EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Feb 12th 2020

©2020 Andrew Gill, 1/30,1745x982

Basic list functions

Remember lists, and list operations? First, look at the types

```
length :: [a] -> Int           -- length of a list
(++)   :: [a] -> [a] -> [a]    -- append two lists
null :: [a] -> Bool            -- is a list empty
head :: [a] -> a               -- take the first element of a list
tail :: [a] -> [a]             -- take the rest of a list
(:)   :: a -> [a] -> [a]       -- add a value to front of list
```

We are going to write all of these today.

©2020 Andrew Gill, 2/30,1745x886

Taking things to bits

Consider these examples:

```
GHCi> let swap (a,b) = (b,a)
GHCi> swap (1,2)
(2,1)
```

What is happening?

- The tuple is being deconstructed, into the variables **a** and **b**.
- A new tuple is being constructed, using **a** and **b**, swapped.

```
GHCi> let reverse [a,b,c] = [c,b,a]
GHCi> reverse [1,2,3]
[3,2,1]
GHCi> reverse [1,2]
*** Exception: <interactive>:2:5-29: Non-exhaustive patterns in function reverse
```

How do we generalize this to work over **any** length of list?

©2020 Andrew Gill, 3/30,1745x886

The truth about lists

`[..,..,..]` is just syntactical sugar for building finite, fixed sized lists.
Instead, we can build list inductively.

- An empty list is constructed by using `[]`.
- An non-empty list is constructed by using a value and another list. This operation is called "cons", and written as infix `:` in Haskell.

The `:` operator associates to the right. This means we can write:

```
1 : 2 : 3 : []
```

This list is **identical** to the list generated by `[1,2,3]`.

```
add0 :: [Int] -> [Int]
add0 xs = 0 : xs  -- could not write using [..,..,..] notation
```

©2020 Andrew Gill, 4/30,1745x886

Table of value identifiers and symbols

What	Syntax-rule	Description	Example
name	start with Upper	Constructor	`True` or `False`
name	start with lower	variable	`x` or `abc`
symbol	start with ':'	infix Constructor	`:`
symbol	not starting with ':'	infix variable	+ or ^
specials		tuples, lists	`(...,...)` or `[1,2,3]`

infix to nonfix: **1 + 2 ⇒ (+) 1 2**

nonfix to infix: **mod x y ⇒ x \`mod\` y**

©2020 Andrew Gill, 5/30,1745x886

Table of type identifiers and symbols

What	Syntax-rule	Description	Example
name	start with Upper	Fixed Type	**`Int`** or **`Bool`**
name	start with lower	type variable	**`a`** is universally quantified
specials		tuple type, list type	**`(...,...)`** or **`[Int]`**

©2020 Andrew Gill, 6/30,1745x886

Pattern matching in Haskell

Both fixed-sized list notation and cons-list notation can be used for pattern matching.

```
head :: [a] -> a              -- take the first element of a list
head (x : xs) = x
tail :: [a] -> [a]            -- take the rest of a list
tail (x : xs) = xs
```

Both notations can be intermixed.

```
null :: [a] -> Bool           -- is a list empty
null []      = True
null (x:xs)  = False
```

- Here, the first equation is attempted, then if it fails, the second.
- This "pattern matching"is a form of control flow

©2020 Andrew Gill, 7/30,1745x886

Haskell functions and recursion

Many Haskell functions are recursive.
Canonical example: factorial function.

```
fac :: Int -> Int
fac 0 = 1
fac n = n * fac (n-1)
```

Another way of writing, using **`if then else`**.

```
fac :: Int -> Int
fac n = if n == 0 then 1 else n * fac (n-1)
```

©2020 Andrew Gill, 8/30,1745x886

Common way of acting over a list

Write a function that adds **1** to every element of a list.

```
adder :: [Int] -> [Int]
adder []       = []
adder (x:xs) = x + 1 : adder xs
```

©2020 Andrew Gill, 9/30,1745x886

Lets write the length function

We count the cons cells, recursively.

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

If a value is ignored, you can say so.

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

©2020 Andrew Gill, 10/30,1745x886

fromto function

Here is what we want to function to do

```
GHCi> fromto (1,10)
[1,2,3,4,5,6,7,8,9,10]
```

First attempt, using tuples

```
fromto :: (Int,Int) -> [Int]
fromto (n,m) = if n > m then [] else n : fromto (n+1,m)
```

©2020 Andrew Gill, 11/30,1745x886

fromto function Curryed (a.k.a. Haskell B Curry)

Can we make this neater?

```
GHCi> fromto 1 10
[1,2,3,4,5,6,7,8,9,10]
```

Second attempt, using currying

```
fromto :: Int -> Int -> [Int]
fromto n m = if n > m then [] else n : fromto (n+1) m
```

©2020 Andrew Gill, 12/30,1745x886

Curry to go

The principle of currying is simple:

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

But what about zip?

```
zip :: [a] -> [b] -> [(a,b)]
```

zip really has type

```
zip :: [a] -> ([b] -> [(a,b)])
```

- Key idea: **->** groups to the right
- **All functions always have one argument**

Let's write the append function

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ...
(x:xs) ++ ys = ...
```

Solution

```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : xs ++ ys
```

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

KU

©2020 Andrew Gill, 15/30,1745x886

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.

©2020 Andrew Gill, 17/30,1745x886

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?

©2020 Andrew Gill, 18/30,1745x886

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`**?

©2020 Andrew Gill, 20/30,1745x886