# PROGRAMMING IN HASKELL



Chapter 4.4 – Closures and Partial Functions

# First Class Functions.. recall



```
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



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

We introduce **genIfEven**:

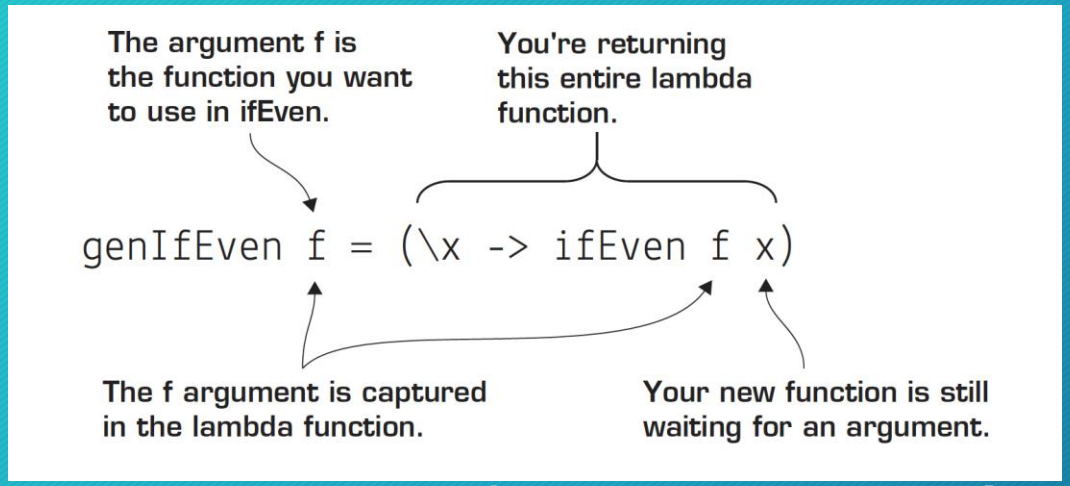
```
genIfEven :: Integral a => (a -> a) -> a -> a

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

# Closures



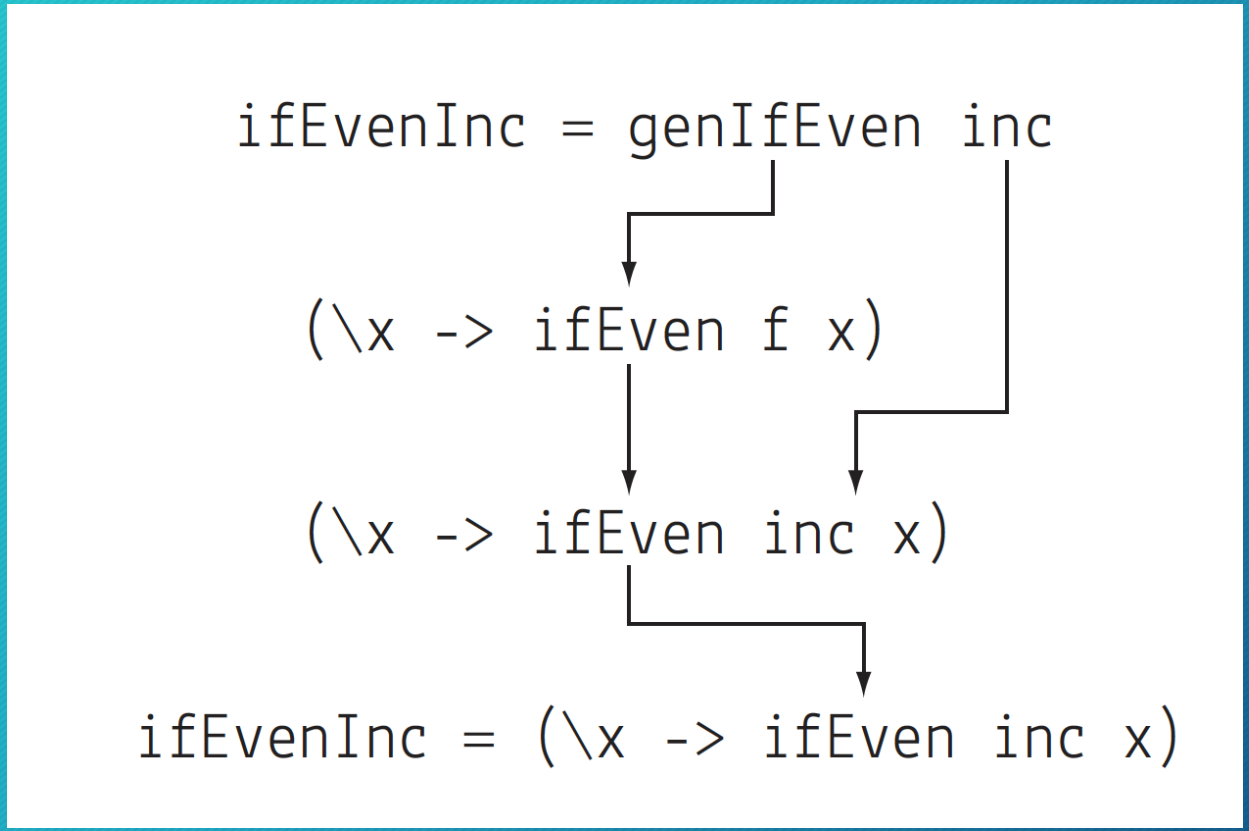
How this works:




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





# Closures and Partial Application



- 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



```
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.


ANY
QUESTIONS?

# Reference

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