Funktionale Programmierung
WS 2017/2018
Prof. Dr. Margarita Esponda

Zwischenklausur (**Lösungen**)

Name: ……………………………………………….. **Vorname:** ……………….………..…………………

Matrikel-Nr: …………………..…………

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.
5) Keine Hilfsmittel sind erlaubt.
6) Die maximale Punktzahl ist 100.

Viel Erfolg!

Aufgabe	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**Summe**	**Note**
Max. Punkte	5	7	8	10	6	14	16	18	16	100	
Punkte											

1. Aufgabe (5 Punkte)

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

1. Lösung

f ist strikt ⇔ f ⊥=⊥ **- - 5 P.**

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

- - 4 P.

2. Aufgabe (7 Punkte)

Betrachten Sie folgende Definition der **foldl1** Funktion:

```
foldl1:: (a->a->a)->[a]->a
foldl1  f  (x:xs) =  foldl  f  x  xs
```

Reduzieren Sie folgenden Ausdruck schrittweise zur Normalform.

foldl1 ((+).(*2)) [1,0,1,0]

Lösung:

```
foldl1 ((+).(*2)) [1,0,1,0] =>  foldl  ((+).(*2)) 1  [0,1,0]
                            =>  foldl  ((+).(*2)) (((+).(*2)) 1 0) [1,0]
                            =>  foldl  ((+).(*2))  2  [1,0]
                            =>  foldl  ((+).(*2)) (((+).(*2)) 2 1)  [0]
                            =>  foldl  ((+).(*2)) 5 [0]
                            =>  foldl  ((+).(*2)) (((+).(*2)) 5 0) []
                            =>  foldl  ((+).(*2)) 10 []           - - 5 P.
                            =>  10                                - - 2 P.
```

3. Aufgabe (8 Punkte)

Betrachten Sie folgende Definition der **iterate**-Funktion:

```
iterate     :: (a -> a) -> a -> [a]
iterate f x  =  x : iterate f ( f x )
```

Definieren Sie damit Funktionen, die folgende unendlichen Listen darstellen:

[1, -1, 1, -1, 1, ...]

[0, 1, 3, 7, 15, 31, 63, ...]

Lösung:

```
plus_minus :: [Integer]
plus_minus = iterate ((*) (-1)) 1                    - - 4 P.

mersenne_zahlen :: [Integer]
mersenne_zahlen = iterate ((+1).(*2)) 0              - - 4 P.
```

4. Aufgabe (10 Punkte)

Definieren Sie eine rekursive, polymorphe Funktion **applyUntil**, 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 in die Ergebnisliste einfügt.

Anwendungsbeispiel:

applyUntil (^2) (>6) [1,5,5,7,1,5] => [1, 25, 25]

1. Lösung

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

2. Lösung

```
applyUntil2 :: (a -> b) -> (a -> Bool) -> [a] -> [b]         —- 1 P.
applyUntil2 f p xs  = map f (takeWhile (not.p) xs)           —- 4 P.
```

5. Aufgabe (6 Punkte)

Die **freq**-Funktion berechnet, wie oft ein angegebenes Element in einer Liste vorkommt.

Anwendungsbeispiele: **freq** 3 [2, 1, 4, 3, 9, 3, 3, 0] => 3
freq 'd' "abcdea abcda" => 2

Definieren Sie eine polymorphe Funktion der **freq** Funktion, unter sinnvoller Verwendung von Listengeneratoren.

1. Lösung:

```
freq :: (Eq a) => a -> [a] -> Int                    - - 2 P.
freq y xs = sum [ 1 | x <- xs, x==y ]                - - 4 P.
```

2. Lösung:

```
freq :: (Eq a) => a -> [a] -> Int                    - - 2 P.
freq y xs = length [ 1 | x <- xs, x==y ]             - - 4 P.
```

6. Aufgabe (14 Punkte)

a) Definieren Sie eine polymorphe Funktion, die unter Verwendung des **Insertsort**-Algorithmus die Elemente einer Liste sortiert.

b) Analysieren Sie die Komplexität Ihrer Funktion bezüglich der Anzahl der Reduktionen.

1. Lösung für a):

```
isort :: (Ord a) => [a] -> [a]
isort [] = []
isort (a:xs) = ins a (isort xs)
               where
                ins a   []   = [a]
                ins a (b:ys)      |  a<= b     = a:(b:ys)
                                  | otherwise = b: (ins a ys)     - - 8 P.
```

2. Lösung für a):

```
isort :: (Ord a) => [a] -> [a]
isort [] = []

            where
                isort' sorted [] = sorted
                isort' sorted (x:xs) = isort' (ins x sorted) xs
                ins a    []   = [a]
                ins a (b:ys)     | a<= b     = a:(b:ys)
                                 | otherwise = b: (ins a ys)     - - 8 P.
```

b) für die 1. Lösung:

Max. Anzahl Reds.

$$isort\ [x_1,x_2,\ldots,x_n] \Rightarrow ins\ x_1\ (isort\ [x_2,x_3,\ldots,x_n])$$ 1 **ins+isort**

...   **...**

$$\Rightarrow ins\ x_1\ (ins\ x_2\ (ins\ x_3\ (\ldots (ins\ x_n\ [\,])\ldots)$$ 1 **ins+isort**

n-Reds

$$\Rightarrow ins\ x_1\ (ins\ x_2\ (ins\ x_3\ (\ldots (ins\ x_{n-1}\ [x_n])\ldots)$$ 1 **ins**

$$\Rightarrow ins\ x_1\ (ins\ x_2\ (ins\ x_3\ (\ldots (ins\ x_{n-2}\ [x_{n-1},x_n])\ldots)$$ 2 **ins**

...   ...

$$\Rightarrow ins\ x_1\ [x_2,x_3,\ldots,x_{n-1},x_n]$$ n **ins**

T(n) = (n+1)+(1+2+..+n

Insertion-Sort

```
isort [] = []
isort (a:xs) = ins a (isort xs)
               where
                 ins a   []                = [a]
                 ins a (b:ys)  |  a<= b    =  a:(b:ys)
                               | otherwise = b: (ins a ys)
```

Eingabe: Liste mit **n** Zahlen

$$\overbrace{\boxed{3}\ \boxed{7}\ \boxed{9}\ \boxed{2}\ \boxed{0}\ \boxed{1}\ \boxed{4}\ \boxed{5}\ \boxed{6}\ \boxed{0}\ \boxed{8}\ \boxed{2}\ \boxed{3}\ \boxed{7}\ \boxed{5}}^{\mathbf{n}}$$

Berechnungsschritte: Reduktionen

Im schlimmsten Fall: T(n) = (n+1)+(1+2+3+...+n

$$= \frac{(n+1)(n+2)}{2}$$

$$= n^2 + 3n + 2$$

Zeitkomplexität: = **O(n²)**

- - 6 P.

7. Aufgabe (16 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 Zeitkomplexität.

Lösung:

```
majority :: (Eq a) => [a] -> Maybe a
majority []  = Nothing                                                       - - 1 P.
majority [x] = Just x                                                        - - 1 P.
majority xs  = if ((freq mm xs)*2) > (length xs) then (Just mm) else Nothing
           where
            (c, mm) = aux (1, head xs) (tail xs)
            aux (n, mm) []  = (n, mm)
            aux (n, mm) (e:es) | e==mm = aux (n+1,mm) es
                               | (e/=mm && n==0) = aux (1, e) es
                               | otherwise = aux (n-1,mm) es               - - 10 P.
```

Eingabegröße ist n = die Länge der Eingabeliste

Komplexität:

Für die Auswertung des Ausdrucks innerhalb der **if-then-else** Anweisung muss der Ausdruck **((freq mm xs)*2) > (length xs)** einmal berechnet werden.

- Für die Berechnung von **mm** wird die Funktion **aux** n-Mal rekursiv aufgerufen.
- der Ausdruck **length xs** hat wiederum einen linearen Aufwand.
- zum Schluss hat die **freq** Funktion auch einen linearen Aufwand.

$$T(n) = c_1 + T_{freq}(n) + T_{length}(n) + T_{aux}(n)$$

$$= c_1 + O(n) + O(n) + O(n)$$

$$= O(n)$$ **- - 4 P.**

8. Aufgabe (18 Punkte)

Betrachten Sie folgende algebraische Datentypen für Wahrheitswerte, natürliche Zahlen und ganze Zahlen mit entsprechenden Funktionsdefinitionen:

```
data  B  =  T  | F  deriving Show
data  Nat  =  Zero  | S Nat  deriving Show
data ZInt = Z Nat Nat  deriving Show

succN ::  Nat -> Nat
succN a = S a

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

a) Definieren Sie für Ihren Wahrheitstyp **B** eine **xorB** Funktion (exklusives Oder).

b) Definieren Sie eine **smaller** (<) und eine **unequal** (/=) Funktion für den Datentyp **Nat**.

c) Definieren Sie die **addN** (Addition), **multN** (Multiplikation) und **powN** (Potenzfunktion) für den Datentyp **Nat** nur unter Verwendung der **foldn** Funktion.

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

Sie dürfen in dieser Aufgabe keine vordefinierten Funktionen oder Datentypen von Haskell verwenden.

Lösung:

```
xorB :: B -> B -> B
xorB T F = T
xorB F T = T
xorB _ _ = F                                -- 2 P.

smaller :: Nat -> Nat -> B
smaller Zero (S a) = T
smaller (S a) (S b) = smaller a b
smaller _ _ = F                             -- 2 P.

unequal :: Nat -> Nat -> B
unequal Zero Zero = F
unequal (S a) (S b) = unequal a b
unequal _ _ = T                             -- 2 P.

addN :: Nat -> Nat -> Nat
addN = foldn succN                          -- 2 P.

-- Oder:
addN = foldn S                              -- 2 P.
```

```
multN :: Nat -> Nat -> Nat
multN m n = foldn (addN m) Zero n                    - - 2 P.

powN :: Nat -> Nat -> Nat
powN Zero Zero = error "not defined"
powN m n = foldn (multN m) (S Zero) n                - - 3 P.

notB :: B -> B
notB T = F
notB F = T                                           - - 2 P.

equalZ :: ZInt -> ZInt -> B
equalZ (Z a b) (Z c d) = notB (unequal (addN b c) (addN a d))  - - 3 P.
```

9. Aufgabe (16 Punkte)

Betrachten Sie folgenden algebraischen Datentyp für einfache Bäume aus der Vorlesung:

```
data SimpleBT = L | N SimpleBT SimpleBT
```

a) (6 P.) Definieren Sie eine Funktion **nodes** und eine Funktion **pfad**, die entsprechend die Anzahl der Knoten und die Pfadlänge einer SimpleBT-Baumstruktur berechnet. Verwenden Sie dafür jeweils folgende zwei Formeln:

Anzahl der Knoten: $|t| = |t_l| + |t_r| + 1$

Pfadlänge: $\pi(t) = \pi(t_l) + \pi(t_r) + |t| - 1$

b) (10 P.) Definieren Sie eine Funktion **balance**, die überprüft, ob ein **SimpleBT**-Baum überall balanciert ist. Wenn Sie zusätzliche Funktionen dafür brauchen, müssen Sie diese auch definieren.

Lösung:

```
nodes :: SimpleBT -> Integer
nodes L = 1
nodes (N leftT rightT) = 1 + nodes leftT + nodes rightT        - - 3 P.

pfad :: SimpleBT -> Integer
pfad L = 0
pfad (N lt rt) = (pfad rt) + (pfad lt) + (nodes (N lt rt)) - 1    - - 3 P.

height :: SimpleBT -> Integer
height L = 0
height (N lt rt) = (max (height lt) (height rt)) + 1

balanced :: SimpleBT -> Bool
balanced  L = True
balanced  (N lt rt) = (balanced lt) && (balanced rt) && height lt == height rt
                                                         - - 5 P. + 5 P.
```

-- 2. Lösung:

```
size :: SimpleBT -> Integer
size L = 1
size (N lt rt) = size lt + size rt + 1

balance2 L = True
balance2 (N lt rt) = (balance2 lt) && (balance2 rt) && size lt == size rt
                                                        - - 5 P. + 5 P.
```

-- 3. Lösung:

```
balance3 tree = (size tree) == (2^((height tree)+1)-1)        - - 6 P. + 2 P. + 2 P.
```

-- 4. Lösung:

```
balance4 tree = fst (balance tree)
                where
                 balance L = (True, 1)
                 balance (N lt rt) = (bLeft && bRight && tLeft==tRight, 1 + tRight)
                                     where
                                     (bLeft,  tLeft)  = balance lt
                                     (bRight, tRight) = balance rt        - - 10 P.
```