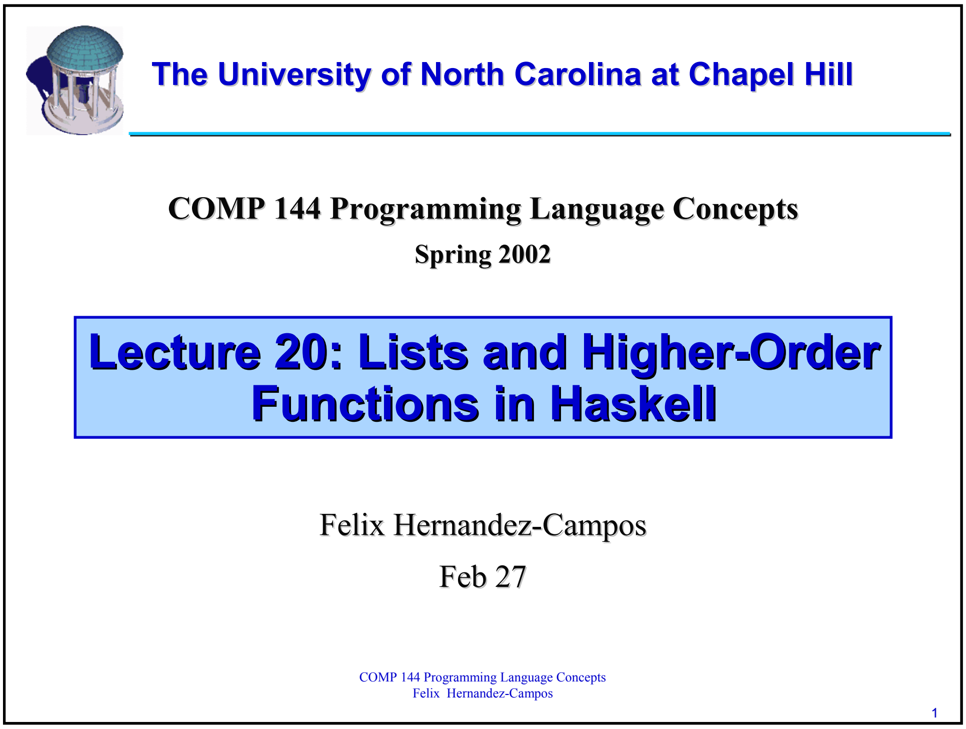


The University of North Carolina at Chapel Hill

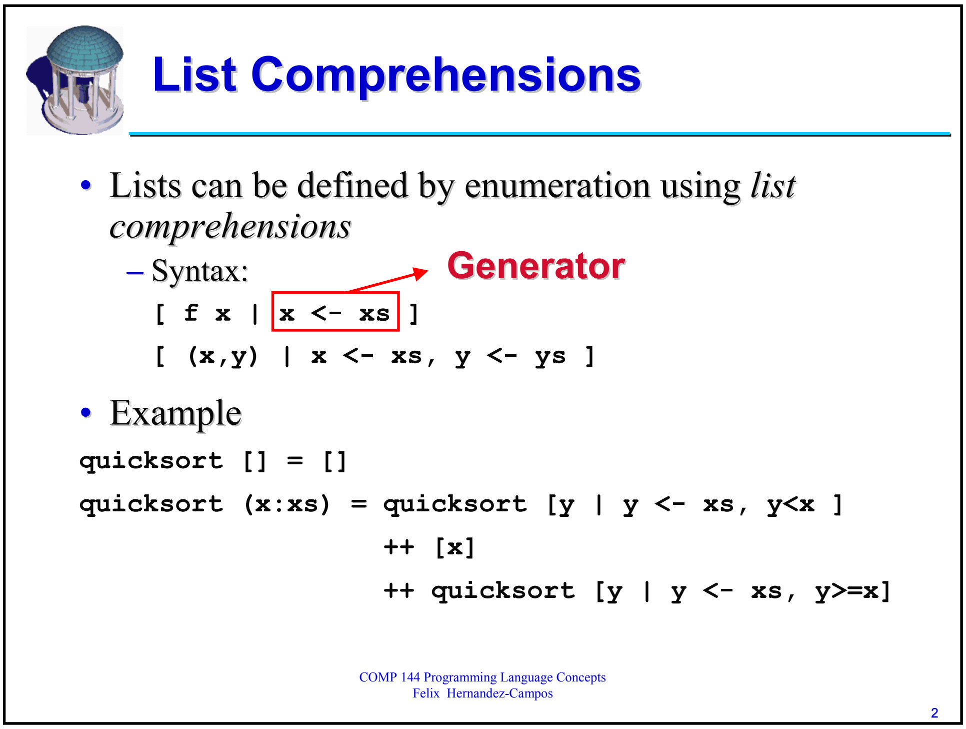
COMP 144 Programming Language Concepts

Spring 2002

Lecture 20: Lists and Higher-Order Functions in Haskell

Felix Hernandez-Campos

Feb 27



List Comprehensions

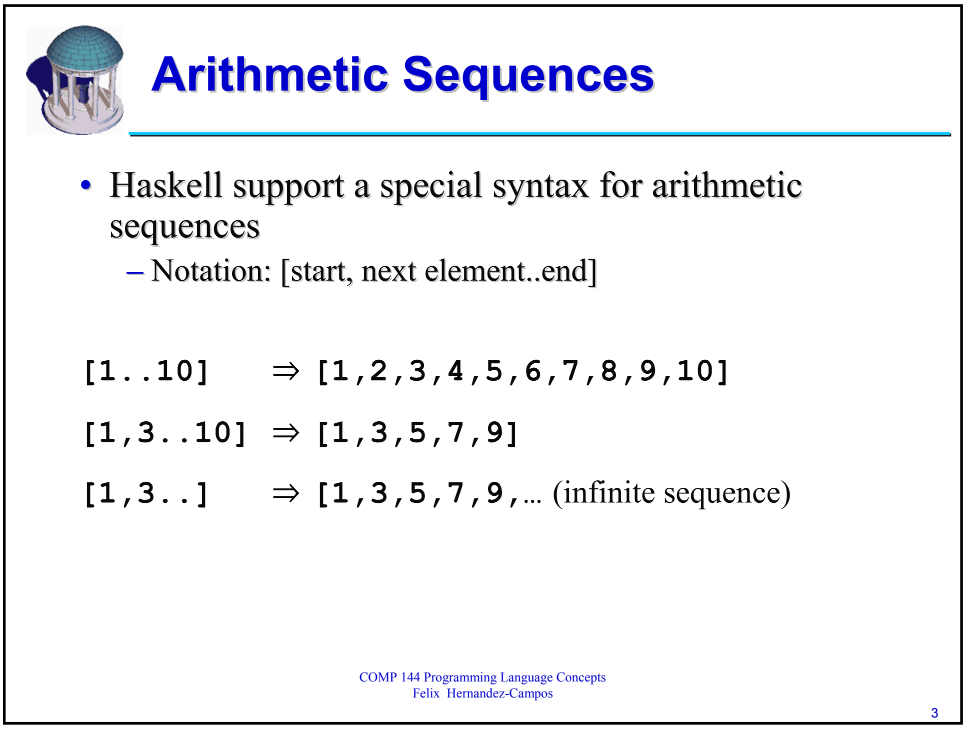
- Lists can be defined by enumeration using *list comprehensions*
  - Syntax:

Generator

```
[ f x | x <- xs ]
[ (x,y) | x <- xs, y <- ys ]
```

- Example

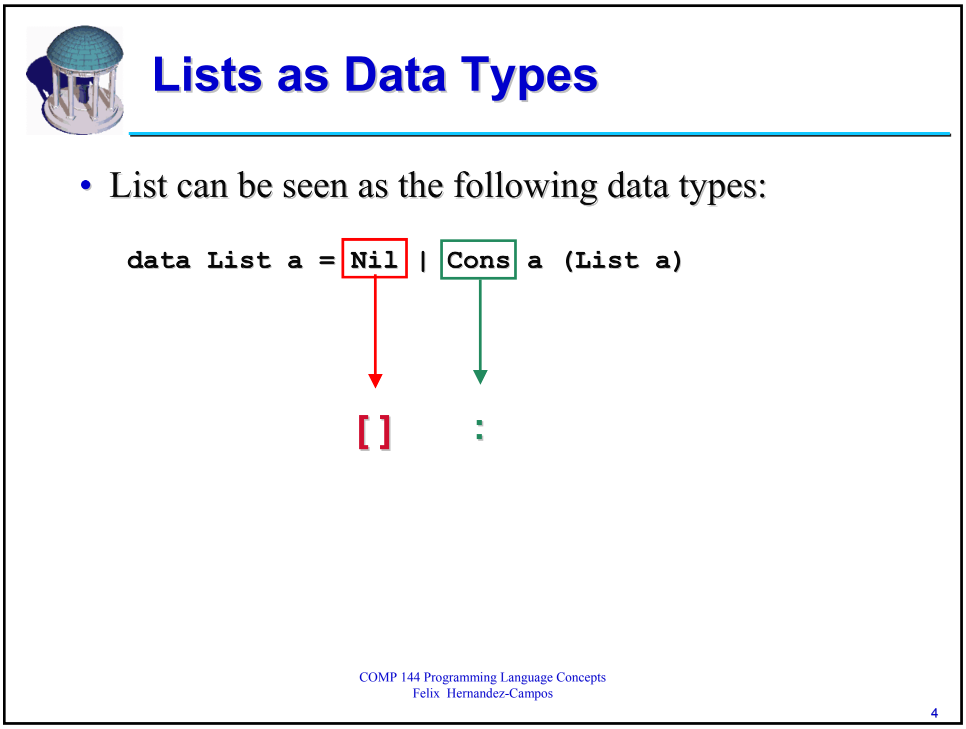
```
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs, y<x ]
                   ++ [x]
                   ++ quicksort [y | y <- xs, y>=x]
```



Arithmetic Sequences

- Haskell support a special syntax for arithmetic sequences
  - Notation: [start, next element..end]

```
[1..10]    ⇒ [1,2,3,4,5,6,7,8,9,10]

[1,3..10]  ⇒ [1,3,5,7,9]

[1,3..]    ⇒ [1,3,5,7,9,… (infinite sequence)
```



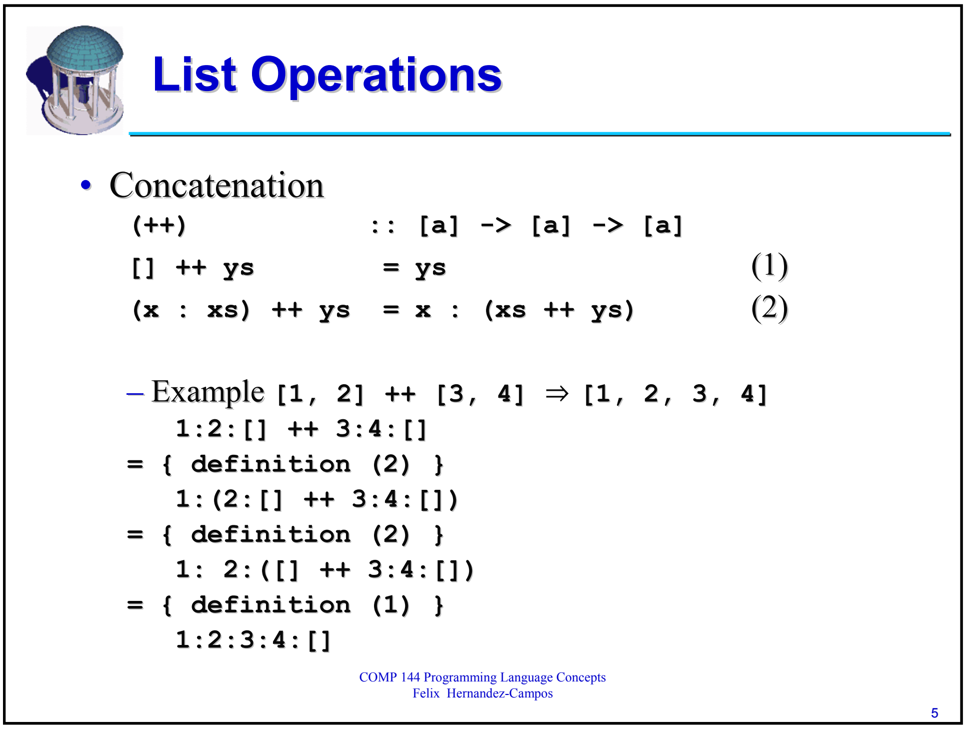
Lists as Data Types

- List can be seen as the following data types:

```
data List a = Nil | Cons a (List a)
```

Nil ↓ []

Cons ↓ :



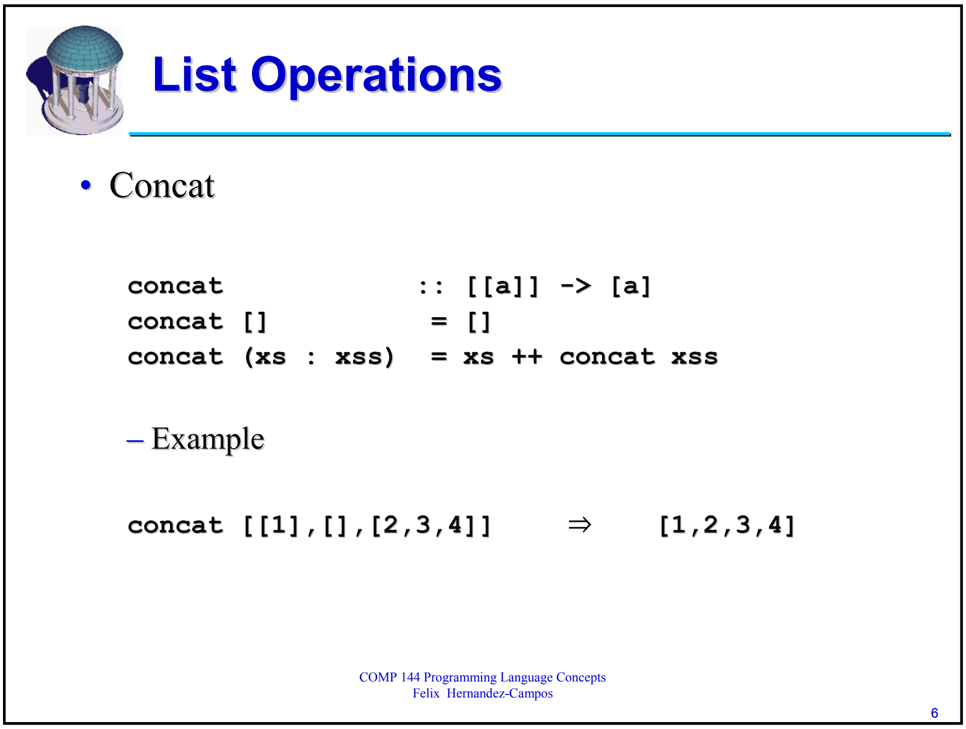
List Operations

- Concatenation

```
(++)              :: [a] -> [a] -> [a]
[] ++ ys           = ys                       (1)
(x : xs) ++ ys     = x : (xs ++ ys)           (2)
```

– Example `[1, 2] ++ [3, 4]` ⇒ `[1, 2, 3, 4]`

```
   1:2:[] ++ 3:4:[]
= { definition (2) }
   1:(2:[] ++ 3:4:[])
= { definition (2) }
   1: 2:([] ++ 3:4:[])
= { definition (1) }
   1:2:3:4:[]
```



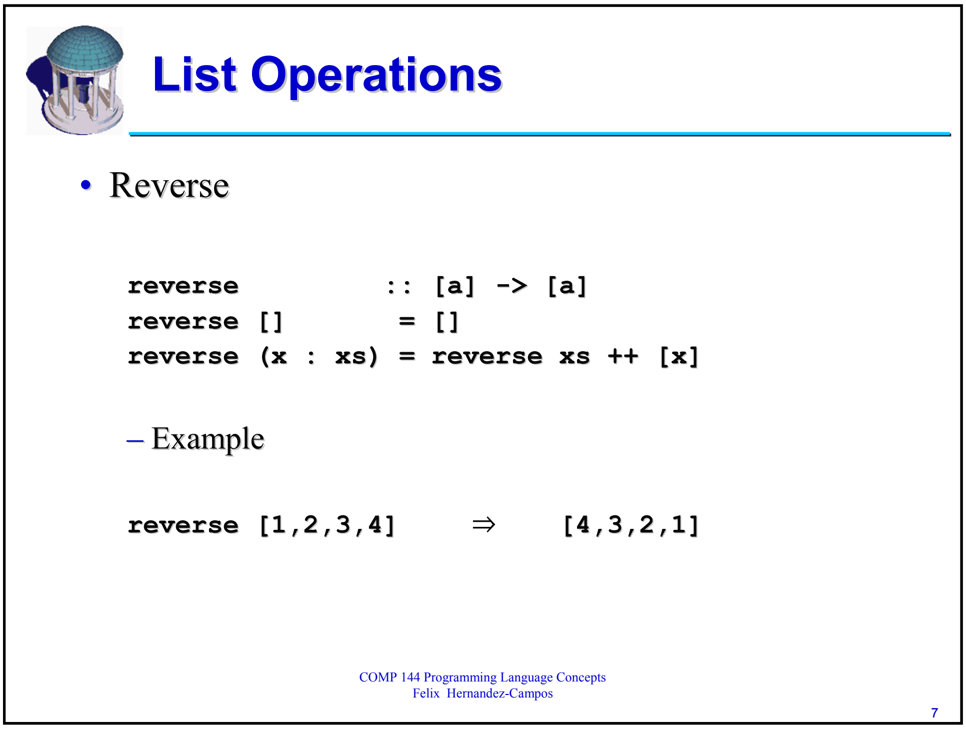
List Operations

- Concat

```
concat               :: [[a]] -> [a]
concat []             = []
concat (xs : xss)     = xs ++ concat xss
```

– Example

```
concat [[1],[],[2,3,4]]     ⇒     [1,2,3,4]
```



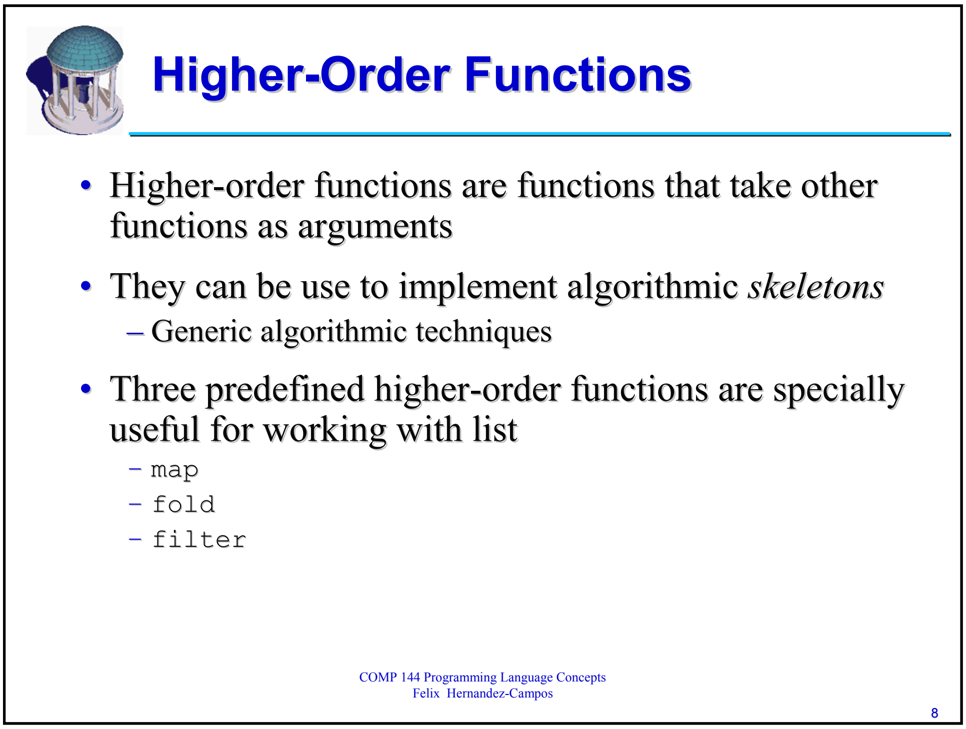
List Operations

- Reverse

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

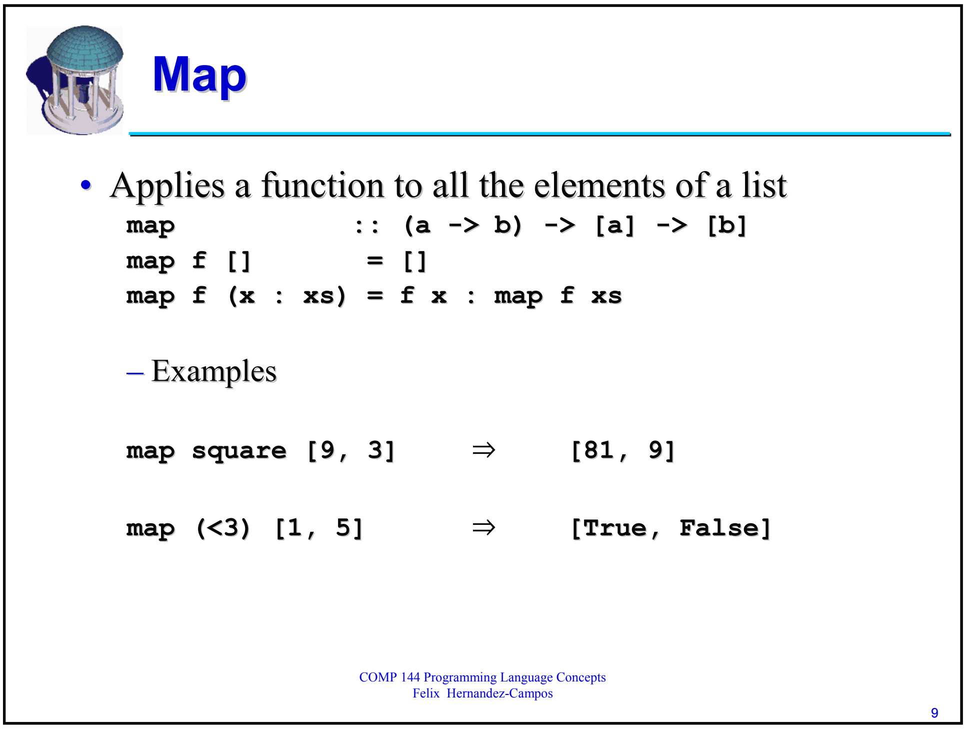
– Example

```
reverse [1,2,3,4]    ⇒    [4,3,2,1]
```



Higher-Order Functions

- Higher-order functions are functions that take other functions as arguments
- They can be use to implement algorithmic *skeletons*
  – Generic algorithmic techniques
- Three predefined higher-order functions are specially useful for working with list
  – `map`
  – `fold`
  – `filter`



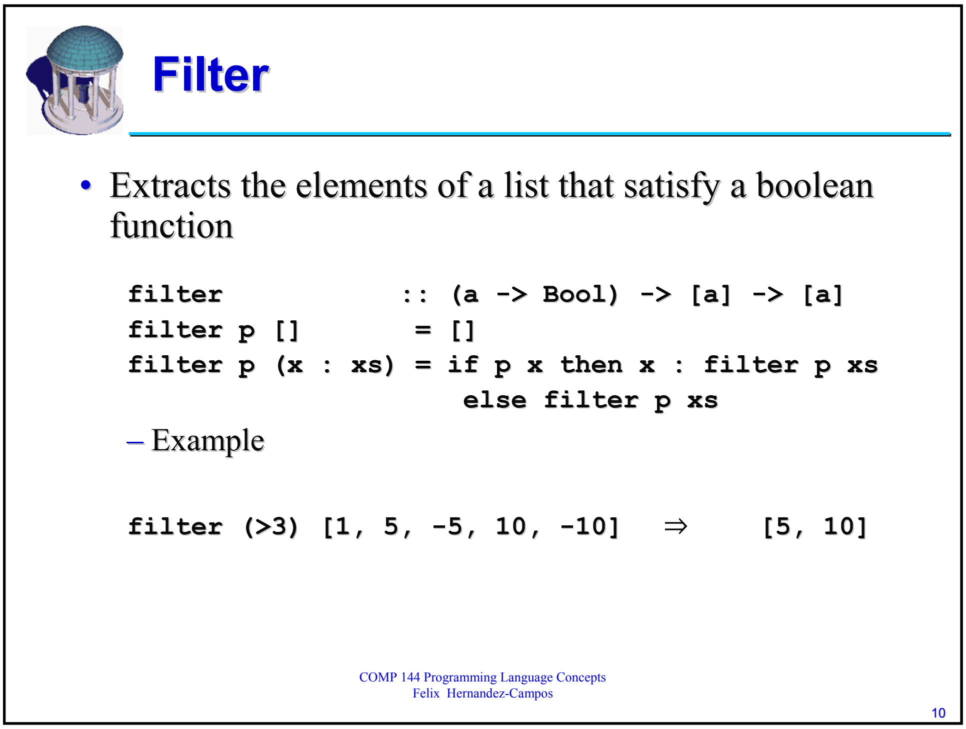
Map

- Applies a function to all the elements of a list

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

  – Examples

```
map square [9, 3]     ⇒     [81, 9]

map (<3) [1, 5]       ⇒     [True, False]
```



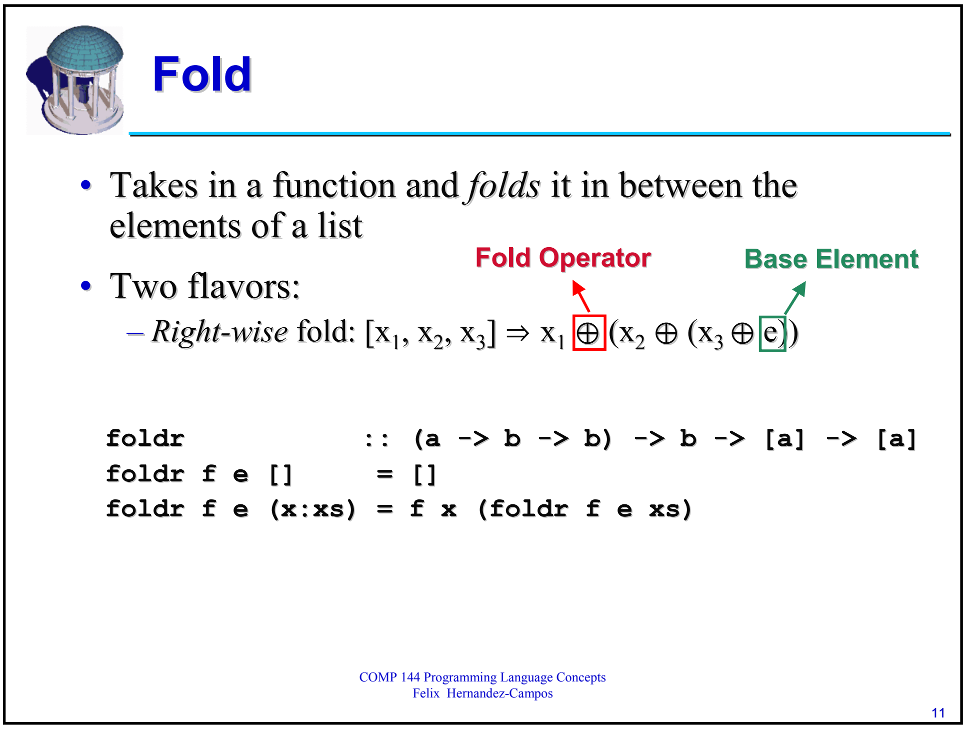
Filter

- Extracts the elements of a list that satisfy a boolean function

```
filter            :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x then x : filter p xs
                      else filter p xs
```

  – Example

```
filter (>3) [1, 5, -5, 10, -10]   ⇒     [5, 10]
```



Fold

- Takes in a function and *folds* it in between the elements of a list
- Two flavors:
  - *Right-wise* fold: $[x_1, x_2, x_3] \Rightarrow x_1 \boxed{\oplus} (x_2 \oplus (x_3 \oplus \boxed{e}))$

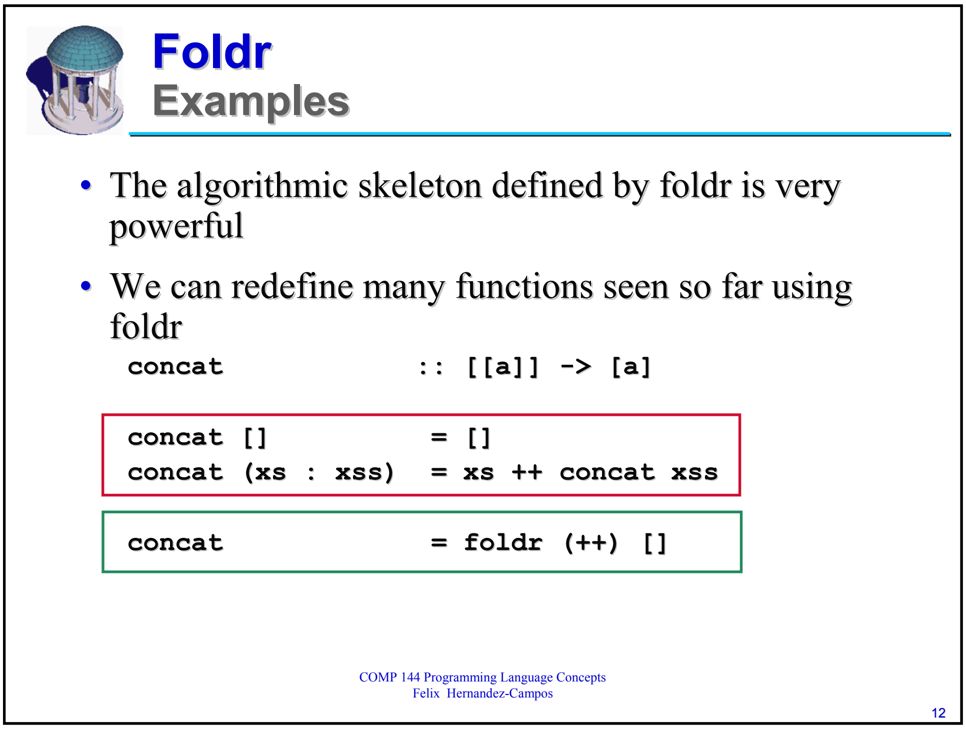
Fold Operator

Base Element

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

COMP 144 Programming Language Concepts
Felix Hernandez-Campos

11



Foldr Examples

- The algorithmic skeleton defined by foldr is very powerful
- We can redefine many functions seen so far using foldr

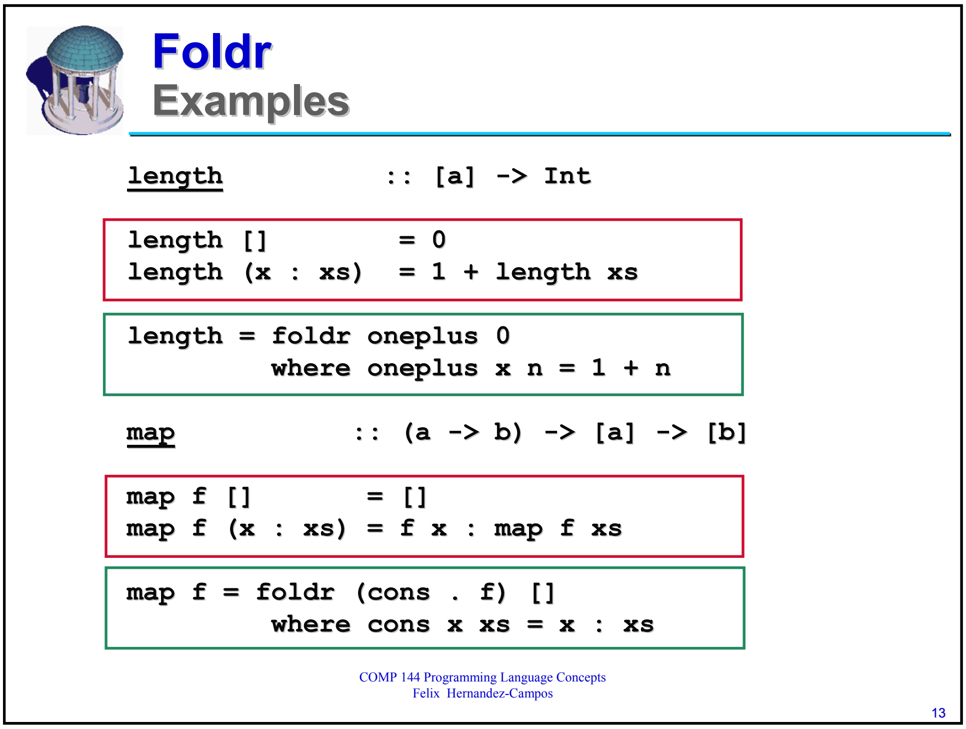
```
concat              :: [[a]] -> [a]
```

```
concat []            = []
concat (xs : xss)    = xs ++ concat xss
```

```
concat               = foldr (++) []
```

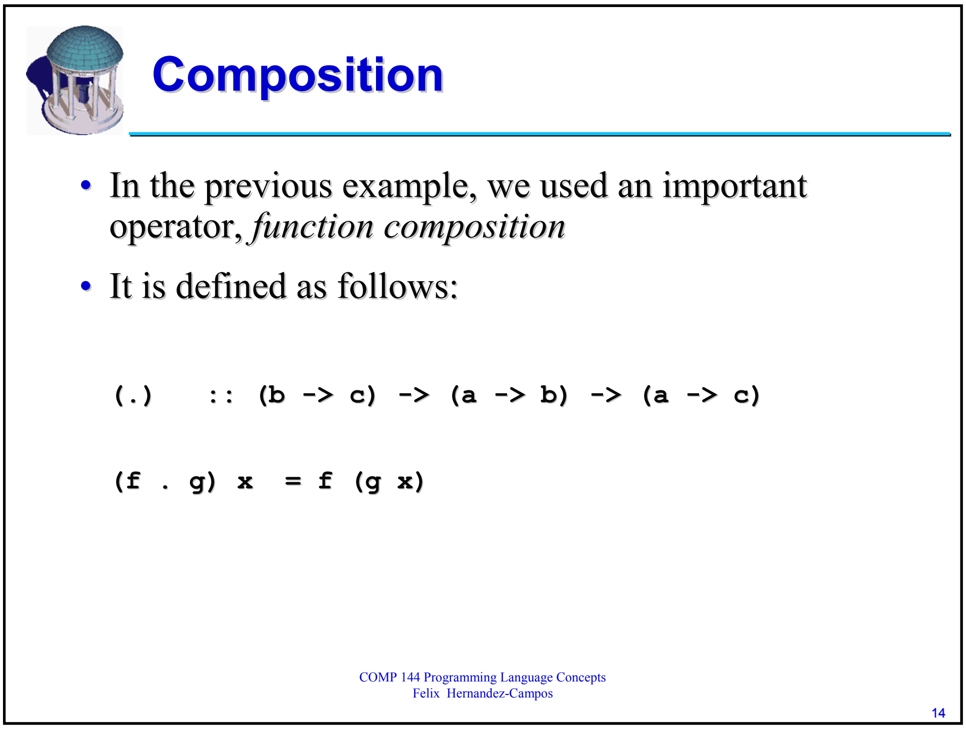
COMP 144 Programming Language Concepts
Felix Hernandez-Campos

12



Foldr
Examples

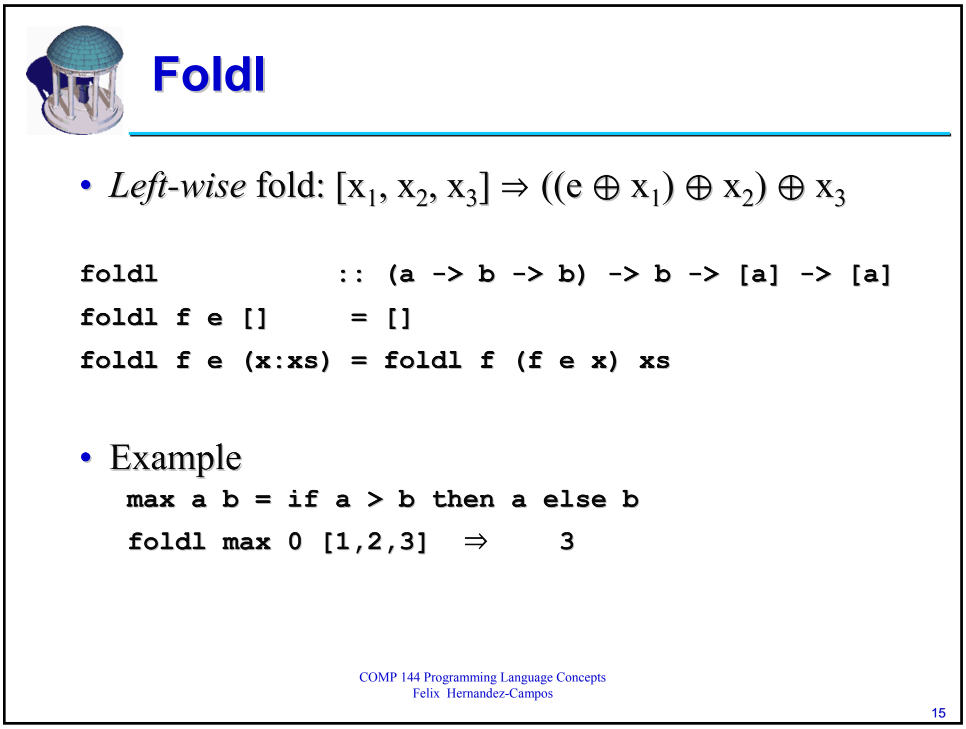
```
length           :: [a] -> Int

length []         = 0
length (x : xs)   = 1 + length xs

length = foldr oneplus 0
         where oneplus x n = 1 + n

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

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

map f = foldr (cons . f) []
         where cons x xs = x : xs
```



Composition

- In the previous example, we used an important operator, *function composition*
- It is defined as follows:

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

(f . g) x  = f (g x)
```



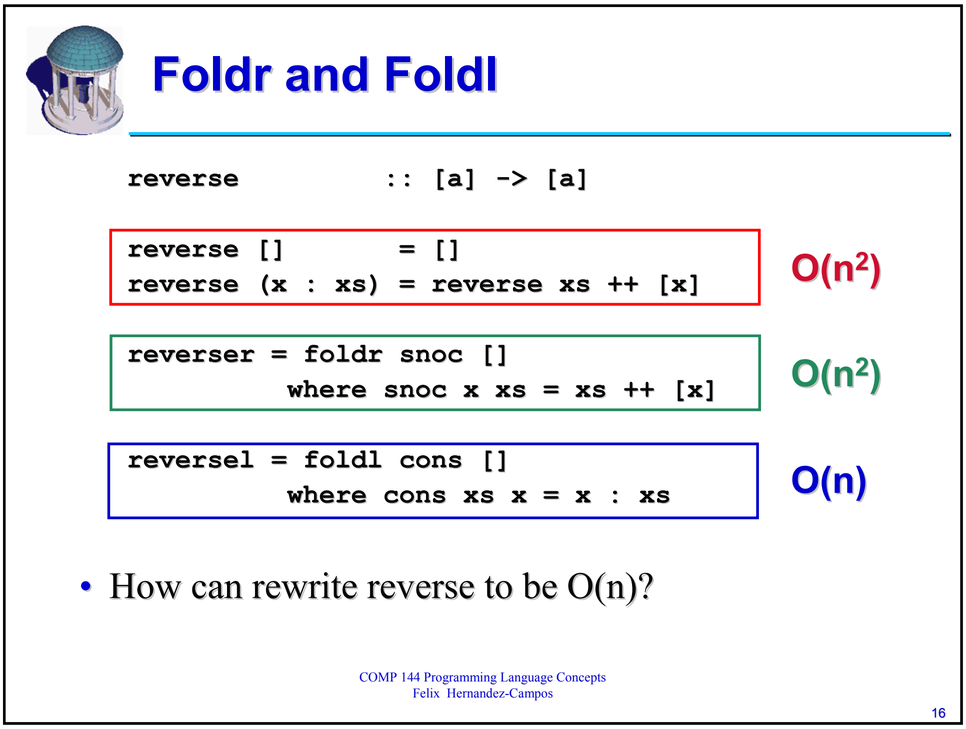
Foldl

- *Left-wise* fold: $[x_1, x_2, x_3] \Rightarrow ((e \oplus x_1) \oplus x_2) \oplus x_3$

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

- Example

```
max a b = if a > b then a else b
foldl max 0 [1,2,3]   ⇒     3
```



Foldr and Foldl

```
reverse          :: [a] -> [a]
```

```
reverse []         = []
reverse (x : xs)  = reverse xs ++ [x]
```

$O(n^2)$

```
reverser = foldr snoc []
             where snoc x xs = xs ++ [x]
```

$O(n^2)$

```
reversel = foldl cons []
             where cons xs x = x : xs
```

$O(n)$

- How can rewrite reverse to be O(n)?

Solution

```
rev    :: [a] -> [a]
rev xs = rev2 xs []

rev2            :: [a] -> [a] -> [a]
rev2 []    ys    = ys
rev2 (x:xs) ys = (rev2 xs) (x:ys)
```

Reading Assignment

- *A Gentle Introduction to Haskell* by Paul Hudak, John Peterson, and Joseph H. Fasel.
  - http://www.haskell.org/tutorial/
  - Read sections 3 and 4 (intro, 4.1-3)