Chapter 4 Exercises

Defining Functions

Connor Baker

Compiled on December 21, 2019 at 9:03pm

4.8 Exercises

1. Using library functions, define a function `halve :: [a] -> ([a],[a])` that splits an even-lengthed list into two halves. For example:

```
> halve [1,2,3,4,5,6]
([1,2,3],[4,5,6])
```

   Solution:

```
halve :: [a] -> ([a],[a])
halve xs = splitAt (((length xs) + 1) `div` 2) xs
```

2. Define a function `third :: [a] -> a` that returns the third element in a list that contains at least this many elements using:

   a. head and tail;
   b. list indexing !!;
   c. pattern matching.

   Solution:

```
thirdHeadAndTail :: [a] -> a
thirdHeadAndTail = head . tail . tail

thirdIndex :: [a] -> a
thirdIndex xs = xs !! 2

thirdPatternMatching :: [a] -> a
thirdPatternMatching (_:_:x:_) = x
```

3. Consider a function `safetail :: [a] -> [a]` that behaves in the same way as tail except that it maps the empty list to itself rather than producing an error. Using `tail` and the function `null :: [a] -> Bool` that decides if a list is empty or not, define `safetail` using:

   a. a conditional expression;
   b. guarded equations;
   c. pattern matching.

   Solution:

```
safetailConditional :: [a] -> [a]
safetailConditional xs = if null xs
                         then []
                         else tail xs

safetailGuarded :: [a] -> [a]
safetailGuarded xs | null xs   = []
                   | otherwise = tail xs

safetailPatternMatch :: [a] -> [a]
```

```
safetailPatternMatch [] = []
safetailPatternMatch (_:xs) = xs
```

4. In a similar way to && in section 4.4, show how the disjunction operator || can be defined in four different ways using pattern matching.

   Solution:

```
or1 :: Bool -> Bool -> Bool
True  `or1` True  = True
True  `or1` False = True
False `or1` True  = True
False `or1` False = True

or2 :: Bool -> Bool -> Bool
False `or2` False = False
_     `or2` _     = True

or3 :: Bool -> Bool -> Bool
False `or3` b = b
True  `or3` _ = True

or4 :: Bool -> Bool -> Bool
a `or4` b | a == b    = b
          | otherwise = True
```

5. Without using any other library functions or operators, show how the meaning of the following pattern matching definition for logical conjunction && can be formalized using conditional expressions:

```
True && True = True
_    && _    = False
```

   Hint: use two nested conditional expressions

   Solution

```
formalConjunction :: Bool -> Bool -> Bool
formalConjunction a b =
    if a then
        if b then
            True
        else False
    else False
```

6. Do the same for the following alternative definition, and note the difference in the number of conditional expressions that are required:

```
True  && b = b
False && _ = False
```

   Solution:

```
formalConjunction :: Bool -> Bool -> Bool
```

```haskell
formalConjunction a b =
    if a then
        b
    else
        False
```

7. Show how the meaning of the following curried function definition can be formalized in terms of lambda expressions:

```haskell
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Solution:

```haskell
mult :: Int -> Int -> Int -> Int
mult = \x -> (\y -> (\z -> x * y * z))
```

8. The Luhn algorithm is used to check bank card numbers for simple errors such as mistyping a digit, and proceeds as follows:

- consider each digit as a separate number;
- moving left, double every other number from the second last;
- subtract 9 from each number that is now greater than 9;
- add all the resulting numbers together;
- if the total is divisible by 10, the card number is valid.

Define a function `luhnDouble :: Int -> Int` that doubles a digit and subtracts 9 if the result is greater than 9. For example:

```haskell
> luhnDouble 3
6
> luhnDouble 6
3
```

Using `luhnDouble` and the integer remainder function mod, define a function `luhn :: Int -> Int -> ↪ Int -> Int -> Bool` that decides if a four-digit bank card number is valid. For example:

```haskell
> luhn 1 7 8 4
True
> luhn 4783
False
```

Solution:

```haskell
luhnDouble :: Int -> Int
luhnDouble n | n <= 5    = 2 * n
             | otherwise = 2 * n - 9

luhn :: Int -> Int -> Int -> Int -> Bool
luhn a b c d =
    (luhnDouble a + b + luhnDouble c + d)
```

```
    `mod` 10 == 0
```