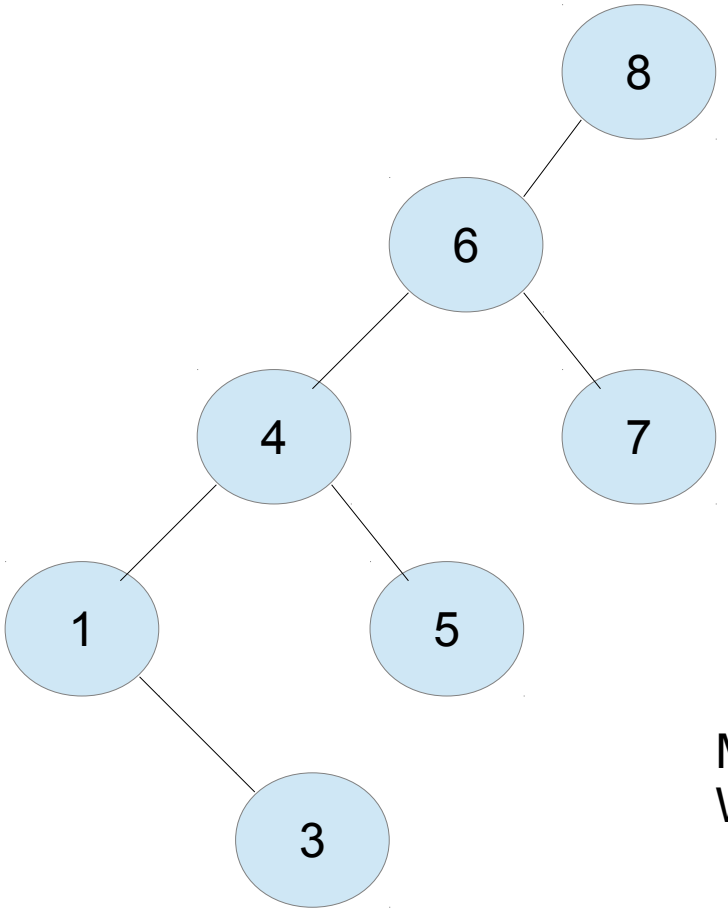
Википедия

# Рекурсивные структуры

```
treeCode :: Tree a -> [Int]
treeCode EmptyTree = []
treeCode (Node val left right) =
    ([0] ++ treeCode left ++ [1]) ++ ([0] ++ treeCode right ++ [1])

>treeCode $ list2tree [8,6,4,1,7,3,5]
[0,0,0,0,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,1]
```

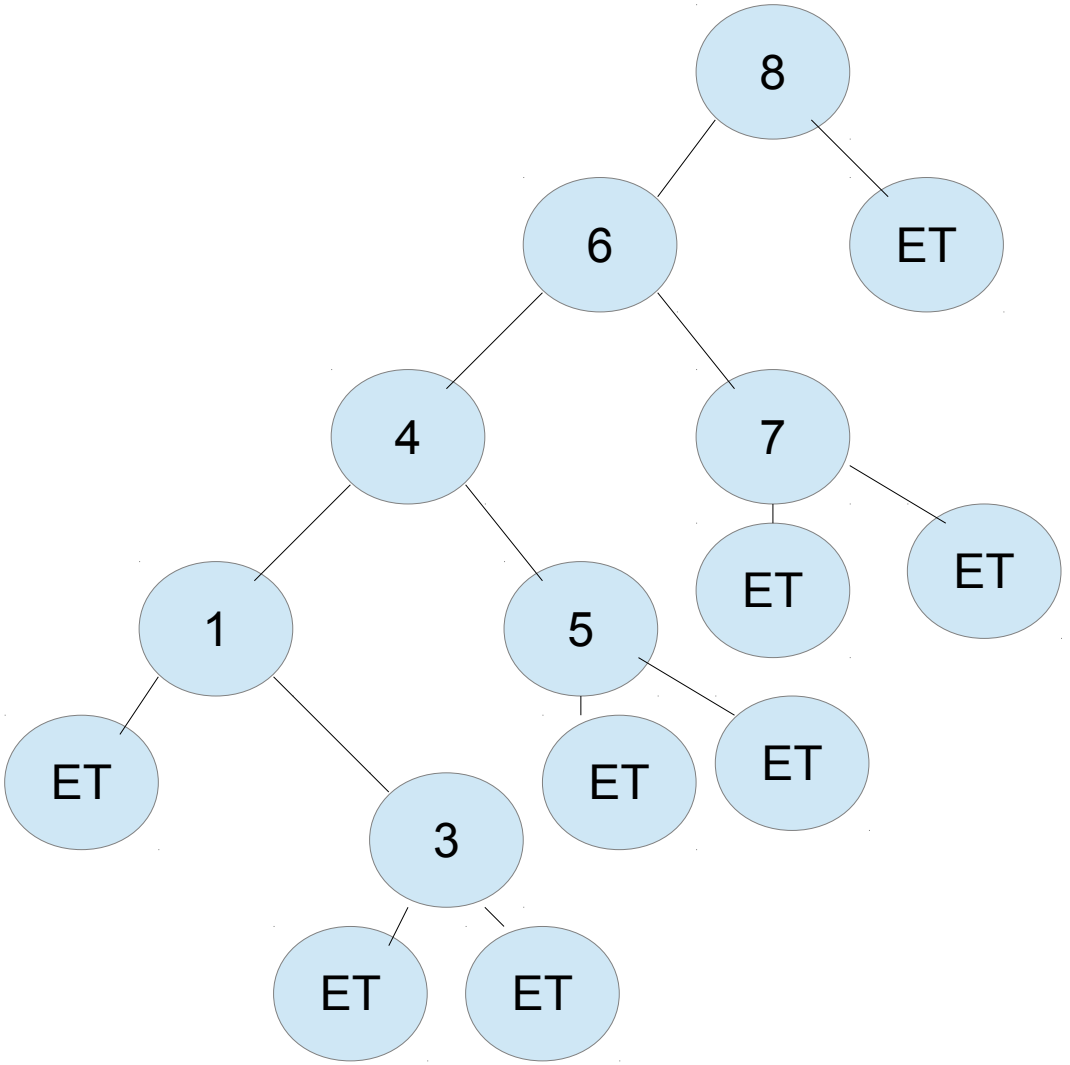
# Рекурсивные структуры




Моя функция не работает!!!111
Why?

[0,0,0,0,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,1]

# Рекурсивные структуры




[0,0,0,0,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,0,1,0,1,1,1,0,1]

# Рекурсивные структуры

list2tree [12, 12, 12, 13, 13, 14]

Как будет выглядеть дерево?

# Рекурсивные структуры

```
data Tree = ?

singleton :: a -> Tree a
singleton x = ?

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
?

list2tree :: (Ord a) => [a] -> Tree a
list2tree = ?
```

# Рекурсивные структуры

```
data Tree a = EmptyTree | Node a Int (Tree a) (Tree a) deriving (Show, Read, Eq)


singleton :: a -> Tree a
singleton x = Node x 1 EmptyTree EmptyTree


treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a i left right)
    | x == a = Node x (i+1) left right
    | x < a  = Node a i (treeInsert x left) right
    | x > a  = Node a i left (treeInsert x right)


list2tree :: (Ord a) => [a] -> Tree a
list2tree = l2t EmptyTree
  where
    l2t acc [] = acc
    l2t acc (head:tail) = l2t (treeInsert head acc) tail
```