Programming in Haskell: Lecture 21

S P Suresh

October 28, 2019

Recursive data types

- Just like we have recursive functions, we can have recursive data types

Recursive data types

- Just like we have recursive functions, we can have recursive data types
- A recursive datatype `t` is one which has some components of the same type `t`

Recursive data types

- Just like we have recursive functions, we can have recursive data types
- A recursive datatype `t` is one which has some components of the same type `t`
- Some constructors of a recursive data type `t` have `t` among their input types, as well as the return type

Example: `Nat`

- Simplest recursive data type

```
data Nat = Zero | Succ Nat


Zero :: Nat
Succ :: Nat -> Nat
```

Functions on Nat

- Check for zero:

```
isZero :: Nat -> Bool
isZero Zero = True
isZero _    = False
```

Functions on Nat

- Check for zero:

```
isZero :: Nat -> Bool
isZero Zero = True
isZero _    = False
```

- Predecessor:

```
pred :: Nat -> Nat
pred Zero     = Zero
pred (Succ n) = n
```

Functions on Nat

- Addition:

```
plus :: Nat -> Nat -> Nat
plus m Zero     = m
plus m (Succ n) = Succ (plus m n)
```

Functions on Nat

- Addition:

```
plus :: Nat -> Nat -> Nat
plus m Zero     = m
plus m (Succ n) = Succ (plus m n)
```

- Multiplication:

```
mult :: Nat -> Nat -> Nat
mult m Zero     = Zero
mult m (Succ n) = plus m (mult m n)
```

Showing Nat

- A custom **show** for Nat:

```
data Nat = Zero | Succ Nat
instance Show Nat where
    show = show . turnToInt
turnToInt :: Nat -> Int
turnToInt Zero = 0
turnToInt (Succ n) = turnToInt n + 1
```

Showing Nat

- A custom `show` for `Nat`:

```
data Nat = Zero | Succ Nat
instance Show Nat where
    show = show . turnToInt
turnToInt :: Nat -> Int
turnToInt Zero = 0
turnToInt (Succ n) = turnToInt n + 1
```

- In `show = show . turnToInt`

Showing Nat

- A custom **show** for Nat:

```
data Nat = Zero | Succ Nat
instance Show Nat where
    show = show . turnToInt
turnToInt :: Nat -> Int
turnToInt Zero = 0
turnToInt (Succ n) = turnToInt n + 1
```

- In `show = show . turnToInt`
  - The left **show** has type `Nat -> String`

Showing Nat

- A custom **show** for Nat:

```
data Nat = Zero | Succ Nat
instance Show Nat where
    show = show . turnToInt
turnToInt :: Nat -> Int
turnToInt Zero = 0
turnToInt (Succ n) = turnToInt n + 1
```

- In `show = show . turnToInt`
    - The left **show** has type `Nat -> String`
    - The left **show** has type `Int -> String`

Example: `List`

- Recursive data types can also be polymorphic

```
List a = Nil | Cons a (List a)
```

Example: `List`

- Recursive data types can also be polymorphic

```
List a = Nil | Cons a (List a)
```

- This is the built-in type `[a]`

Example: List

- Recursive data types can also be polymorphic

```
List a = Nil | Cons a (List a)
```

- This is the built-in type [a]
- Functions are defined as usual on pattern matching:

```
head :: List a -> a
head (Cons x _) = x
```

Example: `List`

- Recursive data types can also be polymorphic

```
List a = Nil | Cons a (List a)
```

- This is the built-in type `[a]`
- Functions are defined as usual on pattern matching:

```
head :: List a -> a
head (Cons x _) = x
```

- Exception on `head Nil`

List

- **List** and **head**

```
List a = Nil | Cons a (List a)
head :: List a -> a
head (Cons x _) = x
```

List

- **List** and **head**

```
List a = Nil | Cons a (List a)
head :: List a -> a
head (Cons x _) = x
```

- Exception on **head** Nil

List

- `List` and `head`

```
List a = Nil | Cons a (List a)
head :: List a -> a
head (Cons x _) = x
```

- Exception on `head Nil`
- Can fix it with custom `head`

```
head :: List a -> Maybe a
head Nil          = Nothing
head (Cons x _) = Just x
```

Binary trees

- A binary tree data structure is defined as follows:

Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree

Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
- A node containing an element with left and right subtrees is a binary tree

Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
- A node containing an element with left and right subtrees is a binary tree
- Type constructor `BTree`

```
data BTree a = Nil
             | Node (BTree a) a (BTree a)
```

Binary trees

- A binary tree data structure is defined as follows:
- The empty tree is a binary tree
- A node containing an element with left and right subtrees is a binary tree
- Type constructor `BTree`

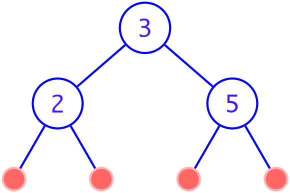
```
data BTree a = Nil
             | Node (BTree a) a (BTree a)
```

- Two value constructors:

```
Nil :: BTree a
Node :: BTree a -> a -> BTree a -> BTree a
```

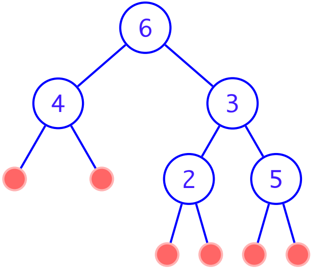
Binary trees

```
Node (Node Nil 2 Nil) 3
     (Node Nil 5 Nil)
```



Binary trees

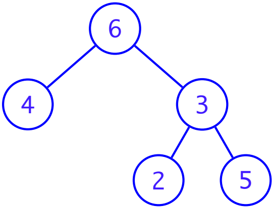
```
Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil) 3
           (Node Nil 5 Nil))
```



Binary trees

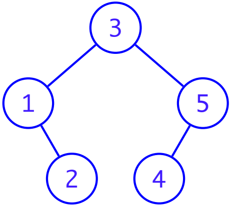
```
Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil) 3
           (Node Nil 5 Nil))
```

- We omit nodes representing `Nil` usually

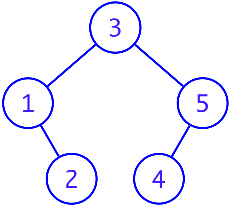


Binary trees

- Yet another binary tree



- Yet another binary tree



- Corresponding `BTree`

```
Node
    (Node Nil 1 (Node Nil 2 Nil))
3
    (Node (Node Nil 4 Nil) 5 Nil)
```

Functions on binary trees

- Number of nodes in a tree

```
size :: BTree a -> Int
size Nil             = 0
size (Node tl x tr) = 1 + size tl + size tr
```

Functions on binary trees

- Number of nodes in a tree

```
size :: BTree a -> Int
size Nil            = 0
size (Node tl x tr) = 1 + size tl + size tr
```

- **Height**: number of nodes on longest path from root

```
height :: BTree a -> Int
height Nil            = 0
height (Node tl x tr) = 1 + max (height tl) (height tr)
```

Functions on binary trees

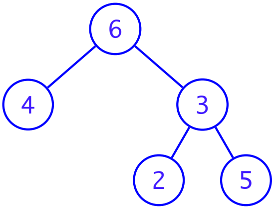
- Reflect the tree on its “vertical axis”

Functions on binary trees

- Reflect the tree on its “vertical axis”

Functions on binary trees

- Reflect the tree on its “vertical axis”



Functions on binary trees

- Reflect the tree on its "vertical axis"

Functions on binary trees

- Reflect the tree on its "vertical axis"

- Haskell code:

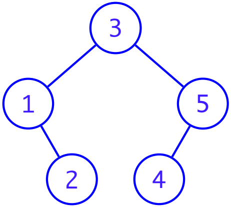
```
reflect :: BTree a -> BTree a
reflect Nil              = Nil
reflect (Node tl x tr) = Node (reflect tr) x (reflect tl)
```

Functions on binary trees

- `levels` – List nodes level by level and from left to right inside each level

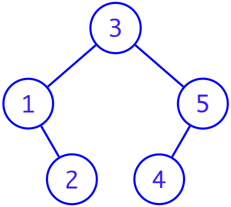
Functions on binary trees

- `levels` – List nodes level by level and from left to right inside each level
- Let `t` be the tree below:



Functions on binary trees

- `levels` – List nodes level by level and from left to right inside each level
- Let `t` be the tree below:



- `levels t = [3,1,5,2,4]`

Functions on binary trees

- `levels`

```
levels :: BTree a -> [a]
levels = concat . levels'
levels' :: BTree a -> [[a]]
levels' Nil                = []
levels' (Node tl x tr) = [x]:join (levels' tl)
                                   (levels' tr)
join :: [[a]] -> [[a]] -> [[a]]
join [] yss                = yss
join xss []                = xss
join (xs:xss) (ys:yss) = xs++ys: join xss yss
```

Creating a binary tree

- Create a binary tree from a list

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible

Creating a binary tree

- Create a binary tree from a list
- As balanced as possible
- Strategy:

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
    - Split the list in two halves

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
  - Split the list in two halves
  - Recursively create a binary tree from each half

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
  - Split the list in two halves
  - Recursively create a binary tree from each half
  - Join them together

Creating a binary tree

- Creating a balanced tree from a list

```
createTree :: [a] -> BTree a
createTree [] = Nil
createTree xs = Node
                (createTree front) x (createTree back)
    where
        n = length xs
        (front, x:back) = splitAt (n `div` 2) xs

levels (createTree [0..14]) =
              [7,3,11,1,5,9,13,0,2,4,6,8,10,12,14]
height (createTree [0..14]) = 4
```

Showing a binary tree

- To be able to show a binary tree, we need to derive a **Show** instance

```
data BTree a = Nil | Node (BTree a) a (BTree a)
    deriving Show

createTree [0..14] =
Node (Node (Node (Node Nil 0 Nil) 1 (Node Nil 2 Nil))
     3 (Node (Node Nil 4 Nil) 5 (Node Nil 6 Nil)))
7 (Node (Node (Node Nil 8 Nil) 9 (Node Nil 10 Nil))
     11 (Node (Node Nil 12 Nil) 13 (Node Nil 14 Nil)))
```

Showing a binary tree

- To be able to show a binary tree, we need to derive a **Show** instance

```
data BTree a = Nil | Node (BTree a) a (BTree a)
    deriving Show

createTree [0..14] =
Node (Node (Node (Node Nil 0 Nil) 1 (Node Nil 2 Nil))
     3 (Node (Node Nil 4 Nil) 5 (Node Nil 6 Nil)))
7 (Node (Node (Node Nil 8 Nil) 9 (Node Nil 10 Nil))
     11 (Node (Node Nil 12 Nil) 13 (Node Nil 14 Nil)))
```

- Not particularly readable!

Showing a binary tree

- To be able to show a binary tree, we need to derive a **Show** instance

```
data BTree a = Nil | Node (BTree a) a (BTree a)
    deriving Show

createTree [0..14] =
Node (Node (Node (Node Nil 0 Nil) 1 (Node Nil 2 Nil))
     3 (Node (Node Nil 4 Nil) 5 (Node Nil 6 Nil)))
7 (Node (Node (Node Nil 8 Nil) 9 (Node Nil 10 Nil))
     11 (Node (Node Nil 12 Nil) 13 (Node Nil 14 Nil)))
```

- Not particularly readable!
- Addressed in the next class