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


```
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

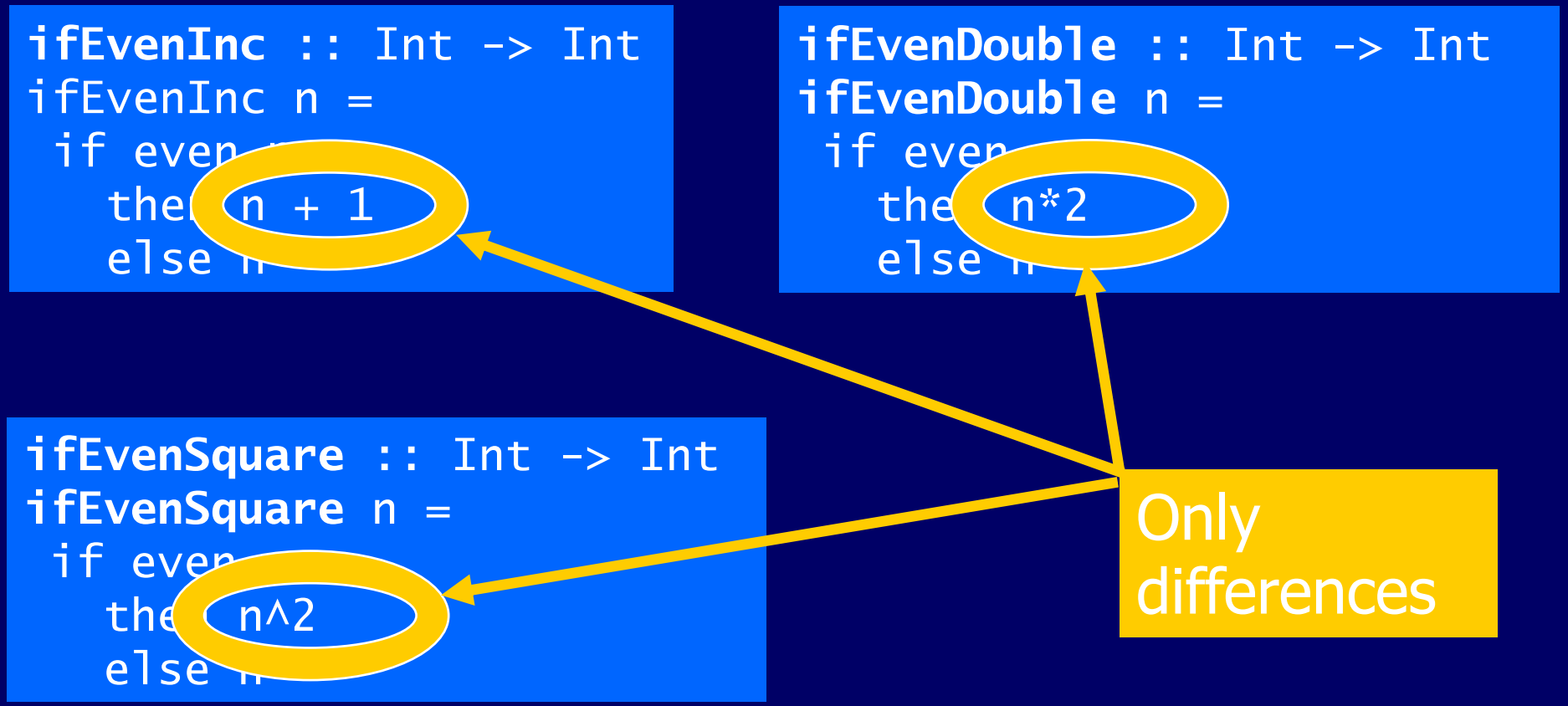
```
sumSquareOrSquareSum'''' :: Int -> Int -> Int
sumSquareOrSquareSum'''' x y =
          let sumSquare = (x^2 + y^2)
              squareSum = (x + y)^2
          in
              if sumSquare > squareSum
              then sumSquare
              else squareSum
```

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



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
```

etc.

Example – sorting 1/3

Sorting is an example where we use:

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

We use:

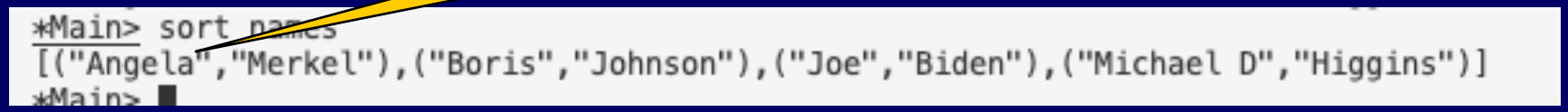
- Pairs/Tuples (fst and snd return the first and second element of the tuples respectively)
- Form (“firstname”, “lastname”)

Example – sorting 2/3

```
import Data.List ( sort )

names :: [(String, String)]
names = [("Angela", "Merkel"), ("Joe", "Biden"),
("Michael D", "Higgins"), ("Boris", "Johnson")]
```

Called as



Default sort is on first element

Example – sorting 3/3

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

names :: [(String, String)]
names = [("Angela", "Merkel"), ("Joe", "Biden"),
("Michael D", "Higgins"), ("Boris", "Johnson")]

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

```
*Main> sortBy compareLastNames names
[("Joe","Biden"),("Michael D","Higgins"),("Boris","Johnson"),("Angela","Merkel")]
*Main>
```

Example – structuring a solution

Problem: We want to write an address generating function.
This function is to take names and addresses from academic colleges(wit, itc, oth), 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

Waterford Institute of Technology(wit):

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
"Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."

e.g. "Adam Able" from wit would be given the address :
"Adam Able , Lower Floor, Main Building, Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."

e.g. "Sean Stack" from wit would be given the address :
"Sean Stack , Middle Floor, Science Building, Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."

Example – structuring a solution

Carlow Institute of Technology(itc):

All staff in Carlow Institute of Technology have the same address:

"Carlow Institute of Technology, Dublin Road, Carlow, Ireland, R93 V960."

e.g. “Adam Able” from itc would be given the address : “Adam Able , Carlow Institute of Technology, Dublin Road, Carlow, Ireland, R93 V960."

Example – structuring a solution

OTH Regensburg(oth):

All staff in OTH Regensburg have the same address:

"OTH Regensburg, Seybothstraße 2,
93053 Regensburg, Germany".

In the address, however, the last name is before the first name.

e.g. "Adam Able" from oth would be given the address :
"Able, Adam, OTH Regensburg, Seybothstraße 2,
93053 Regensburg, Germany".

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 wit, the address depends also on name)

We will use ‘key’s (”wit”, “itc”, “oth”) to indicate the locations.

(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:

```
*Main> addressLetter ("Willie", "Donnelly") "wit"
"Willie  Donnelly,  Lower Floor, Main Building  Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."
*Main> addressLetter ("Mairead", "Meagher") "wit"
"Mairead  Meagher,  Top Floor, Main Building  Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."
*Main> addressLetter ("Catherine", "Moloney") "itc"
"Catherine  Moloney,  Carlow Institute of Technology, Dublin Road, Carlow, Ireland, R93 V960."
*Main> addressLetter ("Markus", "Westner") "oth"
"Westner, Markus,  OTH Regensburg, Seybothstra\223e 2, 93053 Regensburg, Germany."
*Main>
```

Example – structuring a solution

```
getLocation :: String -> (Name -> String)
getLocation location = case location of
        "wit" -> witOffice
        "itc" -> itcOffice
        "oth" -> othOffice
        _ -> (\name -> fst name ++ " " ++ snd name
                ++ ": Address unknown" )
```

Returns the appropriate String producing function

Uses lambda function, expecting another argument

Example – structuring a solution

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

Example – structuring a solution

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

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

Example – structuring a solution

```
data Location = Wit | Itc | Oth
:
:
office :: Location -> String
office Wit = "Waterford institute of Technology, Cork Road, Waterford, Ireland, X91 K0EK."
office Itc = "Carlow Institute of Technology, Dublin Road, Carlow, Ireland, R93 V960."
office Oth = "OTH Regensburg, Seybothstraße 2, 93053 Regensburg, Germany."
```

Example – Overall solution

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


ANY
QUESTIONS?