PROGRAMMING IN HASKELL

Chapter 4.3 – First Class Functions

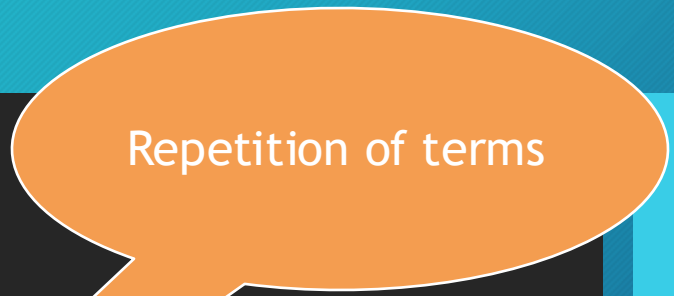
Lambdas, let and where

We wish to write a function that takes two numbers, and returns the greater of the sum of the squares of the numbers $(x^2 + y^2)$ or the square of the sums of the numbers $(x+y)^2$

We will call it **sumSquareOrSquareSum** we will write this using :

- where
- lambda functions, and finally
- let

Starting point




```
sumSquareOrSquareSum :: Int -> Int -> Int
sumSquareOrSquareSum x y =
        if x^2 + y^2 > (x + y)^2
        then x^2 + y^2
        else (x + y)^2
```

We can do better….

Using where


```
sumSquareOrSquareSum' :: Int -> Int -> Int
sumSquareOrSquareSum' x y =
      if sumSquare > squareSum
      then sumSquare
      else squareSum
   where   sumSquare = x^2 + y^2
           squareSum = (x + y)^2
```

- Cleaner
- Easier to read

Towards lambda functions


```
body :: Int -> Int -> Int
body sumSquare squareSum =
     if sumSquare > squareSum
     then sumSquare
     else squareSum

sumSquareOrSquareSum'' :: Int -> Int -> Int
sumSquareOrSquareSum'' x y =
     body (x^2 + y^2) ((x + y)^2)
```

Next, we will rewrite body as a lambda function.

Using lambda functions

Rewrite ‘body as lambda

```
sumSquareOrSquareSum''' :: Int -> Int -> Int
sumSquareOrSquareSum''' x y =
      (\sumSquare squareSum ->
         if sumSquare > squareSum
         then sumSquare
         else squareSum )
            (x^2 + y^2) ((x + y)^2)
```

Using let

Readability of the where clause

```
sumSquareOrSquareSum'''' :: Int -> Int -> Int
sumSquareOrSquareSum'''' x y =
      let  sumSquare = (x^2 + y^2)
           squareSum = (x + y)^2
      in
           if sumSquare > squareSum
           then sumSquare
           else squareSum
```

With the power of the lambda function

First Class Functions

First class functions are functions that can be passed around like any other values, including as arguments to functions.

First Class Functions. example


```
ifEvenInc :: Int -> Int
ifEvenInc n =
 if even n
   then n + 1
   else n
```

```
ifEvenDouble :: Int -> Int
ifEvenDouble n =
 if even n
   then n*2
   else n
```

```
ifEvenSquare :: Int -> Int
ifEvenSquare n =
 if even n
   then n^2
   else n
```

Only differences

First Class Functions. Introduce a function as an argument


```
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
```

even only works on integers

First Class Functions

We can call these functions thus:

```
*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
```

Using lambdas


```
*Main> ifEven (\x -> x*2) 6
12
*Main> ifEven (\x -> x*2) 3
3
*Main> ifEven (\x -> x^2) 3
3
*Main>
```

etc.

Example – sorting 1/3

In this sorting example we use:

- built in functions and also
- pass functions into other (sortBy) functions

We use:

- Pairs/Tuples (fst and snd return the first and second element of the tuples respectively)
- Form (“firstname”, “lastname”)

Example – sorting 2/3


```
names :: [(String, String)]
names = [     ("Hugo", "Keenan"),
              ("Mack", "Hansen"),
              ("James", "Lowe"),
              ("Bundee", "Aki"),
              ("Robbie", "Henshaw"),
              ("Sam", "Prendergast"),
              ("Dan", "Sheehan"),
              ("Jack ", "Crowley") ,
              ("Caelan", "Doris").  ]
```

Default sort is on first element

Called as

```
ghci> sort names
[("Bundee","Aki"),("Caelan","Doris"),("Dan","Sheehan"),("Hugo","Keenan"),("Jack ","Crowley"),("James","Lowe
"),("Mack","Hansen"),("Robbie","Henshaw"),("Sam","Prendergast")]
```

Example - sorting 3/3


```
import Data.List ( sort, sortBy )

names :: [(String, String)]
names = “as before”

compareLastNames :: Ord a => (a, a) -> (a, a) -> Ordering
compareLastNames name1 name2
     | lastName1 > lastName2 = GT
     | lastName1 < lastName2 = LT
     | otherwise = EQ
   where lastName1 = snd name1
         lastName2 = snd name2
```

Called as

```
ghci> sortBy compareLastNames names
[("Bundee","Aki"),("Jack ","Crowley"),("Caelan","Doris"),("Mack","Hansen"),("Robbie","Henshaw")
,("Hugo","Keenan"),("James","Lowe"),("Sam","Prendergast"),("Dan","Sheehan")]
ghci>
```

Example – structuring a solution

Problem: We want to write an address generating function.
This function is to take names and addresses from the three different SETU campuses:

- Waterford Campus ,
- Carlow Campus ,
- Wexford Campus

with slightly different rules for different colleges.

We wish to structure a solution that generalises as much as possible, but also allows for differences.

We will pass functions into other functions to help!

Example – structuring a solution

SETU (Waterford Campus):

Staff are allocated offices based (only) on their last name, i.e.

- < "L" – they are in the "Lower Floor, Main Building",
- otherwise, they are in the "Middle Floor, Science Building".

The remaining part of the address is
"SETU, Cork Road, Waterford, Ireland, X91 K0EK."

Example – structuring a solution

e.g. “Adam Able” from SETU (Waterford) would be given the address :
“Adam Able , Lower Floor, Main Building, SETU, Cork Road, Waterford, Ireland, X91 K0EK."

e.g. “Sean Stack” from SETU (Waterford) would be given the address :
“Sean Stack , Middle Floor, Science Building, SETU, Cork Road, Waterford, Ireland, X91 K0EK."

Example – structuring a solution

SETU (Carlow Campus):

All staff SETU (Carlow Campus) have the same address:

"SETU (Carlow Campus), Dublin Road, Carlow, Ireland, R93 V960."

e.g. “Adam Able” from SETU (Carlow) would be given the address :
“Adam Able , SETU, Dublin Road, Carlow, Ireland, R93 V960."

Example - structuring a solution

SETU (Carlow Campus):

All staff SETU (Carlow Campus) have the same address:

"SETU (Carlow Campus), Dublin Road, Carlow, Ireland, R93 V960."

Example – structuring a solution

SETU (Wexford Campus):

All staff in the Wexford Campus have the same address:

"SETU (Wexford Campus), Summerhill Rd, Townparks, Wexford, Y35 KA07."
In the address, however, the last name is before the first name.

e.g. “Adam Able” from SETU (Wexford Campus) would be given the address :
“Able, Adam, SETU (Wexford Campus), Summerhill Rd, Townparks, Wexford, Y35 KA07.”

Example – structuring a solution

A few notes:

We use the (“Tom”, “Hanks”) structure to model a name and use a type synonym Name to encapsulate this structure.

```
type Name = (String, String)
```

When we write a function, we write the arguments in the order (left to right) from most general to least general. This allows maximum reuse of these functions.

Example – structuring a solution

Each location needs a different address structure. To calculate the full (String) address we need

- location
- name (not so obvious but in case of SETU (Waterford Campus) , the address depends also on name)

We will use the following 'key's to indicate the locations:

- WatSETU",
- "CarSETU" and
- "WexSETU"

(depending on the key, we will call a different function)

Example – structuring a solution

Top-down approach:


```
addressLetter :: Name -> String -> String
addressLetter name location = getLocation location name
```

e.g. of running addressLetter:

```
ghci> addressLetter ("Siobhan", "Drohan") "Wat(SETU)"
"Siobhan  Drohan,  Lower Floor, Main Building  SETU (Waterford Campus), Cork Road, Waterford, Ireland, X91 K0EK."
ghci>
ghci> addressLetter ("Siobhan", "Roche") "Wat(SETU)"
"Siobhan  Roche,  Top Floor, Main Building  SETU (Waterford Campus), Cork Road, Waterford, Ireland, X91 K0EK."
ghci>
ghci> addressLetter ("Catherine", "Moloney") "Car(SETU)"
"Catherine  Moloney,  SETU (Carlow Campus), Dublin Road, Carlow, Ireland, R93 V960."
ghci>
ghci> addressLetter ("Veronica", "Campbell") "Wex(SETU)"
"Campbell, Veronica,  SETU (Wexford Campus), Summerhill Rd, Townparks, Wexford, Y35 KA07."
ghci>
```

Example – structuring a solution


```
getLocation :: String -> (Name -> String)
getLocation location   = case location  of
        "Wat(SETU)" -> watOffice
        "Car(SETU)" -> carOffice
        "Wex(SETU)" -> wexOffice
              _     -> (\name -> fst name ++ "  " ++ snd name ++ ": Address unknown" )
```

Returns the appropriate String producing function

Uses lambda function, expecting another argument

Example – structuring a solution


```
watOffice :: Name -> String
watOffice name =
  if lastName < "L"
    then nametext ++ ",  Lower Floor, Main Building  " ++ office WatSETU
    else nametext ++ ",  Top Floor, Main Building  " ++ office WatSETU
  where
    nametext = fst name ++ "  " ++ snd name
    lastName = snd name
```

Example – structuring a solution


```
carOffice :: Name -> String
carOffice name =    nametext ++ ",  " ++ office CarSETU
  where
    nametext = fst name ++ "  " ++ snd name
```

```
wexOffice :: Name -> String
wexOffice name =  nametext ++ ",  " ++ office WexSETU
  where
    nametext = snd name ++ ", " ++ fst name
```

Example – structuring a solution


```
data  Location = WatSETU | CarSETU| WexSETU
:
:
office :: Location -> String
office WatSETU = "SETU (Waterford Campus), Cork Road, Waterford, Ireland, X91 K0EK."
office CarSETU= "SETU (Carlow Campus), Dublin Road, Carlow, Ireland, R93 V960."
office WexSETU = "SETU (Wexford Campus), Summerhill Rd, Townparks, Wexford, Y35 KA07."
```

Example - Overall solution

The full text of the solution is available in the associated lab


ANY
QUESTIONS?