COMP304 Tutorial 4
05/08/2016

1 Pattern-Matching

Where possible, favour pattern-matching over conditionals or guards. For example, if you want to check the form of a list, you should use pattern-matching. Favour this:

```
count _ [] = 0
count x (y:ys)
   | x == y    = 1 + count x ys
   | otherwise = count x ys
```

Over this:

```
count x ys = if null ys then 0
             else if x == (head ys) then 1 + count x (tail ys)
             else count x (tail ys)
```

2 Type Signatures

Though Haskell doesn't need them, it's a good idea to write type signatures because they serve as documentation for how a function should be used. What's wrong with this signature?

```
count :: Int -> [Int] -> Int
```

It means our count function only works for lists of integers, which isn't ideal because counting up the occurrences of a value is something you can do generally to any type on which equality is defined.

Sometimes it's useful to leave off the type signature, write the function, and then check its signature in `GHCi`. Type inference will tell us the most general signature.

```
*Main> :type count
count :: (Eq a1, Num a) => a1 -> [a1] -> a
```

Type inference says that `count` can return any numeric type; so it could return $\sqrt{2}$. And while that's technically correct, since we're counting things the result should really just be a whole number i.e. an Integer. So a more sensible type signature would be the following:

```
count :: Eq a => a -> [a] -> Int
```

Looking at what type inference says is useful, but if the most general type is *too* general we might have to change it a bit.

3 Requirements

Be careful reading assignment requirements. For example, allPos should return the indices in which an item occurs, indexed from 1. So for example, allPos1[1] should return [1].

If the assignment gives some sample inputs and outputs you should include them in your test.

4 Lazy Evaluation

Sometimes the way you would write a function in a lazily-evaluated language would be inefficient in a strictly-evaluated language. To wit, let's say you wanted to find out if any string in a list had more than 5 characters.

You could map each string to whether they're greater than length 4. Then you could check if any of them are true. We'll do this by passing a lambda to map.

```
Prelude> let strings = ["hello", "how", "are", "you", "my", "mate"]
Prelude> strings
["hello","how","are","you","my","mate"]
Prelude> map (\s -> length s > 4) strings
[True,False,False,False,False,False]
Prelude> or (map (\s -> length s > 4) strings)
True
```

In a strictly-evaluated, imperative language you would loop over each s and check if any of them satisfies length s > 4. On first glance, the version might seem inefficient: you must first construct the list of booleans, then check if any is true.

In a lazily-evaluated language what happens is that map generates the first element True, then passes it to or, which then knows enough to be able to evaluate to True; so the intermediate list of booleans is never fully constructed.

5 Lazy Evaluation 2

Will the following recurse forever?

```
Prelude> let l = [1..]
Prelude> if length l == 0 then ‘‘empty’’ else ‘‘not empty’’
```

You may think that because of lazy evaluation, lengthl == 0 only checks as much of the list as you would need to establish that it is non-empty. This is not the case: in order to apply ==, we need to know what length l is. In order to evaluate length l we must traverse to the end of the list. Lazy evaluation is not "smart" in the sense that it will figure out the best way to evaluate an expression.

The way to check if a list is empty would be to pattern-match it against the empty-list, which is exactly what null xs does.

```
Prelude> if null [1..] then ‘‘empty’’ else ‘‘not empty’’
‘‘not empty’’
```

6 Algebraic Data Types

A list has two forms: the empty list [ ] or an element prepended to a list `x : xs`. In the language of algebraic data-types, we say that list has two constructors. In fact, we can define a list as an algebraic data-type like so:

```
data Seq a = Nil | Cons a (Seq a)
```

This is really what a Haskell list **is**, but a Haskell list a built-in special notation for these things: [ ] is the same as `EmptyList`. `x : xs` is the same as `Cons x s`. A list like $[1, 2, 3]$ is shorthand for `1 : 2 : 3 :` [ ]. If we passed in `1 : 2 : 3 :` [ ] to a function that pattern-matches on the head with `x : xs`, what we're doing is binding `x` to 1 and binding `xs` to 2 : 3 : [ ].

This is called constructing and destructing; we're able to pull apart and construct any list by appealing to its two constructors, `Nil` and `Cons`.

We can rewrite the usual `List` functions using our own algebraic definition of a list.

```
count :: Eq a => a -> Seq a -> Int
count _ Seq = 0
count x (Cons y ys)
   | x == y    = 1 + count ys
   | otherwise = count ys
```

In this case we're pattern-matching on the two different forms of a `Seq`; we're pattern-matching in exactly the same way that we pattern-match on a list.

Note that we need to give names to `Nil` and `Cons` in our definition of `Seq` to make it clear to Haskell when we're pattern-matching on an algebraic data-type.

If we want to construct a `Seq` we can do it like this:

```
*Main> let s = Cons 3 (Cons 3 (Cons 5 Nil))
*Main> count 3 s
2
```

7 Binary Tree

We can define a binary tree as an algebraic data type.

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

The second line says `Tree` is in the `Show` typeclass, which means it can be printed (a bit like `toString` in Java).

To manually build a tree we can do this:

```
example = Node 5 (Node 2 Nil (Node 1 Nil Nil)) (Node 6 Nil Nil)
```

A function to figure out the size of a tree would recurse through the tree adding 1 at each node. We want to pattern-match on the two different forms of the tree. An empty tree has size zero.

```
sizeTree Nil = 0
```

A non-empty tree has size one, plus the size of its subtrees. Since we don't care about what the tree at this point is actually storing we pattern-match it with a wildcard, which says "I don't really care about its value".

```
 sizeTree (Node _ left right) = 1 + (sizeTree left) + (sizeTree right)
```

8 Higher-Order Functions

A semordnilap is a pair of words that spell each other when reversed: ("swap", "paws") is a semordnnilap; so is ("stressed", "dessert"). Given a list of strings, return the longest semordnilap. We'll solve this using a mixture of higher-order functions and list comprehensions.

An algorithm for this is: find the collection of semordsnilaps in the list and then find the longest.

First we find the collection of semordnilaps in the list. We can construct all (distinct) pairs of words in the list which form a semordnilap using a list comprehension.

```
longestSemord strs =
   let allPairs = [(x,y) | x ← strs, y ← strs, x /= y, reverse x == y]
```

If we leave off the inequality check, `allPairs` will contain words paired with themselves (so any palindrome will be considered a semordnilap with itself).

To find the longest semordnilap we can use a fold. Our combination function will take the longest semordnilap so far, compare the length of one of its words to the next one in the list, and take the longer semodrnilap. Note that we are comparing pairs of words.

Since finding the longest semordnilap in an empty list doesn't make sense, we'll use `foldl1` which operates on non-empty lists; therefore we don't have to pass a unit value.

To make things more clear we'll use a let-binding to give a name to that combination function.

```
longestSemord strs =
   let allPairs = [(x,y) | x ← strs, y ← strs, x /= y, reverse x == y]
        longer (a,b) (c,d) = if length a > length c then (a,b) else (c,d)
    in foldl1 longer semords
```

Warning: make sure you line up all the assignments in the `let` expression, and make sure you line up `let` and `in`. Do NOT mix tabs and spaces or `GHCi` (probably) won't be able to parse it.

Now we can find the longest semordnilap:

```
*Main> longestSemord ["stressed", "desserts", "deliver", "reviled", "hello"]
("desserts","stressed")
```