Programming in Haskell: Lecture 10

S P Suresh

September 11, 2019

Higher-order functions

- Most functions produce a function as result

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`
- Hence `apply :: (a -> b) -> a -> b`

Higher-order functions

- Most functions produce a function as result
- We can also pass functions as arguments
- Example: `apply f x = f x`
- What is its type?
- A generic function `f` has type `a -> b`
- Second argument `x` is also input to `f`
- Output `apply f x` is the same as `f x`
- Hence `apply :: (a -> b) -> a -> b`
- Same as the built-in `($)`

The built-in function `map`

```
capitalize :: String -> String
capitalize ""     = ""
capitalize (c:cs) = toUpper c: capitalize cs

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list

The built-in function `map`

```
capitalize :: String -> String
capitalize ""     = ""
capitalize (c:cs) = toUpper c: capitalize cs

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list
- Built in function `map` achieves this

The built-in function **map**

```haskell
capitalize :: String -> String
capitalize ""      = ""
capitalize (c:cs) = toUpper c: capitalize cs

sqrList :: [Integer] -> [Integer]
sqrList []        = []
sqrList (x:xs)    = x^2 : sqrList xs
```

- Common pattern: apply a function `f` to each member in a list
- Built in function **`map`** achieves this
- **`map`** `f [x0, x1, ..., xk] ---> [f x0, f x1, ..., f xk]`

The built-in function **map**

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

The built-in function map

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

- Given a list of lists, sum the lengths of inner lists

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

The built-in function map

- Some examples

```
map (+ 3) [2,6,8] = [5,9,11]
map (* 2) [2,6,8] = [4,12,16]
map (^2) [1,2,3,4] = [1,4,9,16]
```

- Given a list of lists, sum the lengths of inner lists

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

- Can be written using `map` as:

```
sumLength l = sum (map length l)
```

The built-in function `map`

- The function `map`

```
map f []     = []
map f (x:xs) = f x: map f xs
```

The built-in function `map`

- The function `map`

```
map f []     = []
map f (x:xs) = f x: map f xs
```

- What is the type of `map`?

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

The built-in function **filter**

- Select all even numbers from a list

```
allEvens :: [Int] -> [Int]
allEvens []                  = []
allEvens (x:xs) | even x     = x: allEvens xs
                | otherwise = allEvens xs
```

The built-in function **filter**

- Select all even numbers from a list

```
allEvens :: [Int] -> [Int]
allEvens []                   = []
allEvens (x:xs) | even x      = x: allEvens xs
                | otherwise = allEvens xs
```

- Abstract pattern:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x:xs) | p x         = x: filter p xs
                | otherwise = filter p xs
allEvens = filter even
```

Combining **map** *and* **filter**

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

Combining **map** *and* **filter**

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

- Extract all vowels in a string and capitalize them

```
capVows :: String -> String
capVows = map toUpper . filter isVow
isVow c = c `elem` "aeiou"
```

Combining **map** *and* **filter**

- Squares of even numbers in a list

```
sqrEvens :: [Int] -> [Int]
sqrEvens l = map (^2) $ filter even l
```

- Extract all vowels in a string and capitalize them

```
capVows :: String -> String
capVows = map toUpper . filter isVow
isVow c = c `elem` "aeiou"
```

- `(.)` denotes function composition: `(f . g) e = f (g e)`

New lists from old

- Set comprehension in mathematics

New lists from old

- Set comprehension in mathematics
- $M = \{x^2 \mid x \in L, \text{even}(x)\}$

New lists from old

- Set comprehension in mathematics
- $M = \{x^2 \mid x \in L, \text{even}(x)\}$
- Generates a new set M from a given set L

New lists from old

- Set comprehension in mathematics
- $M = \{x^2 \mid x \in L, \text{even}(x)\}$
- Generates a new set M from a given set L
- Haskell allows this almost verbatim:

```
m = [x^2 | x <- l, even x]
```

New lists from old

- Set comprehension in mathematics
- $M = \{x^2 \mid x \in L, \text{even}(x)\}$
- Generates a new set M from a given set L
- Haskell allows this almost verbatim:

```
m = [x^2 | x <- l, even x]
```

- List comprehension, combines `map` and `filter`

Examples

- All divisors of `x`

```
divisors x = [y | y <- [1..x], x `mod` y == 0]
```

Examples

- All divisors of `x`

```
divisors x = [y | y <- [1..x], x `mod` y == 0]
```

- All primes below `x`

```
primes x = [y | y <- [1..x], divisors y == [1,y]]
```

Examples

- Can use multiple generators

Examples

- Can use multiple generators
- Pairs of integers below `10`

```
[(x,y) | x <- [1..10], y <- [1..10]]
```

Examples

- Can use multiple generators
- Pairs of integers below `10`

```
[(x,y) | x <- [1..10], y <- [1..10]]
```

- Like nested loops, later generators move faster

```
[(1,1), (1,2), ..., (1,10), (2,1), ..., (2,10),
                        ..., (10,1), ..., (10,10)]
```

- All Pythagorean triples below `100`

```
[(x,y,z) | x <- [1..100],
           y <- [1..100],
           z <- [1..100],
           x^2 + y^2 == z^2]
```

Examples

- All Pythagorean triples below `100`

```
[(x,y,z) | x <- [1..100],
           y <- [1..100],
           z <- [1..100],
           x^2 + y^2 == z^2]
```

- Oops, that has duplicates!

```
[(x,y,z) | x <- [1..100],
           y <- [(x+1)..100],
           z <- [(y+1)..100],
           x^2 + y^2 == z^2]
```

Examples

- All Pythagorean triples below `100`

```
[(x,y,z) | x <- [1..100],
           y <- [1..100],
           z <- [1..100],
           x^2 + y^2 == z^2]
```

- Oops, that has duplicates!

```
[(x,y,z) | x <- [1..100],
           y <- [(x+1)..100],
           z <- [(y+1)..100],
           x^2 + y^2 == z^2]
```

- Later lists can refer to earlier generators

Examples

- The built-in function `concat`

```
concat ls = [x | l <- ls, x <- l]
```

Examples

- The built-in function **concat**

```
concat ls = [x | l <- ls, x <- l]
```

- Given a list of lists, extract the head of all even-length non-empty lists

```
headEvens ls = [head l | l <- ls, length l > 0,
                                  even (length l)]
```

Examples

- The built-in function **concat**

```
concat ls = [x | l <- ls, x <- l]
```

- Given a list of lists, extract the head of all even-length non-empty lists

```
headEvens ls = [head l | l <- ls, length l > 0,
                                  even (length l)]
```

- Can use patterns instead of names

```
headEvens ls = [x | (x:xs) <- ls, even (length (x:xs))]
```

Translating list comprehension

- List comprehension can be written in terms of `map`, `filter` and `concat`

Translating list comprehension

- List comprehension can be written in terms of `map`, `filter` and `concat`
- A list comprehension has the form

```
    [e | q1, q2, ..., qN]
```

Translating list comprehension

- List comprehension can be written in terms of **`map`**, **`filter`** and **`concat`**
- A list comprehension has the form

```
[e | q1, q2, ..., qN]
```

- Each `qi` is:

Translating list comprehension

- List comprehension can be written in terms of **`map`**, **`filter`** and **`concat`**
- A list comprehension has the form

```
[e | q1, q2, ..., qN]
```

- Each `qi` is:
  - either a boolean condition `b`

Translating list comprehension

- List comprehension can be written in terms of **`map`**, **`filter`** and **`concat`**
- A list comprehension has the form

```
[e | q1, q2, ..., qN]
```

- Each `qi` is:
  - either a boolean condition `b`
  - or a generator `p <- l`, where `p` is a pattern and `l` is a list-valued expression

Translating list comprehension

- A boolean condition acts as a filter

```
[e | b, Q] = if b then [e | Q] else []
```

Translating list comprehension

- A boolean condition acts as a filter

```
[e | b, Q] = if b then [e | Q] else []
```

- Depends only on generators or qualifiers to its left

Translating list comprehension

- A boolean condition acts as a filter

```
[e | b, Q] = if b then [e | Q] else []
```

- Depends only on generators or qualifiers to its left
- A generator `p <- l` produces a list of candidates

```
[e | p <- l, Q] = concat $ map f l
    where
        f p     = [e | Q]
        f _     = []
```

Translating list comprehension

- A boolean condition acts as a filter

```
[e | b, Q] = if b then [e | Q] else []
```

- Depends only on generators or qualifiers to its left
- A generator `p <- l` produces a list of candidates

```
[e | p <- l, Q] = concat $ map f l
    where
        f p     = [e | Q]
        f _     = []
```

- `concat $ map f l` is very common

Translating list comprehension

- A boolean condition acts as a filter

```
[e | b, Q] = if b then [e | Q] else []
```

- Depends only on generators or qualifiers to its left
- A generator `p <- l` produces a list of candidates

```
[e | p <- l, Q] = concat $ map f l
    where
        f p     = [e | Q]
        f _     = []
```

- `concat $ map f l` is very common
- Built-in function: `concatMap f l = concat $ map f l`

Translation example

```
     [n^2 | n <- [1..7], even n]
---> concatMap f [1..7]
         where f n = [n^2 | even n]
---> concatMap f [1..7]
         where f n = if even n then [n^2] else []
---> concat [[], [4], [], [16], [], [36], []]
---> [4, 16, 36]
```

Example: generating primes

- Start with the infinite list `[2,3,4,...]`

Example: generating primes

- Start with the infinite list `[2,3,4,...]`
- The head is a prime

Example: generating primes

- Start with the infinite list `[2,3,4,...]`
- The head is a prime
- Remove all its multiples from the tail and recursively compute primes in that list

Example: generating primes

- Start with the infinite list `[2,3,4,...]`
- The head is a prime
- Remove all its multiples from the tail and recursively compute primes in that list
- Haskell program:

```
primes = sieve [2..]
    where
    sieve (p:xs) = p:sieve [x | x <- xs, x `mod` p /= 0]
```

Example: generating primes

- Start with the infinite list `[2,3,4,...]`
- The head is a prime
- Remove all its multiples from the tail and recursively compute primes in that list
- Haskell program:

```haskell
primes = sieve [2..]
    where
    sieve (p:xs) = p:sieve [x | x <- xs, x `mod` p /= 0]
```

- The n^{th} prime is `primes!!(n-1)`

Example: generating primes

```
     primes
---> sieve [2..]
---> 2:sieve [x | x <- [3..], x `mod` 2 /= 0]
---> 2:sieve (3:[x | x <- [4..], x `mod` 2 /= 0])
---> 2:3:sieve [y | y <- [x | x <- [4..], x `mod` 2 /= 0], y `mod` 3 /= 0]
---> 2:3:sieve [y | y <- [x | x <- [5..], x `mod` 2 /= 0], y `mod` 3 /= 0]
---> 2:3:sieve [y | y <- 5:[x | x <- [6..], x `mod` 2 /= 0], y `mod` 3 /= 0]
---> 2:3:sieve (5:[y | y <- [x | x <- [6..], x `mod` 2 /= 0],
                                         y `mod` 3 /= 0])
---> 2:3:5:sieve [z <- [y | y <- [x | x <- [6..], x `mod` 2 /= 0],
                                      y `mod` 3 /= 0],
                                      z `mod` 5 /= 0]
---> ...
```

The built-in function `takeWhile`

- **`take`** `n l` returns `n`-element prefix of list `l`

The built-in function **`takeWhile`**

- **`take`** `n l` returns `n`-element prefix of list `l`
- Instead, use a property to determine the prefix

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10]  = [8]
takeWhile (< 10) [8,1,9,10] = [8,1,9]
```

The built-in function **`takeWhile`**

- **`take`** `n l` returns `n`-element prefix of list `l`
- Instead, use a property to determine the prefix

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10]  = [8]
takeWhile (< 10) [8,1,9,10] = [8,1,9]
```

- `position c s` returns the first position in `s` where `c` occurs (or **`length`** `s`):

```
position c s = length $ takeWhile (/= c) s
```

The built-in function **takeWhile**

- `take n l` returns `n`-element prefix of list `l`
- Instead, use a property to determine the prefix

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile (> 7) [8,1,9,10]  = [8]
takeWhile (< 10) [8,1,9,10] = [8,1,9]
```

- `position c s` returns the first position in `s` where `c` occurs (or `length s`):

```
position c s = length $ takeWhile (/= c) s
```

- `dropWhile` is the analogue of `drop`

`zip` *and* `zipWith`

- `zip` forms a list of pairs from two lists

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

zip *and* zipWith

- zip forms a list of pairs from two lists

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

- zipWith combines two lists using a function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f   []     _      = []
zipWith f   _      []     = []
zipWith f   (x:xs) (y:ys) = f x y:zipWith f xs ys
```

`zip` *and* `zipWith`

- `zip` forms a list of pairs from two lists

```
zip :: [a] -> [b] -> [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y):zip xs ys
```

- `zipWith` combines two lists using a function

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f   []     _      = []
zipWith f   _      []     = []
zipWith f   (x:xs) (y:ys) = f x y:zipWith f xs ys
```

- `zipWith (+) [0,2,4,6,8] [1,3,5,7] = [1,5,9,13]`