CS 1187 – Homework 01

Solutions and Grading Key – 80 Points

Dr. Isaac D. Griffith

March 20, 2022

Part 01 – Haskell (20 Points)

Exercise 16 – (3 points)

A Bit is an integer that is either 0 or 1. A Word is a list of bits that represents a binary number. Here are some binary values that can be represented by Words:

```
[1,0] => 2
[1,0,0,1] => 9
[1,1,1] => 7
```

We can define functions that are the Bit equivalent of `or` and `and` as follows:

```
bitOr :: Int -> Int -> Int
bitOr 0 0 = 0
bitOr x y = 1

bitAnd :: Int -> Int -> Int
bitAnd 1 1 = 1
bitAnd x y = 0
```

Now it is possible to take the 'bitwise' *and* of two words as follows:

```
bitwiseAnd [1,0,0] [1,0,1]
 => [bitAnd 1 1, bitAnd 0 0, bitAnd 0 1]
 => [1,0,0]

bitwiseAnd [0,0,0] [1,1,0]
 => [0,0,0]
```

Write a function `bitwiseAnd` that takes two Words and creates a third Word that is the bitwise *and* of the two Words.

Solution:

```
bitwiseAnd :: [Int] -> [Int] -> [Int]
bitwiseAnd a b = zipWith bitAnd a b
```

Grading:

- (1 Point) Function type definition: `bitwiseAnd :: [Int] -> [Int] -> [Int]`
- (1 Point) Function uses `zipWith`
- (1 Point) Function makes use of `bitAnd`

Exercise 22 – (3 Points)

Write a function

```
f :: [Int] -> Int -> [Int]
```

that takes a list of Int values and an Int and returns a list of indexes at which that Int appears.

Solution:

```
f :: [Int] -> Int -> [Int]
f [] _ = []
f lst y = g lst y 0

g :: [Int] -> Int -> Int -> [Int]
g [] _ _ = []
g (x:xs) v idx = if x == v then idx : (g xs v (idx + 1))
                 else g xs v (idx + 1)
```

Exercise 23 – (4 Points)

Write a list comprehension that produces a list giving all of the positive integers that are *not* squares in the range 1 to 20.

Solution:

```
[x :: Integer | x <- [1..20], (floor (sqrt $ fromIntegral x))^2 /= x]
```

Grading:

- (1 Point) – typed value: `x :: Integer`
- (1 Point) – generator: `x <- [1..20]`
- (2 Point) – filter: `(floor (sqrt $ fromIntegral x))^2 /= x`

Exercise 25 – (5 Points)

Write a function using `foldr` that takes a list and removes each instance of a given letter.

Solution:

```
remLetter :: String -> Char -> String
remLetter [] _ = []
remLetter s c = foldr (check c) "" s

check :: Char -> Char -> String -> String
check c x y = if x == c then y
              else x:y
```

Exercise 27 – (5 Points)

Using `foldl`, write a function

```
maybeLast :: [a] -> Maybe a
```

that takes a list and returns the last element in it if there is one, otherwise it returns `Nothing`.

Solution:

```
f :: a -> a -> a
f x y = y

maybeLast :: [a] -> Maybe a
maybeLast [] = Nothing
maybeLast (x:xs) = Just $ foldl f x xs
```

Grading:

- (1 Point) type definition: `maybeLast :: [a] -> Maybe a`
- (1 Point) empty list pattern: `maybeLast [] = Nothing`
- (1 Point) list pattern using `foldl`
- (1 Point) list pattern uses Just
- (1 Point) `foldl` is correct and uses secondary function (or lambda)

Part 02 - Equational Reasoning (50 Points)

Exercise 16 – (19 Points)

Using equational reasoning, prove the following (using your implementation):

```
1.) bitwiseAnd [0,0,0] [1,1,0] => [0,0,0]
2.) bitwiseAnd [1,0,0,1,1] [1,1,1,1] => [1,0,0,1]
```

Solution:

```
bitwiseAnd [0,0,0] [1,1,0]
  = zipWith bitAnd [0,0,0] [1,1,0]                  { bitwiseAnd }
  = bitAnd 0 1 : zipWith bitAnd [0,0] [1,0]         { zipWith }
  = bitAnd 0 1 : bitAnd 0 1 : zipWith bitAnd [0] [0]  { zipWith }
  = bitAnd 0 1 : bitAnd 0 1 : bitAnd 0 0 : []       { zipWith }
  = bitAnd 0 1 : bitAnd 0 1 : 0 : []                { bitAnd }
  = bitAnd 0 1 : 0 : 0 : []                         { bitAnd }
  = 0 : 0 : 0 : []                                  { bitAnd }
  = [0,0,0]                                         { cons }
```

```
bitwiseAnd [1,0,0,1,1] [1,1,1,1]
  = zipWith bitAnd [1,0,0,1,1] [1,1,1,1]                                        { bitwiseAnd }
  = bitAnd 1 1 : zipWith bitAnd [0,0,1,1] [1,1,1]                               { zipWith }
  = bitAnd 1 1 : bitAnd 0 1 : zipWith bitAnd [0,1,1] [1,1]                      { zipWith }
  = bitAnd 1 1 : bitAnd 0 1 : bitAnd 0 1 : zipWith bitAnd [1,1] [1]             { zipWith }
  = bitAnd 1 1 : bitAnd 0 1 : bitAnd 0 1 : bitAnd 1 1 : zipWith bitAnd [1] [] { zipWith }
  = bitAnd 1 1 : bitAnd 0 1 : bitAnd 0 1 : bitAnd 1 1 : []                      { bitAnd }
  = bitAnd 1 1 : bitAnd 0 1 : bitAnd 0 1 : 1 : []                               { bitAnd }
  = bitAnd 1 1 : bitAnd 0 1 : 0 : 1 : []                                        { bitAnd }
  = bitAnd 1 1 : 0 : 0 : 1 : []                                                 { bitAnd }
  = 1 : 0 : 0 : 1 : []                                                          { bitAnd }
  = [1,0,0,1]                                                                   { cons }
```

Grading:

- (8 Points) reasoning for `bitwiseAnd [0,0,0] [1,1,0]`
- (11 Points) reasoning for `bitwiseAnd [1,0,0,1,1] [1,1,1,1]`

Exercise 25 – (21 Points)

Using equational reasoning, prove the following (using your implementation):

```
1.) input string: "bcad", letter to remove 'a'
    yields the following string: "bcd"
2.) input string: "AAA", letter to remove 'B'
    yields the following string: "AAA"
3.) input string: "", letter to remove 'a'
    yields the following string: ""
```

Solution:

```
remLetter "bcad" 'a'
  = foldr (check 'a') "" "abcada"                                 { remLetter.2 }
  = check 'a' 'b' (check 'a' 'c' (check 'a' 'a'
        (check 'a' 'd' "")))                                      { foldr }
  = check 'a' 'b' (check 'a' 'c' (check 'a' 'a'
        (if 'd' == 'a' then "" else 'd':"")))                     { check }
  = check 'a' 'b' (check 'a' 'c' (check 'a' 'a' 'd':""))          { eval }
  = check 'a' 'b' (check 'a' 'c'
        (if 'a' == 'a' then 'd':"" else 'a':'d':""))              { check }
  = check 'a' 'b' (check 'a' 'c' 'd':"")                          { eval }
  = check 'a' 'b' (if 'c' == 'a' then 'd':""
        else 'c':'d':"")                                          { check }
  = check 'a' 'b' 'c':'d':"")                                     { eval }
  = if 'b' == 'a' then 'c':'d':"" else 'b':'c':'d':""             { check }
  = 'b':'c':'d':""                                                { eval }
  = "bcd"                                                         { cons }

remLetter "AAA" 'B'
  = foldr (check 'B') "" "AAA"                                    { remLetter.2 }
  = check 'B' 'A' (check 'B' 'A' (check 'B' 'A' ""))              { foldr }
  = check 'B' 'A' (check 'B' 'A' (if 'A' == 'B' then "" else 'A':"")) { check }
  = check 'B' 'A' (check 'B' 'A' "")                              { eval }
  = check 'B' 'A' (if 'A' == 'B' then "" else 'A':"")             { check }
  = check 'B' 'A' ""                                              { eval }
  = if 'A' == 'B' then "" else 'A':""                             { check }
  = ""                                                            { eval }

remLetter "" 'a'
  = remLetter [] 'a'                                              { String def. }
  = []                                                            { remLetter.1 }
  = ""                                                            { String def. }
```

Grading:

- (11 Points) reasoning for `remLetter "bcad" 'a'`
- (8 Points) reasoning for `remLetter "AAA" 'B'`
- (2 Points) reasoning for `remLetter "" 'a'`

Exercise 27 – (10 Points)

Using equational reasoning, prove the following (using your implementation):

```
1.) maybeLast [1,1,1,0,0,0,2] => Just 2
2.) maybeLast [] => Nothing
```

Solution:

```
maybeLast [1,1,1,0,0,0,2]
  = Just $ foldl f 1 [1,1,0,0,0,2]             { maybeLast.1 }
  = Just $ f (f (f (f (f (f 1 1) 1) 0) 0) 0) 2  { foldl }
  = Just $ f (f (f (f (f 1 1) 0) 0) 0) 2        { f }
  = Just $ f (f (f (f 1 0) 0) 0) 2              { f }
  = Just $ f (f (f 0 0) 0) 2                    { f }
  = Just $ f (f 0 0) 2                          { f }
  = Just $ f 0 2                                { f }
  = Just 2                                      { f }

maybeLast []
  = Nothing                                     { maybeLast.2 }
```

Grading:

- (9 Points) reasoning for `maybeLast [1,1,1,0,0,0,2]`
- (1 Point) reasoning for `maybeLast []`