Funktionale Programmierung
WS 2019/2020
Prof. Dr. Margarita Esponda

Zwischenklausur

Name: …….. **Vorname:** ……..……..……

Matrikel-Nr: ……..……

Die maximale Punktzahl ist 100.

Aufgabe	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**Summe**	**Note**
Max. Punkte	8	10	12	10	8	14	16	16	6	100	
Punkte											

Wichtige Hinweise:

1) Schreiben Sie in allen Funktionen die entsprechende Signatur.
2) Verwenden Sie die vorgegebenen Funktionsnamen, falls diese angegeben werden.
3) Die Zwischenklausur muss geheftet bleiben.
4) Schreiben Sie Ihre Antworten in den dafür vorgegebenen freien Platz, der unmittelbar nach der Frage steht.

Viel Erfolg!

1. Aufgabe (8 Punkte)

Reduzieren Sie folgende zwei Haskell-Ausdrücke zur **Normalform**. Schreiben Sie mindestens **drei** Reduktionsschritte auf oder begründen Sie Ihre Antwort.

a) [x | xs<-["zwei", "drei", "vier"], x<-xs, (x/='e'), (x/='i')]

b) ((foldr (+) 1) **.** (map (div 4))) [1..5]

Lösung:

a) [x | xs<-["zwei", "drei", "vier"], x<-xs, (x/='e'), (x/='i')]

=> ['z', 'w', 'e', 'i', 'd', 'r', 'e', 'i', 'v', 'i', 'e', 'r']

=> ['z', 'w', 'd ', 'r', 'v','r'] **- - 3 P.**

=> "zwdrvr" **- - 1 P.**

b) ((foldr (+) 1) . (map (div 4))) [1..5]

=> (foldr (+) 1) ((map (div 4)) [1..5])

=> (foldr (+) 1) [4, 2, 1, 1, 0]

=> (+) 4 ((foldr (+) 1) [2, 1, 1, 0])

=> (+) 4 ((+) 2 (foldr (+) 1 [1, 1, 0])) **- - 3 P.**

=> 9 **- - 1 P.**

2. Aufgabe (10 Punkte)

Definieren Sie eine polymorphe Funktion **subList**, die eine Liste und zwei natürliche Zahlen **n**, **m** als Argument bekommt und die Teilliste der Länge **m** beginnend bei Position **n** berechnet:

Anwendungsbeispiel: **subList** "Beispiel" 2 5 => "ispie"

1. Lösung:

```
subList :: Int -> Int -> [a] -> [a]                    -- 2 P
subList  n    m  [] = []
subList  0    m  xs = take m xs
subList (n+1) m  (x:xs) = subList n m xs               -- 8 P
```

2. Lösung:

```
subList :: Int -> Int -> [a] -> [a]                    -- 2 P
subList  n    m  [] = []
subList  n    m  (x:xs) = take m (drop n (x:xs))       -- 8 P
```

3. Aufgabe (12 Punkte)

Betrachten Sie folgende rekursive Funktion, die die maximale Anzahl der Teilflächen berechnet, die entstehen können, wenn ein Kreis mit **n** geraden Linien geteilt wird.

```
maxSurfaces :: Int -> Int
maxSurfaces 0 = 1
maxSurfaces n = maxSurfaces (n - 1) + n
```

a) Definieren Sie eine Funktion, die mit Hilfe einer endrekursiven Funktion, genau die gleiche Berechnung realisiert.

b) Welche Vorteile hat die endrekursive Funktion gegenüber der nicht endrekursiven Lösung?

a) Lösung: **- - 8 P**

```
end_maxSurfs :: Int -> Int
end_maxSurfs n = end_maxSurfs' 0 n
                 where
                 end_maxSurfs' acc 0 = acc + 1
                 end_maxSurfs' acc n = end_maxSurfs' (acc + n) (n-1)
```

b) Lösung: **- - 4 P**

Die Ausdrücke werden nicht großer, weil die Argumente reduziert werden bevor die Berechnung wieder in die Rekursion hinein geht (Pattern-Matching). Dadurch wird zwischendurch weniger Speicherplatz verbraucht.
In beide Funktionen ist die Komplexität linear O(n), weil nur n Rekursive Aufrufe und 2*n Additionen/Subtraktionen stattfinden. Aber endrekursive Definitionen können im Haskell optimiert werden, indem die Kette der rekursiven Aufrufen mit einer while-Schleife ersetzt wird.

```
maxSurfaces 4 => (maxSurfaces 3) + 4
              => ((maxSurfaces 2) + 3) + 4
              => (((maxSurfaces 1)+ 2) + 3) + 4
              => ((((maxSurfaces 0) + 1)+ 2) + 3) + 4
              => (1 + 1)+ 2) + 3) + 4
              => ....

end_maxSurfs 5  => end_maxSurfs' 0 4
                => end_maxSurfs' 4 3
                => end_maxSurfs' 7 2
                => end_maxSurfs' 9 1
                => end_maxSurfs' 10 0
                => 11
```

4. Aufgabe (10 Punkte)

Definieren Sie eine rekursive, polymorphe Funktion **mapUntil**, die als Argumente eine Funktion **f** (f :: a -> b), eine Prädikat-Funktion **p** (p :: a->Bool) und eine Liste bekommt und solange die Elemente der Liste das Prädikat **nicht** erfüllen, die Funktion **f** auf die Elemente der Liste anwendet und diese in der Ergebnisliste einfügt.

Anwendungsbeispiel: **mapUntil** (*3) (>5) [1,5,5,7,1,5] => [3,15,15]

1. Lösung:

```
mapUntil :: (a -> b) -> (a -> Bool) -> [a] -> [b]          - - 2 P
mapUntil f p [] = []                                       - - 1 P
mapUntil f p (x:xs) | not (p x) = f x : mapUntil f p xs    - - 7 P
                    | otherwise = []
```

2. Lösung:

```
mapUntil :: (a -> b) -> (a -> Bool) -> [a] -> [b]          - - 2 P
mapUntil f p xs = map f (takeWhile (not.p) xs)             - - 8 P
```

5. Aufgabe (8 Punkte)

Definieren Sie eine Funktion **sumPowerTwo**, die die Summe der Quadrate aller Zahlen zwischen **1** und **n** unter Verwendung der **foldl** und **map**-Funktionen berechnet. Sie dürfen in Ihrer Definition keine Listengeneratoren verwenden.

1. Lösung

```
sumPowerTwo :: Int -> Int                                          -- 1 P
sumPowerTwo n = foldl (+) 0 (map (^2) (take n (iterate (+1) 1)))   -- 7 P
```

2. Lösung

```
sumPowerTwo :: Integer -> Integer                                  -- 1 P
sumPowerTwo n = foldl (+) 0 (map (^2) [1..n])                      -- 7 P
```

3. Lösung

```
sumPowerTwo :: Integer -> Integer
sumPowerTwo n = foldl (+) 0 (map g [1..n] )
                    where
                    g x = x*x
```

4. Lösung

```
sumPowerTwo :: Integer -> Integer                                  -- 1 P
sumPowerTwo n = foldl (+) 0 (map (\a -> a*a) [1..n])               -- 7 P
```

5. Lösung

```
sumPowerTwo :: Integer -> Integer                                  -- 1 P
sumPowerTwo n = (foldl (+) 0 . map (^2)) [1..n]                    -- 7 P
```

6. Lösung

```
sumPowerTwo :: Integer -> Integer                                  -- 1 P
sumPowerTwo n = let ns = [1..n] in (foldl (+) 0 . map (^2)) ns     -- 7 P
```

7. Lösung

```
sumPowerTwo :: Integer -> Integer                                  -- 1 P
sumPowerTwoWhere n = (foldl (+) 0 . quads) n
                         where
                         quads n = map (^2) [1..n]                 -- 7 P
```

6. Aufgabe (14 Punkte)

Ein Element einer Liste von ***n*** Objekten stellt die absolute Mehrheit der Liste dar, wenn das Element mindestens $\left(\frac{n}{2}+1\right)$ mal in der Liste vorkommt.

Definieren Sie eine **majority** Funktion, die mit **linearem** Aufwand das Majority-Element der Liste findet, wenn eines existiert oder sonst **Nothing** zurückgibt.

Die Signatur der Funktion soll wie folgt aussehen:

majority :: (Eq a) => [a] -> **Maybe** a

Begründen Sie Ihre Antwort bezüglich der Komplexität.

O(n) Lösung:

```
majority :: (Eq a) => [a] -> Maybe a
majority []   = Nothing
majority [x]  = Just x
majority xs | (freq l_maj xs) > half = (Just l_maj)
            | otherwise = Nothing
              where
              (c, l_maj) = local_maj (1, head xs) (tail xs)
              local_maj (n, m) [] = (n, m)
              local_maj (n, m) (e:es) | e==m = local_maj (n+1,m) es
                                      | (e/=m && n==0) = local_maj (1, e) es
                                      | otherwise = local_maj (n-1,m) es
              half = div (length xs) 2

freq :: Eq a => a -> [a] -> Int
freq elem xs = sum [ 1 | x<-xs, x==elem ]
```

- - 10 P

Begründung:

$T_{majority}(n) = T_{local\_maj}(n) + T_{freq}(n) + T_{half}(n)$

$= \quad c_1*n + c_2*n + c_3$

$= \quad (c_1 + c_2)*n + c_3$

$= \quad O(n)$

- - 4 P

O(n²) Lösungsbeispiele:

```
majority' :: (Eq a) => [a] -> Maybe a
majority'  [] = Nothing
majority'  xs = if mll==[] then Nothing else Just (head mll)
          where
           mll = [e|e<-xs, length [m|m<-xs, m==e]>len]
           len = (div (length xs) 2)
```

- - nur 5 P

```
majority' :: (Eq a) => [a] -> Maybe a
majority' []   = Nothing
majority' [x] = Just x
majority' xs  = maj xs xs
   where
      maj  []    _  = Nothing
      maj (x:xs) ys | (length (filter (==x) ys)) > len = Just x
                    | otherwise = maj xs ys
      len = (div (length xs) 2)
```

- - nur 5 P

Korrekte Komplexitätsanalyse: **- - 4 P**

7. Aufgabe (16 Punkte)

Betrachten Sie folgende algebraische Datentypen und Funktionen:

```
data  B  =  F  | T  deriving Show

data  Nat  =  Zero  | S Nat  deriving Show

data ZInt = Z Nat Nat  deriving Show

succN :: Nat -> Nat
succN n = S n

addN ::  Nat-> Nat-> Nat
addN n Zero = n
addN n (S m) = succN (addN n m)

multN :: Nat -> Nat -> Nat
multN _ Zero  = Zero
multN n (S m) = addN n (multN n m)

foldn :: (Nat -> Nat) -> Nat -> Nat -> Nat
foldn  h  c  Zero  = c
foldn  h  c  (S n) = h (foldn  h  c  n)
```

a) Definieren Sie damit eine **(<)** und eine **(==)** Funktion für den Datentyp **Nat**.

b) Definieren Sie die Potenzfunktion **(^)** für den Datentyp **Nat** (natürliche Zahlen) unter Verwendung der **foldn** Funktion.

c) Definieren Sie die **(/=)** und (Funktion für den Datentyp **ZInt** (ganze Zahlen).

Wenn Sie zusätzliche Funktionen für Ihre Definitionen in a), b) und c) brauchen, müssen Sie diese auch selber definieren.

Lösung a):

```
(<<) :: Nat -> Nat -> B                                   - - 1 P
(<<) Zero (S _)  = T
(<<) (S a) (S b) = (<<) a b
(<<)  _    _     = F                                      - - 3 P

eqN :: Nat -> Nat -> B                                    - - 1 P
eqN Zero  Zero  = T
eqN (S a) (S b) = eqN a b
eqN  _     _    = F                                      - - 3 P
```

Lösung b):

```
powN' :: Nat -> Nat -> Nat   -- the case 0^0 is not defined!   - - 1 P.
powN' b = foldn (multN b) (S Zero)                             - - 3 P.
```

Lösung c):

```
notB :: B -> B
notB T = F
notB F = T                                                - - 1 P

eqZ :: ZInt -> ZInt -> B
eqZ (Z a b) (Z c d) = eqN (addN b c) (addN a d)           - - 1 P

neqZ :: ZInt -> ZInt -> B
neqZ a b = notB (eqZ  a  b)                               - -  2 P
```

2. Lösung c):

```
notB :: B -> B
notB T = F
notB F = T                                                - - 1 P

neqZ :: ZInt -> ZInt -> B
neqZ (Z a b) (Z c d) = notB (eqN (addN b c) (addN a d))   - - 3 P
```

8. Aufgabe (16 Punkte)

Betrachten Sie folgenden algebraischen Datentyp für binäre Suchbäume:

```
data BSearchTree a = Nil | Node a (BSearchTree a) (BSearchTree a)
```

Definieren Sie eine Funktion **insert**, die einen Baum **t** :: BSearchTree a und ein Element **x** :: a, als Argument bekommt und **x** im Baum eingefügt wird.

```
insert :: (Ord a) => a -> BSearchTree a -> BSearchTree a
```

Lösung:

```
insert:: (Ord a) => BSTree a -> a -> BSTree a
insert Nil k  = Node k Nil Nil
insert (Node l  leftTree rightTree ) k
                    |k <= l = (Node l  (insert leftTree k) rightTree)
                    |otherwise = (Node l leftTree (insert rightTree k))
```

Definieren Sie eine Funktion **oneChild**, die entscheidet, ob der Baum Knoten mit nur einem Kind beinhaltet, wobei **Nil** nicht als Kind zählt.

```
oneChild :: (Ord a) => BSearchTree a -> Bool
```

Lösung:

```
oneChild :: (Ord a) => BSTree a -> Bool
oneChild Nil = False
oneChild (Node _ Nil (Node _ lt rt)) = True
oneChild (Node _ (Node _ lt rt) Nil) = True
oneChild (Node _ lt  rt) = oneChild lt || oneChild rt
```

9. Aufgabe (6 Punkte)

Wann können Sie eine einstellige Funktion als **strikt** bezeichnen? Geben Sie die formale Definition, die in der Vorlesung besprochen worden ist, an.

Lösung:

f ist strikt $\Leftrightarrow$ f $\perp = \perp$ **- - 6 P.**

Lösung: Eine Funktion ***f*** *ist* nach einem ihrer Argumente ***a*** strikt, wenn für die Auswertung der Funktion die Auswertung von ***a*** notwendig ist.

- - nur 4 P.