Table of Contents

Higher-Order Functions

Benson Joeris
bjoeris@gmail.com
bjoeris.wordpress.com

pluralsight
hardcore developer training

Overview

- Functions with function arguments
- Create functions on the fly
  - Partial function application
  - Lambda expressions
  - Function composition

Table of Contents

Functions as Values

- Passed as function arguments
- Created on the fly
- Higher-Order Functions

Functions as Values

```
pass3 f = f 3
```

```
add1 x = x + 1
```

GHCi> `pass3 add1`

Result: `4 -- add1 3 = 3 + 1 = 4`

Functions as Values

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

```
add1 x = x + 1
mult2 x = 2 * x
```

GHCi> `compose add1 mult2 4`

Result: `9 -- add1 (mult2 4) = (2 * 4) + 1 = 9`

Functions as Values

```
always7 x = 7
```

```
always7' = const 7
```

GHCi>
```
(const 7) 5
```
Result: 7

Table of Contents

Partial Application

```
int foo(int x, int y, int z) {
  return x + y + z;
}
```

```
foo_1_2 = foo(1,2);
```

Partial Application

```
foo x y z = x + y + z
```

```
foo_1_2 = foo 1 2
```

```
GHCi> foo_1_2 3
Result: 6 -- 1 + 2 + 3
```

Partial Application

```
pass x f = f x
pass3 = pass 3
```

Partial Application

- Arguments given in order

Table of Contents

Operators

- +, *, :, ++ are all functions
- (+), (*), (:), (++)

```
GHCi> (+) 5 3
Result: 8
```

Operators

```
pass_3_4 f = f 3 4
```

GHCi> `pass_3_4 (+)`

Result: `7`

Operator Definitions

```
(a,b) .+ (c,d) = (a + c, b + d)
```

Partially Applying Operators

```
plus1 = (+) 1
```

```
plus1' = (1+)
plus1'' = (+1)
```

Turning functions into operators

```
GHCi> mod 10 2
Result: 0
```

```
GHCi> 10 `mod` 2
Result: 0
```

Table of Contents

Map

- Applies a function to every element in a list

Map

GHCi> `map length ["hello","abc","1234"]`

Result: `[5,3,4]`

Map

GHCi> `map (1+) [1,3,5,7]`

Result: `[2,4,6,8]`

Map

```
double = map (2*)
```

Table of Contents

Filter

- Tests each element, keeps those that pass

Filter

```
notNull xs = not (null xs)
```

```
GHCi> filter notNull ["","abc","","hello",""]
Result: ["abc","hello"]
```

Filter

```
isEven x = x `mod` 2 == 0
removeOdd = filter isEven
```

Filter

```
GHCi> map snd (filter fst
                 [(True,1),(False,7),(True,11)])
Result: [1,11]
```

Table of Contents

Fold

- Combines all the values in a list
- Two versions
  - foldl
  - foldr

foldl

GHCi>
```
foldl (+) 0 [1,2,3,4]
```
Result:
```
10 -- 0 + 1 + 2 + 3 + 4 = 10
```

foldl

```
showPlus s x = "(" ++ s ++ "+" ++ (show x) ++ ")"
```

```
GHCi> showPlus "(1+2)" 3
Result: "((1+2)+3)"
```

```
GHCi> foldl showPlus "0" [1,2,3,4]
Result: ((((0 + 1) + 2) + 3) + 4)
```

foldr

GHCi>
```
foldr (+) 0 [1,2,3,4]
```
Result:
```
10 -- 1 + 2 + 3 + 4 + 0 = 10
```

foldr

```
showPlus' x s = "(" ++ (show x) ++ "+" ++ s ++ ")"
```

```
GHCi> foldr showPlus' "0" [1,2,3,4]
Result: (1+(2+(3+(4+0))))
```

```
GHCi> foldl showPlus "0" [1,2,3,4]
Result: ((((0+1)+2)+3)+4)
```

foldl vs foldr

```
GHCi> foldl (-) 0 [1,2,3]
Result: -6
```

$$((0-1)-2)-3 = ((-1)-2)-3 = (-3)-3 = -6$$

```
GHCi> foldr (-) 0 [1,2,3]
Result: 2
```

$$1-(2-(3-0)) = 1-(2-3) = 1-(-1) = 1+1 = 2$$

foldl vs foldr

- foldl: slightly faster
- foldr: infinite lists

Table of Contents

Zip

GHCi>
```
zip [1, 2, 3] [4, 5, 6]
```
Result:
```
[(1,4), (2,5), (3,6)]
```

Zip

GHCi>
```
zip [1, 2] [3, 4, 5, 6]
```
Result:
```
[(1,3), (2,4)]
```

zipWith

```
GHCi> zipWith (+) [1,2,3] [4,5,6]
Result: [5,7,9]
```

zipWith3

```
plus3 x y z = x + y + z
```

GHCi> `zipWith3 plus3 [1,2,3] [4,5,6] [7,8,9]`

Result: `[12,15,18]`

Table of Contents

Lambda Expressions

```
plus3 x y z = x + y + z
```

- Clutter
- Disrupts flow

Lambda Expressions

```
GHCi> zipWith3 (\ x y z -> x + y + z)
                [1,2,3] [4,5,6] [7,8,9]
Result: [12,15,18]
```

Lambda Expressions

GHCi>
```
map (\ x -> 2 * x) [1,2,3]
```
Result:
```
[2,4,6]
```

GHCi>
```
map (2*) [1,2,3]
```
Result:
```
[2,4,6]
```

GHCi>
```
map (\x -> 2 * x + 1) [1,2,3]
```
Result:
```
[3,5,7]
```

When to use Lambda Expressions

- Too simple: partial application
- Too complex: named function

Table of Contents

Function Operators

- (.) - Function Composition
- ($) - Function Application

Function Composition (.)

```
stringLength = length . show
```

```
GHCi> stringLength 120
Result: 3
```

Function Composition (.)

```
stringLength = length . show
```

```
stringLength' x = length (show x)
```

Function Composition (.)

```
notNull = not . null
```

Function Composition (.)

```
f a b = a + b
g x = 2 * x
```

```
f . g
```

```
g . f
```

Function Application ($)

```
f $ x = f x
```

Function Application ($)

```
f $ g x = f (g x)
```

```
f $ g $ h $ k x = f (g (h (k x)))
```

Function Application ($)

```
GHCi> map (\f -> f 3) [(+1), (\x -> 2*x + 3), (*2)]
Result: [4,9,6]
```

```
GHCi> map ($3) [(+1), (\x -> 2*x + 3), (*2)]
Result: [4,9,6]
```

```
GHCi> zipWith ($) [(+1), (\x -> 2*x + 3), (*2)]
                  [1,2,3]
Result: [2,7,6]
```

Table of Contents

Summary

- Functions are values
- Operators are functions
- Partial Application
- Higher-Order Functions
  - map
  - filter
  - fold
  - zipWith
- Lambda Expressions
- Function Operators
  - (.) - Composition
  - ($) - Application