EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Feb 14th 2020

Types

	()	Int	[A]	(A,B)	A→B
[...]	[()]	[Int]	[[A]]	[(A,B)]	[A→B]
(...,C)	((),C)	(Int,C)	([A],C)	((A,B),C)	(A→B,C)
C → ...	C → ()	C → Int	C → [A]	C → (A,B)	C → (A→B)
... → C	() → C	Int → C	[A] → C	(A,B) → C	(A→B) → C

take, with Curry

We can create a custom version of **`take`**

```
GHCi> :t take
Int -> [a] -> [a]
GHCi> let f = take 5
GHCi> :t f
f :: [a] -> [a]
GHCi> f [10..20]
[10,11,12,13,14]
```

This works, because of Currying.

Curry

The principle of currying is simple:

- All you can do is apply a function to an argument;
- and every function accepted just one argument.

So:

- Pass the first argument;
- get back a **new and customized** function that accepted the second argument.

So:

- we pass **5** to **`take`**;
- and get back a **new and customized** function that **`take`**'s 5 elements.

Filtering

Consider

```
GHCi> filter odd [1..10]
[1,3,5,7,9]
```

How might we construct a function that filters out even numbers?

```
GHCi> let f = ...
```

- What is the type of this function?

The truth about filtering

- **`filter`** takes two arguments, a **function** and a list.
- It returns the elements, in order, that match the predicate.

```
filter :: (a -> Bool) -> [a] -> [a]
```

OR

```
filter :: (a -> Bool) -> ([a] -> [a])
```

- This use of functions-as-arguments is called **higher-order functions**.
- This pervasive use of functions is why this class is called functional programming.

map

`map` is one of the most important functions in functional programming.

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

- What can we tell from the type?
- What can we use **`map`** for?
- Can we nest **`map`**?
- Can we write **`map`**?

Other Higher Order functions

```
flip :: (a -> b -> c) -> b -> a -> c
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

- **`flip`** turn around the arguments
- **`curry`** takes a function that takes a 2-tuple and Currys it.
- **`uncurry`** takes a function that uses currying, and provides a 2-tuples API.

KU

Sections

Sections are a way of building a specialized function from an infix function.

```
GHCi> :t (+)
(+) :: Num a => a -> a -> a
GHCi> :t (+ 1)
(+ 1) :: Num a => a -> a
GHCi> :t (1 +)
(1 +) :: Num a => a -> a
```

- Parenthesis around an infix operator, (+), gives a nonfix function
- Parenthesis around an infix operator and an argument gives a partially applied function.

(+) 1 2 ⇒ **1 + 2**	(1 +) 2 ⇒ **1 + 2**	(+ 1) 2 ⇒ **2 + 1**

Caveat: **`(– 2)`** is negative two, not subtract two. It is the one exception here. Use **`negate 2`** instead.

Dot and Dollar

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

- **.** composes two functions. The data flows from right to left.
- **$** is an infix version of function application.

Both these higher-order functions allow chaining of function.

```
GHCi> map (*2) $ map (+1) $ [1..10]
????
GHCi> let f = map (*2) . map (+1)
GHCi> f [1..10]
????
```

- use **$** when you are providing the final argument
- use **.** when composing functions

KU