Funktionale Programmierung WS 16/17
Prof. Dr. Margarita Esponda

Klausur (Solutions)

Name: ..... **Vorname:**..... **Matrikel-Nr:**.....

Aufgab	**A1**	**A2**	**A3**	**A4**	**A5**	**A6**	**A7**	**A8**	**A9**	**A10**	**Max. Summe**	**Note**
Max. Punkte	9	11	10	6	14	8	12	7	15	8	**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 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.

Viel Erfolg!

1. Aufgabe (9 Punkte)

Betrachten Sie folgende Funktionsdefinitionen:

```
apply :: (a -> b, a -> c) -> a ->  (b, c)
apply (f, g) x = (f x, g x)

cross :: (a -> b, c -> d) -> (a, c) -> (b, d)
cross (f, g) = apply (f . fst, g . snd)
```

Was ist der **Wert** folgender Ausdrücke? Begründen Sie Ihre Antwort oder schreiben Sie mindestens vier Zwischenschritte Ihrer Berechnungen auf.

a) cross ((mod 2), (==1) . (div 9)) (4, 7)

Lösung:

```
cross ( (mod 2), (==1) . (div 9) ) (4, 7)
    => apply ( (mod 2) . fst, ( ((==1) . (div 9)) . snd ) (4,7)
    => ( ((mod 2) . fst) (4,7), ( ((==1) . (div 9)) . snd ) (4,7) )
    => ( (mod 2) (fst (4,7)),  ((==1) . (div 9)) (snd (4,7)) )
    => ( (mod 2) 4,  ((==1) . (div 9)) 7 )
    => ( 2,  ((==1) . (div 9)) 7 )
    => ( 2,  (==1) ((div 9) 7) )
    => ( 2,  (==1) 1 )                                   -- 3 P.
    => (2, True)                                         -- 2 P.
```

b) foldl (\ys x-> x:ys) [] (take 3 [1..])

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

2. Aufgabe (11 Punkte)

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

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

a) Definieren Sie eine ***uncurrifizierte*** Version der *zipWith* Funktion.

b) 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]                    -- 2 P.
zipWith_b (f, [], _) = []                                        -- 1 P.
zipWith_b (f, _, []) = []                                        -- 1 P.
zipWith_b (f, (x:xs), (y:ys)) = (f x y): zipWith_b (f, xs, ys)   -- 1 P.

zipWith_a ::(a -> b -> c) -> [a] -> [b] -> [c]                   -- 2 P.
zipWith_a  f  xs  ys = [ f a b | (a,b) <- zip xs ys ]            -- 4 P.
```

3. Aufgabe (10 Punkte)

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

Anwendungsbeispiele: **freq** 3 [2, 1, 4, 8, 9, 3, 3, 0] => 2
freq 'a' "abcdea abda" => 4

Schreiben Sie eine Definition der **freq**-Funktion, unter sinnvoller Verwendung der **foldl**-Funktion.

1. Lösung

```
freq' :: (Eq a) => a -> [a] -> Int                    -- 2 P.
freq' a  = foldl count_reps 0                         -- 8 P.
           where
             count_reps n y   | y==a      = n+1
                              | otherwise = n
```

2. Lösung

```
freq :: (Eq a) => a -> [a] -> Int                     -- 2 P.
freq a  = foldl (\x y -> if (y==a) then x+1 else x) 0 -- 8 P.
```

4. Aufgabe (6 Punkte)

Betrachten Sie folgendes Datentyp-Synonym mit entsprechenden Funktionsdefinitionen für Zahlenmengen:

```
type Set = [Int]

inSet :: Int -> Set -> Bool
inSet e  []  = False
inSet e (x:xs) | e == x = True
               | otherwise = inSet e xs

(\\) :: Set -> Set -> Set
(\\) setA setB = [x| x<-setA, not (inSet x setB)]
```

b) Analysieren Sie die Komplexität der (\\) Funktion.

Lösung:

Die **(\\)** Funktion ist so definiert, dass jedes Element **x** aus der Menge ***setA*** in der Menge ***setB*** gesucht wird. Im besten Fall hat setA oder setB nur ein Element, weil dann die Komplexität gleich O(n) ist.

Im schlimmsten Fall haben setA und setB die gleiche Anzahl von Elementen und kein Element von ***setA*** befindet sich in ***setB***, weil dann jedes Element x von setA mit allen Elementen der Menge setB verglichen wird.

Wenn n die Anzahl der Elemente in setA und in setB ist bzw. die Länge der Listen ist, ist die Komplexität der (\\) Funktion gleich $O(n^2)$.

$$
\begin{aligned}
T_{(\backslash\backslash)}(n,n) &= T_{(\backslash\backslash)}(n-1,n) + c_1 \cdot T_{(inSet)}(n) \; + \; c_2 \\
&= T_{(\backslash\backslash)}(n-1,n) + O(n) + c_2 && \text{-- 4 p.} \\
&= O(n^2) && \text{-- 2 p.}
\end{aligned}
$$

mit c_1 = Zeitkosten eines rekursiven Aufrufs der inSet Funktion

c_2 = Zeitkosten der not + Generierung eines Elements des Listengenerators

5. Aufgabe (14 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 eine **smaller** (<) und eine **equal** (==) Funktion für den Datentyp **Nat**.

Lösung:

```
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.

1. Lösung:

```
power :: Nat -> Nat -> Nat
power m = foldn (mult m) (S Zero)
```
(hier ist 0^0=1 und wird als Lösung akzeptiert)
- - 4 P.

2. Lösung:

```
power :: Nat -> Nat -> Nat
power Zero Zero = error "... not defined"
power m n = foldn (mult m) (S Zero) n
```
- - 4 P.

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

Lösung:

```
(<<) :: ZInt -> ZInt -> B
(<<)  (Z a b) (Z c d) = smaller (add b c) (add a d)
```

- - 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.

7. Aufgabe (12 *Punkte*)

Betrachten Sie folgende Funktionsdefinitionen:

```
length :: [a] -> Int
length [] = 0                          . . . . . length.1
length (x:xs) = 1 + length xs          . . . . . length.2

replicate :: Int -> a -> [a]
replicate 0 _ = []                     . . . . . replicate.1
replicate (n+1) x = x: replicate n x   . . . . . replicate.2
```

Beweisen Sie mittels vollständiger Induktion über n folgende Eigenschaft:

length (replicate n x) = n

Induktionsanfang: n = 0

length (replicate 0 x) $\stackrel{?}{=}$ 0

length (replicate 0 x) = length []
replicate.1
= 0
length.0

- - 3 P.

I.V. für n=k gilt length (replicate k x) = n **- - 3 P.**

Induktionsschritt: n = k+1

length (replicate (k+1) x) $\stackrel{?}{=}$ (k+1)

length (replicate (k+1) x) = length (x: replicate k x)
replicate.2
= 1 + length (replicate k x)
length.2
= 1 + k
I.V.
= k + 1

$\Rightarrow$ length (replicate n x) = n $\forall\, n \in \mathbb{N}$ **- - 6 P.**

8. Aufgabe *(7 Punkte)*

Definieren Sie eine **λ**-Funktion **IN**, die überprüft, ob ein Element in einer Liste vorhanden ist.

Sie können dabei die Funktionen **NIL** (Testet, ob die Liste leer ist), **=** (Testet nach Gleichheit), **HEAD**, **TAIL,** und **Y** (Fixpunkt-Operator) als gegeben verwenden.

Die Funktionen für die Wahrheitswerte **F** und **T** müssen Sie selber definieren

Lösung:

$$T \equiv \lambda x.\lambda y.x$$
$$F \equiv \lambda x.\lambda y.y$$

$$Y(\lambda rx\ell.\{NIL\}\ell F(= x(\{HEAD\}\ell)T(rx(\{TAIL\}\ell))))$$

T -- 1 P.
F -- 1 P.
IN -- 5 P.

9. Aufgabe (15 Punkte)

a) *(2 P.)* Was ist ein Kombinator?

b) *(3 P.)* Schreiben Sie die **I**, **K** und **S** - Kombinatoren als Lambda-Ausdrücke.

c) *(4 P.)* Reduzieren Sie folgenden Kombinatoren-Ausdruck zur Normalform. Begründen Sie die einzelnen Schritte.

$$SI(KIS)(SKI)$$

c) *(6 P.)* Zeigen Sie, dass folgende Lambda- und Kombinatoren-Ausdrücke äquivalent sind.

$$\lambda x.\lambda y.\, xy \;\equiv\; I$$

Verwenden Sie dafür die Transformations oder Eliminierungs-Regeln, die am Ende der Klausur vorgegeben sind.

Lösung a): **- - 2 p.**

Ein Kombinator ist eine primitive Funktion (Lambda-Ausdruck) ohne freie Variablen.

Lösung b): **- - 3 p.**

$I \equiv \lambda x.x$
$K \equiv \lambda x.\lambda y.x$ oder $\lambda xy.x$
$S \equiv \lambda x.\lambda y.\lambda a.xa(ya)$ oder $\lambda xya.xa(ya)$

Lösung c): **- - 4 p.**

$SI(KIS)(SKI) \Rightarrow_S I(SKI)\ ((KIS)(SKI))$
$\Rightarrow_I (SKI)\ ((KIS)(SKI))$
$\Rightarrow_k (SKI)\ (I)(SKI)$
$\Rightarrow_I (SKI)\ (SKI)$
$\Rightarrow_S K(SKI)(I(SKI))$
$\Rightarrow_K (SKI)$

Lösung d): **- - 6 p.**

$$T[\lambda x.\lambda y.\, xy] \Rightarrow_{7)} T[\lambda x.T[\lambda y.\, xy]]$$
$$\Rightarrow_{5)} T[\lambda x.\, T[x]]$$
$$\Rightarrow_{1)} T[\lambda x.\, x]$$
$$\Rightarrow_{3)} I$$

oder mit der Eliminierung-Regeln

$T[\lambda x.\lambda y.x(y)]$ $\Rightarrow_{2)}$ elim. x $[\lambda y.x(y)]$
$\Rightarrow_{5)}$ elim. x [elim. y $[x(y)]]$
$\Rightarrow_{6)}$ elim. x $[T[x]]$
$\Rightarrow_{0)}$ elim. x $[x]$
$\Rightarrow_{3)} I$

10. Aufgabe (8 Punkte)

a) Zeigen Sie, dass die Funktion **gerade**, die entscheiden kann, ob eine natürliche Zahl gerade ist oder nicht, primitiv rekursiv ist. Das bedeutet, wenn Sie für die Definition andere Hilfsfunktionen verwenden, müssen Sie auch zeigen, dass diese 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):

--2 P.

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

$$not\ \ =\ \ isZero$$ **--1 P.**

--2 P.

$$gerade\ (\ 0\ )\ \ =\ C_1^0$$
$$gerade\ (\ S(n)\ )\ \ =\ \ not\ (\ \pi_1^2\ (\ gerade\ (\ n\ ),\ n\ )\ )$$

Lösung b):

```
isZero :: PRFunction                                   --2 P.
isZero  = pr isZero (const 1) (const 0)

nott :: PRFunction
nott  = isZero

gerade :: PRFunction                                   --2 P.
gerade = pr gerade (const 1) (compose nott [(p 1)])
```

```
-- 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)	$T[x]$	$\Rightarrow x$	
2)	$T[(E_1\ E_2)]$	$\Rightarrow (T[E_1]\ T[E_2])$	
3)	$T[\lambda x.x]$	$\Rightarrow I$	
4)	$T[\lambda x.E]$	$\Rightarrow (K\ T[E])$	wenn $x \notin FV(E)$
5)	$T[\lambda x.E\ x]$	$\Rightarrow (T[E])$	wenn $x \notin FV(E)$
6)	$T[\lambda x.(E_1\ E_2)]$	$\Rightarrow (S\ T[\lambda x.E_1]\ T[\lambda x.E_2])$	falls $x \in FV(E_1)$ oder $x \in FV(E_2)$
7)	$T[\lambda x.\lambda y.E]$	$\Rightarrow T[\lambda x.T[\lambda y.E]]$	falls $x \in FV(E)$

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 $[E_2]$)