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

Klausur

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 Klausur 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	10	9	7	12	12	8	12	16	14	100	
Punkte											

1. Aufgabe (10 Punkte)

a) (6 P.) Welche Auswertungsstrategie wird in der Haskell-Programmiersprache verwendet? Erläutern Sie die Auswertungsstrategie anhand eines Beispiels. Welche Vorteile hat eine funktionale Programmiersprache damit?

b) (4 P.) Was ist ein **statisches Typsystem** im Kontext von Programmiersprachen? Welche Vorteile hat eine Programmiersprache damit?

Lösung:

a) Nach Bedarf (Lazy-evaluation) -- **1P**.

Lazy-evaluation ist eine optimierte Auswertungsvariante von call-by-name und wird in Haskell und anderen funktionalen Sprachen verwendet.

Beispiel:

```
g x = 2 * x
f x = x * x

f (g 5)  ⇒  (g 5) * (g 5)  where g 5 = 2*5
         ⇒  (g 5) * (g 5)  where g 5 = 10
         ⇒  10 * 10
         ⇒  100
```

call-by-need ist eine Art call-by-name-Auswertungsstrategie mit Gedäch

-- **3P**.

Vorteile:

- Wenn eine Normalform existiert, wird diese erreicht. -- **2P**.

b) Der Datentyp der Funktionen wird statisch während der Übersetzungszeit des Programms abgeleitet. -- **2P**.

Vorteile:

- Datentyp-Fehler werden früher erkannt
- durch die Reduzierung der Typ-Überprüfung reduziert sich die Ausführungszeit
- Typ-Ableitung (Typ-Inferenz) ist möglich

-- **2P**.

2. Aufgabe (9 Punkte)

Betrachten Sie folgende Funktionsdefinition:

```
com :: (a -> a) -> (a -> a) -> a -> (a, a)
com f g x = ((f · g) x, (g · f) x)
```

Reduzieren Sie folgenden Ausdruck schrittweise zur Normalform:

a) com (* 10) (mod 10) 7

Lösung:

com (* 10) (mod 10) 7

=> ((* 10) · (mod 10)) 7, ((mod 10) · (* 10)) 7

=> ((* 10) ((mod 10) 7), ((mod 10) ((* 10) 7)

=> ((* 10) (3), ((mod 10) 70) **- - 4 P.**

=> (30, 10) **- - 1 P.**

b) foldl (\ys x-> x:ys) [] (take 4 [1..])

Lösung:

=> (foldl (\ys x-> x:ys) [] [1,2,3, 4]

=> (foldl (\ys x-> x:ys) 1:[] [2,3, 4]

=> (foldl (\ys x-> x:ys) 2:[1] [3, 4]

=> (foldl (\ys x-> x:ys) 3:[2,1] [4]

=> (foldl (\ys x-> x:ys) 4:[3,2,1] []

=> (foldl (\ys x-> x:ys) [4,3,2,1] [] **- - 3 P.**

=> [4, 3,2,1] **- - 1 P.**

3. Aufgabe (7 Punkte)
Schreiben Sie eine Funktion, die mit Hilfe einer endrekursiven Funktion die Fibonacci-Zahlen mit linearem Aufwand berechnet.

Lösung 1):

```
fib: Integer -> Integer                                  - - 1 P.
fib n = quickFib  0  1  n                                - - 1 P.
        where
              quickFib a  b  0  = a                      - - 1 P.
              quickFib a  b  n  = quickFib b (a+b) (n-1) - - 4 P.
```

Lösung 2):

```
nextFib :: (Integer, Integer) -> (Integer, Integer)      - - 1 P.
nextFib  (a,b)  =  (b, a+b)                              - - 1 P.

fibTuple n | n==0         =  (0, 1)                      - - 1 P.
           | otherwise    =  nextFib (fibTuple (n-1))    - - 3 P.

fib n = fst ( fibTuple n )                               - - 1 P.
```

4. Aufgabe (12 Punkte)

Betrachten Sie folgendes Anwendungsbeispiel der ***currifizierten*** **zipWith** Standardfunktion von Haskell:

zipWith (^) [1, 2, 3] [2, 3, 4] => [1, 8, 81]

a) (6 P.) Definieren Sie eine polimorphe ***uncurrifizierte*** Version der **zipWith** Funktion.

b) (6 P.) Schreiben Sie eine sinnvolle Definition der **zipWith** Funktion mit Hilfe von Listengeneratoren. Sie dürfen dabei andere Standardfunktionen von Haskell verwenden.

Lösungen a), b):

```
zipWith_b :: ((a -> b -> c), [a], [b]) -> [c]            - - 1,5 P.
zipWith_b (f, [], _) = []
zipWith_b (f, _, []) = []                                - - 1,5 P.
zipWith_b (f, (x:xs), (y:ys)) = (f x y): zipWith_b (f, xs, ys)   - - 3 P.


zipWith_a ::(a -> b -> c) -> [a] -> [b] -> [c]           - - 1 P.
zipWith_a  f  xs  ys = [ f a b | (a,b) <- zip xs ys ]    - - 5 P.
```

5. Aufgabe (12 Punkte)

Betrachten Sie folgende algebraische Datentypen und Funktionen:

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

add ::  Nat-> Nat-> Nat
add a  Zero = a
add a  (S b) = add (S a) b

mult :: Nat -> Nat -> Nat
mult  _ Zero  = Zero
mult  a (S b) = add a (mult a b)

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 die **xor**, **smaller** (<) und **equal** (==) Funktionen für den Datentyp **B** und **Nat**.

Lösung:

```
xor :: B -> B -> B
xor T  T  =  F
xor F  F  =  F
xor T  F  = T
xor F  T  = F
```
- - 2 P.

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

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

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

Lösung:

```
power :: Nat -> Nat -> Nat
power Zero Zero = error  "undefined"
power m = foldn (mult m) (S Zero)
```
- - 4 P.

6. Aufgabe (8 Punkte)

Betrachten Sie folgenden algebraischen Datentyp für einfache binäre Bäume:

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

Definieren Sie damit folgende Funktionen:

```
height :: SimpleBT -> Integer      - - berechnet die Höhe des Baumes
balanced :: SimpleBT -> Bool       - - entscheidet, ob der Baum vollständig
                                   - - bzw. balanciert ist oder nicht.
```

Wenn Sie dafür zusätzliche Funktionen brauchen, müssen Sie diese auch definieren.

Lösung:

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

- - 4 P.

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

- - 4 P.

2. Lösung:

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

balanced :: SimpleBT -> Bool
balanced L = True
balanced (N lt rt) = (balanced lt) && (balanced rt) && size lt == size rt
```

3. Lösung:

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

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

balanced :: SimpleBT -> Bool
balanced  tree  =  (size tree) == (2^((height tree)+1)-1)
```

4. Lösung:

```
balanced :: SimpleBT -> Bool
balanced tree = fst (balanced' tree)
        where
        balanced' L = (True, 1)
        balanced' (N lt rt) = (bLeft && bRight && dLeft == dRight, 1+dRight)
                where
                (bLeft,  dLeft  ) = balanced' lt
                (bRight, dRight) = balanced' rt
```

7. Aufgabe (12 Punkte)

Betrachten Sie folgenden algebraischen Datentyp mit entsprechenden Funktionsdefinitionen:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

size :: Tree a -> Int
size (Leaf _ ) = 1                          ..... size.1
size (Node lt rt) = size rt + size lt       ..... size.2

depth :: Tree a -> Int
depth (Leaf _ ) = 1                         ..... depth.1
depth (Node lt rt) = max (depth lt) (depth rt) + 1 ..... depth.2
```

Beweisen Sie mit Hilfe von struktureller Induktion, dass folgende Eigenschaft gilt:

$$\text{size t} \leq 2^{\text{depth } t}$$

Lösung:

Induktionsanfang: t = (Leaf _)

size (Leaf _) $\overset{?}{<=}$ $2^{\text{depth (Leaf \_)}}$

size (Leaf _) $=_{\text{size.1}}$ 1

$<=_{\text{algebra}}$ 2^1

$=_{\text{depth.1}}$ $2^{\text{depth (Leaf \_)}}$ **- - 3 P.**

Induktionsvoraussetzung: für t = Node lt rt gilt

I.V. size lt <= $2^{\text{depth lt}}$

size rt <= $2^{\text{depth rt}}$ **- - 1 P.**

Induktionsschritt: t = Node lt rt

size (Node lt rt) $\overset{?}{<=}$ $2^{\text{depth (Node lt rt)}}$

size (Node lt rt) $=_{\text{size.2}}$ size lt + size rt

$<=_{\text{I.V.}}$ $2^{\text{depth lt}} + 2^{\text{depth rt}}$

$<=_{\text{algebra}}$ $2*2^{\text{max(depth lt, depth rt)}}$

$=_{\text{algebra}}$ $2^{\text{max(depth lt, depth rt) + 1}}$

$=_{\text{depth.2}}$ $2^{\text{depth (Node lt rt)}}$

damit ist size (Node lt rt) <= $2^{\text{depth (Node lt rt)}}$ bewiesen.

+Die Behauptung gilt dann für alle **t** endlichen Bäume. **- - 8 P.**

8. Aufgabe *(16 Punkte)*

a) Reduzieren Sie den folgenden Lambda-Ausdruck zur Normalform:

$$(\lambda xy.\, xy(\lambda xy.\, \mathrm{y}))(\lambda xy.\, x)(\lambda xy.\, \mathrm{y})$$

Lösung:

$$\begin{aligned}(\lambda xy.\, xy(\lambda xy.\, \mathrm{y}))(\lambda xy.\, x)(\lambda xy.\, \mathrm{y}) &\Rightarrow_\beta (\lambda y.\, (\lambda xy.\, \mathrm{x})\ y\ (\lambda xy.\, y))(\lambda xy.\, \mathrm{y}) \\ &\Rightarrow_\beta (\lambda xy.\, \mathrm{x})(\lambda xy.\, \mathrm{y})(\lambda xy.\, y) \\ &\Rightarrow_\beta (\lambda y.\, (\lambda xy.\, \mathrm{y}))(\lambda xy.\, y) \\ &\Rightarrow_\beta (\lambda xy.\, \mathrm{y}) \\ &\equiv F\end{aligned}$$

- - **(5 P.)**

b) Schreiben Sie einen Lambda-Ausdruck, der die Subtraktion von zwei **natürlichen** Zahlen berechnet. Sie können dabei die Funktion **P** (Vorgänger) als gegeben verwenden.

Lösung:

$$(\lambda nm.\, \mathrm{m}(P(n))$$

- - (1 P.)

c) Definieren Sie einen Lambda-Ausdruck, der die **GGT**-Funktion von Euclid (größter gemeinsamer Teiler von zwei positiven natürlichen Zahlen) berechnet.

```
ggt p q | p==q = p
        | p>q  = ggt (p-q) q
        | p<q  = ggt p (q-p)
```

Sie können dabei die Funktionen **CSUB** (Subtraktion für natürliche Zahlen), **Y** (Fixpunkt-Operator) und die Vergleichsoperationen **<**, **>**, **=** als gegeben verwenden.

Lösung:

$$GGT \equiv Y(\lambda rpq.= pqp(> pq(r(\{CSUB\}pq)q)(rp(\{CSUB\}qp))))$$

- - (5 *P.*)

c) Zeigen Sie, dass folgende Lambda- und Kombinatoren-Ausdrücke äquivalent sind.

$$\lambda x.\lambda y.(xx) \quad \equiv \quad S(KK)(SII)$$

1. Lösung:

$$\begin{aligned}
T\left[\lambda x.\lambda y.(xx)\right] &\Rightarrow_{7} T\left[\lambda x.\, T\left[\lambda y.(xx)\right]\right] \\
&\Rightarrow_{4} T\left[\lambda x.\, K\ T\left[xx\right]\right] \\
&\Rightarrow_{2,8} T\left[\lambda x.\, K\ (xx)\right] \\
&\Rightarrow_{6} S\ T\left[(\lambda x.\, K)\right]\ T\left[(\lambda x.\, xx)\right] \\
&\Rightarrow_{4} S\ (KK)\ T\left[(\lambda x.\, xx)\right] \\
&\Rightarrow_{6} S\ (KK)\ (S\ T\left[(\lambda x.x)\right]\ T\left[(\lambda x.x)\right]) \\
&\Rightarrow_{2,2} S\ (K\ K)\ (S\ I\ I)
\end{aligned}$$

- - (5 P.

2. Lösung:

$$(\lambda x.\lambda y.(xx))\ a\ b \ \Rightarrow\ (\lambda y.(aa))\ b \ \Rightarrow\ a\ a$$

$$\begin{aligned}
S(KK)(SII)\ a\ b \ &\Rightarrow\ (K\ K)\ a\ ((SI\ I)\ a)\ b \\
&\Rightarrow\ K\ ((SI\ I)\ a)\ b \\
&\Rightarrow\ (S\ I\ I)\ a \\
&\Rightarrow I\ a\ (I\ a) \\
&\Rightarrow\ a\ a
\end{aligned}$$

oder **- - (5 P.)**

9. Aufgabe (14 Punkte)

a) Zeigen Sie, dass die Funktion **half**, die eine natürliche Zahl durch zwei teilt (ganzzahlige Division) primitiv rekursiv ist. Das bedeutet, wenn Sie für die Definition andere Hilfsfunktionen verwenden, müssen Sie auch zeigen, dass diese Hilfsfunktionen primitiv rekursiv definierbar sind.

b) Definieren Sie zusätzlich Ihre Funktionen unter Verwendung der in der Vorlesung definierten **z**, **s**, **p**, **compose** und **pr** Haskell-Funktionen.

Lösung a): - - (7 P.)

$$isZero\ (\ 0\ ) \quad = \quad C_1^0$$
$$isZero\ (\ S(n)\ ) \ = \ Z\ (isZero(n),\ n\ )$$

$$add\ (\ 0,\ m)\ =\ \pi_1^1(m)$$
$$add\ (S(n),\ m)\ =\ S(\pi_1^3(add(n,m),n,m))$$

$$not\ =\ isZero$$

$$odd\ (\ 0\ ) \quad = C_0^0$$
$$odd\ (\ S(n)\ ) \quad = not(\pi_1^2(\ odd\ (\ n\ ),\ n\ ))$$

$$half\ (\ 0\ ) \quad = \quad C_0^0$$
$$half\ (\ S(n)\ ) \quad = \quad add(\pi_1^2(half(n),n),\ odd(\pi_2^2(half(n),n)))$$

Lösung b): - - (7 P.)

```
isZero :: PRFunction
isZero  = pr isZero (const 1) (const 0)

add :: PRFunction
add  = pr add (p 1) (compose s [(p 1)])

not :: PRFunction
not  = isZero

odd :: PRFunction
odd [n] = pr odd (const 0) (compose not [(p 1)]) [n]

half :: PRFunction
half [n] = pr half (const 0) (compose add [(p 1),(compose odd [(p 2)])]) [n]
```

```
-- Null-Funktion
z  :: [Integer] -> Integer
z xs = 0

-- Nachfolger-Funktion
s  :: [Integer] -> Integer
s [x] = x+1

-- Projektions-Funktionen
p :: Integer -> [Integer] -> Integer
p 1 (a:b) = a
p n (a:b) = p (n-1) b

type  PRFunction  =  ( [Integer] → Integer )
-- Kompositionsschema
compose :: PRFunction -> [PRFunction] -> [Integer] -> Integer
compose  f  gs  xs  =  f  [ g  xs | g <- gs ]

-- Rekursionsschema
pr :: PRFunction -> PRFunction -> PRFunction -> [Integer] -> Integer
pr  rec  g  h  (   0  :xs)  =  g  xs
pr  rec  g  h  ((n+1):xs) =  h ( (rec (n:xs)):n:xs )
```

Transformationsregeln, um Lambda-Terme in SKI-Terme zu verwandeln.

1)	T[x]	=> x	
2)	T[(E_1 E_2)]	=> (T[E_1] T[E_2])	
3)	T[λx.x]	=> I	
4)	T[λx.E]	=> (K T[E])	wenn x∉FV(E)
5)	T[λx.E x]	=> (T[E])	wenn x∉FV(E)
6)	T[λx.(E_1 E_2)]	=> (S T[λx.E_1] T[λx.E_2])	falls x∈FV(E_1) oder x∈FV(E_2)
7)	T[λx.λy.E]	=> T[λx.T[λy.E]]	falls x∈FV(E)

Transformations- bzw. **Eliminierungsregeln**

0) $T[x] \Rightarrow x$
   $T[I] \Rightarrow I$
   $T[K] \Rightarrow K$
   $T[S] \Rightarrow S$

1) $T[E_1\ E_2] \Rightarrow T[E_1]\ T[E_2]$

2) $T[\lambda x.E] \Rightarrow$ elim. x $[E]$

3) elim. x $[x] \Rightarrow I$

4) elim. x $[y] \Rightarrow K\ y$ *wenn* $x \neq y$
   elim. x $[I] \Rightarrow K\ I$
   elim. x $[K] \Rightarrow K\ K$
   elim. x $[S] \Rightarrow K\ S$

5) elim. x $[\lambda y.E] \Rightarrow$ elim. x [elim. y $[E]$]

6) elim. x $[E\ x] \Rightarrow T[E]$ *wenn* $x \notin FV(E)$

7) elim. x $[E_1\ E_2] \Rightarrow S$ (elim. x $[E_1]$) (elim. x $[E_2]$)