Week 5

Higher Order Functions

Introduction

- **Functions take values as inputs and return values as outputs.**
- In Haskell, functions are first-class values.
  - Functions can be passed as inputs to other functions.
  - Functions can be outputs of other functions (via partial application).
  - Functions can be anonymous (defined without names).
- Functions that accept or return functions are known as **higher-order functions**.

Associativity of `->`

- In type declarations, `->` is **right-associative**:

```
a -> b -> c  ≡ a -> (b -> c)
```

- Multi-argument functions are "*curried*"—arguments are provided one at a time.
- Note the distinction:
  - `a -> (b -> c)` ≠ `(a -> b) -> c`

Partial Application: Motivation

- Example with a one-argument function:

```
f :: a -> b
x :: a
f x :: b
```

- Example with a two-argument function:

```
g :: a -> b -> c     -- equivalent to g :: a -> (b -> c)
x :: a
g x :: b -> c        -- Now a function awaiting the second argument
```

Examples of Partial Application

`take`

```
take :: Int -> ([a] -> [a])
```

```
take 5 :: [a] -> [a]

(take 5) "Desmond" == "Desmo"
(take 5) [1,2,3]       == [1,2,3]
```

`replicate`

```
replicate :: Int -> a -> [a]
replicate 2 :: a -> [a]

(replicate 2) "Go"       == ["Go","Go"]
(replicate 2) (True,2.3) == [(True,2.3),(True,2.3)]
```

`(+)` (addition)

```
(+) :: Int -> (Int -> Int)
(+) 2 :: Int -> Int

((+) 2) 3 == 5
-- Alternate notations: (2+) or (+2)
```

`(:)` (cons)

```
(:) :: a -> ([a] -> [a])
(:) 'a'   :: [Char] -> [Char]
(:) 2     :: [Int]  -> [Int]

(0:) []      == [0]
(0:) [8,4,5] == [0,8,4,5]
```

Summary: Parentheses and Application

- Right-associativity of `->`:

  ```
  Int -> [a] -> [a] ≡ Int -> ([a] -> [a])
  ```

- In expressions, function application is **left-associative**:

  ```
  take 5 "Desmond" == (take 5) "Desmond"
  ```

Functions as Inputs: Motivation

- Example list:

  ```
  [2.35, 2.40, 2.00, 2.65, 3.12, 2.77]
  ```

- Two tasks:
  - Round each `Float` to `Int` → `[2, 2, 2, 3, 3, 3]`

- Check if each measurement is `> 3` → `[False, False, False, False, True, False]`

Using Recursion

```
roundEach :: [Float] -> [Int]
roundEach []     = []
roundEach (x:xs) = roundFloatInt x : roundEach xs

eachGreaterThan3 :: [Float] -> [Bool]
eachGreaterThan3 []     = []
eachGreaterThan3 (x:xs) = (>3) x : eachGreaterThan3 xs
```

Introducing `map`

Motivation

Functions like `roundEach` and `eachGreaterThan3` share a common pattern:

- Apply a function `f` to each element of a list → collect the results.

Defining `map`

```
myMap :: (a -> b) -> [a] -> [b]
myMap f ls = case ls of
  []     -> []
  (x:xs) -> f x : myMap f xs

-- Now available in Prelude as `map`.
```

Using `map` to simplify:

```
roundEach        = map roundFloatToInt
eachGreaterThan3 = map (>3)
```

Function Composition

Motivation

Compose `f` and `g`:

- Apply `f` to `x`, then apply `g` to `f x` → `g (f x)`.

Type of composition:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
-- Alternatively written as:
-- (.) :: (b -> c) -> (a -> b) -> a -> c
```

```
compo :: (b -> c) -> (a -> b) -> a -> c
compo g f x = g (f x)
```

Example:

- Given `length :: [a] -> Int`, `even :: Int -> Bool`:

```
lengthEven :: [a] -> Bool
lengthEven = even . length

-- Equivalent to:
lengthEven xs = even (length xs)
```

Other Useful Higher-Order Functions

`filter`

```
filter :: (a -> Bool) -> [a] -> [a]
filter f xs = [ x | x <- xs, f x ]

-- Example:
filter even [1,2,3,2,4,3] == [2,2,4]
```

`zipWith`

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

-- Example:
zipWith take [1,2,3] ["one","two","three"] == ["o","tw","thr"]
```

`concatMap`

```
concatMap :: (a -> [b]) -> [a] -> [b]

-- Example:
concatMap (take 2) ["one","two","three"] == "ontwth"
```

Exercises

- Define your own versions:
  - `myFilter`, using explicit recursion.
  - `myZipWith`, using `zip`, `map`, and anonymous functions.
  - `myConcatMap`, using `map`, `concat`, composition, and partial application.

Anonymous Functions (Lambdas)

Three ways to pass a function to a higher-order function:

- **Partial application**:

```
filter (==0) [1,1,0,0]
```

- **Named function**:

```
map sndOfTriple [(1,2,3), (3,2,1)]

sndOfTriple :: (a,b,c) -> b
sndOfTriple (_, y, _) = y
```

- **Anonymous function (lambda)**:

```
map (\(_, y, _) -> y) [(1,2,3), (3,2,1)]
```

Syntax:

```
\ var1 var2 ... varN -> expr

-- Example:
(\x -> x + 1)
(\x y -> x + y)
(\(x,y) -> x < y)
```

Exercise: Filtering with Lambda

Define:

```
f :: [(Int, Int)] -> [(Int, Int)]
```

that keeps only pairs where the first component is less than the second:

```
f [(2,1), (2,5), (3,4), (3,3)] == [(2,5), (3,4)]
```

Using `filter` with lambda:

```
f pairs = filter (\(x,y) -> x < y) pairs
-- Or simply:
f = filter (\(x,y) -> x < y)
```