Lists & Functional Patterns Exercises

Exercise 1

The sequence of remainders, 1011, provides the binary representation of the integer 13. It is easy to implement this procedure using recursion:

```
intTobin :: Int -> [Bit]
```

For example:

```
> intTobin 13
   [1,0,1,1]
```

Exercise 2

Define a function `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` that alternately applies its two argument functions to successive elements in a list, in turn about order. For example:

Lists & Functional Patterns Exercises

```
> altMap (+10) (+100) [0,1,2,3,4]

  [10,101,12,103,14]
```

Exercise 3

1. Show how the list comprehension `[ f x | x <- xs, p x]` can be re-expressed using the higher order functions map and filter order functions map and filter .

2. Without looking at the definitions from the standard prelude, define the following higher-order library functions on lists.

a. Decide if all elements of a list satisfy a predicate:
`all :: (a -> Bool) -> [a] -> Bool`

b. Decide if any element of a list satisfies a predicate:
`any :: (a -> Bool) -> [a] -> Bool`

c. Select elements from a list while they satisfy a predicate:

Lists & Functional Patterns Exercises

```
takeWhile :: (a -> Bool) -> [a] -> [a]
```

d. Remove elements from a list while they satisfy a predicate:

```
dropWhile :: (a -> Bool) -> [a] -> [a]
```

Exercise 4

Fold has a secret twin brother named unfold which undoes what fold does. In this post, we will see what unfold is and how it is related to fold. unfoldr builds a list from a seed value while foldr reduces a list to a summary value.

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

unfoldr takes the element and returns Nothing if it is done producing the list or returns Just (a, b), in which case, a is a prepended to the list and b is used as the next element in a recursive call.

For example,

Lists & Functional Patterns Exercises

```
> unfoldr (\x -> if x == 0 then Nothing else
   Just (x, x-1)) 10 [10,9,8,7,6,5,4,3,2,1]
```

Rewrite the function IntToBin using unfold

```
intoToBin :: Int -> [Bit]
intoToBin = undefined
```

Exercise 3

Write a recursive function named replaceThe which takes a text/string, breaks it into words and replaces each instance of “the” with “a”. It’s intended only to replace exactly the word “the". notThe is a suggested helper function for accomplishing this.

```
> notThe "the"
   Nothing

> notThe "blahtheblah"
```

Lists & Functional Patterns Exercises

```
   Just "blahtheblah"

> notThe "woot"
   Just "woot"

notThe :: String -> Maybe String
notThe = undefined

> replaceThe "the cow loves us"
   "a cow loves us"

replaceThe :: String -> String
replaceThe = undefined
```

Exercise 4

Implement run-length encoding and decoding. Run-length encoding (RLE) is a simple form of data compression, where runs (consecutive data elements) are replaced by just one data value and count.

For example we can represent the original 53 characters with only 13.

```
"WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWWWWB"
-> "12WB12W3B24WB"
```

RLE allows the original data to be perfectly reconstructed from the compressed data, which makes it a lossless data compression.

```
"AABCCCDEEEE" -> "2AB3CD4E" -> "AABCCCDEEEE"
```

For simplicity, you can assume that the unencoded string will only contain the letters A through Z
(either lower or upper case) and whitespace. This way data to be encoded will never contain any
numbers and numbers inside data to be decoded always represent the count for the following
character.

Exercise 5

Given a number, determine whether or not it is valid per the Luhn formula. The Luhn algorithm is a simple checksum formula used to wada

Lists & Functional Patterns Exercises

validate a variety of identification numbers, such as credit card numbers. The task is to check if a given string is valid.

- Validating a Number Strings of length 1 or less are not valid.
- Spaces are allowed in the input, but they should be stripped before checking.
- All other non-digit characters are disallowed.

Example 1: valid credit card number

```
4539 3195 0343 6467
```

The first step of the Luhn algorithm is to double every second digit starting from the right. We will be doubling

```
4_3_3_9_0_4_6_6_
```

If doubling the number results in a number greater than 9 then subtract 9 from the product. The
results of our doubling:

```
8569 6195 0383 3437
```

Then sum all of the digits:

```
8+5+6+9+6+1+9+5+0+3+8+3+3+4+3+7 = 80
```

If the sum is evenly divisible by 10, then the number is valid. This number is valid!

Example 2: invalid credit card number

```
8273 1232 7352 0569
```

Lists & Functional Patterns Exercises

Double the second digits, starting from the right

```
7253 2262 5312 0539
```

Sum the digits

```
7+2+5+3+2+2+6+2+5+3+1+2+0+5+3+9 = 57
```

57 is not evenly divisible by 10, so this number is not valid.

Exercise 6

1. Define a function named split. This function behaves like the function words from the Data.List package. It takes a string as an argument and returns a list of words in the string.

2. Define a function named join, which is the inverse of the split function.

3. Define a function unsignedDigitsInBaseToDecimal, which generalises the unsignedBinaryToDecimal function to convert numbers in bases 2 through 10 to decimal integers. The new function expects an integer base (2 through 10) and a string of digits in that base as arguments. It returns the unsigned integer represented by the arguments. Hint: Exercise 1

4. Define a function frombaseTenToBaseX :: Int -> Base -> [Int] , which generalises the intToBin function to convert numbers from base 10 to the given base. Hint: Exercise 1
Note: You should define the Base synonym type.

Lists & Functional Patterns Exercises

Example:

```
> frombaseTenToBaseX 13 (Base 2)
  [1011]
```

Exercise 7

1. Define a new type named Book to represent books. The components of a book are a unique identifier, a title, an author, and a Sum type CheckedStatus to indicate whether or not a book has been checked out.

2. Define a set of functions to manipulate the Book type of the previous exercise. The function newBook returns a book with the given identifier, author and title (it's not checked out by default). The functions identifier, author, title, and isCheckedOut access and return the values of a book's components. The function checkOutOrReturn resets the isCheckedOut component to its logical negation.

3. Use the Data.List.lookup function in the definition of the lookup function for the AssociationList type.

4. Define the removeKey function for the AssociationList type.

Lists & Functional Patterns Exercises

5. Define two functions, keys and values, for the AssociationList type. The keys function returns a list of keys, and the values function returns a list of values.

Note: The AssociationList type is a list of tuples where book identifiers are first elements of the tuples and the actual book are the second elements

Exercise 8: small library for Maybe

Write functions implementing the following type declarations:

```
isJust :: Maybe a -> Bool
isJust = undefined

> isJust (Just 1)
   True
> isJust Nothing
   False

isNothing :: Maybe a -> Bool
isNothing = undefined

> isNothing (Just 1)
   False
> isNothing Nothing
   True

mayybee :: b -> (a -> b) -> Maybe a -> b
```

Lists & Functional Patterns Exercises

```
> mayybee 0 (+1) Nothing
   0
> mayybee 0 (+1) (Just 1)
   2


fromMaybe :: a -> Maybe a -> a

> fromMaybe 0 Nothing
   0
> fromMaybe 0 (Just 1)
   1


catMaybes :: [Maybe a] -> [a]

> catMaybes [Just 1, Nothing, Just 2]
   [1, 2]
> let xs = take 3 $ repeat Nothing
> catMaybes xs
   []


flipMaybe :: [Maybe a] -> Maybe [a]

> flipMaybe [Just 1, Just 2, Just 3]
   Just [1, 2, 3]
> flipMaybe [Just 1, Nothing, Just 3]
```

```
  Nothing
```

Exercise 9: small library for Either

Write each of the following functions. If more than one possible unique function exists for the type, use common sense to determine what it should do.

1. Try to eventually arrive at a solution that uses foldr to implement the following function.

```haskell
lefts' :: [Either a b] -> [a]
lefts' = undefined
```

2. Same as the last one. Use foldr to implement

```haskell
rights' :: [Either a b] -> [b]
rights' = undefined
```

3.

```haskell
partitionEithers' :: [Either a b] -> ([a], [b])
partitionEithers' = undefined
```

4.

```haskell
partitionEithers' :: [Either a b] -> ([a], [b])
partitionEithers' = undefined
```

5.

```haskell
eitherToMaybe :: Either a b -> Maybe b
eitherToMaybe = undefined
```

6.

```haskell
maybeToEither :: a -> Maybe b -> Either a b
maybeToEither = undefined
```

Exercise 10 : Validate the sentence

1. Use the Maybe type to write a function that counts the number of vowels in a string and the number of consonants. If the number of vowels exceeds the number of consonants, the function returns Nothing. In many human languages, vowels rarely exceed the number of consonants so when they do, it may indicate the input isn't a word (that is, a valid input to your dataset):

```haskell
newtype Word' = Word' String
```

vowels = "aeiou"

Lists & Functional Patterns Exercises

```haskell
mkWord :: String -> Maybe Word'
mkWord = undefined
```

2. Write the function

```haskell
validateSentence :: [Word'] -> [Maybe Word']
```

then write the function

```haskell
validateSentence' :: [Word'] -> Maybe [Word']
```

Exercise 11

Natural number in haskell can be represented with the following data structure:

```haskell
data Nat =
     Zero
   | Succ Nat
  deriving (Eq, Show)
```

1. Write a function that take a value of type Nat and returns the corresponding Integer value

```haskell
natToInteger :: Nat -> Integer
natToInteger = undefined
```

2. Using the Maybe type, write a function that takes an Integer and return a Nat.

Exercise 12

1. Write safer versions of the following functions using Maybe then Either sum types
   - head
   - last
   - divideBy

2. What is the difference between Maybe and Either? Indicate in which situation each of them is more suitable than the other: Justify your answer.

3. Given the type

```
data ParseDigitError = NotADigit Char deriving Show
```

Define a digit parsing function with the following declaration:

```
parseDigit :: Char -> Either ParseDigitError Int
parseDigit = undefined
```