List comprehension is for "whoosh"-style programming.
Recursion is for "element-at-a-time" programming - like loops in other languages.
Before looking recursion, it's necessary to understand lists better.
Cons and append

Cons takes an element and a list.
Append takes two lists.

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

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"

(:) 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).

Cons (:) is special: any list can be written using : and [], in only one way.

Notice: the definition of lists is SELF-REFERENTIAL.
It is a WELL-FOUNDED definition because it defines a complicated list, \( x : xs \), in terms of a simpler list, \( xs \),
and ultimately in terms of the simplest list of all, [].
A list of numbers

null :: [a] -> Bool tells if a list is empty or not.
head :: [a] -> a gives the first element in a list.
tail :: [a] -> [a] gives the remainder of a list, after the first element.
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

This shows two ways of writing the same function (squares of the numbers in a list). The second version is RECURSIVE: it defines squaresRec in terms of itself. The definition is well-founded because:

- squaresRec (x:xs) is defined in terms of squaresRec xs - xs is simpler than x:xs.
- this reduces squaresRec eventually to squaresRec [], the BASE CASE, which is not recursive.

The recursive definition of squaresRec has two cases, just like the recursive definition of lists.
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

We use PATTERN MATCHING to discriminate cases and to extract the components of a constructed list. Notice the correspondence to the definition of lists.

This is exactly the same, written without using pattern matching.
How recursion works—squaresRec

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

squaresRec [1,2,3]  

Here's an example - we'll look at the computation, step by step.
How recursion works—squaresRec

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

squaresRec [1,2,3]
=

squaresRec (1 : (2 : (3 : [])))  This is what [1,2,3] means.
How recursion works—squaresRec

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

squaresRec [1,2,3]
=
      squaresRec (1 : (2 : (3 : [])))
    =    { x = 1, xs = (2 : (3 : [])) }  
       1*1 : squaresRec (2 : (3 : []))

Does the first equation apply? No
Does the second equation apply? Yes! It matches if x=1 and xs= (2:(3:[])).
We replace the expression on the left-hand side of the equation with the expression on the right-hand side.
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 : [])))
  =
  { x = 2, xs = (3 : [])}
  1*1 : (2*2 : squaresRec (3 : []))

The same thing applies to the expression squaresRec (2 : (3 : [])).
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 : [])))
=
  {  x = 3, xs = []  }
  1*1 : (2*2 : (3*3 : squaresRec []))

Likewise for the expression squaresRec (3 : []).
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 : []))

Now the first equation finally applies.
How recursion works—squaresRec

```haskell
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 : []))
```

We can do the multiplications. (We could have done them earlier.)
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]

Here is the same thing, written using list notation.
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 7.6.3: http://www.haskell.org/ghc/  ?? for help
*Main> quickCheck prop_squares
+++ OK, passed 100 tests.
*Main>

We can use QuickCheck to check that both definitions compute the same function.
Part III

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

Comprehension

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

Recursion

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

We can use GUARDS in recursive definitions too - here is the notation.
otherwise is just another name for True.
Haskell checks the cases in order to decide which to use.
Pattern matching and conditionals

Pattern matching with guards

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

Conditionals with binding

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

Again, you can do it without pattern matching and with if-then-else instead of guards.
How recursion works—oddsRec

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

oddsRec [1,2,3]  
```

Again, let's look at an example of computation, step by step.
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 : []))) This is what [1,2,3] means.
How recursion works—oddsRec

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

oddsRec \ [1,2,3]

= 

oddsRec \ (1 : (2 : (3 : []))))

= 

{ \ x = 1, \ xs = (2 : (3 : [])), \ \text{odd} \ 1 \ = \ True \ }

1 : \text{oddsRec} \ (2 : (3 : []))

The second equation applies, with \(x=1\) and \(xs = 2:(3:[])\). And then the first guard is satisfied.
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 : []))
                  =
                  { x = 2, xs = (3 : []), odd 2 = False }
                  1 : oddsRec (3 : [])

The same thing applies to the expression oddsRec (2 : (3 : [])).
This time the second guard is satisfied.
How recursion works—oddsRec

```haskell
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 : [])
= { x = 3, xs = [], odd 3 = True }
1 : (3 : oddsRec [])
```

Likewise for the expression oddsRec (3 : []). The first guard is satisfied.
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 : [])

Now the first equation finally applies.
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]
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

ghci odds.hs
GHCi, version 7.6.3: http://www.haskell.org/ghc/  ?? for help
*Main> quickCheck prop_odds
+++ OK, passed 100 tests.
*Main>
Part IV

Accumulation: Sum a list
Sum

\[
\text{sum :: [Int] -> Int}
\]

\[
\text{sum [] = 0}
\]

\[
\text{sum (x:xs) = x + sum xs}
\]

Here is an example that can't be done using comprehension. (sum is built into Haskell - we don't need to define it ourselves.)

Computing this example step by step.
Sum

\[
\text{sum :: [Int] -> Int}
\]

\[
\text{sum [] = 0}
\]

\[
\text{sum (x:xs) = x + sum xs}
\]

\[
\text{sum [1,2,3] = sum (1 : (2 : (3 : [])))}
\]
Sum

\[ \text{sum} :: [\text{Int}] \rightarrow \text{Int} \]
\[ \text{sum} \ [\] \quad = \quad 0 \]
\[ \text{sum} \ (x:xs) \quad = \quad x \ + \ \text{sum} \ \text{xs} \]

\[ \text{sum} \ [1,2,3] \]
\[ = \]
\[ \text{sum} \ (1 : (2 : (3 : []))) \]
\[ = \quad \{ x = 1, \ \text{xs} = (2 : (3 : [])) \} \]
\[ 1 \ + \ \text{sum} \ (2 : (3 : [])) \]
Sum

\[
\begin{align*}
\text{sum} & : [\text{Int}] \to \text{Int} \\
\text{sum} \ [] & = 0 \\
\text{sum} \ (x:xs) & = x + \text{sum} \ xs
\end{align*}
\]

\[
\begin{align*}
\text{sum} \ [1,2,3] \\
&= \\
&= \text{sum} \ (1 : (2 : (3 : []))) \\
&= \\
&= 1 + \text{sum} \ (2 : (3 : [])) \\
&= \{x = 2, \ xs = (3 : [])\} \\
&= 1 + (2 + \text{sum} \ (3 : []))
\end{align*}
\]
Sum

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

\[
\text{sum} \; [] = 0
\]

\[
\text{sum} \; (x:xs) = x + \text{sum} \; xs
\]

\[
\text{sum} \; [1,2,3]
\]

\[
= \text{sum} \; (1 : (2 : (3 : [])))
\]

\[
= 1 + \text{sum} \; (2 : (3 : []))
\]

\[
= 1 + (2 + \text{sum} \; (3 : []))
\]

\[
= \{ x = 3, \; xs = [] \}
\]

\[
1 + (2 + (3 + \text{sum} \; []))
\]
Sum

\[
\text{sum} : : \left[\text{Int} \right] \rightarrow \text{Int} \\
\text{sum} \left[\right] = 0 \\
\text{sum} \left(x : xs\right) = x + \text{sum} \; xs
\]

\[
\text{sum} \left[1, 2, 3\right] \\
= \text{sum} \left(1 : (2 : (3 : \left[\right]))\right) \\
= 1 + \text{sum} \left(2 : (3 : \left[\right])\right) \\
= 1 + (2 + \text{sum} \left(3 : \left[\right]\right)) \\
= 1 + (2 + (3 + \text{sum} \left[\right]\right)) \\
= 1 + (2 + (3 + 0))
\]
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 :: \intlist \rightarrow \int} \\
\text{product \[]} = 1 \\
\text{product (x:xs) = x \times \text{product xs}}
\]

\[
\text{product [1,2,3]} = \\
\text{product (1 : (2 : (3 : [])))} = \\
1 \times \text{product (2 : (3 : []))} = \\
1 \times (2 \times \text{product (3 : [])}) = \\
1 \times (2 \times (3 \times \text{product []})) = \\
1 \times (2 \times (3 \times 1)) = 6
\]
Part V

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

Comprehension

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

Recursion

\[
\text{sumSqOddRec} :: [\text{Int}] \rightarrow \text{Int} \\
\text{sumSqOddRec} \; [] \; = \; 0 \\
\text{sumSqOddRec} \; (x:xs) \; \mid \; \text{odd} \; x \; = \; x^2 + \text{sumSqOddRec} \; xs \\
\mid \; \text{otherwise} \; = \; \text{sumSqOddRec} \; xs
\]

Here's a recursive definition of the sum of the squares of the odd numbers in a list.
How recursion works—sumSqOddRec

\[
\text{sumSqOddRec} :: \text{[Int]} \rightarrow \text{Int} \\
\text{sumSqOddRec} \ [\] \quad = \quad 0 \\
\text{sumSqOddRec} \ (x:xs) \mid \text{odd } x \quad = \quad x^2 + \text{sumSqOddRec } xs \\
| \text{otherwise} \quad = \quad \text{sumSqOddRec } xs
\]

\text{sumSqOddRec} \ [1,2,3] \quad \text{Computing this example step by step.}
How recursion works—sumSqOddRec

sumSqOddRec :: [Int] -> Int
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
| otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3] = sumSqOddRec (1 : (2 : (3 : [])))
How recursion works—sumSqOddRec

sumSqOddRec :: [Int] -> Int
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
| otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3] =
    sumSqOddRec (1 : (2 : (3 : [])))
=  
    { x = 1, xs = (2 : (3 : [])), odd 1 = True }
    1*1 + sumSqOddRec (2 : (3 : []))
How recursion works—sumSqOddRec

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

\[
\text{sumSqOddRec} [1,2,3]
\]
\[
= \text{sumSqOddRec} (1 : (2 : (3 : [])))
\]
\[
= 1 \times 1 + \text{sumSqOddRec} (2 : (3 : []))
\]
\[
= \{ x = 2, \; xs = (3 : []), \; \text{odd } 2 = \text{False} \} \]
\[
= 1 \times 1 + \text{sumSqOddRec} (3 : [])
\]
How recursion works—sumSqOddRec

sumSqOddRec :: [Int] -> Int
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | odd x = x*x + sumSqOddRec xs
                  | otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3]
= sumSqOddRec (1 : (2 : (3 : [])))
= 1*1 + sumSqOddRec (2 : (3 : []))
= 1*1 + sumSqOddRec (3 : [])
= 1*1 + (3*3 : sumSqOddRec [])
How recursion works—sumSqOddRec

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

\[
\text{sumSqOddRec} \ 	ext{[]} \quad = \quad 0
\]

\[
\text{sumSqOddRec} \ (x:x) \quad | \quad \text{odd} \ x \quad = \quad x^2 + \text{sumSqOddRec} \ x
\quad | \quad \text{otherwise} \quad = \quad \text{sumSqOddRec} \ x
\]

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

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

\[
\quad = \\
\quad 1^2 + \text{sumSqOddRec} \ (2 : (3 : []))
\]

\[
\quad = \\
\quad 1^2 + \text{sumSqOddRec} \ (3 : [])
\]

\[
\quad = \\
\quad 1^2 + (3^2 + \text{sumSqOddRec} \ [])
\]

\[
\quad = \\
\quad 1^2 + (3^2 + 0)
\]
How recursion works—\(\text{sumSqOddRec}\)

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

\[
\text{sumSqOddRec} \; [] = 0
\]

\[
\text{sumSqOddRec} \; (x:xs) \mid \text{odd} \; x = x \times x + \text{sumSqOddRec} \; xs
\]

\[
\text{sumSqOddRec} \; \text{otherwise} = \text{sumSqOddRec} \; xs
\]

\[
\text{sumSqOddRec} \; [1,2,3]
\]

\[
= \text{sumSqOddRec} \; (1 : (2 : (3 : [])))
\]

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

\[
= 1 \times 1 + \text{sumSqOddRec} \; (3 : [])
\]

\[
= 1 \times 1 + (3 \times 3 + \text{sumSqOddRec} \; [])
\]

\[
= 1 \times 1 + (9 + 0)
\]

\[
= 1 + (9 + 0)
\]
How recursion works—sumSqOddRec

sumSqOddRec :: [Int] -> Int
sumSqOddRec [] = 0
sumSqOddRec (x:xs) | odd x = x * x + sumSqOddRec xs
                 | otherwise = sumSqOddRec xs

sumSqOddRec [1,2,3] =
                     = sumSqOddRec (1 : (2 : (3 : [])))
                     = 1 * 1 + sumSqOddRec (2 : (3 : []))
                     = 1 * 1 + sumSqOddRec (3 : [])
                     = 1 * 1 + (3 * 3 + sumSqOddRec [])
                     = 1 * 1 + (3 * 3 + 0)
                     = 1 + (9 + 0)
                     = 10