Part I

Lists and Recursion
Cons and append

(:) :: a -> [a] -> [a] -- cons takes an element and a list
(++) :: [a] -> [a] -> [a] -- append takes two lists

1 : [2,3] = [1,2,3]
[1] ++ [2,3] = [1,2,3]
[1,2] ++ [3] = [1,2,3]
'l' : "ist" = "list"
"l" ++ "ist" = "list"
"li" ++ "st" = "list"

[1,2] : 3 -- type error!
1 ++ [2,3] -- type error!
[1,2] ++ 3 -- type error!
"l" : "ist" -- type error!
'l' ++ "ist" -- type error!

(:) is pronounced cons, for construct
(++) is pronounced append
Lists

Every list can be written using only (: ) and [].

\[ [1, 2, 3] = 1 : (2 : (3 : [])) \]

"list" = ['l','i','s','t']
    = 'l' : ('i' : ('s' : ('t' : [])))

A recursive definition: A list is either

- empty, written [], or
- constructed, written \( x : xs \), with head \( x \) (an element), and tail \( xs \) (a list).

So every list matches exactly one of the following two patterns

\[ [ ] \quad \text{-- only matches the empty list} \]
\[ ( x : xs ) \quad \text{-- matches any non-empty list} \]

We can use any two distinct variables in the cons pattern

\[ ( \text{head} : \text{tail} ) \quad \text{-- matches any non-empty list} \]
Patterns

List patterns can be used in definitions

myList = [ 0, 1, 2, 3, 4 ]
( x : xs ) = myList
[ a, b, c, d, e ] = myList  -- matches lists of length 5
[ p, q, r ] = myList  -- matches lists of length 3

> ( x : xs ) = [ 0, 1, 2, 3, 4 ]
> x
0
> xs
[ 1, 2, 3, 4 ]
> [ a, b, c, d, e ] = [ 0, 1, 2, 3, 4 ]
> c
2
> [ p, q, r ] = [ 0, 1, 2, 3, 4 ]
*** Exception: ...  -- pattern and value must match!
Recursion

A list is either

- **empty**, written \[\], or
- **constructed**, written \(x:xs\), with *head* \(x\) (an element), and *tail* \(xs\) (a list).

"Brexit means Brexit." Theresa May
Recursion versus meaningless self-reference

A *list* is either

- *empty*, written `[ ]`, or
- *constructed*, written `x:xs`, with *head* `x` (an element), and *tail* `xs` (a list).

“Brexit means Brexit.”

Theresa May
A list of numbers

> null  [1,2]
False
> head  [1,2]
1
> tail  [1,2]
[2]
> null  [2]
False
> head  [2]
2
> tail  [2]
[]
> null  []
True
Part II

Mapping: Square every element of a list
Two styles of definition—squares

Comprehension

squares :: [Int] -> [Int]
squares xs = [ x*x | x <- xs ]

Recursion

squaresRec :: [Int] -> [Int]
squaresRec [] = []
squaresRec (x:xs) = x*x : squaresRec xs
Pattern matching and conditionals

Pattern matching

squaresRec :: [Int] -> [Int]
squaresRec [] = []
squaresRec (x:xs) = x * x : squaresRec xs

Conditionals with binding

squaresCond :: [Int] -> [Int]
squaresCond ws =
  if null ws then
    []
  else
    let
      x = head ws
      xs = tail ws
    in
      x * x : squaresCond xs
How recursion works—squaresRec

squaresRec :: [Int] -> [Int]
squaresRec [] = []
squaresRec (x:xs) = x * x : squaresRec xs

squaresRec [1,2,3]  
= squaresRec (1 : (2 : (3 : [])))  
= 1 * 1 : squaresRec (2 : (3 : []))  
= 1 * 1 : (2 * 2 : squaresRec (3 : []))  
= 1 * 1 : (2 * 2 : (3 * 3 : squaresRec []))  
= 1 * 1 : (2 * 2 : (3 * 3 : []))  
= 1 : (4 : (9 : []))  
= [1, 4, 9]
QuickCheck

-- squares.hs
import Test.QuickCheck

squares :: [Int] -> [Int]
squares xs = [ x*x | x <- xs ]

squaresRec :: [Int] -> [Int]
squaresRec [] = []
squaresRec (x:xs) = x*x : squaresRec xs

prop_squares :: [Int] -> Bool
prop_squares xs = squares xs == squaresRec xs

[jitterbug]dts: ghci squares.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
> quickCheck prop_squares
+++ OK, passed 100 tests.
Part III

Filtering: Select odd elements from a list
Two styles of definition—odds

Comprehension

\[
\text{odds} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{odds} \; \text{xs} \; = \; [ \; x \; | \; x \; \leftarrow \; \text{xs}, \; \text{odd} \; x \; ]
\]

Recursion

\[
\text{oddsRec} :: [\text{Int}] \rightarrow [\text{Int}] \\
\text{oddsRec} \; [] \; = \; [] \\
\text{oddsRec} \; (x:xs) \; | \; \text{odd} \; x \; = \; x \; : \; \text{oddsRec} \; \text{xs} \\
\quad \mid \; \text{otherwise} \; = \; \text{oddsRec} \; \text{xs}
\]
Pattern matching and conditionals

Pattern matching with guards

\[
\text{oddsRec} :: \text{[Int]} \rightarrow \text{[Int]}
\]
\[
\text{oddsRec} [] = []
\]
\[
\text{oddsRec} (x:xs) \mid \text{odd } x = x : \text{oddsRec} \ x s
\]
\[
\quad \mid \text{otherwise } = \text{oddsRec} \ x s
\]

Conditionals with binding

\[
\text{oddsCond} :: \text{[Int]} \rightarrow \text{[Int]}
\]
\[
\text{oddsCond} \ \text{ws} =
\]
\[
\quad \text{if null ws then}
\]
\[
\quad \quad []
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{let}
\]
\[
\quad \quad \quad x = \text{head} \ \text{ws}
\]
\[
\quad \quad \quad xs = \text{tail} \ \text{ws}
\]
\[
\quad \quad \text{in}
\]
\[
\quad \quad \quad \text{if odd } x \ \text{then}
\]
\[
\quad \quad \quad \quad x : \text{oddsCond} \ \text{xs}
\]
\[
\quad \quad \quad \text{else}
\]
\[
\quad \quad \quad \quad \text{oddsCond} \ \text{xs}
\]
How recursion works—oddsRec

oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x = x : oddsRec xs
               | otherwise = oddsRec xs

oddsRec [1,2,3] =
    oddsRec (1 : (2 : (3 : []))))
= 1 : oddsRec (2 : (3 : [])))
= 1 : oddsRec (3 : [])
= 1 : (3 : oddsRec [])
= 1 : (3 : [])
= [1,3]
QuickCheck

-- odds.hs
import Test.QuickCheck

odds :: [Int] -> [Int]
odds xs = [ x | x <- xs, odd x ]

oddsRec :: [Int] -> [Int]
oddsRec [] = []
oddsRec (x:xs) | odd x = x : oddsRec xs
        | otherwise = oddsRec xs

prop_odds :: [Int] -> Bool
prop_odds xs = odds xs == oddsRec xs

[jitterbug]dts: ghci odds.hs
GHCi, version 8.0.2: http://www.haskell.org/ghc/  ?? for help
> quickCheck prop_odds
+++ OK, passed 100 tests.
Part IV

Accumulation: Sum a list
Sum

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

sum [1,2,3] =
  sum (1 : (2 : (3 : [])))
  =
  1 + sum (2 : (3 : []))
  =
  1 + (2 + sum (3 : []))
  =
  1 + (2 + (3 + sum []))
  =
  1 + (2 + (3 + 0))
  =
  6
Product

\[
\text{product} :: \text{[Int]} \rightarrow \text{Int}
\]

\[
\text{product} \ [\] \quad = \quad 1
\]

\[
\text{product} \ (x:xs) \quad = \quad x \times \text{product} \ xs
\]

\[
\text{product} \ [1,2,3]
\]

\[
= \quad \text{product} \ (1 : (2 : (3 : [])))
\]

\[
= \quad 1 \times \text{product} \ (2 : (3 : []))
\]

\[
= \quad 1 \times (2 \times \text{product} \ (3 : []))
\]

\[
= \quad 1 \times (2 \times (3 \times \text{product} \ []))
\]

\[
= \quad 1 \times (2 \times (3 \times 1))
\]

\[
= \quad 6
\]
Part V

Putting it all together:
Sum of the squares of the odd numbers in a list
Two styles of definition

Comprehension

\[
\text{sumSqOdd} :: [\text{Int}] \to \text{Int} \\
\text{sumSqOdd} \; \text{xs} = \text{sum} \; [x^2 \mid x \leftarrow \text{xs}, \; \text{odd} \; x]
\]

Recursion

\[
\text{sumSqOddRec} :: [\text{Int}] \to \text{Int} \\
\text{sumSqOddRec} \; [] = 0 \\
\text{sumSqOddRec} \; (x : \text{xs}) \mid \text{odd} \; x = x^2 + \text{sumSqOddRec} \; \text{xs} \\
\mid \text{otherwise} = \text{sumSqOddRec} \; \text{xs}
\]
How recursion works—sumSqOddRec

\[
\text{sumSqOddRec} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sumSqOddRec} [] = 0 \\
\text{sumSqOddRec} (x:xs) | \text{odd} x = x \times x + \text{sumSqOddRec} \; xs \\
| \text{otherwise} = \text{sumSqOddRec} \; xs
\]

\[
\text{sumSqOddRec} \; [1,2,3] \\
= \text{sumSqOddRec} \; (1 : (2 : (3 : []))) \\
= 1 \times 1 + \text{sumSqOddRec} \; (2 : (3 : [])) \\
= 1 \times 1 + \text{sumSqOddRec} \; (3 : []) \\
= 1 \times 1 + (3 \times 3 + \text{sumSqOddRec} \; []) \\
= 1 \times 1 + (3 \times 3 + 0) \\
= 1 + (9 + 0) \\
= 10
\]