

Funktionale Programmierung WS 15/16  
Prof. Dr. Margarita Esponda

## Klausur (Lösungen)

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

Aufgabe	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	Max. Summe	Note
Max. Punkte	8	8	8	10	8	12	14	8	12	12	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.

## Viel Erfolg!

### 1. Aufgabe (8 Punkte)

- a) (2 P.) Was verstehen Sie unter statischer Typisierung im Kontext von Programmiersprachen?

Der Datentyp der Funktionen wird statisch während der Übersetzungszeit des Programms abgeleitet.

- b) (2 P.) Nennen Sie die zwei wichtigsten Vorteile von statischen Typ-Systemen.

- Datentyp-Fehler werden früher erkannt und nicht erst zur Laufzeit.
- durch die Reduzierung der Typ-Überprüfung reduziert sich die Ausführungszeit.
- Typ-Ableitung kann gemacht werden. (Typ-Inferenz).

- c) (4 P.) Erläutern Sie die Auswertungsstrategie nach Bedarf (Lazy-Evaluation). Welche Vorteile hat eine Programmiersprache, die diese Auswertungsstrategie verwendet?

Lazy-evaluation ist eine optimierte Auswertungsvariante von call-by-name.

In der call-by-name Auswertungsstrategie werden Ausdrücke von außen nach innen ausgewertet. D.h. Argumente werden nur bei Bedarf reduziert.

Der wichtigste Vorteil der Lazy-Evaluation ist, dass, wenn ein Ausdruck eine Normalform besitzt, diese immer erreicht wird.

Diese Art der Auswertung ist effizienter und terminiert immer, wenn eine Normalform existiert.

**2. Aufgabe** (8 Punkte)

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

- a) `length [(x,y) | x<-[1..5], y<-[5,4..1], x>y]`  
 b) `((foldr (+) 5).(map (\x -> div x 3))) [1..8]`

a)

```
=> length [(x,y) | x<- [1,2,3,4,5], y<-[5,4,3,2,1], x>y]
=> length [(2,1), (3,2), (3,1), (4,3), (4,2), (4,1), (5,4), (5,3), (5,2), (5,1)]
=> 10
```

b)

```
=> ((foldr (+) 5).(map (\x -> div x 3))) [1,2,3,4,5,6,7,8]
=> (foldr (+) 5) ((map (\x -> div x 3)) [1,2,3,4,5,6,7,8])
=> (foldr (+) 5) [0, 0, 1, 1, 1, 2, 2, 2]
=> 14
```

**3. Aufgabe** (8 Punkte)

Betrachten Sie folgende Funktionsdefinition, die eine beliebig lange Liste von Primzahlen berechnet:

```
primes = sieb [2..]
      where
        sieb (p:xs) = p:sieb[k | k<-xs, (mod k p)>0]
```

Programmieren Sie damit eine **primeOne** Funktion, die die ersten **n** Primzahlen berechnet, die mit der Ziffer **1** enden.

Anwendungsbeispiel: **primeOne** 6 => [11, 31, 41, 61, 71, 101]

Lösung:

```
primeOne :: Int -> [Integer]
primeOne n = take n [x|x<-primes, (mod x 10)==1]
```

**4. Aufgabe** (10 Punkte)

Betrachten Sie folgende **max1** und **max2** Funktionsdefinitionen, die das größte Element einer Liste berechnen.

```

max1 [x] = x
max1 (x:xs) | all (<=x) xs = x
               | otherwise = max1 xs

all p xs = foldl (&&) True (map p xs)

max2 [x] = x
max2 (a:b:xs) = aux a (b:xs)
  where
    aux a [] = a
    aux a (b:xs) | a>b = aux a xs
                  | otherwise = aux b xs

```

Analysieren Sie die Komplexität der **max1** und **max2** Funktionen.

<pre> <b>max1</b> [x] = x <b>max1</b> (x:xs)   all (&lt;=x) xs = x                  otherwise = <b>max1</b> xs </pre>	Anzahl der Reduktionen
$\text{max1 } [x_1, x_2, \dots, x_{n-1}, x_n] \Rightarrow \text{all } (\leq x_1) [x_2, \dots, x_{n-1}, x_n]$	(n)
$\Rightarrow \text{all } (\leq x_2) [x_3, \dots, x_{n-1}, x_n]$	(n-1)
$\Rightarrow \text{all } (\leq x_3) [x_4, \dots, x_{n-1}, x_n]$	(n-2)
$\dots$	
$\Rightarrow \text{all } (\leq x_{n-2}) [x_{n-1}, x_n]$	3
$\Rightarrow \text{all } (\leq x_{n-1}) [x_n]$	2
$\text{max1 } [x_n] \Rightarrow x_n$	1
Summe aller Reduktionen:	$(n) \cdot (n+1) / 2$

$$T(n) = c \left( \frac{1}{2} n^2 + \frac{1}{2} n \right) \quad \text{mit } c = \text{Zeitkosten einer Reduktion}$$

$$T(n) = O(n^2)$$

```

max2 [x] = x
max2 (a:b:xs) = aux a (b:xs)
  where
    aux a [] = a
    aux a (b:xs) | a>b = aux a xs
                  | otherwise = aux b xs

```

	Anzahl der Reduktionen
$\text{max2 } [x_1, x_2, \dots, x_{n-1}, x_n] \Rightarrow \text{aux } x_1 (x_2, : [x_3, \dots, x_{n-1}, x_n])$	1
$\Rightarrow \text{aux } x_2 (x_3, : [x_4, \dots, x_{n-1}, x_n])$	1
$\Rightarrow \text{aux } x_3 (x_4, : [x_5, \dots, x_{n-1}, x_n])$	1
$\dots$	$\dots$
$\Rightarrow \text{aux } x_i (x_n, : [])$	1
$\dots$	$\dots$
$\Rightarrow \text{aux } x_j []$	1

$$T(n) = c \cdot n + 1$$

$$T(n) = O(n)$$

**5. Aufgabe** (8 Punkte)

Betrachten Sie folgenden algebraischen Datentyp und folgende Funktionsdefinition:

```
data Nat = Zero | S Nat deriving Show
```

```
add :: Nat -> Nat -> Nat
```

```
add a Zero = a
```

```
add a (S b) = add (S a) b
```

Definieren Sie damit folgende Funktionen:

```
gerade :: Nat -> Bool
```

```
fibonacci :: Nat -> Nat
```

Lösung:

```
gerade Zero      = True
```

```
gerade (S (S a)) = gerade a
```

```
gerade _         = False
```

```
fibonacci Zero = Zero
```

```
fibonacci (S Zero) = S Zero
```

```
fibonacci (S (S b)) = add (fibonacci b) (fibonacci (S b))
```

**6. Aufgabe** (12 Punkte)

Betrachten Sie folgenden algebraischen Datentyp für binäre Suchbäume:

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

Definieren Sie eine Funktion **complete**, die einen Baum **t** als Argument bekommt und entscheidet, ob der Baum vollständig ist oder nicht.

**complete** :: (Ord a) => BSearchTree a -> Bool

**-- 1. Lösung:**

height :: BSearchTree a -> Integer

height Nil = 0

height (Node x lt rt) = (max (height lt) (height rt)) + 1

complete Nil = True

complete (Node x lt rt) = (complete lt) && (complete rt) && height lt == height rt

**-- 2. Lösung:**

size :: BSearchTree a -> Integer

size Nil = 1

size (Node x lt rt) = size lt + size rt + 1

complete2 Nil = True

complete2 (Node x lt rt) = (complete2 lt) && (complete2 rt) && size lt == size rt

**-- 3. Lösung:**

complete3 tree = (size tree) == (2^((height tree)+1)-1)

**-- 4. Lösung:**

complete4 tree = fst (balanced tree)

where

balanced Nil = (True, 1)

balanced (Node x lt rt) = (bLeft && bRight && dLeft==dRight, 1+dRight)

where

(bLeft, dLeft) = balanced lt

(bRight, dRight) = balanced rt

**7. Aufgabe** (14 Punkte)

Betrachten Sie folgende Funktionsdefinitionen:

$\text{drop } 0 \text{ xs} = \text{xs}$  ..... (drop.1)

$\text{drop } (n+1) [] = []$  ..... (drop.2)

$\text{drop } (n+1) (x:\text{xs}) = \text{drop } n \text{ xs}$  ..... (drop.3)

$\text{take } 0 \text{ xs} = []$  ..... (take.1)

$\text{take } (n+1) [] = []$  ..... (take.2)

$\text{take } (n+1) (x:\text{xs}) = x:(\text{take } n \text{ xs})$  ..... (take.3)

$(++) [] \text{ ys} = \text{ys}$  ..... ((+).1)

$(++) (x:\text{xs}) \text{ ys} = x:(\text{xs} ++ \text{ys})$  ..... ((+).2)

Beweisen Sie mittels struktureller Induktion über die Liste  $\text{xs}$  folgende Eigenschaft:

$$\text{take } n \text{ xs} ++ \text{drop } n \text{ xs} = \text{xs}$$

Geben Sie für alle Beweisschritte die verwendeten Beziehungen oder die Gesetze an.

**Lösung:**

**I.A.  $\text{xs} = []$**

**zz.  $\text{take } n [] ++ \text{drop } n [] = []$**

1.Fall.  $n=0$

**zz.  $\text{take } 0 [] ++ \text{drop } 0 [] = []$**

$$\begin{aligned} \text{take } 0 [] ++ \text{drop } 0 [] &= [] ++ [] \\ &\quad \text{take.1, drop.1} \\ &= [] \\ &\quad (+).1 \end{aligned}$$

2.Fall.  $n > 0$

**zz.  $\text{take } n [] ++ \text{drop } n [] = []$**

$$\begin{aligned} \text{take } n [] ++ \text{drop } n [] &= [] ++ [] \\ &\quad \text{take.2, drop.2} \\ &= [] \\ &\quad (+).1 \end{aligned}$$

**I.V. für  $\text{xs} = \text{xs}'$  gilt  $\text{take } n \text{ xs}' ++ \text{drop } n \text{ xs}' = \text{xs}'$**

**Induktionsschritt.  $xs = (x:xs')$** 

$$\text{zz. } \text{take } n \ (x:xs') \ ++ \ \text{drop } n \ (x:xs') \ = \ (x:xs')$$

1. Fall.  $n=0$ 

$$\text{zz. } \text{take } 0 \ (x:xs') \ ++ \ \text{drop } 0 \ (x:xs') \ = \ (x:xs')$$

$$\text{take } 0 \ (x:xs') \ ++ \ \text{drop } 0 \ (x:xs') \ = \ [] \ ++ \ (x:xs')$$

*take.1, drop.1*

$$= \ (x:xs')$$

*(++) .1*2. Fall  $n>0$ 

$$\text{take } n \ (x:xs') \ ++ \ \text{drop } n \ (x:xs') \ = \ x:(\text{take } (n-1) \ xs') \ ++ \ \text{drop } n \ (x:xs')$$

*take.3*

$$= \ x:(\text{take } (n-1) \ xs') \ ++ \ \text{drop } (n-1) \ xs'$$

*drop.3*

$$= \ x:((\text{take } (n-1) \ xs') \ ++ \ \text{drop } (n-1) \ xs')$$

*(++) .1*

$$= \ (x:xs')$$

*I.V*dann gilt die Eigenschaft für alle  $xs :: a$  endliche Listen.**8. Aufgabe** (8 Punkte)

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

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

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

Lösung:

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

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

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

**9. Aufgabe** (12 Punkte)

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

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

b) (3 P.) Reduzieren Sie folgenden Kombinatoren-Ausdruck zur Normalform. Begründen Sie die einzelnen Schritte.

Lösung:

$$\begin{aligned}
SI(\underline{K}IS)(SKI) &\Rightarrow \underline{S}II(SKI) \\
&\Rightarrow \underline{I}(SKI)(\underline{I}(SKI)) \\
&\Rightarrow (\underline{S}KI)(SKI) \\
&\Rightarrow \underline{K}(SKI)(I(SKI)) \\
&\Rightarrow SKI
\end{aligned}$$

c) (7 P.) Zeigen Sie, dass folgende Lambda- und Kombinatoren-Ausdrücke äquivalent sind.

$$\lambda x.\lambda y.(xx) \equiv S(KK)((SI)I)$$

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

Lösung:

$$\begin{aligned}
\lambda x.\lambda y.(xx) &\Rightarrow_{\gamma)} T[\lambda x.T[\lambda y.(xx)]] \\
&\Rightarrow_{4)} T[\lambda x.(K \ T[xx])] \\
&\Rightarrow_{2)} T[\lambda x.(K \ T[x] \ T[x])] \\
&\Rightarrow_{1)1)} T[\lambda x.(K \ xx)] \\
&\Rightarrow_{6)} S \ T[\lambda x.K] \ T[\lambda x.xx] \\
&\Rightarrow_{4)1)6)} S \ (KK) \ (S \ T[\lambda x.x] \ T[\lambda x.x]) \\
&\Rightarrow_{3)3)} S \ (KK) \ (SII) \\
&\Rightarrow S \ (KK) \ ((SI)I)
\end{aligned}$$



**10. Aufgabe** (12 Punkte)

- a) Zeigen Sie, dass die Funktion **min**, die die kleinste Zahl von zwei natürlichen Zahlen berechnet, 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.

Folgende Definitionen ist vorgegeben:

$$\begin{aligned} \text{add} (0, m) &= \pi_1^1(m) \\ \text{add} (S(n), m) &= S(\pi_1^3(\text{add}(n, m), n, m)) \end{aligned}$$

Lösung:

a)

$$\begin{aligned} \text{pred} (0) &= C_0^0 \\ \text{pred} (S(n)) &= \pi_2^2(\text{pred}(n), n) \end{aligned}$$

$$\text{sub}(m, n) = \text{sub}'(\pi_2^2(m, n), \pi_1^2(m, n))$$

$$\text{sub}'(0, m) = \pi_1^1(m)$$

$$\text{sub}'(S(n), m) = \text{pred}(\pi_1^3(\text{sub}'(n, m), n, m))$$

$$\text{min}(x, y) = \text{sub}(\pi_1^2(x, y), \text{sub}(\pi_1^2(x, y), \pi_2^2(x, y)))$$

b)

```
predd :: PRFunction
predd = pr predd (const 0) (p 2)
```

```
sub :: PRFunction
sub = compose sub' [(p 2), (p 1)]
```

```
sub' :: PRFunction
sub' = pr sub' (p 1) (compose predd [(p 1)])
```

```
min :: PRFunction
min = compose sub [(p 1), compose sub [(p 1), (p 2)]]
```

```

-- 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  $x \in FV(E)$