Notes of Learn You a Haskell for Great Good

Sean Go

December 2017

Chapter 1

Introduction

ghci

1.1 Function

```
4+5.0 = 9.0
succ 9 = 10
succ 9 + max 5 4 + 1 = 16  =  (succ 9) + (max 5 4) + 1
succ 9 * 10 = 100, succ (9 * 10) = 91
div 92 10 = 92 `div` 10 = 9
```

Listing 1.1: call function

1.2 File

```
doubleMe x = x + x
doubleUs x y = (x + y) * 2
doubleSmallNumber x = if x > 100
                        then x
                        else x*2
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
conanO'Brien = "It's a-me, Conan O'Brien!"
```

Listing 1.2: baby.hs

doubleMe(doubleMe(doubleMe(1))) = 8
But i do not know how to write doubleUs(doubleUs(), doubleUs()), it caused error

1.3 List

1.3.1 Basic let, ++, :, !!

```
Prelude> let lostNumbers = [4,8,15,16,23,42]
Prelude> lostNumbers
[4,8,15,16,23,42]
Prelude> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
Prelude> "hello" ++ " " ++ "world"
"hello world"
Prelude> ['w','o'] ++ ['o','t']
"woot"
Prelude> 'A':" SMALL CAT"
"A SMALL CAT"
Prelude> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
Prelude> [1,2,3] == 1:2:3:[]
True
Prelude> "Steve Buscemi" !! 6
'B'
Prelude> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
Prelude> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
Prelude> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
Prelude> b !! 2
[1,2,2,3,4]
```

Listing 1.3: Basic List

1.3.2 Compare

Lists can be compared if the items they contain can be compared. When using <, <=, >= and > to compare two lists, they are compared in lexicographical order.

```
Prelude> [3,2,1] > [2,1,0]
True
Prelude> [3,2,1] > [2,10,100]
True
```

```
Prelude> [3,2,1] > [2,10]
True
Prelude> [3,2] > [2,10,100]
True
Prelude> [3] > [2,10,100]
True
Prelude> [2] > [2,10,100]
False
Prelude> [3,4,2] < [3,4,3]
True
Prelude> [3,4,2] == [3,4,2]
True
```

Listing 1.4: Compare List

1.3.3 More: head,tail,last,init,null,reverse, take,maximum,minimum,sum,product,elem

Here are some more basic list functions, followed by examples of their usage.

```
ghci> head [5,4,3,2,1]
5
ghci> tail [5,4,3,2,1]
[4,3,2,1]
ghci> last [5,4,3,2,1]
1
ghci> init [5,4,3,2,1]
[5,4,3,2]
ghci> length [5,4,3,2,1]
5
ghci> null [1,2,3]
False
ghci> null []
True
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
ghci> take 3 [5,4,3,2,1]
```

```
[5,4,3]
ghci> take 1 [3,9,3]
[3]
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
ghci> maximum [1,9,2,3,4]
9
ghci> minimum [8,4,2,1,5,6]
1
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

Listing 1.5: More Option

1.4 Range

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
ghci> [13,26..24*13]
[13,26,39,52,65,78,91,104,117,130,143,156,
 169,182,195,208,221,234,247,260,273,286,299,312]
ghci> take 24 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143,156,
 169,182,195,208,221,234,247,260,273,286,299,312]
ghci> take 10 (cycle [1,2,3])
[1,2,3,1,2,3,1,2,3,1]
```

```
ghci> take 12 (cycle "LOL ")
"LOL LOL LOL "
ghci> take 10 (repeat 5)
[5,5,5,5,5,5,5,5,5,5]
ghci> replicate 3 10
[10,10,10]
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

Listing 1.6: Range

1.5 Comprehension

$\{2 \cdot x | x \in N, x < 10\}$

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]

boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]

ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]

ghci> [x+y | x <- [1,2,3], y <- [10,100,1000]]
[11,101,1001,12,102,1002,13,103,1003]

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]

ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]


ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]


length' xs = sum [1 | _ <- xs]


removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"


ghci> let xxs = [[1,3,5,2,3,1,2,4,5],[1,2,3,4,5,6,7,8,9],
                 [1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

Listing 1.7: Comprephension

1.6 Tuples

1.6.1 Tuples

```
ghci> (1, 3)
(1,3)
ghci> (3, 'a', "hello")
(3,'a',"hello")
ghci> (50, 50.4, "hello", 'b')
(50,50.4,"hello",'b')

ghci> [(1,2),(8,11,5),(4,5)]
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Listing 1.8: Tuples

1.6.2 Pair, fst, snd, zip

```
ghci> fst (8, 11)
8
```

```
ghci> fst ("Wow", False)
"Wow"

ghci> snd (8, 11)
11
ghci> snd ("Wow", False)
False

ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1..5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

Listing 1.9: Pair

1.6.3 Application, Find Right Triangle

We'll use Haskell to find a right triangle that fits all of these conditions:

1. The lengths of the three sides are all integers.
2. The length of each side is less than or equal to 10.
3. The triangle¡s perimeter (the sum of the side lengths) is equal to 24.

```
ghci> let triples = [ (a,b,c) | c <- [1..10], a <- [1..10], b <- [1..10] ]
ghci> let rightTriangles = [ (a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a],
a^2 + b^2 == c^2]
ghci> let rightTriangles' = [ (a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a],
a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(6,8,10)]
```

Listing 1.10: Find Right Triangle

Chapter 2

Belive The Type

2.1 Explicit Type Declaration, :t, ::,

```haskell
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool


removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]

addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Listing 2.1: Explicit Type Declaration

2.2 Common Haskell Types

Char, Int, Integer, Float, Double, Bool, Tuples

2.3 Type Variables

```
ghci> :t head
head :: [a] -> a

ghci> :t fst
fst :: (a, b) -> a
```

Listing 2.2: type variable

a and *b* ... are *type variables*, can be of any type.

2.4 Type Classes

A *type class* is an interface that defines some behavior. If a type is an instance of a type class, then it supports and implements the behavior the type class describes.

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

Listing 2.3: Type Classes

the => symbol. Everything before this symbol is called a *class constraint*. The *Eq* type class provides an interface for testing for equality.

2.4.1 Eq

==, / =

2.4.2 Ord

>, <, >=, <=

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool

:t (<=)
(<=) :: Ord a => a -> a -> Bool
```

Listing 2.4: Ord

2.4.3 The Show Type Class

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Listing 2.5: show

2.4.4 The Read Type Class

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Listing 2.6: read

```
ghci> :t read
read :: (Read a) => String -> a
```

Listing 2.7: type of read

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
\\
ghci> [read "True", False, True, False]
[True, False, True, False]
```

Listing 2.8: example read

2.4.5 The Enum Type Class

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

Listing 2.9: enum type class

2.4.6 The Bounded Type Class

Instances of the Bounded type class have an upper bound and a lower bound, which can be checked by using the minBound and maxBound functions:

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False

ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Listing 2.10: bounded

2.4.7 The Num Type Class

Num is a numeric type class. Its instances can act like numbers. Let's examine the type of a number:

```
ghci> :t 20
20 :: (Num t) => t

ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

```

ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

Listing 2.11: Num Type Class

2.4.8 The Integral Type Class

```
fromIntegral :: (Num b, Integral a) => a -> b
length :: [a] -> Int
ghci> fromIntegral (length [1,2,3,4]) + 3.2
```

Listing 2.12: Integreal Type Class

2.4.9 Some Final Notes on Type Classes

Because a type class defines an abstract interface, one type can be an instance of many type classes, and one type class can have many types as instances. For example, the Char type is an instance of many type classes, two of them being Eq and Ord, because we can check if two characters are equal as well as compare them in alphabetical order. Sometimes a type must first be an instance of one type class to be allowed to become an instance of another. For example, to be an instance of Ord, a type must first be an instance of Eq. In other words, being an instance of Eq is a prerequisite for being an instance of Ord. This makes sense if you think about it, because if you can compare two things for ordering, you should also be able to tell if those things are equal.

Chapter 3

Syntax In Functions

3.1 Pattern Matching

Pattern matching is used to specify patterns to which some data should conform and to deconstruct the data according to those patterns.

```haskell
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"

sayMe :: Int -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"

factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)

charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"

ghci> charName 'a'
"Albert"
ghci> charName 'b'
```

```
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in function charName
```

Listing 3.1: Pattern matching

3.1.1 Pattern Matching with Tuples

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)

addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

ghci> :t addVectors
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)

first :: (a, b, c) -> a
first (x, _, _) = x
second :: (a, b, c) -> b
second (_, y, _) = y
third :: (a, b, c) -> c
third (_, _, z) = z
```

Listing 3.2: Pattern Matching with Tuples

3.1.2 Pattern Matching with Lists and List Comprehensions

```
ghci> let xs = [(1,3),(4,3),(2,4),(5,3),(5,6),(3,1)]
ghci> [a+b | (a, b) <- xs]
[4,7,6,8,11,4]

head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x

ghci> head' [4,5,6]
4
ghci> head' "Hello"
```

```
'H'

tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: " ++ show x
                  ++ " and " ++ show y

ghci> tell [1]
"The list has one element: 1"
ghci> tell [True,False]
"The list has two elements: True and False"
ghci> tell [1,2,3,4]
"This list is long. The first two elements are: 1 and 2"
ghci> tell []
"The list is empty"

badAdd :: (Num a) => [a] -> a
badAdd (x:y:z:[]) = x + y + z
ghci> badAdd [100,20]
*** Exception: examples.hs:8:0-25: Non-exhaustive patterns in function badAdd
```

Listing 3.3: Pattern Matching with Lists and List Comprehensions

3.1.3 As-patterns

```
firstLetter :: String -> String
firstLetter "" = "Empty string, whoops!"
firstLetter all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]

ghci> firstLetter "Dracula"
"The first letter of Dracula is D"
```

Listing 3.4: empty

3.2 Guards

```
bmiTell :: => Double -> String
bmiTell bmi
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise = "You're a whale, congratulations!"


bmiTell :: Double -> Double -> String
bmiTell weight height
    | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
    | weight / height ^ 2 <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | weight / height ^ 2 <= 30.0 = "You're fat! Lose some weight, fatty!"
    | otherwise = "You're a whale, congratulations!"


ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"


max' :: (Ord a) => a -> a -> a
max' a b
    | a <= b = b
    | otherwise = a

myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a == b = EQ
    | a <= b = LT
    | otherwise = GT

ghci> 3 `myCompare` 2
GT
```

Listing 3.5: guards

3.3 Where

```
bmiTell :: Double -> Double -> String
bmiTell weight height
    | bmi <= 18.5 = "You're underweight, you emo, you!"
    | bmi <= 25.0 = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
```

```haskell
    | otherwise = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2


bmiTell :: Double -> Double -> String
    bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat = "You're fat! Lose some weight, fatty!"
    | otherwise = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          skinny = 18.5
          normal = 25.0
          fat = 30.0
```

Listing 3.6: where

3.3.1 where's Scope

```haskell
greet :: String -> String
greet "Juan" = niceGreeting ++ " Juan!"
greet "Fernando" = niceGreeting ++ " Fernando!"
greet name = badGreeting ++ " " ++ name
        where niceGreeting = "Hello! So very nice to see you,"
            badGreeting = "Oh! Pfft. It's you."

badGreeting :: String
badGreeting = "Oh! Pfft. It's you."
niceGreeting :: String
niceGreeting = "Hello! So very nice to see you,"
greet :: String -> String
greet "Juan" = niceGreeting ++ " Juan!"
greet "Fernando" = niceGreeting ++ " Fernando!"
greet name = badGreeting ++ " " ++ name
```

Listing 3.7: where's Scope

3.3.2 Pattern Matching with where

```haskell
bmiTell :: Double -> Double -> String
    bmiTell weight height
    | bmi <= skinny = "You're underweight, you emo, you!"
    | bmi <= normal = "You're supposedly normal. Pffft, I bet you're ugly!"
    | bmi <= fat = "You're fat! Lose some weight, fatty!"
    | otherwise = "You're a whale, congratulations!"
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)

initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

Listing 3.8: empty

3.3.3 Functions in where Blocks

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2

*Main> calcBmis [(10, 10)]
[0.1]
*Main> calcBmis [(10, 10), (20, 10)]
[0.1,0.2]
*Main> calcBmis [(10, 10), (20, 10), (12, 90)]
[0.1,0.2,1.4814814814814814e-3]
```

Listing 3.9: Functions in where Blocks

3.4 let It Be

```
cylinder :: Double -> Double -> Double
cylinder r h =
let sideArea = 2 * pi * r * h
topArea = pi * r ^ 2
in sideArea + 2 * topArea

ghci> 4 * (let a = 9 in a + 1) + 2
42

ghci> [let square x = x * x in (square 5, square 3, square 2)]
[(25,9,4)]

ghci> (let a = 100; b = 200; c = 300 in a*b*c,
        let foo="Hey "; bar = "there!" in foo ++ bar)
(6000000,"Hey there!")

ghci> (let (a, b, c) = (1, 2, 3) in a+b+c) * 100
```

```
600
```

Listing 3.10: let

3.4.1 let in comprehensions

```haskell
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]

calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi > 25.0]
```

Listing 3.11: empty

3.4.2 let in GHCi

```
ghci> let zoot x y z = x * y + z
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: ‘boot’
```

Listing 3.12: let int ghci

3.5 case Expressions

```haskell
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

Listing 3.13: case expression

```haskell
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x


head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x

describeList :: [a] -> String
describeList ls = "The list is " ++ case ls of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."

describeList :: [a] -> String
describeList ls = "The list is " ++ what ls
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

Listing 3.14: example

Chapter 4

Hello Recurision!

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list!"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)


replicate' :: Int -> a -> [a]
replicate' n x
    | n <= 0 = []
    | otherwise = x : replicate' (n-1) x

take' :: (Num i, Ord i) => i -> [a] -> [a]
take' n _
    | n <= 0 = []
take' _ [] = []
take' n (x:xs) = x : take' (n-1) xs


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

repeat' :: a -> [a]
repeat' x = x:repeat' x

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

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x = True
    | otherwise = a `elem'` xs


quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerOrEqual = [a | a <- xs, a <= x]
        larger = [a | a <- xs, a > x]
    in quicksort smallerOrEqual ++ [x] ++ quicksort larger

ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"  abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

Listing 4.1: Recurision

Listing 4.2: empty

Listing 4.3: empty

Listing 4.4: empty

Listing 4.5: empty

Listing 4.6: empty

Listing 4.7: empty

Listing 4.8: empty

Listing 4.9: empty

Listing 4.10: empty

Listing 4.11: empty

Listing 4.12: empty

Listing 4.13: empty

Listing 4.14: empty

Listing 4.15: empty

Listing 4.16: empty

Listing 4.17: empty

Listing 4.18: empty

Listing 4.19: empty

Listing 4.20: empty

Listing 4.21: empty

Listing 4.22: empty

Listing 4.23: empty

Listing 4.24: empty

Listing 4.25: empty

Listing 4.26: empty

Listing 4.27: empty

Listing 4.28: empty

Listing 4.29: empty

Listing 4.30: empty

Listing 4.31: empty

Listing 4.32: empty

Listing 4.33: empty

Listing 4.34: empty

Listing 4.35: empty

Listing 4.36: empty

Appendix A

First Appendix

Last note