Parametric Polymorphism
Week 4 Friday

COMP1100/1130

Review of Recursion: a mystery function

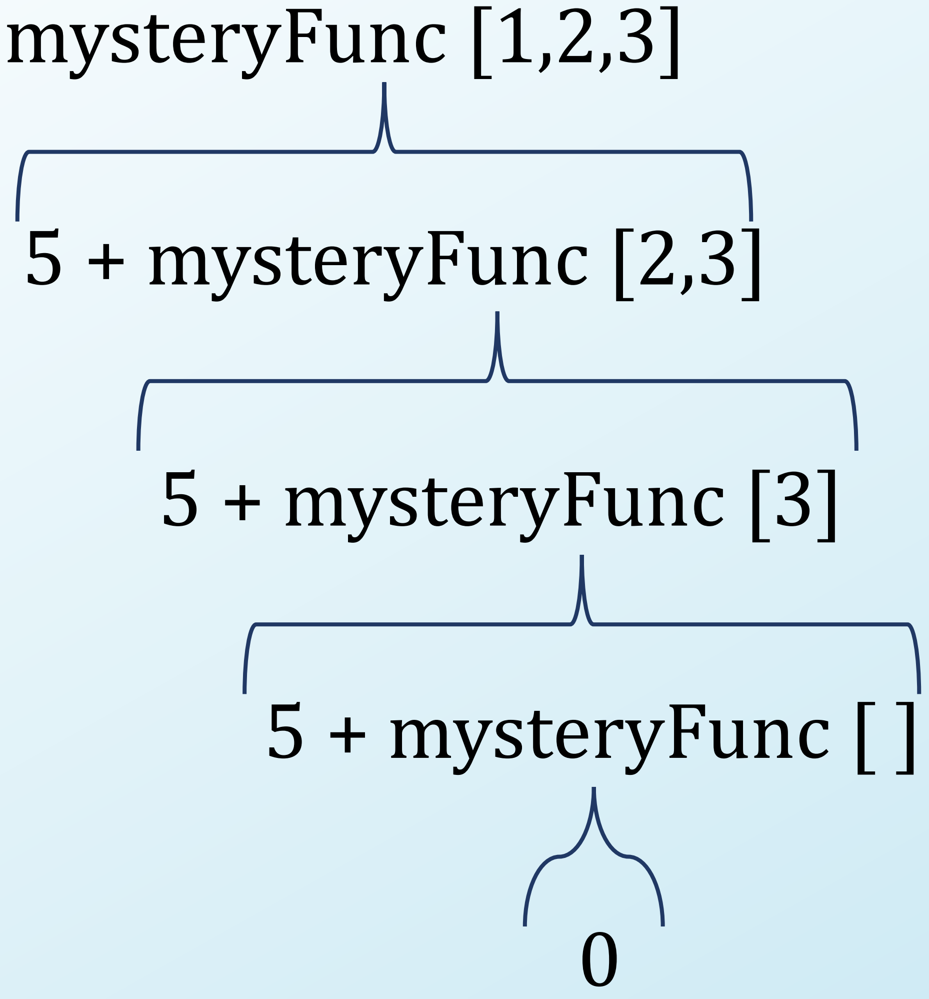
Consider the following function:

```haskell
mysteryFunc :: [Int] -> Int
mysteryFunc list = case list of
  []    -> 0
  _:xs -> 5 + mysteryFunc xs
```

Can you explain what the result is?

Review of Recursion

Let’s step through it!



mysteryFunc [1,2,3]

5 + mysteryFunc [2,3]

5 + mysteryFunc [3]

5 + mysteryFunc [ ]

0

```
mysteryFunc :: [Int] -> Int
mysteryFunc list = case list of
  []   -> 0
  _:xs -> 5 + mysteryFunc xs
```

Result = 5 + 5 + 5 + 0 = 15

A Closer Look at MysteryFunc

Consider the following function:

```
mysteryFunc :: [Int] -> Int
mysteryFunc list = case list of
  []   -> 100
  _:xs -> 5 + mysteryFunc xs
```

What does this change?

Do you get the same result for mysteryFunc [1,2] and mysteryFunc [1000,2000]? Why?

Changing it to take a List of Strings

Consider the following function:

```
mysteryFunc :: [String] -> Int
mysteryFunc list = case list of
  []    -> 100
  _:xs -> 5 + mysteryFunc xs
```

How about now? What does this change?

What is the result for mysteryFunc [“hello”, “goodbye”]?

Does it work with any list?

Consider the following function:

```
mysteryFunc :: [Bool] -> Int
mysteryFunc list = case list of
  []   -> 100
  _:xs -> 5 + mysteryFunc xs
```

How about now? What does this change?

What is the result for mysteryFunc [True, False, True, True]?

Generalising MysteryFunc

Consider the following function:

```
mysteryFunc :: [a] -> Int
mysteryFunc list = case list of
  []   -> 100
  _:xs -> 5 + mysteryFunc xs
```

What does the [a] mean? This means any type.

Try it with mysteryFunc [1,2] and mysteryFunc[True, True].

Getting the head of a list

The head function returns the head of a list. It doesn’t matter what type of elements the list has:

```
head :: [a] -> a
```

What happens if we give it an empty list [ ] ?

How can we prevent this?

Let's try to write it

```
myHead :: [a] -> a
myHead list = case list of
  x:_ -> x
```

Why are there warnings?

What should we do in the [ ] case?

The Maybe type

```
data Maybe a = Nothing | Just a
```

Now we can return Nothing!

This is instantiated depending on the type, e.g. as follows:

```
data Maybe String = Nothing | Just String
data Maybe int = Nothing | Just int
data Maybe Bool = Nothing | Just Bool
```

An Improved Head Function

```
improvedHead :: [a] -> Maybe a
improvedHead list = case list of
  [] -> Nothing
  x:_ -> Just x
```

Now let’s try it with an empty list!

Another Polymorphic Data Type

Tuples can contain elements of any type.

Each of the elements can be of different types.

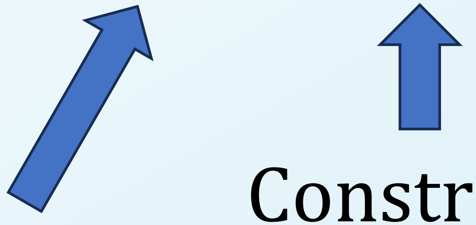
Examples:

```
    (1,2,3,4)
    (1,“2”,3,4)
    (1,“2”,True, False)
    (1,“2”,(4,5), False)
    (1,“2”,(), False)
    (1,“2”,(True, 2), False)
```

Another Polymorphic Data Type

Defining Pairs:

```
data (,) a b = (,) a b
```



Type variables

Constructor

We usually write it as (a,b)

```
first :: (a, b) -> a
first (x,_) = x
```

What is the return type? Why?

Checking the types

Is this correct? Why/why not?

```
first :: (a, b) -> b
first (x,_) = x
```

Remember that `(a, b) -> a` is talking about the types, not saying that it has to be the *same* `a` object. Think about this:

```
addFour :: (Int, String) -> Int
addFour (x,_) = x + 4
```

Another Polymorphic Data Type

Defining Lists:

```
data [] a = [] | a : [a]
```

This is how lists are defined recursively.

e.g. 5 : 4 : 7 : 9 : [ ]

[5,4,7,9] This is syntactic sugar.

5 : (4 : 7 : 9 : [ ])

5 : (4 : (7 : 9 : [ ]))

5 : (4 : (7 : (9 : [ ])))

When to use Parametric Polymorphism

When should you use parametric polymorphism?

```
mysteryFunc :: [a] -> Int
mysteryFunc list = case list of
  []   -> 100
  _:xs -> 5 + mysteryFunc xs
```

Think about whether the function needs a particular type of list.

Checking the types

Could this be done with parametric polymorphism?

```
addFour :: (Int, String) -> Int
addFour (x,_) = x + 4
```

The x has to be a number.

Next Lecture

We’ll look at how to define standard list functions provided in the prelude using parametric polymorphism.