

[illegible]

Lösung Aufgabe 1 (Verschiedenes):**(4 Punkte)**

- a) Erklären Sie so kurz wie möglich den Begriff “Endständige Rekursion”.

LÖSUNG: Jeder rekursive Aufruf liefert direkt ein Endergebnis.

ODER: Ergebnisse aus rekursiven Aufrufen werden nicht mehr weiterverarbeitet.

ODER: Nach der Rückkehr aus einem rekursiven Aufruf findet keine weitere Berechnung statt.

- b) Geben Sie einen Haskell Ausdruck an, welcher eine unendliche Liste Ihrer Wahl definiert!

LÖSUNG:

```
let ones = 1 : ones in ones
-- oder
let nums = 0 : (map (+1) nums) in nums
-- oder noch viel einfacher
[1..]
```

- c) Geben Sie *eine* Haskell Typdeklaration für einen polymorphen *und* rekursiven Datentyp an (beides gleichzeitig). Sie dürfen sich irgendeine ausdenken; egal ob es diese in der Standardbibliothek gibt oder nicht.

LÖSUNG: Hier sind drei verschiedene akzeptable Antworten (eine Antwort reicht):

```
data Foo a    = Foo (Foo a) -- sinnloser Typ ohne endliche Werte, genügt hier aber
data List a   = Nil | Cons a (List a)
data Tree a b = Leaf a | Node b [Tree a b]
```

Viele haben hier eine Typsignatur einer polymorphen Funktion angegeben, was natürlich keinen Punkt bringt.

- d) Nennen Sie einen Vorteil von Call-By-Name gegenüber Call-By-Value.

LÖSUNG: Mögliche Antworten:

- Kann unnötige Berechnungen vermeiden.
- Kann auch noch einen Wert liefern/terminieren, wenn Call-By-Value nicht terminiert.

Lösung Aufgabe 2 (Auswertung):**(6 Punkte)**

- a) Rechnen Sie aus, zu welchem Wert die folgenden Haskell-Ausdrücke vollständig auswerten. Geben Sie nur das Endergebnis an, etwaige Nebenrechnungen bitte deutlich abtrennen:

(i) `(4 == 5.0):([])`

— `[False]` bekannt als A1-5a —

(ii) `[[z,y] | x<-[1..4], y<-[1,3..7], let z = x-1, x==y]`

— `[[0,1],[2,3]]` —

(iii) `(\x y-> y) "yes" (\z -> "no") "which?"`

— `"no"` ähnlich zu A1-5f —

- b) Werten Sie folgenden Ausdruck schrittweise vollständig aus, unterstreichen Sie dabei den reduzierten Redex. Sie dürfen eine beliebige Auswertestrategie verwenden, für volle Punktzahl müssen Sie aber Call-By-Value oder Call-By-Name verwenden.

Beispiel: `(\x -> 43 + x) (negate 1) ~> (\x -> 43 + x) (-1) ~> 43 + (-1) ~> 42`

Definitionen:

```
const x y = x
negate x  = -x
```

Ausdruck zum Auswerten:

`const const (negate 1) (negate 2) 3 ~>`

LÖSUNG: Funktionsanwendung ist implizit links geklammert, d.h. auszuwerten ist

`((const const) (negate 1)) (negate 2) 3`

Fehlerhafte Klammerung war hier der häufigste Fehler.

Call-By-Name:

`const const (negate 1) (negate 2) 3 ~> const (negate 2) 3 ~> negate 2 ~> -2`

Call-By-Value:

`const const (negate 1) (negate 2) 3 ~> const const (-1) (negate 2) 3
~> const const (-1) (-2) 3 ~> const (-2) 3 ~> -2`

- c) Welche Auswertestrategie haben Sie in Aufgabenteil b verwendet?

— (Für den Punkt musste konsequent Call-By-Value oder Call-By-Name angewendet worden sein.) —

Lösung Aufgabe 3 (Induktion):**(5 Punkte)**

Wir betrachten folgende Funktionsdefinitionen:

```
length :: [Char] -> Int
length [] = 0 -- (LZ)
length (x:xs) = 1 + length xs -- (LS)

insert :: Char -> [Char] -> [Char]
insert x [] = [x] -- (IN)
insert x (y:ys) | x <= y = x : y : ys -- (IK)
                 | otherwise = y : insert x ys -- (IG)
```

Beweisen Sie mit Induktion über die Länge der Liste, dass für alle Zeichen c und alle Strings s die folgende Gleichung gilt:

$$\text{length} (\text{insert } c \ s) = 1 + \text{length } s$$

Hinweise: Formen Sie beide Seiten der geforderten Gleichung schrittweise in den exakt gleichen Term um. Begründen Sie jeden Umformungsschritt durch Angabe des Kürzels der verwendeten Gleichung, also (LZ), (LS), (IN), (IK) oder (IG), und (IH) bei Verwendung der Induktionshypothese! Beachten Sie die Pattern-Guards bei Verwendung von (IK) oder (IG) und begründen Sie kurz, warum der jeweilig Fall eintritt.

LÖSUNG: Diese Aufgabe kennen wir in leicht anderer Formulierung als A6-3a.

Der Beweis wird mit Induktion über die Länge der Liste s geführt. *Wichtig:* Die andere Variable c verbleibt als Variable von einem beliebigen Wert, denn wir wollen die Gleichung ja auch für beliebige Werte von c beweisen!

Induktionsanfang für eine Liste der Länge 0: Es gilt also $s = []$.

$\text{length} (\text{insert } c \ []) = (\text{IN}) = \text{length } [c] = (\text{LS}) = 1 + \text{length } []$

Damit haben wir die linke Seite der geforderten Gleichung direkt in die rechte Seite überführt. Den Schritt $1 + \text{length } [] = (\text{LZ}) = 1 + 0 = 1$ benötigen wir also gar nicht mehr.

Induktionsschritt für Liste der Länge $n > 0$: Da die Liste s in diesem Fall nicht leer ist, können wir dem Kopf und Rumpf von s eigene, frische Namen geben: Es sei $s = (h:t)$. Damit ist t eine Liste mit einer Länge kleiner als n . Somit dürfen wir als Induktionshypothese (IH) die Gleichung $\text{length} (\text{insert } d \ t) = 1 + \text{length } t$ für ein beliebiges Zeichen d verwenden.

Wir müssen die Gleichung $\text{length} (\text{insert } c \ (h:t)) = 1 + \text{length } (h:t)$ beweisen. Zu Beginn möchten wir gerne (IK) oder (IG) anwenden. Da wir keine Information haben, welcher Fall eintritt, müssen wir einfach beide Fälle betrachten. Dies tun wir einer Fallunterscheidung:

Fall 1: $c \leq h$

$\text{length} (\text{insert } c \ (h:t)) = (\text{IK}) = \text{length } (c:h:t) = (\text{LS}) = 1 + \text{length } (h:t)$

Die Induktionshypothese benötigen wir in diesem einfachen Fall also gar nicht.

Fall 2: $c > h$

$\text{length} (\text{insert } c \ (h:t)) = (\text{IG}) = \text{length } (h:(\text{insert } c \ t)) = (\text{LS}) = 1 + \text{length} (\text{insert } c \ t) = (\text{IH}) = 1 + 1 + \text{length } t = (\text{LS}) = 1 + \text{length } (h:t)$

Wer Schwierigkeiten mit dem letzten Umformungsschritt hat, sollte diesen einfach von rechts-nach-links lesen.

Lösung Aufgabe 4 (Abstiegssfunktion):**(5 Punkte)**

Wir wollen mithilfe einer geeigneten Abstiegssfunktion zeigen, dass die folgende rekursive Funktion `foo :: (Int,Int,Int) -> Int`, gegeben in Haskell Notation, immer terminiert:

```
foo (x, y, z)
  | x > 10, z > 0  = 2 * foo (x-1, y+3, z+2)
  | x > 0, z > 20 = 4 * foo (x+3, y-1, z `div` 2)
  | otherwise = y
```

- a) Zeigen Sie, dass die Funktion $m'(x, y, z) = \max(2x + z, 0)$ keine geeignete Abstiegssfunktion für den Terminationsbeweis von `foo` ist.

LÖSUNG: Wir wählen als Argument zum Beispiel $(50, 50, 50)$ (viele andere Werte sind auch möglich). Damit sind wir im ersten Fall der Funktionsgleichung. Wenn m' eine Abstiegssfunktion wäre, dann müsste $m'(50, 50, 50) > m'(49, 50, 52)$ gelten.

Dies gilt aber nicht, wie wir durch nachrechnen beweisen:

$$\begin{aligned} m'(50, 50, 50) &= \max(100 + 50, 0) = 150 \\ m'(49, 50, 52) &= \max(98 + 52, 0) = 150 \end{aligned}$$

Wir haben also ein Funktionsargument, für den der Wert von m' für einen rekursiven Aufruf nicht echt kleiner wird. Somit ist m' keine geeignete Abstiegssfunktion.

Das Argument "weil y in m' nicht vorkommt" nutzt gar nichts; wie wir im zweiten Aufgabenteil sehen, muss nicht jedes Argument in der Abstiegssfunktion berücksichtigt werden.

- b) Finden Sie einen Wert $c > 0$ so dass $m(x, y, z) = \max(cx + z, 0)$ eine Abstiegssfunktion ist und beweisen Sie dies.

Hinweise: Auf die Bedingungen AUF und DEF verzichten wir hier. Weiterhin dürfen Sie folgende Abschätzung verwenden: Wenn $z > 20$ dann gilt auch $(z \text{ 'div' } 2) < z - 10$

LÖSUNG: Wir wählen als Abstiegssfunktion $m(x, y, z) = \max(3x + z, 0)$ mit $m : \text{Int} \times \text{Int} \times \text{Int} \rightarrow \mathbb{N}$.

Der erste rekursive Aufruf erfolgt für $x > 10$ und $z > 0$. Damit gilt in diesem Fall einfach $\max(3x + z, 0) = 3x + z$ und auch $\max(3(x-3) + (z+2), 0) = 3(x-3) + (z+2)$. Wir rechnen:

$$\begin{aligned} m(x, y, z) &= \max(3x + z, 0) = 3x + z > 3x + z - 1 = 3x - 3 + z + 2 = 3(x-1) + (z+2) \\ &= \max(3(x-1) + (z+2), 0) = m(x-1, y+3, z+2) \end{aligned}$$

Der zweite rekursive Aufruf erfolgt, für $x > 0$ und $z > 20$. Damit gilt in diesem Fall erneut $\max(3x + z, 0) = 3x + z$ und insbesondere auch $\max(3(x+3) + z \text{ 'div' } 2, 0) = 3(x+3) + z \text{ 'div' } 2$, da bei der ganzzahligen Division einer positiven Zahl größer als 1 wieder eine positive Zahl als Ergebnis herauskommt.

Wir rechnen:

$$\begin{aligned} m(x, y, z) &= \max(3x + z, 0) = 3x + z > 3x + z - 1 = 3x + 9 + z - 10 > 3x + 9 + (z \text{ 'div' } 2) \\ &= 3(x+3) + (z \text{ 'div' } 2) = \max(3(x+3) + (z \text{ 'div' } 2), 0) = m(x+3, y-1, z \text{ 'div' } 2) \end{aligned}$$

Wird der Wert von y in der Abstiegssfunktion berücksichtigt, kann man im Gegensatz zu unserem Lösungsvorschlag hier die max-Funktion nicht mehr so einfach verschwinden lassen!

Lösung Aufgabe 5 (Maybe):**(6 Punkte)**

Gegeben ist folgende Datentypdeklaration aus der Standardbibliothek:

```
data Maybe a = Nothing | Just a
```

- a) Implementieren Sie die Funktion `fromMaybe :: a -> Maybe a -> a` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek.

Diese Funktion liefert das erste Argument zurück, falls das zweite Argument `Nothing` war; ansonsten wird der im zweiten Argument "verpackte" Wert zurückgegeben.

LÖSUNG:

```
fromMaybe x Nothing = x
fromMaybe _ (Just y) = y
```

- b) Implementieren Sie eine Funktion `andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool` zu Fuss, d.h. ohne Verwendung von Funktionen der Standardbibliothek; lediglich die Funktion `&& :: Bool -> Bool -> Bool` dürfen Sie verwenden. Die Funktion `andMaybe` soll zwei in `Maybe` verpackte boolsche Werte mit dem logischen UND verknüpfen. Falls einer der Werte nicht vorliegt, so soll auch kein Wahrheitswert zurückgegeben werden.

Der Typkonstruktor `Maybe` kann als Monade aufgefasst werden. Die volle Punktzahl erhalten Sie für diese Aufgabe nur dann, wenn Ihre Lösung die DO-Notation sinnvoll einsetzt. Die Verwendung von `return :: a -> m a` ist erlaubt.

LÖSUNG: Ohne DO-Notation:

```
andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool
andMaybe (Just a) (Just b) = Just (a && b)
andMaybe _      _         = Nothing
```

Mit DO-Notation (siehe Vorlesungsfolie 12-14):

```
andMaybe :: Maybe Bool -> Maybe Bool -> Maybe Bool
andMaybe ma mb = do a <- ma
                    b <- mb
                    return $ a && b
```

- c) Geben Sie die Definition der Methode `return :: a -> Maybe a` aus der Instanzdeklaration von `Maybe` zur Klasse `Monad` an.

```
instance Monad Maybe where
    return x = Just x
```

Lösung Aufgabe 6 (Datenstrukturen):**(5 Punkte)**

Gegeben ist folgende Datentypdeklaration zur Repräsentation von Typen:

```
data TTyp = TVar Char | TInt | TListe TTyp | TTupel [TTyp]
  deriving (Eq, Show)
```

Hinweis: In dieser Aufgabe dürfen Sie alle Funktionen der Standardbibliothek verwenden.

- a) Schreiben Sie eine Funktion `rename :: (Char -> Char) -> TTyp -> TTyp` welche alle `Char`-Zeichen in einem Wert des Typs `TTyp` gemäß einer gegebenen Funktion umbenennt. *Beispiel:*

```
> let t = TListe (TTupel [TVar 'a', TInt, TVar 'c'])
t :: TTyp
> let s = \c -> if c=='a' then 'b' else c
s :: Char -> Char
> rename s t
TListe (TTupel [TVar 'b', TInt, TVar 'c'])
```

LÖSUNG: Diese Teilaufgabe war schon aus der Probeklausur bekannt.

```
rename :: (Char -> Char) -> TTyp -> TTyp
rename r (TVar v)      = TVar $ r v
rename _ TInt          = TInt
rename r (TListe tt)   = TListe $ rename r tt
rename r (TTupel tts) = TTupel $ map (rename r) tts
```

Bemerkung:

In dieser Aufgabe ging es eigentlich nur um Bäume. Der Bekanntheit wegen wurde hier ein zur damaligen Aufgabe “Unifikation II” sehr ähnlicher Baum-Datentyp wiederverwendet.

Natürlich konnte man auch ganz ordentlich eine **Functor** Instanz definieren, und `rename = fmap` definieren, was ja ebenfalls behandelt wurde.

- b) Schreiben Sie eine Funktion `count :: TTyp -> Int`, welche die Anzahl der `TInt`-Knoten in einem Wert des Typs `TTyp` berechnet. *Beispiel:*

```
> count (TTupel [TVar 'a', TListe (TInt), TInt, TTupel [TInt]])
3
```

LÖSUNG: Diese Teilaufgabe sogar noch einfacher. Es war sehr verwunderlich, dass viele Teilnehmer die erste Aufgabe gut lösen konnten, aber diese nahezu identische zweite Teilaufgabe überhaupt nicht.

```
count :: TTyp -> Int
count (TVar _) = 0
count TInt     = 1
count (TListe t) = count t
count (TTupel l) = sum $ map count l
-- Zahlreiche Varianten möglich, z.B.:
-- foldr ((+).count) 0 l
```

Lösung Aufgabe 7 (Funktionen höherer Ordnung):**(5 Punkte)**

- a) Implementieren Sie zu Fuss, d.h. ohne Verwendung von Funktionen aus der Standardbibliothek, die Infix-Funktion zur Funktionskomposition: `(.) :: (b -> c) -> (a -> b) -> a -> c`
Beispiel: `((+1).(*2)) 3 == 7`

LÖSUNG: Siehe Folie 7-25:

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

- b) Schreiben Sie eine Funktion `compose :: [a -> a] -> a -> a` welche eine Liste von Funktion von rechts-nach-links auf einen Startwert anwendet:
Beispiel: `compose [(+1),(*2)] 3 == 7`

Sie dürfen alle Funktionen aus der Standardbibliothek einsetzen. Um die volle Punktzahl zu erhalten, dürfen Sie keine direkte Rekursion verwenden! Verwenden Sie stattdessen Funktionen der Standardbibliothek, wie z.B. `foldr :: (a -> b -> b) -> b -> [a] -> b`

LÖSUNG: Einfache Lösung mit direkter Rekursion:

```
compose []      x = x
compose [f]     x = f x
compose (f:fs) x = f (compose fs x)
```

Hier vier alternative Lösungsmöglichkeiten mit `foldr`:

```
compose1      = foldr (.) id

compose2 fs x = foldr (\f acc -> f acc) x fs

compose3 fs x = foldr ($) x fs

compose4      = flip $ foldr ($)
```


Lösung Aufgabe 8 (Typen):**(7 Punkte)**

- a) Geben Sie jeweils den allgemeinsten Typ des gegebenen Haskell-Ausdrucks an, inklusive etwaiger Typklassen Einschränkungen. Bitte nur das Ergebnis hinschreiben. Nebenrechnungen bitte deutlich abtrennen.

(i) `fst ('7', (()))`

Char

(ii) `[False] : []`

[[Bool]] siehe A6a aus der Probeklausur

(iii) `(\f x y -> (f y) ++ show x)`

Show a => (b -> String) -> a -> b -> String

- b) Geben Sie einen allgemeinsten Unifikator für folgende Typgleichung an:

$\{\alpha \rightarrow \beta \rightarrow \text{Int} = \beta \rightarrow \gamma\}$ $[\beta/\alpha, (\beta \rightarrow \text{Int})/\gamma]$

- c) Beweisen Sie folgendes Typurteil unter Verwendung der zur Erinnerung auf Seite 9 angegeben Typregeln in einer der beiden in der Vorlesung behandelten Notationen (Herleitungsbaum oder lineare Schreibweise).

(iv) $\Gamma \vdash \text{let } (x, y) = (f, \text{True}) \text{ in } x y :: \text{Int}$ wobei $\Gamma = \{f :: \text{Bool} \rightarrow \text{Int}\}$

LÖSUNG: Als Herleitungsbaum aufgeschrieben:

$$\frac{\frac{\Gamma \vdash f :: \text{Bool} \rightarrow \text{Int}}{\Gamma \vdash (f, \text{True}) :: (\text{Bool} \rightarrow \text{Int}, \text{Bool})} (\text{VAR}) \quad \frac{\Gamma \vdash \text{True} :: \text{Bool}}{\Gamma \vdash (f, \text{True}) :: (\text{Bool} \rightarrow \text{Int}, \text{Bool})} (\text{BOOL})}{\Gamma \vdash \text{let } (x, y) = (f, \text{True}) \text{ in } x y :: \text{Int}} (\text{PAIR-INTRO}) \quad \frac{\frac{\Delta \vdash x :: \text{Bool} \rightarrow \text{Int}}{\Delta \vdash x y :: \text{Int}} (\text{VAR}) \quad \frac{\Delta \vdash y :: \text{Bool}}{\Delta \vdash x y :: \text{Int}} (\text{APP})}{\Delta \vdash x y :: \text{Int}} (\text{PAIR-ELIM})$$

wobei $\Delta = \{f :: \text{Bool} \rightarrow \text{Int}, x :: \text{Bool} \rightarrow \text{Int}, y :: \text{Bool}\}$

Typregeln:

$$\frac{}{\Gamma \vdash x :: \Gamma(x)} \quad (\text{VAR})$$

Alternative Schreibweise für Var:

$$\frac{x :: A \in \Gamma}{\Gamma \vdash x :: A} \quad (\text{VAR})$$

$$\frac{c \text{ ist eine Integer Konstante}}{\Gamma \vdash c :: \mathbf{Int}} \quad (\text{INT})$$

$$\frac{c \in \{\mathbf{True}, \mathbf{False}\}}{\Gamma \vdash c :: \mathbf{Bool}} \quad (\text{BOOL})$$

$$\frac{\Gamma \vdash e_1 :: A \rightarrow B \quad \Gamma \vdash e_2 :: A}{\Gamma \vdash e_1 e_2 :: B} \quad (\text{APP})$$

$$\frac{\Gamma, x :: A \vdash e :: B}{\Gamma \vdash \lambda x \rightarrow e :: A \rightarrow B} \quad (\text{ABS})$$

$$\frac{\Gamma \vdash e_1 :: A \quad \Gamma \vdash e_2 :: B}{\Gamma \vdash (e_1, e_2) :: (A, B)} \quad (\text{PAIR-INTRO})$$

$$\frac{\Gamma \vdash e_1 :: (B, C) \quad \Gamma, x :: B, y :: C \vdash e_2 :: A}{\Gamma \vdash \mathbf{let} (x, y) = e_1 \mathbf{in} e_2 :: A} \quad (\text{PAIR-ELIM})$$

$$\frac{\Gamma \vdash e_1 :: A \quad \Gamma, x :: A \vdash e_2 :: C}{\Gamma \vdash \mathbf{let} x = e_1 \mathbf{in} e_2 :: C} \quad (\text{LET})$$

$$\frac{\Gamma \vdash e_1 :: \mathbf{Bool} \quad \Gamma \vdash e_2 :: A \quad \Gamma \vdash e_3 :: A}{\Gamma \vdash \mathbf{if} e_1 \mathbf{then} e_2 \mathbf{else} e_3 :: A} \quad (\text{COND})$$

