# CS 457/557: Functional Languages

Lecture 3: Lists by many means…

Mark P Jones and Andrew Tolmach
Portland State University

# Why Study Lists?

- Lists are a heavily used data structure in many functional programs

- Special syntax is provided to make programming with lists more convenient

- Lists are a special case / an example of:
  - An algebraic datatype (coming soon)
  - A parameterized datatype (coming soon)
  - A monad (coming, but a little later)

# What is a List?

- An ordered collection (multiset) of values
  - [1,2,3,4], [4,3,2,1], [1,1,2,2,3,3,4,4] are distinct lists of integers
- A list of type [T] contains zero or more elements of type T
  - [True, False] :: [Bool]
  - [1,2,3] :: [Integer]
  - ['a', 'b', 'c'] :: [Char]
  - [[],[1],[1,2],[1,2,3]] :: [[Integer]]
- All elements have the same type:
  - [True, 2, 'c'] is not a valid list

# Naming Convention

- We often use a simple naming convention:
- If a typical value in a list is called x, then a typical list of such values might be called xs (i.e., the plural of x)
- ... and a list of lists of values called x might be called xss
- A simple convention, minimal clutter, and a useful mnemonic too!

# How do you make a list?

- The empty list, [], which has type [a] for any (element) type a
- Enumerations: [$e_1$, $e_2$, $e_3$ , $e_4$]
- Arithmetic Sequences:
  - [$elem_1$ .. $elem_3$]
  - [$elem_1$, $elem_2$ .. $elem_3$]
  - Only works for certain element types: integers, booleans, characters, …
  - (omit last element to specify an "infinite list")

# … continued

- Using list comprehensions:
  - [ 2*x+1 | x <- [1,3,7,11] ]
- Using prelude/library functions:
  - ++
  - reverse
  - take, takeWhile, drop, dropWhile, map, …
  - …
- Using constructor functions:
  - [] and (:)  (“nil” and “cons”)

# Prelude Functions

```
(++)       :: [a] -> [a] -> [a]
reverse    :: [a] -> [a]
take       :: Int -> [a] -> [a]
drop       :: Int -> [a] -> [a]
takeWhile  :: (a -> Bool) -> [a] -> [a]
dropWhile  :: (a -> Bool) -> [a] -> [a]
iterate    :: (a -> a) -> a -> [a]
repeat     :: a -> [a]
…
```

# map

- map :: (a -> b) -> [a] -> [b]
- map f xs produces a new list by applying the function f to each element in the list xs
- map (1+) [1,2,3] = [2,3,4]
- map even [1,2,3] = [False, True, False]
- map id xs = xs, for any list xs
- We can also think of map as a function that turns functions of type (a -> b) into list transformers of type ([a] -> [b])
- incAll :: [Int] -> [Int]
- incAll = map (1+)
- incAll [1,2,3] = [2,3,4]

The "identity" function

```
id    :: a -> a
id x   = x
```

# Aside: Applicative Syntax

- "Function application groups to the left"
  f x y z = ((f x) y) z

- "Function type arrows group to the right"
  a -> b -> c -> d = a -> (b -> (c -> d))

- If f :: a -> b and x :: a, then f x :: b

- If f :: a -> b -> c, x :: a, and y :: b, then (f x) :: (b -> c) and (f x y) = (f x) y :: c

# Aside: “Curried” Functions

- We can think of a function f :: a -> b -> c in two different ways:
  - f takes two arguments (one of type a and one of type b) and returns a result of type c
  - f takes one argument (of type a) and returns a function (of type b -> c) as its result
- A function that takes its arguments one at a time is described as a curried function
- (All Haskell library functions work this way …)
- Named after Haskell Curry (although some think we should call it “Schönfinkeling” after Moses Schönfinkel who used the idea earlier …)

# Aside: Uncurried Functions

- We can force programmers to supply multiple arguments at the same time by using tuples:

```
add        :: (Int, Int) -> Int
add (x,y)  = x + y
```

- However, Haskell's syntax encourages the use of curried functions:
  - Fewer parentheses needed in many cases
  - More flexibility from the use of partial applications (i.e., when some of the trailing arguments to a function are omitted)
  - May be more efficient (avoids making a tuple)

- (tuples are also use to return "multiple results")

# filter

- filter :: (a -> Bool) -> [a] -> [a]
- filter even [1..10] = [2,4,6,8,10]
- filter (<5) [1..100] = [1,2,3,4]
- filter (<5) [100,99..1] = [4,3,2,1]

- We can think of filter as mapping predicates/ functions of type (a -> Bool), to list transformers of type [a] -> [a]

- keepEvens:: [Int] -> [Int]
- keepEvens= filter even
- keepEvens [1..10] = [2,4,6,8,10]

# Higher-Order Functions

- A function that takes functions as arguments or returns a function as its result is called a higher-order function

- map and filter are higher-order functions:

- map (map (1+)) [[1], [2,3,4], [5,6]]
  = [map (1+) [1],
  map (1+) [2,3,4],
  map (1+) [5,6]]
  = [[2], [3,4,5], [6,7]]

# Aside: Composition

- ```
  (.)          :: (b -> c) -> (a -> b) -> (a -> c)
  (f . g) x    = f (g x)
  ```
- Good for describing "pipelines"
- Example:
  ```
  toOdd        = (1+) . (2*)
  toOdd x      = 1 + 2*x
  ```
- The first definition is said to be "point-free" because it doesn't mention the argument x by name
  ```

# Example: Grouping

```
group :: Int -> [a] -> [[a]]
group n
    = takeWhile (not . null)
    . map (take n)
    . iterate (drop n)
```

["abc", "def", "g"]

↑

["abc", "def", "g", "", "", "", …]

↑

["abcdefg", "defg", "g", "", "", "", …]

↑

"abcdefg"

# Example: Grouping

```
group 3
    =  takeWhile (not . null)
    .  map (take 3)
    .  iterate (drop 3)
```

# Example: Grouping

```
group 3 "abcdefg"

   = (takeWhile (not . null)

     . map (take 3)

     . iterate (drop 3)) "abcdefg"
```

# Example: Grouping

```
group 3 "abcdefg"
    = takeWhile (not . null)
          (map (take 3)
              (iterate (drop 3) "abcdefg"))
```

# Example: Grouping

```
group 3 "abcdefg"
  = takeWhile (not . null)
      (map (take 3)
          ["abcdefg", "defg", "g", "", "", ...])
```

# Example: Grouping

group 3 "abcdefg"

= takeWhile (not . null)

["abc", "def", "g", "", "", ...]

# Example: Grouping

group 3 "abcdefg"

= ["abc", "def", "g"]

# Aside: Lambda Notation

- The syntax \vars -> expr denotes a function that takes arguments vars and returns the corresponding value of expr

- Referred to as a <u>lambda expression</u> after the corresponding construct in $\lambda$-calculus

- Examples:
  - (\x -> x + 1) 3  = 4
  - (\x y -> (x + y) * (x - y)) 4 2 = 12
  - map (\x -> 1 + 2*x) [1,2,3] = [3,5,7]
  - filter p . filter q  =  filter (\x -> q x && p x)

# List Comprehensions

General form:

- [ expression | qualifiers ]

where qualifiers are either:

- Generators: pat <- expr; or
- Guards: expr; or
- Local definitions: let defns

Works like a kind of generalized "for loop"

# Examples

```
[ x*x | x <- [1..6] ]
        = [ 1, 4, 9, 16, 25, 36 ]

[ x | x <- [1..27], 28 `mod` x == 0 ]
        = [ 1, 2, 4, 7, 14 ]

[ m | n <- [1..5], m <-[1..n] ]
        = [ 1, 1,2, 1,2,3, 1,2,3,4, 1,2,3,4,5 ]
```

# Applications

- Some familiar functions:

  ```
  map f xs       = [ f x | x <- xs ]
  filter p xs    = [ x | x <- xs, p x ]
  ```

- Can you define take, head, or (++) using a comprehension?

# Laws of Comprehensions

[ x | x <- xs ] = xs

[ e | x <- xs ] = map (\x -> e) xs

[ e | True ] = [ e ]

[ e | False ] = []

[ e | $gs_1$, $gs_2$ ] = concat [ [ e | $gs_2$] | $gs_1$ ]

# Example

```
[ (x,y) | x <- [1,2], y <- [1,2] ]

= concat
      [ [ (x,y) | y <- [1,2]] | x <- [1,2] ]

= concat
      [ map (\y -> (x,y)) [1,2] | x <- [1,2] ]

= concat
      (map (\x ->
          map (\y -> (x,y)) [1,2]) [1,2])
```

# Constructor Functions

- What if you can't find a function in the prelude that will do what you want to do?

- Every list takes the form:
  - [], an empty list
  - (x:xs), a non-empty list whose first element is x, and whose tail is xs

- Equivalently: the list type has two "constructor functions":
  - The constant [] :: [a]
  - The operator (:) :: a -> [a] -> [a]

- Using "pattern matching", we can also take lists apart …

# Functions on Lists

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

head          :: [a] -> a
head (x:xs) = x

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

# Recursive Functions in Prelude

```
last            :: [a] -> a
last (x:[])     = x
last (x:y:xs)   = last (y:xs)

init            :: [a] -> [a]
init (_:[])     = []
init (x:y:xs)   = x : init (y:xs)

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

# … continued

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

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

# … continued

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

first matching pattern "wins"

```
unzip              :: [(a,b)] -> ([a],[b])
unzip []            = ([],[])
unzip ((l,r):xs) = (l:ls,r:rs)
      where (ls,rs) = unzip xs
```

nested pattern

local definition

# … and more

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

in List library

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

user defined

# Using the List Library

- Data.List is one of several standard Haskell Libraries
- To use Data.List functions:
  - In the interpreter: :l Data.List
  - In a .hs or .lhs file: import Data.List
- Many useful functions are defined in this library.
- Browse via http://downloads.haskell.org/~ghc/latest/docs/html/libraries/ for full details.

# Summary

- There are many ways to construct and manipulate list values in functional languages like Haskell
- Higher-order functions capture common patterns of computations
- List comprehensions are especially compact
- Pattern matching and recursion support arbitrary computations on lists