CS4450 Lecture 14 Notes

Wednesday, September 19, 2018 2:54 PM

Lambda expressions & Higher-Order Functions

Chapter 6

Nameless functions in Haskell

So far, whenever we've defined a Haskell function, we've given it a name.

Lambda functions allow you to define nameless functions.

```
inc x = x + 1
```

To make the above a lambda expression:

```
\ x -> x + 1
```

Using nameless functions:

Lambda expressions may be applied in exactly the same way as declared functions

- ```
  (\ x -> x + 1) 1
  2
  ```

- ```
  map (\ x -> x + 1) [1,2,3]
  [2,3,4] -- This is the same thing as if we mapped inc on to there.
  ```

Haskell is a lambda calculus. Lambda calculi are a vital tool in PL research.

The Map Function

The higher-order library function called **map** applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
```

For example:

```
> map (+1) [1,3,5,7]

[2,4,6,8]
```

```
map inc :: [Int] -> [Int]
```

The map function can be defined in a particularly simple manner using a list comprehension:

```
map f xs = [f x | x <- xs]
```

A more standard way of defining this would be:

```
map f []     = []
map f (x:xs) = f x : map f xs
```

<u>The Filter Function</u>

The higher-order library function **filter** selects every element from a list that satisfies a predicate.

```
filter :: (a -> Bool) -> [a] -> [b]
```

Filter is shown in the slides, it can be defined using recursion and guards.

These recursive patterns are good because they allow for more efficient code.

<u>An exercise</u>

Write a program that sums integers from right to left (right-assoicative order)

```
iadd []     = 0
iadd (x:xs) = x + iadd xs
```

Write a program that appends strings from right to left this time

```
sapp [] = ""
sapp (x:xs) = x ++ sapp xs
```

We would write the above on a homework that asks this because it's a good way to do it.

Both have default values (0 and ""). In addition, both have a binary operator (either + or ++)

Write a function **gen** that says "If I apply gen to 0, I get iadd. If I apply gen to "", I get sapp."

This function below produces a "design pattern"

```
gen f d []     = d
gen f d (x:xs) = f x (gen f d xs)
```

gen here is actually **foldr**, which is a function in the Haskell Prelude (the default Haskell library)

The foldr code will be posted on Canvas. The type of foldr is:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

We'll come back to the type of foldr and discuss why it has this type in due time.

Note: We could also write iadd as

```
iadd = foldr (+) a
```

<u>Yet another exercise</u>

Say we want to take any list of strings ["one", "two", "three"] and we want to join them, separated by a space.

Attempt #1:

Attempt #2:

```
space x y = x ++ " " ++ y
Join l      = foldr space "" l
```

We have an extra space at the end with the above method. You can use pattern matching to fix this (shown on the Canvas slides)

Check out slide 31 on the Chapter 6 slides for a graphical image of **foldr**.

As far as I'm aware, every list in Haskell ends with the empty list []. It's like a null terminator in C, at least that's my understanding.

The higher-order library function **foldr** (fold right) encapsulates this simple pattern of recursion, with the function XOR and the value v as arguments.

For example:

```
sum = foldr (+) 0
```

Defining the function length using foldr:

```
length :: [a] -> Int
length []       = 0
length (_:xs) = 1 + length xs

lplus _ v = 1 + v

length = foldr lplus 0
  where
    lplus _ v = 1 + v
```

Code will be posted along with the Lambda function!