An Introduction to Higher-Order Functions

Prof. Susan Older

14 October 2019

Déjà Vu: Doubling Each Element of a List

Multiply every element of a list by 2:

```
-- by recursion
doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

```
-- by list comprehension
doubleAll :: [Int] -> [Int]
doubleAll lst = [ 2*x | x <- lst]
```

$$\begin{array}{rcccccc} & x & x & x & x & \\ \texttt{doubleAll}\ [& 5, & 3, & 8, & 7] & \\ \rightsquigarrow & \downarrow & \downarrow & \downarrow & \downarrow & 2*x \\ [& 10, & 6, & 16, & 14] & \end{array}$$

Capitalizing Each Element of a List

Convert each element of string to uppercase:

```
-- by recursion
capitalize :: String -> String
capitalize [] = []
capitalize (c:cs) = toUpper c : capitalize cs
```

```
-- by list comprehension
capitalize :: String -> String
capitalize cs = [ toUpper c | c <- cs]
```

$$\begin{array}{rcccccc} & c & c & c & c & \\ \texttt{capitalize}\ [& 'r', & '2', & 't', & '!'] & \\ \rightsquigarrow & \downarrow & \downarrow & \downarrow & \downarrow & \texttt{toUpper c} \\ [& 'R', & '2', & 'T', & '!'] & \end{array}$$

"Listifying" Each Element of a List

Add the components of each pair and make a list:

```
-- by recursion
addPairs :: [(Int,Int)] -> [[Int]]
addPairs [] = []
addPairs ((m,n):rest) = [m+n] : addPairs rest
```

```
-- by list comprehension
addPairs :: [(Int,Int)] -> [[Int]]
addPairs pairList = [[m+n] | (m,n) <- pairList]
```

$$\begin{array}{rcccccc} & (m,n) & (m,n) & (m,n) & (m,n) & \\ \texttt{addPairs}\ [& (2,3), & (8,0), & (1,2), & (10,6)] & \\ \rightsquigarrow & \downarrow & \downarrow & \downarrow & \downarrow & \texttt{[m+n]} \\ [& [5], & [8], & [3], & [16]] & \end{array}$$

Capturing the Computational Pattern: Mapping via map

Haskell takes it one step further and defines a general way to do this:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

```
map :: (a -> b) -> [a] -> [b]
map f lst = [ f x | x <- lst]
```

map is defined in the Prelude:

$$\begin{array}{rcccccc} & & x & x & & x & \\ \texttt{map f} & [& a_1, & a_2, & \cdots, & a_k] & \\ \rightsquigarrow & & \downarrow & \downarrow & & \downarrow & \texttt{f x} \\ & [& f\ a_1, & f\ a_2, & \cdots, & f\ a_k] & \end{array}$$

Examples using map

```
> map fst [(1,False), (3,True), (-5, False)]
> map length [[1,5,6], [], [3,4]]
> map product [[1,5,6], [], [3,4]]
> map toUpper "We want cake! Where's our cake?"
```

- `map fst [(1,False), (3,True), (-5, False)]`
  `= [fst (1,False), fst (3,True), fst (-5, False)]`
  `= [1, 3, -5]`
- `map length [[1,5,6], [], [3,4]]`
  `= [length [1,5,6], length [], length [3,4]]`
  `= [3,0,2]`
- `map product [[1,5,6], [], [3,4]]`
  `= [product [1,5,6], product [], product [3,4]]`
  `= [30,1,12]`

Another Common Computational Pattern: Filtering

Grab elements that satisfy a given property:

```
getEvens :: [Int] -> [Int]
getEvens lst = [ x | x <- lst, even x]
```

```
halves :: [(Int,Int)] -> [(Int,Int)]
halves pairs = [(m,n) | (m,n) <- pairs, n == 2*m]
```

```
getLowers :: String -> String
getLowers cs = [ c | c <- cs, isLower c]
```

Haskell includes the following built-in function:

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


Filter in Action: getEvens

Filtering numbers from a list:

```
getEvens :: [Int] -> [Int]
getEvens lst = filter even lst

-- getEvens lst = [ x | x <- lst, even x]
```

$$\begin{array}{rcccccc} & & x & x & x & x & \\ \texttt{getEvens} & [& 10, & 13, & 8, & 2] & \\ \rightsquigarrow & & \downarrow & \downarrow & \downarrow & \downarrow & \textit{even } x == \textit{True?} \\ & [& 10, & & 8, & 2] & \end{array}$$

Filter in Action: `getLowers`

Filtering characters from a list:

```
getLowers :: String -> String
getLowers cs = filter isLower cs

        --   = [ c | c <- cs, isLower c]
```

```
                  c    c    c    c
getLowers  [ 'a', 'B', '?', 'r']
      ⇝     ↓    ↓    ↓    ↓     isLower c == True?
           [ 'a',                'r']
```


Recap: Another Look at `filter`

Haskell includes the following built-in function:

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

`filter`'s argument `p` is a predicate:
that is, `p` is a function that returns a `Bool`.

Examples of `filter`'s use:

```
> filter even [10,13,1,12,26,33]
> filter isLower "a37\nbZ8"
> filter isDigit "a37\nbZ8"
> filter isControl "a37\nbZ8"
```


Functions as First-Class Values

In Haskell, functions are first-class values, which means that they are treated the same as all other values:

- Can be passed as arguments to functions
- Can be returned as results of functions
- Can be bound to variables
- Can be expressed without being given a name
- Can appear in tuples, lists, etc.

⋮

A function that accepts functions as arguments or returns functions as results (or both!) is said to be higher order:

Both `map` and `filter` are higher order.

Anonymous Functions: What's in a Name?

You don't have to name every expression in Haskell:

Consider `max (length [1,3,6]) (product [2,4,6])`:

- `(length [1,3,6])` is an expression that returns an `Int`
- `(product [2,4,6])` is an expression that returns an `Int`
- We can use these expressions without giving them names.

You don't have to name every function either:

1. `(\x -> 4*x+1) 100` ⇝ `401`
2. `(\a b -> 10*b + a) 7 3` ⇝ `37`
3. `map (\x -> 4*x+1) [3,5,10]` ⇝ `[13,21,41]`
4. `map (\(x,y) -> 10*x + y) [(3,1),(2,6),(5,2),(12,7)]`
   ⇝ `[31,26,52,127]`
5. `filter (\(x,y) -> x > y) [(3,1),(2,6),(5,2),(12,7)]`
   ⇝ `[(3,1),(5,2),(12,7)]`