

PROGRAMMING IN HASKELL

0



Chapter 4.4 - Closures and Partial Functions

First Class Functions.. recall

1


```
inc :: Num a => a -> a
inc n = n + 1
```

```
double :: Num a => a -> a
double n = n * 2
```

```
square :: Num a => a -> a
square n = n ^ 2
```

```
ifEven :: Integral a => (a->a) -> a -> a
ifEven f n =
  if even n
    then f n
    else n
```

However, we are still repeating code



```
*Main> ifEven square 4
16
*Main> ifEven inc 4
5
*Main> ifEven inc 5
5
*Main> ifEven double 4
8
*Main> ifEven double 5
5
*Main> ifEven square 4
16
*Main> ifEven square 5
5
```


Use of lambdas leading to closures

2

We would like to write a function that will return **ifEvenX** (where x is double etc.).

We introduce **genIfEven**:

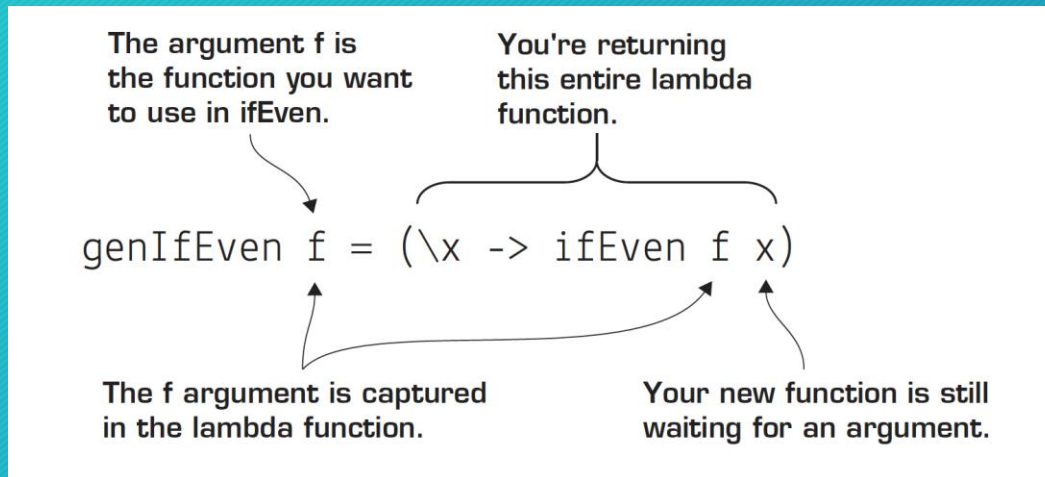
```
genIfEven :: Integral a => (a -> a) -> a -> a
```

```
genIfEven f = (\x -> ifEven f x)
```

Closures

3

How this works:



You pass in a function and return a lambda function. The function `f` is captured inside the lambda function. When you capture a value inside a lambda function, this is referred to as a *closure*.

Example - genIfEven inc

4

```
ifEvenInc = genIfEven inc
           |
           |
           v
(\x -> ifEven f x)
           |
           |
           v
(\x -> ifEven inc x)
           |
           |
           v
ifEvenInc = (\x -> ifEven inc x)
```

The diagram illustrates the transformation of the function `ifEvenInc`. It starts with the definition `ifEvenInc = genIfEven inc`. Arrows show the substitution of `inc` into the function `ifEven` within the `genIfEven` function. The first line of code is `ifEvenInc = genIfEven inc`. An arrow points from `inc` to the `ifEven` function in the second line, `(\x -> ifEven f x)`. Another arrow points from `inc` to the `ifEven` function in the third line, `(\x -> ifEven inc x)`. Finally, an arrow points from the third line to the final definition, `ifEvenInc = (\x -> ifEven inc x)`.

Closures and Partial Application

5

- ❑ Closures are powerful and useful. But the use of lambda function to create the closure can make it less clear.
- ❑ We can use Partial Application which is cleaner and easier to read

```
add4 :: Num n => n -> n -> n -> n -> n  
add4 a b c d = a + b + c + d
```

```
addXto3 :: Num n => n -> n -> n -> n -> n  
addXto3 x = (\b c d -> add4 x b c d)
```

```
addXYto2 :: Num n => n -> n -> n -> n -> n  
addXYto2 x y = (\c d -> add4 x y c d)
```


Closures and Partial Application

6

```
mystery = add4 5
```

This returns a function that expects the remaining 3 arguments.

```
anotherMystery = add4 5 4
```

This returns a function that expects two arguments.

This is called Partial Application and is clearer to the reader.



Reference

Based on material from 'Get Programming in Haskell', Will Kurt.