COMP105 Lecture 22

Recursive Types

Recap: Custom types

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

```
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

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

Some examples:

```
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**

```
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
```

Exercises

1. Write a function
   `ourElem :: Eq a => List a -> a -> Bool` that
   implements the `elem` function for our list type

2. Write a function `ourRange :: Int -> List Int` that takes
   one argument `n > 1` and outputs a value using our list type
   that contains all numbers between 1 and n in descending order

3. Create a recursive custom type `ThreeList` that implements a
   list type that can hold three different values