

# Functional Programming G5031 Topic 5 – Currying and Lambdas

## Outline

- Curried Functions
- Lambdas

## Curried Functions (1)

- Named after the mathematician Haskell Curry
- Functions with multiple arguments can take tuples or take their arguments one at a time by returning functions as results:

```
add :: (Int, Int) -> Int
add' :: Int -> (Int -> Int)
```

– We've seen this style already ... now we get more detail on how it works

## Curried Functions (2)

```
add :: (Int, Int) -> Int
add (x, y) = x + y
```

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

- The same result but a different process to get there:

– Intermediate function: `add' x` takes single argument `y` and returns the result

## Curried Functions (3)

- Functions with more than two arguments can be curried by returning nested functions:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

- mult takes Int x, returns function mult x
- mult x takes Int y returns function mult x y
- mult x y takes Int z and returns the result x\*y\*z

## Curried Functions (4)

- `Int -> Int -> Int`  
means  
`Int -> (Int -> Int)`  
but `->` is right-associative so we don't need brackets

- Functions associate to the left:

```
mult x y z
means
((mult x) y) z
```

## How Is This Useful? (1)

- All functions can be reduced to functions with a single argument (and some known internal values) ... which is a nice simple form, e.g.:

```
add :: Int -> (Int -> Int)
add x y = x + y
```

add 4 - the function which adds 4 to another number

## How Is This Useful? (2)

- So, Curried functions are more flexible than functions on tuples, because useful functions can be made by partially applying a curried function

- We used partial application by using the undefined constant in exercise 2

```
listSum :: [Float] -> Float
listSum xs = foldl (+) 0 xs
```

- Can be simplified to:

```
listSum = foldl (+) 0
```

## Similarly

- We defined the append function (++) using foldr in a previous lecture:

```
(xs ++ ys) = foldr (:) ys xs
```

- this can be rewritten using currying to:

```
(++ ys) = foldr (:) ys
```

## A More Complex Example (1)

- Take the function:

```
reverse :: [a] -> [a]
reverse xs = foldl revOp [] xs
  where revOp acc x = x:acc
```

- Use prelude function flip to redefine revOp:

```
flip :: (a->b->c) -> b -> a -> c
flip f x y = f y x
revOp acc x = flip (:) acc x
```

## A More Complex Example (1)

- Curry revOp twice:

```
revOp acc x = flip (:) acc x
```

becomes:

```
revOp = flip (:)
```

- Curry reverse:

```
reverse xs = foldl revOp [] xs
```

becomes:

```
reverse = foldl (flip (:)) []
```

- Which is how its defined in the prelude

## Sections (1)

- An operator written between its two arguments, e.g.:

```
> 1+2
```

```
3
```

- ... can be converted into a curried function written before its two arguments by using parentheses (as we've already seen), e.g.:

```
> (+) 1 2
```

```
3
```

## Sections (2)

- This convention also allows one of the arguments of the operator to be included in the parentheses, e.g.:

> (+1) 2  
3

- In general, if  $\oplus$  is some operator then functions of the form  $(\oplus)$ ,  $(x\oplus)$ , and  $(\oplus y)$  are called sections

## Why Sections?

- Useful functions can sometimes be constructed in a simple way using sections, e.g.:

- (+1) – successor / increment
- (\*2) – double
- (/2) – half
- (1/) – reciprocal

## Lambda Expressions (1)

- Functions can be constructed without naming the functions by using lambda expressions:

-  $\lambda x \rightarrow x+1$

- The nameless function that takes a number,  $x$ , and returns the result  $x+1$

- Layout is:

Lambda( $\lambda$ ) arguments  $\rightarrow$  result

## Lambda Expressions (2)

- The use of the  $\lambda$  symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based
- In maths nameless functions are usually denoted using  $\lambda$ , e.g.:  $x \lambda x+1$
- In Haskell  $\lambda$  becomes backslash:  $\backslash$ , e.g.:

$\backslash x \rightarrow x+1$

## Example Lambda (1)

- AddNum takes n and returns a function h which adds m to n:

```
addNum :: Int -> Int -> Int
```

```
addNum n = h
```

```
  where h m = n+m
```

- which is a bit indirect ...  
 lambda functions let us write:

```
addNum n = (\m -> n+m)
```

## Example Lambda (2)

```
Main> :t addNum
```

```
addNum :: Num a => a -> a -> a
```

```
Main> :t addNum 2
```

addNum 2 returns a function which adds 2 to its argument

```
addNum 2 :: Num a => a -> a
```

```
Main> :t addNum 2 3
```

addNum 2 3 applies 3 to the add-to-2 lambda expression

```
addNum 2 3 :: Num a => a
```

```
Main> addNum 2 3
```

## Why Are Lambdas Useful (1)

- Lambda expressions can be used to give a formal meaning to functions defined using currying, e.g.:

```
add x y = x+y
```

- means:

```
add = λx -> (λy -> x+y)
```

## Why Are Lambdas Useful (2)

- Lambda expressions are also useful when defining functions that return functions as results, e.g.:

```
compose f g x = f (g x)
```

- Is more naturally defined by:

```
compose f g = \x -> f (g x)
```

## Why Are Lambdas Useful (3)

- Lambda expressions can be used to avoid naming functions that are only referenced once, e.g.:

```
map f [0..99]
  where f x = x*2 + 1
- Can be simplified to
map (\x -> x*2 + 1) [0..99]
```

## Length Defined Using Lambda and Foldr

- Although the pattern of recursion described by foldr is simple it can be used with lambdas to define a wide range of functions, e.g.:

- Length:

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

- Can be rewritten:

```
length = foldr (\x n -> 1+n) 0
- as foldr f z (x:xs) = f x (foldr f z xs)
```

## Reverse Defined Using Lambda and Foldr

- Reverse:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
- Can be rewritten:
reverse
  = foldr (\x xs -> xs ++ [x]) []
```

## Composition: Lambdas in Higher Order Functions

- The library function `.` returns the composition of 2 functions as a 1 function:

```
(.) :: (b->c) -> (a->b) -> (a->c)
```

```
f . g = \x -> f (g x)
```

- e.g.:

```
odd :: Int -> Bool
```

```
odd = not . Even
```

- Remember how functions are applied – f takes as argument the result of g, hence (b->c) -> (a->b) in type signature

## Summary

- Curried functions
  - Partial application of functions
  - Sections
- Lambda Expressions
  - Formal meaning to curried functions
  - Functions that return functions
  - Avoid naming functions that are only used once