EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Feb 10th 2020

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.

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?

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
```

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`**

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]`**

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

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)
```

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
```

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
```

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)
```

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
```

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
```

Bring back lists

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
```

What might a function of this type do? **`? :: [[a]] -> [a]`**

```
concat :: [[a]] -> [a]                -- flatten a list
```

What might this function of this type do? **`? :: [(a,b)] -> ([a],[b])`**

```
unzip :: [(a,b)] -> ([a],[b])   -- split a list of pairs
into a pair of lists
```