Muhammad Hargi Muttaqin
191524027

1. filt :: [a] -> (a -> Bool) -> (a -> b) -> [b]
   filt xs p f = map f (filter p xs)

```
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, modules loaded: Main.
*Main> filt [1,4,3,2] even (+1)
[5,3]
*Main> filt [1,4,3,2] even (*2)
[8,4]
*Main> filt [1,4,3,2] odd (*2)
[2,6]
*Main> filt [1,4,3,2] odd (-2)

<interactive>:8:1:
    Non type-variable argument in the constraint: Num (a -> b)
    (Use FlexibleContexts to permit this)
    When checking that `it' has the inferred type
      it :: forall b a. (Integral a, Num (a -> b)) => [b]
*Main> filt [1,4,3,2] odd (+2)
[3,5]
*Main>
```

2. a. all' :: (a -> Bool) -> [a] -> Bool
   all' p = and.map p

```
*Main> :load 2.hs
[1 of 1] Compiling Main             ( 2.hs, interpreted )
Ok, modules loaded: Main.
*Main> all odd [2,4,4,6]
False
*Main> all odd [2,4,4,8]
False
*Main> all even [2,4,4,8]
True
*Main> any even [1,2,3,5]
True
*Main>
```

   b. any' :: (a -> Bool) -> [a] -> Bool
   any' p = or.map p

```
Prelude> :load 2b.hs
[1 of 1] Compiling Main             ( 2b.hs, interpreted )
Ok, modules loaded: Main.
*Main> any even [1,2,4,5]
True
*Main> any odd [2,4,5,6]
True
*Main> any odd [2,4,4,6]
False
*Main>
```

   c. takeWhile' :: (a -> Bool) -> [a] -> [a]
   takeWhile' _ [] = []
   takeWhile' p (x:xs) | p x = x : takeWhile' p xs
                       | otherwise = []

```
*Main> :load 2c.hs
[1 of 1] Compiling Main             ( 2c.hs, interpreted )

2c.hs:4:1: Warning: Tab character
Ok, modules loaded: Main.
*Main> takeWhile (<3) [1,2,3,4,5,6]
[1,2]
*Main> takeWhile (<1) [1,2,3,4,5,6]
[]
*Main> takeWhile (<5) [1,2,3,4,5,6]
[1,2,3,4]
*Main>
```

   d. dropWhile' :: (a -> Bool) -> [a] -> [a]
   dropWhile' _ [] = []
   dropWhile' p (x:xs) | p x = x : dropWhile' p xs
                       | dropWhile = x:xs

```
*Main> :load 2d.hs
[1 of 1] Compiling Main             ( 2d.hs, interpreted )

2d.hs:4:1: Warning: Tab character
Ok, modules loaded: Main.
*Main> takeWhile (<5) [1,2,3,4,5,6]
[1,2,3,4]
*Main> dropWhile(<5) [1,2,3,4,5,6]
[5,6]
*Main> dropWhile(<2) [1,2,3,4,5,6]
[2,3,4,5,6]
*Main>
```

3. mapF :: (a -> b) -> [a] -> [b]

Muhammad Hargi Muttaqin
191524027

```
mapF f  = foldr ((:).f) []

filterF ::  (a -> Bool) -> [a] -> [a]
filterF p = foldr (\x xs -> if p  x then x : xs else xs) []
```

```
*Main> :load 3.hs
[1 of 1] Compiling Main             ( 3.hs, interpreted )
Ok, modules loaded: Main.
*Main> mapF (*4) [1,2,3,4,5,6,7,8,9]
[4,8,12,16,20,24,28,32,36]
*Main> filterF (<9) [1,2,3,4,5,6,7,8,9,0]
[1,2,3,4,5,6,7,8,0]
*Main>
```

4.
```
dec2int :: [Int] -> Int
dec2int = foldl (\v -> \x -> 10*v + x) 0
```

```
*Main> :load 4.hs
[1 of 1] Compiling Main             ( 4.hs, interpreted )
Ok, modules loaded: Main.
*Main> dec2int [1,2,3,4]
1234
*Main> dec2int [10,12,13,14]
11344
*Main>
*Main> dec2int [2,3,4,5]
2345
*Main>
```

5.
```
curry' :: ((a,b) -> c) -> a -> b -> c
curry' f = \x y -> f (x, y)

uncurry' :: (a -> b -> c) -> ((a, b) -> c)
uncurry' f =  \(x, y) -> f x y
```

```
*Main> :load 5.hs
[1 of 1] Compiling Main             ( 5.hs, interpreted )
Ok, modules loaded: Main.
*Main> curry fst 1 9
1
*Main> uncurry mod (10,3)
1
*Main> uncurry div (10,3)
3
*Main>
```

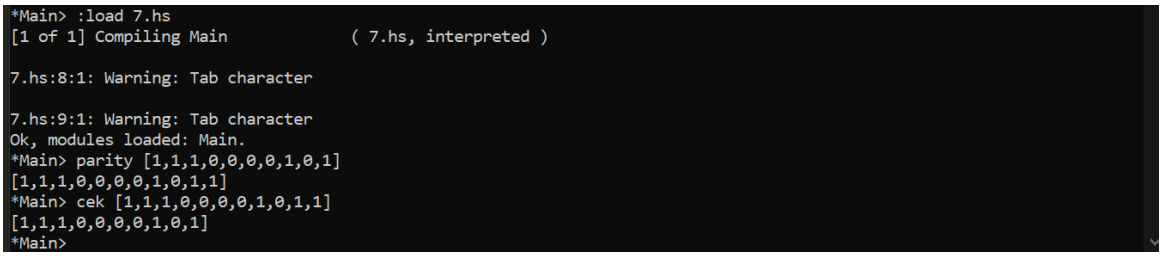
6.
```
unfold p h t x | p x = []
                      | otherwise = h x : unfold p h t (t x)
nt2bin :: Int -> [Int]
int2bin = unfold (==0) (`mod` 2) (`div` 2)

chop8 :: [Int] -> [[Int]]
chop8 = unfold (==[]) (take 8) (drop 8)

map' :: Eq a => (a -> b) -> [a] -> [b]
map' f = unfold (==[]) (f.head) (tail)

iterate :: (Int -> Int) -> Int -> [Int]
iterate f = unfold (>144) id f
```

```
*Main> :load 6.hs
[1 of 1] Compiling Main             ( 6.hs, interpreted )

6.hs:2:1: Warning: Tab character
Ok, modules loaded: Main.
*Main> map (*2) [1..9}

<interactive>:46:15: parse error on input `}'
*Main> map (*2) [1..9]
[2,4,6,8,10,12,14,16,18]
*Main> chop8 [1,1,1,0,0,0,1,1,0]
[[1,1,1,0,0,0,1,1],[0]]
*Main> int2bin 10
[0,1,0,1]
*Main>
```

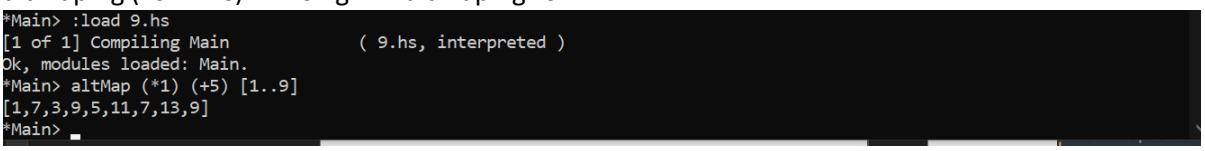
Muhammad Hargi Muttaqin
191524027

7. parity :: [Int] -> [Int]
   parity xs | even (sum xs) = xs ++ [0]
   |otherwise = xs ++ [1]

   cek :: [Int] -> [Int]
   cek [] = []
   cek xs | even (sum xs) = init xs
   | otherwise = error "Error"



8. –
9. altMap :: (a->b) -> (a -> b) -> [a] -> [b]
   altMap _ _ [] = []
   altMap f _ (x0 : []) = f x0 : []
   altMap f g (x0:x1:[]) = f x0 : g x1 : []
   altMap f g (x0:x1:xs) = f x0 : g x1 : altMap f g xs



10. altMap :: (a->b) -> (a -> b) -> [a] -> [b]
    altMap _ _ [] = []
    altMap f _ (x0 : []) = f x0 : []
    altMap f g (x0:x1:[]) = f x0 : g x1 : []
    altMap f g (x0:x1:xs) = f x0 : g x1 : altMap f g xs

    luhnDouble :: Int -> Int
    luhnDouble n
    | n*2 < 9 = n*2
    | otherwise = (n*2 - 9)

    luhn :: [Int] -> Bool
    luhn ns | mod ((sum. altMap id luhnDouble . reverse) ns) 10 == 0 = True
    | otherwise = False

Muhammad Hargi Muttaqin
191524027

```
10.hs:14:1: Warning: Tab character
Ok, modules loaded: Main.
*Main> luhn [9,8,7,6,5,4,3,2,1,2,3,4,5,6,7,8,9]
False
*Main> luhn [9,8,1]
False
*Main> luhn [5,8,1]
False
*Main> luhn [5,2,2,1]
False
*Main> luhn [9,8,7,6,5,4,3,2,1,2,3,4,5,6,7,8,]

<interactive>:7:39: parse error on input `]'
*Main> luhn [9,8,7,6,5,4,3,2,1,2,3,4,5,6,7,8]
False
*Main> luhn [9,8,7,6,5,4,3,2,1,2,3,4,5,6,7]
False
*Main> luhn [1,2,3,4,5,6,7,8,9,8,7,6,5,4,3,2]
False
*Main> luhn [6,0,1,3,4,0,0,2,3,4,1,2,4,3,2,1]
False
*Main> luhn [5,2,2,1,8,4,2,1,3,1,4,2,1,9,3,2]
True
*Main>
```

8.9

1. type Nat = Int

addN :: Nat -> Nat -> Nat

addN m n = m + n

multipNat :: Nat -> Nat -> Nat

multipNat 0 n = 0

multipNat n 0 = 0

multipNat m n = addN n (multipNat (m-1)n)

```
Prelude> :load 1.hs
[1 of 1] Compiling Main             ( 1.hs, interpreted )
Ok, modules loaded: Main.
*Main> multipleNat 10 0

<interactive>:3:1:
    Not in scope: `multipleNat'
    Perhaps you meant `multipNat' (line 6)
*Main> multipNat 10 0
0
*Main> multipNat 10 10
100
*Main> multipNat 2 26
52
*Main>
```

2. data Tree a = Leaf a | Node (Tree a) a (Tree a)

t :: Tree Int

t = Node (Node (Leaf 1) 3 (Leaf 4)) 5 (Node (Leaf 6) 7 (Leaf 9))

occurs :: Ord a => a -> Tree a -> Bool

occurs x (Leaf y) = x == y

occurs x (Node l y r) = case compare x y of LT -> occurs x l

EQ -> True

Muhammad Hargi Muttaqin
191524027

GT -> occurs x r

```
*Main> :load 2.hs
[1 of 1] Compiling Main             ( 2.hs, interpreted )
Ok, modules loaded: Main.
*Main> occours 3 t

<interactive>:10:1:
    Not in scope: `occours'
    Perhaps you meant `occurs' (line 7)
*Main> occurs 3 t
True
*Main> occurs 1 t
True
*Main> occurs 9 t
True
*Main> occurs 10 t
False
*Main>
```

3. data TreeBal a = LeafBal a | NodeBal (TreeBal a) (TreeBal a) deriving Show

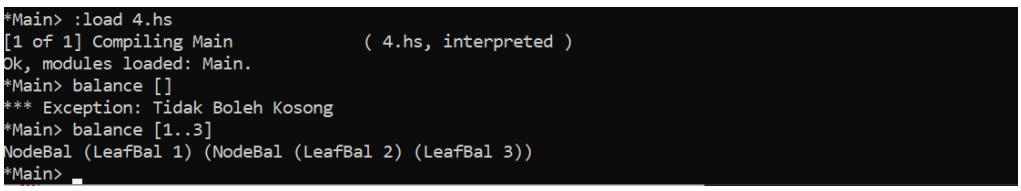
```
tBal :: TreeBal Int
tBal = NodeBal (NodeBal (LeafBal 1) (LeafBal 4))
               (NodeBal (LeafBal 6) (LeafBal 9))
tBal2 :: TreeBal Int
tBal2 = NodeBal
    (NodeBal
        (LeafBal 4)
        (NodeBal
            (LeafBal 13)
            (LeafBal 0)))
    (NodeBal
        (NodeBal
            (LeafBal 333)
            (NodeBal
                (LeafBal 6)
                (LeafBal 9)))
        (LeafBal 42))
isBal :: TreeBal Int -> Bool
isBal (LeafBal _) = True
isBal (NodeBal l r) = (diffLeaves <= 1) && isBal l && isBal r
    where
diffLeaves = abs (numLeaves l - numLeaves r)
```

Muhammad Hargi Muttaqin
191524027

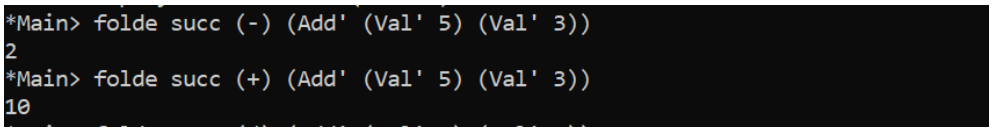
```
numLeaves :: TreeBal Int -> Int

numLeaves (LeafBal _) = 1

numLeaves (NodeBal l r) = numLeaves l + numLeaves r
```

4. data TreeBal a = LeafBal a | NodeBal (TreeBal a) (TreeBal a) deriving Show

```
balance     :: [a] -> TreeBal a

balance []  =  error " Tidak Boleh Kosong"

balance [y] =  LeafBal y

balance ys  =  NodeBal (balance (fst (halveL ys))) (balance (snd (halveL ys)))


halveL :: [a] -> ([a],[a])

halveL [] = ([],[])

halveL ys = splitAt (half ys) ys

 where

 half :: [a] -> Int

 half ys = div (length ys) 2
```



5.
```
data Expr' = Val' Int | Add' Expr' Expr'

folde :: (Int -> i) -> (i -> i -> i) -> Expr' -> i

folde f g (Val' x)   = f x

folde f g (Add' x y) = g (folde f g x) (folde f g y)
```



Muhammad Hargi Muttaqin
191524027

6. data Expr' = Val' Int | Add' Expr' Expr'

```
folde :: (Int -> i) -> (i -> i -> i) -> Expr' -> i

folde f g (Val' x)   = f x

folde f g (Add' x y) = g (folde f g x) (folde f g y)


evalE           :: Expr' -> Int

evalE (Val' k)     =  k

evalE (Add' e1 e2) =  folde toEnum (+) (Add' e1 e2)


numVals           :: Expr' -> Int

numVals (Val' k)     =  1

numVals (Add' e1 e2) =  numVals e1 + numVals e2
```