Lists and recursion

Functional Programming

Utrecht University

Goals

- More list functions
- **Recursion**
- List comprehensions

Chapters 5 and 6 from Hutton's book

From previous lectures

Primitives for building lists

- `[]  :: [a]` is the empty list
- `(:) :: a -> [a] -> [a]` (the “cons” constructor)
    - Build a list by putting an element at the front
- When we write `[1, 2, 3]` the compiler translates it to `1 : 2 : 3 : []`

Pattern matching over lists

```
length []     = 0
length (_:xs) = 1 + length xs
```

From previous lectures

Useful list functions

```
null :: [a] -> Bool

head :: [a] -> a
tail :: [a] -> [a]

reverse :: [a] -> [a]
(++)    :: [a] -> [a] -> [a]

sum :: Num a => [a] -> a

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

Foldable in the interpreter

If you ask for the type of `sum` in `ghci`, you get

```
sum :: (Foldable t, Num a) => t a -> a
```

- This is a *more generic* version of `sum`
- "Adding up all elements" works for other containers
  - Think of sets or (binary) trees

How to obtain the types shown here

```
> :t sum
sum :: (Num a, Foldable t) => t a -> a
> :t +d sum
sum :: [Integer] -> Integer
```

Recursion

Recursion on natural numbers

Recursion = defining something in terms of itself

```
fac 0 = 1
fac n = n * fac (n - 1)
```

```
0 * m = 0
n * m = m + (n - 1) * m
```

- A case for 0 or 1
- A recursive case where the value of n is computed from the same function applied to $n - 1$

Does our product work?

```
0 * m = 0                -- (1)
n * m = m + (n - 1) * m  -- (2)

2 * 4
= -- apply (2)
4 + (2 - 1) * 4
= -- perform substraction
4 + 1 * 4
= -- apply (2) and perform substraction
4 + (4 + 0 * 4)
= -- apply (1)
4 + (4 + 0)
= -- perform additions
8
```

Recursion needs a base case

without base case:

```
fac n = n * fac (n-1)  -- (1)
-- No more equations

fac 1
= -- apply (1), what else?
1 * fac 0
= -- apply (1)
1 * 0 * fac (-1)
= -- apply (1)
1 * 0 * (-1) * fac (-2)
= -- apply (1)
...
```

Recursion needs the argument to get smaller

argument does not get smaller:

```
replicate 0 _ = []                    -- (1)
replicate n x = x : replicate n x     -- (2)

replicate 2 'a'
= -- apply (2)
'a' : replicate 2 'a'
= -- apply (2)
'a' : 'a' : replicate 2 'a'
= -- apply (2)
...
```

Recursion on Lists

```
length []       = 0
length (_ : xs) = 1 + length xs

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

Does our concatenation work?

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

```
[1, 2] ++ [3, 4]
= -- remove syntactic sugar for [1, 2]
(1 : 2 : []) ++ [3, 4]
= -- apply (2)
1 : ((2 : []) ++ [3, 4])
= -- apply (2)
1 : (2 : ([] ++ [3, 4]))
= -- apply (1)
1 : 2 : [3, 4]
= -- resugar the resulting list
[1, 2, 3, 4]
```

Hutton's recipe for recursion

1. Define the type
2. Enumerate the cases
3. Define the simple (base) cases
4. Define the other (recursive) cases
    - This part involves most of the thinking
    - The main question:

      can I obtain the value of the function if I know its result for a smaller part (e.g. for the tail of the list)?
5. Generalize and simplify
    - Remove duplicate equations
    - Pattern match only as necessary
    - Infer a more general type

Cooking `sum`

Cooking `sum`

1. Define the type

```
sum :: [Int] -> Int
```

2. Enumerate the cases

```
sum []     = _
sum (x:xs) = _
```

Cooking `sum`

3. Define the simple (base) cases

```haskell
sum []     = 0
```

4. Define the other (recursive) cases
   - If I know the result of `sum xs`, can I get `sum (x:xs)`?
   - Just add the head element to that result!

```haskell
sum (x:xs) = x + sum xs
```

5. Generalize and simplify
   - In this case our definition works for any numeric type

```haskell
sum :: Num a => [a] -> a
```

Cooking `elem`

`elem x xs` tells you whether `x` is an element of `xs`

```
> 1 `elem` [1,2]
True
> 3 `elem` [1,2]
False
> 2 `elem` []
False
```

We usually write `elem` infix to make it look like $1 \in [1, 2]$

Cooking `elem`

1. Define the (approximate) type

```
elem :: Int -> [Int] -> Bool
```

2. Enumerate the cases

```
elem x []     = _
elem x (y:ys) = _
```

3. Define the simple (base) cases

```
elem x []     = False
```

Cooking `elem`

4. Define the other (recursive) cases
    - We need to distinguish between x equal to y or not
        - Remember: we cannot repeat a variable in a pattern
    - If it is, we stop; otherwise, we continue further

```
elem x (y:ys) | x == y    = True
              | otherwise = elem x ys
```

5. Generalize and simplify
    - We only use `(==)` to inspect values, so Eq is enough

```
elem :: Eq a => a -> [a] -> Bool
```

Cooking take

`take n xs` gets the first `n` elements of list `xs`, or the entire list if there are less than those

```
> take 2 [1,2,3]
[1,2]
> take 0 [1,2,3]
[]
> take 4 [1,2,3]
[1,2,3]
```

Cooking `take`

1. Define the type
    - The type of the elements of the list does not matter

```haskell
take :: Int -> [a] -> [a]
```

2. Enumerate the cases
    - We can match on both the number and list

```haskell
take 0 []     = _
take 0 (x:xs) = _
take n []     = _
take n (x:xs) = _
```

Cooking `take`

3. Define the simple (base) cases

    - If there are no elements to take, we obtain an empty list

```
take 0 []     = []
take 0 (x:xs) = []
take n []     = []
```

4. Define the other (recursive) cases

    - If we have taken 1 element from `x:xs`, there are only `n-1` left to take from `xs`

```
take n (x:xs) = x : take (n-1) xs
```

Cooking take

4. We have the following until now

```haskell
take 0 []     = []
take 0 (x:xs) = []
take n []     = []
take n (x:xs) = x : take (n-1) xs
```

5. Generalize and simplify
    - When the number is 0, the list does not matter
    - If the list is empty, the number does not matter

```haskell
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n-1) xs
```

Question

Define list difference

```haskell
(\\) :: Eq a => [a] -> [a] -> [a]
```

- Return all elements in the first list *except* if they appear in the second

```
> [1,2] \\ [1]
[2]
> [1,2] \\ [2,3,4]
[1]
> [] \\ [1,2,3]
[]
```

Question

Define list difference

```haskell
(\\) :: Eq a => [a] -> [a] -> [a]
```

- Return all elements in the first list *except* if they appear in the second

```
> [1,2] \\ [1]
[2]
> [1,2] \\ [2,3,4]
[1]
> [] \\ [1,2,3]
[]
```

Hint: use `elem` to detect if an element appears in the second

Cooking `init`

`init xs` gives you all the elements except for the last

```
> init [1,2,3]
[1,2]
> init []
*** Exception: Prelude.init: empty list
```

Cooking `init`

`init xs` gives you all the elements except for the last

```
> init [1,2,3]
[1,2]
> init []
*** Exception: Prelude.init: empty list
```

1. Define the type

   ```haskell
   init :: [a] -> [a]
   ```

2. Enumerate the cases

   - The empty list should yield an error

   ```haskell
   init []     = error "empty list in init"
   init (x:xs) = _
   ```

Cooking `init`

- Here is the trick, we need to distinguish whether we have just one element in the list – and we are finished – or we need to get more elements
    - We do this by further pattern matching

2. Enumerate the cases

```
init (x:[]) = _
init (x:xs) = _
```

3. Define the simple (base) cases

```
init (x:[]) = []
```

4. Define the other (recursive) cases

```
init (x:xs) = x : init xs
```

Cooking `init`

5. Generalize and simplify
    - We can use `[x]` to match a one-element list
    - We do not care about that single element → use _

```
init :: [a] -> [a]
init []     = error "empty list in init"
init [_]    = []
init (x:xs) = x : init xs
```

Cooking sorted

`sorted xs` returns `True` if and only if the elements in the list are in ascending order

```
> sorted [1,2,3]
True
> sorted [2,1,3]
False
> sorted []
True
```

Cooking sorted

`sorted xs` returns `True` if and only if the elements in the list are in ascending order

```
> sorted [1,2,3]
True
> sorted [2,1,3]
False
> sorted []
True
```

1. Define the type

   ```
   sorted :: [Int] -> Bool
   ```

2. Enumerate the cases

   ```
   sorted []     = _
   sorted (x:xs) = _
   ```

Cooking sorted

3. Define the simple (base) cases

```
sorted []      = True
```

4. Define the other (recursive) cases
    - We need to compare the first and second elements
        - We need further pattern matching
    - If they are in the right relation, we check further

```
sorted (x:[])    = True
sorted (x:y:ys) | x <= y    = sorted (y:ys)
                | otherwise = False
```

Cooking sorted

5. Generalize and simplify
    - As before, we can use `[x]` instead of `x:[]`
    - We are reusing the whole `y:ys` in the right-hand side
        - We can give it a name using @
        - We avoid matching and rebuilding the list

```
sorted []  = True
sorted [_] = True
sorted (x : xs@(y : _))
  | x <= y    = sorted xs
  | otherwise = False
```

Cooking `zip`

`zip xs ys` turns two lists into a list of tuples

```
> zip [1,2] [3,4]
[(1,3),(2,4)]
> zip [1,2] [3,4,5]
[(1,3),(2,4)]
```

If one of the lists runs out of elements, we stop

Cooking zip

`zip xs ys` turns two lists into a list of tuples

```
> zip [1,2] [3,4]
[(1,3),(2,4)]
> zip [1,2] [3,4,5]
[(1,3),(2,4)]
```

If one of the lists runs out of elements, we stop

Try yourself!

Cooking zip

1. Define the type

```haskell
zip :: [a] -> [b] -> [(a,b)]
```

2. Enumerate the cases

```haskell
zip []     []     = _
zip []     (y:ys) = _
zip (x:xs) []     = _
zip (x:xs) (y:ys) = _
```

3. Define the simple (base) cases

```haskell
zip []     []     = []
zip []     (y:ys) = []
zip (x:xs) []     = []
```

Cooking zip

4. Define the other (recursive) cases

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

5. Generalize and simplify
    - If one of the lists is empty, we don't care about the other

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

Cooking `merge`

Given two *sorted* lists `xs` and `ys`, `merge xs ys` produces a new sorted list from those elements

- This is the basis of a sorting algorithm called MergeSort

```
> merge [1,4] [2,3,5]
[1,2,3,4,5]
> merge [] [2,3,5]
[2,3,5]
```

Cooking `merge`

1. Define the type

```haskell
merge :: [Int] -> [Int] -> [Int]
```

2. Enumerate the cases

```haskell
merge []     []     = _
merge (x:xs) []     = _
merge []     (y:ys) = _
```

   - In the last case we have to decide which number is larger

```haskell
merge (x:xs) (y:ys)
  | x <= y          = _
  | otherwise       = _
```

Cooking `merge`

3. Define the simple (base) cases

```
merge []     []     = []
merge (x:xs) []     = x:xs
merge []     (y:ys) = y:ys
```

4. Define the other (recursive) cases
   - Choose the smallest one and merge the rest

```
merge (x:xs) (y:ys)
  | x <= y          = x : merge xs (y:ys)
  | otherwise       = y : merge (x:xs) ys
```

Cooking `merge`

5. Generalize and simplify
    - This function works for any type which can be ordered
    - In the case of an empty list, we just return the other list
    - We can give names to complete lists to avoid duplication

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys   = ys
merge xs []   = xs
merge xss@(x:xs) yss@(y:ys)
  | x <= y    = x : merge xs yss
  | otherwise = y : merge xss ys
```

cooking inits

`inits xs` returns the initial segments of `xs`, that is, all the lists which are prefixes of the original one

```
> inits [1,2,3]
[[],[1],[1,2],[1,2,3]]
> inits []
[[]]
```

1. Define the type

   ```
   inits :: [a] -> [[a]]
   ```

2. Enumerate the cases

   ```
   inits []     = _
   inits (x:xs) = _
   ```

Cooking initial segments

3. Define the simple (base) cases

```
inits []     = [[]]
```

4. Define the other (recursive) cases

    - Suppose you have `[1,2,3]`, that is, `1 : [2,3]`
    - The initial segments of `[2,3]` are `[[],[2],[2,3]]`, what do you do with the 1?
    - If you put the 1 in front of every list, you get `[[1],[1,2],[1,2,3]]`
    - We are almost there! We are just missing the extra empty list at the front

```
inits (x:xs) = [] : prefixWith x (inits xs)
```

Cooking initial segments

```
prefixWith             :: a -> [[a]] -> [[a]]
prefixWith p []        = []
prefixwith p (ys:yss) = (p:ys) : prefixWith p yss
```

Cooking initial segments

```
prefixWith            :: a -> [[a]] -> [[a]]
prefixWith p []       = []
prefixwith p (ys:yss) = (p:ys) : prefixWith p yss
```

`prefixWith p yss` prefixes every list in yss with a p. Reuse!

```
prefixWith p yss = map (p:) yss
```

Cooking initial segments

```
prefixWith             :: a -> [[a]] -> [[a]]
prefixWith p []        = []
prefixwith p (ys:yss) = (p:ys) : prefixWith p yss
```

`prefixWith p yss` prefixes every list in yss with a p. Reuse!

```
prefixWith p yss = map (p:) yss
```

Use map:

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

Cooking `reverse`

`reverse xs` gives the same elements in reverse order

```
> reverse [1,2,3]
[3,2,1]
```

1. Define the type

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

2. Enumerate the cases

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

Cooking `reverse`

3. Define the simple (base) cases

```
reverse []     = []
```

4. Define the other (recursive) cases

    - Suppose you get `[1,2,3]`, which you split as 1 and `[2,3]`
    - The reverse of `[2,3]` is `[3,2]`, where do you put the 1?
    - At the end of the reversed list!

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

Problem with reverse `reverse`

- This definition is **very inefficient**
  - Each time you call `(++)`, you need to traverse the whole list, since the new element goes at the end
  - If the list has n elements, the amount of steps is

$$n - 1 + n - 2 + n - 3 + ... + 1 = \frac{n \cdot (n - 1)}{2} = \mathcal{O}(n^2)$$

Solution: use an accumulator

- There is a standard technique to solve this problem: using an **accumulator**
  1. Introduce a local definition with an additional parameter (the accumulator)
  2. Figure out the invariant:

     invariant: accumulator contains solution for all elements seen so far.
  3. Follow Hutton's recipe, but
     - Do not pattern match on the accumulator
     - Return the accumulator in the base case
     - Update the accumulator in the recursive steps
  4. Initialize the accumulator in the main call

`sum` with accumulator

Define `sum` using an accumulator

sum with accumulator

Define `sum` using an accumulator

```
sum [1,2,3,4] = 1 + sum [2,3,4]
              = 1 + 2 + sum [3,4]
              = 1 + 2 + 3 + sum [4]
              = 1 + 2 + 3 + 4 + sum []
```

sum with accumulator

Define `sum` using an accumulator

```
sum [1,2,3,4] = 1 + sum [2,3,4]
              = 1 + 2 + sum [3,4]
              = 1 + 2 + 3 + sum [4]
              = 1 + 2 + 3 + 4 + sum []
```

- *Observation:* Always of the form ‘a + sum xs’
- Introduce the function `sum'` that has as invariant:

```
sum' acc xs == acc + sum xs
```

Implementing `sum'`

- invariant: `sum' acc xs == acc + sum xs`

```
sum'               :: Int -> [Int] -> Int
sum' acc []        = _
sum' acc (x:xs)    = _
```

Implementing `sum'`

- invariant: ‘`sum' acc xs == acc + sum xs`

```
sum'               :: Int -> [Int] -> Int
sum' acc []        = _
sum' acc (x:xs) = _
```

Invariant tells us that:

```
sum'               :: Int -> [Int] -> Int
sum' acc []        = acc
sum' acc (x:xs) = sum' (acc + x) xs
```

Implementing `sum'`

- invariant: ‘`sum' acc xs == acc + sum xs`

```
sum'              :: Int -> [Int] -> Int
sum' acc []       = _
sum' acc (x:xs) = _
```

Invariant tells us that:

```
sum'              :: Int -> [Int] -> Int
sum' acc []       = acc
sum' acc (x:xs) = sum' (acc + x) xs
```

so:

```
sum    :: [Int] -> Int
sum xs = sum' 0 xs
```

sum with accumulator

Define `sum` using an accumulator.

We can also apply η-reduction and use a *case* expression.

```
sum :: [Int] -> Int
sum = sum' 0
  where
    sum'          :: Int -> [Int] -> Int
    sum' acc xs = case xs of
                    []     -> acc
                    (x:xs) -> sum' (acc+x) xs
```

reverse with an accumulator

1. Introduce a local definition with an additional parameter to hold the interim result

```
reverse xs = _
  where
    reverse'          :: [a] -> [a] -> [a]
    reverse' acc xs = _
```

reverse with an accumulator

2. Figure out the invariant

```
reverse [1,2,3,4]
    = reverse [2,3,4] ++ [1]
    = (reverse [3,4]  ++ [2]) ++ [1]
    = reverse [3,4]   ++ ([2] ++ [1])
    = ...
```

reverse with an accumulator

2. Figure out the invariant

```
reverse [1,2,3,4]
    = reverse [2,3,4] ++ [1]
    = (reverse [3,4]  ++ [2]) ++ [1]
    = reverse [3,4]   ++ ([2] ++ [1])
    = ...
```

Invariant:

```
reverse' acc xs == reverse xs ++ acc
```

reverse with an accumulator

3. Follow Hutton's recipe, but
   - Do not pattern match on the accumulator
   - Return the accumulator in the base case
   - Update the accumulator in the recursive steps

```
reverse xs = _
  where
    reverse' acc []     = acc
    reverse' acc (x:xs) = reverse' (x:acc) xs
```

4. Initialize the accumulator in the main call
   - When we start, we haven't accumulated any element yet

```
reverse xs = reverse' [] xs
  where
    reverse' acc []     = acc
    reverse' acc (x:xs) = reverse' (x:acc) xs
```

List comprehensions

List comprehensions

`[ expr | x <- list ]`

Succint notation for building *new* lists from *old* ones

```
addone :: Num a => [a] -> [a]
addone xs = [x + 1 | x <- xs]
```

- “For each x in `xs`, return `x + 1`”
- Very similar to mathematical notation

$$\{x + 1 \,|\, x \in xs\}$$

Guards

```
[ expr | x <- list, condition ]
```

```haskell
-- Check is a number is divisible by 2
even :: Integer -> Bool

sumeven :: [Integer] -> Integer
sumeven xs = sum [x | x <- xs, even x]
```

- "Take all `x` in `xs` such that `x` is even"
- The result of a comprehension is another list
  - We can further consume it with other functions
  - In this case, we use `sum`

–

Inits with a list comprehension

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

or

```
inits []     = [[]]
inits (x:xs) = [] : [ x:rs | rs <- inits xs]
```

More List comprehensions; Pattern matching

`[ expr | pattern <- list ]`

```haskell
heads :: [[a]] -> [a]
heads xs = [y | (y:_) <- xs]
```

- Only includes those elements which match the pattern
  - In this case, non-empty lists

    ```
    > heads [[1,2],[],[3,4,5]]
    [1,3]
    ```
- We can introduce new names, as we do with usual pattern matching
  - In this case, we refer to the head in the result

Multiple clauses

We can have multiple generators and guards

- Generators provide every possible combination

```
> [(x,y) | x <- [1,2], y <- [3,4]]
[(1,3),(1,4),(2,3),(2,4)]
```

- Generators and conditions may refer to each other

```
> [(x,y) | x <- [1,2,3], y <- [1,2,3], x <= y]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
> [(x,y) | x <- [1,2,3], y <- [x .. 3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Prime numbers up to a bound

- *Problem:* Compute all primes $\leq n$

Prime numbers up to a bound

- *Problem:* Compute all primes $\leq n$

1. A number x is a prime iff ($x \geq 2$ and) it has exactly two factors
2. f is a factor of x if the remainder of $\frac{x}{f}$ is zero

Prime numbers up to a bound

- *Problem:* Compute all primes $\leq n$

1. A number x is a prime iff ($x \geq 2$ and) it has exactly two factors
2. f is a factor of x if the remainder of $\frac{x}{f}$ is zero

Good style: divide the problem in parts and refine it

```
primes   :: Int -> [Int]
primes n = [ x | x <- [2 .. n], isPrime x ]
  where isPrime x = _
```

Prime numbers up to a bound

- *Problem:* Compute all primes $\leq n$

1. A number x is a prime iff ($x \geq 2$ and) it has exactly two factors
2. f is a factor of x if the remainder of $\frac{x}{f}$ is zero

Good style: divide the problem in parts and refine it

```
primes   :: Int -> [Int]
primes n = [ x | x <- [2 .. n], isPrime x ]
  where isPrime x = length (factors x) == 2
        factors x = _
```

Prime numbers up to a bound

- *Problem:* Compute all primes $\leq n$

1. A number x is a prime iff ($x \geq 2$ and) it has exactly two factors
2. f is a factor of x if the remainder of $\frac{x}{f}$ is zero

Good style: divide the problem in parts and refine it

```
primes   :: Int -> [Int]
primes n = [ x | x <- [2 .. n], isPrime x ]
  where isPrime x = length (factors x) == 2
        factors x = [f | f <- [1 .. x]
                       , x `mod` f == 0
                    ]
```

(Functional) QuickSort

- Divide and conquer approach
  1. Pick a pivot
  2. Partition the elements smaller and larger than the pivot
  3. Sort those partitions
  4. Put together the list

(Functional) QuickSort

- Divide and conquer approach
  1. **Pick a pivot**
     - The first element in the list works
  2. Partition the elements smaller and larger than the pivot
  3. Sort those partitions
  4. Put together the list

```
quicksort []            = []
quicksort (pivot:rest) = undefined
```

(Functional) QuickSort

- Divide and conquer approach
    1. Pick a pivot
    2. **Partition the elements**
    3. Sort those partitions
    4. Put together the list

```
quicksort []            = []
quicksort (pivot:rest) = undefined
  where smaller = [x | x <- rest, x <= pivot]
        larger  = [x | x <- rest, x >  pivot]
```

(Functional) QuickSort

- Divide and conquer approach
  1. Pick a pivot
  2. Partition the elements smaller and larger than the pivot
  3. **Sort those partitions**
  4. **Put together the list**

```
quicksort []           = []
quicksort (pivot:rest) =
  quicksort smaller ++ [pivot] ++ quicksort larger
  where smaller = [x | x <- rest, x <= pivot]
        larger  = [x | x <- rest, x >  pivot]
```

Question

Define `replicate` using comprehensions

Question

Define `replicate` using comprehensions

```
replicate :: Int -> a -> [a]
replicate n x = [x | _ <- [1 .. n]]
```

More List Functions

Cooking final segments

`tails xs` returns the final segments of `xs`, that is, all the lists which are suffixes of the original one

```
> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
> tails   [2,3]
[       [2,3],[3],[]]
> tails     [3]
[             [3],[]]
```

```
tails :: [a] -> [[a]]
tails []        = [[]]
tails ts@(_:xs) = ts : tails xs
```

Final segments using initial segments

Final segments of `xs` seem related to initial segments of `reverse xs`

```
> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
> inits [3,2,1]
[[],[3],[3,2],[3,2,1]]
```

- There are two problems with the second result
  1. Each of the inner lists is reversed
  2. The whole outer list is reversed
- Let’s fix this and give an alternative definition of `tails`

Final segments using initial segments

- To reverse *each* of the inner lists we use a list comprehension

```
> [reverse i | i <- inits [3,2,1]]
[[],[3],[2,3],[1,2,3]]
```

- This leads to this final definition

```
tails xs = reverse [reverse i
                   | i <- inits (reverse xs)]
```

Fizzbuzz

- Write fizzbuzz using direct recursion; test if some element is divisible by `n` (and by `m`) only once.

```
fizzbuzz :: (Int, Int) -> [Int]
         -> ([Int], [Int], [Int])
```

A call of the form `fizzbuzz (m, n) xs` should return a triple with a list in each element:

- The first list contains elements of `xs` divisible by `m`
- The second list those divisible by `n` (and not by `m`)
- The third list should contain the rest

Fizzbuzz

```
fizzbuzz (m,n) xs = fb xs
  where
    fb []     = ([],[],[])
    fb (x:xs) = case ( x `mod` m == 0
                     , x `mod` n == 0
                     ) of
                  (True, _   ) -> (x:ms,ns,  rs)
                  (_   , True) -> (ms,  x:ns,rs)
                  (_   , _   ) -> (ms,  ns,  x:rs)
      where
        (ms,ns,rs) = fb xs
```

Fizzbuzz

```
fizzbuzz (m,n) xs = fb xs
  where
    fb []     = ([],[],[])
    fb (x:xs) = case ( x `mod` m == 0
                     , x `mod` n == 0
                     ) of
                  (True, _    ) -> (x:ms,ns,  rs)
                  (_    , True) -> (ms,  x:ns,rs)
                  (_    , _    ) -> (ms,  ns,  x:rs)
        where
          (ms,ns,rs) = fb xs
```

- Exercise: write fizzbuzz using a comprehensions

Final words

Defining recursive functions is like riding a bicycle: it looks easy when someone else is doing it, may seem impossible when you first try to do it yourself, but becomes simple and natural with practice.

– From ”Programming in Haskell”

- On the other hand, don’t get too attached to recursion
- Many of these examples have better implementations using *higher-order functions*
  - Which happens to be the topic for next lecture!