Functional Programming: Assignment 5

Group 60

Lucas van der Laan
s1047485

1 Warm-up

1

- const x y :: a -> b -> a
- ($->) :: a -> (a -> b) -> b
- oper :: (Fractional a) => [char] -> a -> a -> a
- mapMap :: (a -> b) -> [[a]] -> [[b]]
- without :: (a -> Bool) -> [a] -> [a]
- on :: (b -> b -> c) -> (a -> b) -> a -> a -> c

2

f1: (Input + 1) * 5

f2: (Input * 5) + 1

f3: It sets the maximum of the input to 100 and the minimum to 0, if it's inbetween, it will just result the input.

f4: Returns true if the length of the list is smaller than 5, false otherwise

What if we change f4 to (5<): We now check the other side of the equation, so it will do the opposite, it basically makes it bigger than 5.
What if we change f4 to ((<)5): We now apply < to 5, which means that it will again do bigger than.

3

1.

```haskell
onlyElem :: (Eq a) => a -> [a] -> Bool
onlyElem n = (== 1) . length . filter (== n)
```

2.

```haskell
onlyOnce :: (a -> Bool) -> [a] -> Bool
onlyOnce f = (== 1) . length . filter f
```

3.

```haskell
onlyElem :: (Eq a) => a -> [a] -> Bool
onlyElem n = onlyOnce (== n)
```

4

Tried some stuff, nothing worked.

5

```haskell
and :: [Bool] -> Bool
and = foldr (&&) True

or :: [Bool] -> Bool
or = foldr (||) False

elem :: (Eq a) => a -> [a] -> Bool
elem n = foldr (\x -> (||) (n == x)) False

maximum :: (Ord a) => [a] -> a
maximum [] = error "An empty list has no maxmium"
maximum xs = foldl1 max xs

fromList :: (Ord a) => [a] -> Tree a
fromList = foldr insert Leaf

fromBits :: [Integer] -> Integer
fromBits = foldr (\x acc -> 2 * acc + x) 0
```

I prefer the written out version of recursion, as it is easier to comprehend.

2 Mandatory

6

```haskell
--define using the _list design pattern_
compose :: [a -> a] -> a -> a
compose [] = id
compose (x:xs) = x . compose xs

--define using `foldr`
compose' :: [a -> a] -> (a -> a)
compose' = foldr (.) id
```

It computes the multiplication of all the numbers between 1 and n
compose (map (*) [1..n]) 1 => compose ([(
x -> x * 1), (
x -> x * 2), .. n]) 1
So it then applies all these functions to the new result, with the first doing it on 1
So this becomes 1 * 1 * 2 * 3 * 4 * .. * n

```haskell
--define in terms of *only* `map` and `compose`
foldr' :: (a -> b -> b) -> b -> [a] -> b
foldr' func input xs = compose (map func xs) input
```

7

```haskell
bits :: Int -> [Int]
bits = unfoldr (\x -> if x == 0 then Nothing else Just (x `mod` 2, x `div` 2))

zip :: [a] -> [b] -> [(a,b)]
zip xs ys = unfoldr go seed
  where
    seed = (xs, ys)
    go (xs, []) = Nothing
    go ([], ys) = Nothing
    go (x:xs, y:ys) = Just ((x, y), (xs, ys))

take :: Int -> [a] -> [a]
take n xs = unfoldr go seed
  where
    seed = (n, xs)
    go (0, xs) = Nothing
    go (n, []) = Nothing
    go (n, x:xs) = Just(x, (n-1,xs))

primes :: [Integer]
primes = unfoldr go [2..]
  where
    -- No need for empty list case, since we always put [2..]
    go (p:xs) = Just(p, [n | n <- xs, n `mod` p /= 0])
```

```haskell

apo :: (t -> Either [a] (a, t)) -> t -> [a]
apo f seed = case f seed of
  Left l        -> l
  Right (a,ns) -> a : apo f ns

(++) :: [a] -> [a] -> [a]
xs ++ ys = apo go xs
  where
    go [] = Left ys
    go (x:xs) = Right (x, xs)

insert :: (Ord a) => a -> [a] -> [a]
insert x = apo go
  where
    go [] = Left [x]
    go (y:ys)
      | x <= y = Left (x:y:ys)
      | otherwise = Right (y,ys)

unfoldrApo :: (t -> Maybe (a, t)) -> t -> [a]
unfoldrApo f seed = apo go seed
  where
    go seed = case f seed of
      Nothing -> Left []
      Just x -> Right x
```

8

See WordState.hs, lines of code are too long.