COMP105 Lecture 22

Recursive Types

Outline

Today

- Recursive data types
- Custom lists
- Trees

Relevant book chapters

- Programming In Haskell Chapter 8
- Learn You a Haskell Chapter 8

Recap: Custom types

You can use custom types **inside** other custom types

```haskell
data Point = Point Int Int deriving(Show)

data Shape =   Circle Point Float
             | Rect Point Point
                         deriving(Show)
```

```
ghci> Rect (Point 1 2) (Point 3 4)
Rect (Point 1 2) (Point 3 4)
```

Recursive custom types

In a **recursive** custom type, some constructors contain the type itself

```haskell
data IntList = Empty | Cons Int IntList
                       deriving(Show)
```

Some examples:

```haskell
Empty -- []

Cons 1 (Empty) -- [1]

Cons 1 (Cons 2 Empty) -- [1,2]
```

Recursive custom types

Here is a more **general** list using a type variable

```
data List a = Empty | Cons a (List a)
                             deriving(Show)
```

Examples:

```
ghci> :t Cons 'a' (Cons 'b' Empty) -- ['a', 'b']
List Char

ghci> :t Empty -- []
List a
```

Recursive custom types

Two argument constructors can be made **infix** with backticks

```
ghci> 1 `Cons` (2 `Cons` Empty)
Cons 1 (Cons 2 Empty)
```

This just reimplements the standard Haskell syntax

```
ghci> 1 : (2 : [])
[1,2]
```

Functions

We can write **functions** for our custom list type

```
our_head :: List a -> a

our_head Empty       = error "Empty list"
our_head (Cons x _) = x
```

```
ghci> our_head (1 `Cons` (2 `Cons` Empty))
1
```

Functions

```haskell
our_tail :: List a -> List a

our_tail Empty       = error "Empty list"
our_tail (Cons _ x) = x
```

```
ghci> our_tail (1 `Cons` (2 `Cons` Empty))
Cons 2 Empty
```

Functions

We can write **recursive** functions on recursive types

```
our_sum  :: List Int -> Int

our_sum Empty       = 0
our_sum (Cons x xs) = x + our_sum xs
```

```
ghci> our_sum (1 `Cons` (2 `Cons` Empty))
3
```

Custom Lists

So far we've just re-implemented the Haskell list type

- Here is a new list type that can contain **two different types**

```haskell
data TwoList a b = TwoEmpty
                 | ACons a (TwoList a b)
                 | BCons b (TwoList a b)
                              deriving (Show)
```

```
gchi> :t 'a' `ACons` (False `BCons` TwoEmpty)
TwoList Char Bool
```

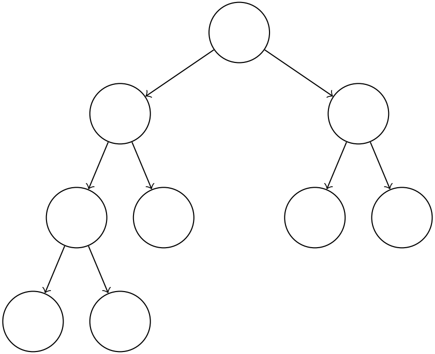
Exercise

What do these functions **do**, and what are their **types**?

```
mystery TwoEmpty         = 0
mystery (ACons _ xs)  = 1 + mystery xs
mystery (BCons _ xs)  = 1 + mystery xs
```

```
mystery2 TwoEmpty       = ""
mystery2 (ACons x xs) = (show x) ++ mystery2 xs
mystery2 (BCons _ xs) = mystery2 xs
```

Trees

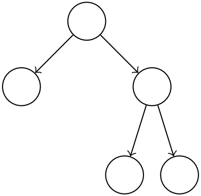


A **tree** is composed of

- Leaf nodes
- Branch nodes

A tree type in Haskell

```
data Tree = Leaf | Branch Tree Tree deriving (Show)
```



```
Branch Leaf (Branch Leaf Leaf)
```

Recursion on trees

We can write **recursive** functions that process trees

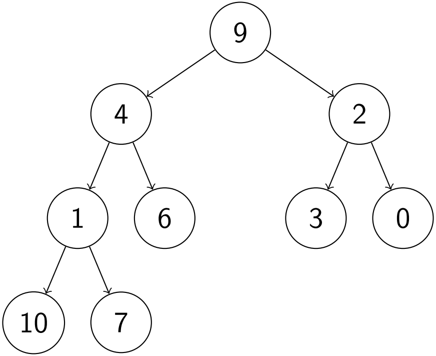
- Usually the recursive case will process both branches

```
size :: Tree -> Int

size (Leaf) = 1
size (Branch x y) = 1 + size x + size y
```

```
ghci> size (Branch Leaf (Branch Leaf Leaf))
5
```

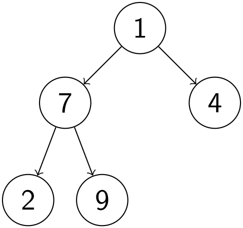
Trees with data



Nodes in a tree often hold **data**

Trees with data

```
data DTree a = DLeaf a
             | DBranch a (DTree a) (DTree a)
                                     deriving (Show)
```




```
DBranch 1 (DBranch 7 (DLeaf 2) (DLeaf 9)) (DLeaf 4)
```

Recursion on trees with data

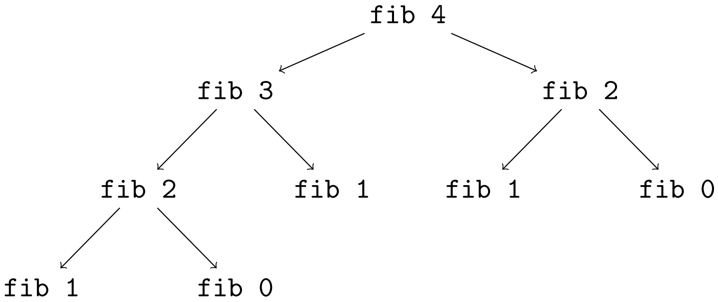
```
tree_sum :: Num a => DTree a -> a

tree_sum (DLeaf x) = x
tree_sum (DBranch x l r) = x + tree_sum l + tree_sum r
```

```
ghci> tree_sum (DBranch 11 (DLeaf 2) (DLeaf 9))
22
```

Example: Fibonacci numbers

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```



Example: Fibonacci numbers

How many recursive calls does the code make?

- Let's build the call tree

```
fib_tree :: Int -> Tree

fib_tree 0 = Leaf
fib_tree 1 = Leaf
fib_tree n = Branch (fib_tree (n-1)) (fib_tree (n-2))
```

```
ghci> fib_tree 4
Branch (Branch (Branch Leaf Leaf) Leaf)
                              (Branch Leaf Leaf)
```

Example: Fibonacci numbers

```
fib_calls n = size (fib_tree n)
```

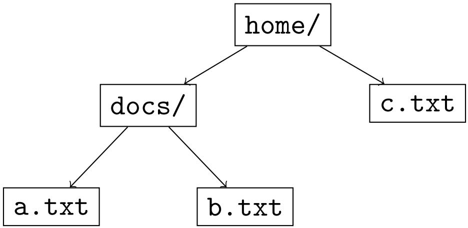
```
ghci> fib_calls 10
177

ghci> fib_calls 20
21891

ghci> fib_calls 30
2692537
```

Example: Finding a file

Suppose that we have a directory structure



Write a function that, given a filename finds the path to that file

Example: Finding a file

We can formulate the files as a data tree

```
let fs =
  DBranch "home/"
     (DBranch "docs/" (DLeaf "a.txt") (DLeaf "b.txt"))
     (DLeaf "c.txt")
```

Example: Finding a file

Note that the file might not exist

- So we will use the maybe type

```
ghci> find_file "a.txt" fs
Just "home/docs/a.txt"
```

```
ghci> find_file "d.txt" fs
Nothing
```

Example: Finding a file

```
find_file file (DLeaf x)
    | x == file = Just file
    | otherwise = Nothing


find_file file (DBranch x l r) =
    let
        left = find_file file l
        right = find_file file r
    in
        case (left, right) of
            (Just y, _) -> Just (x ++ y)
            (_, Just y) -> Just (x ++ y)
            (_, _)      -> Nothing
```

Exercise

```
mystery3 Leaf = 1
mystery3 (Branch l r) = 1 + max (mystery3 l) (mystery3 r)
```

```
mystery4 (DLeaf x) = x
mystery4 (DBranch x l r) = mystery4 l ++ x ++ mystery4 r
```

Exercises

1. Write a function `mul2 :: List Int -> List Int` that multiplies each element in a `List` by 2.

2. Write a function `length_as :: TwoList a b -> Int` that counts the number of times that `ACons` is used in the input list.

3. Write a data type `ThreeList` that is a list that contains three different types.

4. Write a function `depth :: Tree -> Int` that returns the length of the longest path from the root to a leaf.

Exercises

5. Write a function `sum_leaves :: DTree Int -> Int` that returns the sum of all the leafs of a data tree (the numbers on the branch nodes should be ignored)

6. Write a function `tree_mul2 :: DTree Int -> DTree Int` that multiplies each number in the data tree by 2.

7. (*) The "finding a file" example only works properly if the file occurs only once in the directory structure. So if there are two files named `a.txt`, then only one of them will be found.
   Write a function `find_files` that returns a list containing all instances of the file in the directory structure. It should return an empty list if there are no instances of the file.

Summary

- Recursive data types
- Custom lists
- Trees

Next time: IO