Disclaimer: Keine Garantie auf Richtigkeit, aber vielleicht hilft es ja beim Erarbeiten vom Stoff.

Teil I, Kapitel 1 ‘Motivation’

2. Wie kann die mathematische Funktion zwei-hoch, in Zeichen $2^{\cdot}$:

$$2^{\cdot} : \mathbb{N} \to \mathbb{N}$$

$$\forall n \in \mathbb{N}.\ 2^n = \begin{cases} 1 & \text{falls } n = 0 \\ 2 * 2^{n-1} & \text{falls } n > 0 \end{cases}$$

in Haskell implementiert werden?

```
F2 :: Integer → Integer
F2 0 = 1
F2 n = 2 * 2**(n-1)
```

3. Was passiert, wenn Ihre Haskell-Funktion aus der vorigen Aufgabe mit einem negativen Wert aufgerufen wird? Non exaustive pattern error

4. Wie könnte eine (einfache) Fehlerbehandlung für Ihre Implementierung von $2^{\cdot}$ aussehen?

```
F2 n
        | n < 0 = error “function undefined for negative values”
        | otherwise = 2 * 2**(n-1)
```

5. Können Sie syntaktisch andere Varianten angeben, Ihre Haskell-Funktion aus Aufgabe 2 zu definieren? Wenn ja, welche? Welche Sprachmittel nutzen diese Varianten? Welches Sprachmittel haben Sie in Aufgabe 2 genutzt?
Guards (bewachte Ausdrücke)
Andere Varianten:
If-then-else (bedingte Ausdrücke):

```
F2 n = if n == 0 then 1
                else if n < 0 then error “Hier könnte Ihre Werbung stehen.”
                        else 2 * 2**(n-1)
```

6. Imperative und funktionale Sprachen kennen Fallunterscheidungen. Was ist der große Unterschied? Konzeptuell? Bedeutungsmäßig?
???

7. Können Sie die Begriffe curryfiziert und uncurryfiziert für Funktionen anhand eines Beispiels erklären?

Curryfizierte Funktionen konsumieren alle Argumente einzeln. Dies bietet den Vortei einer partiellen Auswertung.
Uncurryfizierte Funktionen fassen Argumente als Tupel zusammen, hierbei können die Argumente nicht einzeln konsumiert werden. (außer es wird vorher curry angewandt)

8. Was sind die Werte der (Listen-) Ausdrücke:
(a)`[5..10] = [5,6,7,8,9,10]`
(b)`[5,8..15] = [5,8,11,14]`
(c)`[‘a‘,‘e‘,..] = [‘a’, ‘e’, ‘i’,… zuerst weiteres Alphabet dann ASCII-Zeichen]`
(d)`[‘z‘..‘a‘] = Alphabet rückwärts`
(e)`[n | n <- [1..10], n mod 2 == 1] = [1,3,5,7,9]`

11. Was ist mit von Neumann-Stil gemeint? Programmieren das aus der von-Neumann-Architektur beruht

12. Was ist mit dem Schlagwort Softwarekrise gemeint? Phänomen dass Software der Hardware hinterherhinkt

13. Welche drei Hauptziele sollte Software-Entwicklung erfüllen, ob ingenieurmäßig genannt werden zu dürfen? verifiziert, wartbar, erweiterbar

14. Welche Eigenschaften sind kennzeichnend für
(a) imperative
Unterscheidung Anweisung und Ausdrücke, Seiteneffekte, befehlsbasiertes Programmieren, zeitliche Reihenfolge wichtig für das Ergebnis
(b) funktionale Programmierung
nur Ausdrücke, gleichungsbasiertes und ergebnisorientiertes Programmieren
keine Seiteneffekte (rein funktional), zeitliche Reihenfolge egal

15. Wo wird funktionale Programmierung heute eingesetzt? Können Sie einige Beispiele von Einsatzfeldern nennen? “überall”, Banken, Webinterfaces, Spiele

16. Haskell ist eine von vielen funktionalen Programmiersprachen. Können Sie einige weitere funktionale Sprachen nennen? Welche? LISP, SASL, ML, Caml

18. Wonach, nach wem ist Haskell benannt? Haskell Brooks Curry

19. Was unterscheidet sog. gewöhnliche und literate Haskell-Skripte konzeptuell voneinander? Kommentare müssen extra gekennzeichnet werden oder alles was nicht anders gekennzeichnet ist, ist ein Kommentar. (literates → alles was nicht mit > gekennzeichnet ist Kommentar) (gewöhnliches -- und {- … -} ist Kommentar)

20. Wie kann man
(a) einzeilige --
(b) mehrzeilige Kommentare {- .. - }
in Haskell-Programme einfügen?

21. Wie sind Identifikatoren in Haskell aufgebaut? Welche Zeichen dürfen sie enthalten? Welche nicht? Nichtleer, Klein- & Großbuchstaben und Ziffern und ‘ und & und _
Zahlen nicht am Anfang; Funktionen und Typvariablen werden klein geschrieben, Typnamen und Funktionen groß

22. Was berechnet der Euklidische Algorithmus? GgT

23. Was hat Eratosthenes mit dem nach ihm benannten Sieb zu berechnen vorgeschlagen? Primzahlenrechner

25. Welche Eigenschaften müssen m und n erfüllen, damit der Binomialkoeffizient (nm) definiert ist? Nicht negative ganze Zahlen sein

26. Was gibt der Binomialkoeffizient (nm) an? Er gibt an, auf wie viele verschiedene Arten man bestimmte Objekte aus einer Menge von verschiedenen Objekten auswählen kann (ohne Zurücklegen, ohne Beachtung der Reihenfolge)

27. Welche Randbedingungen gelten dabei für m und n? Siehe 25?

28. Was ist mit einem hierarchischen System von Rechenvorschriften gemeint? Ein Element basiert auf einer anderen Rechenvorschrift

29. Was mit einem wechselweise rekursiven? Zwei Rechenvorschriften basieren aufeinander

30. Geben Sie je ein Beispiel für ein hierarchisches und ein wechselweise rekursives System von Rechen-vorschriften an. ggt und mod; isOdd und isEven

31. Was sind Hoogle und Hayoo? Suchmaschinen für Haskellbefehle

32. Was ist Leksah? Eine Haskell IDE

33. Was sind Hugs und GHCi? GHCi interaktive Umgebung für GHC, Hugs byte code interpreter für Haskell

34. Was ist GHC? Glasgow Haskell Compiler

35. Funktionale Programmierung ist das Rechnen mit Funktionen. Was ist damit gemeint? Alles ist in Funktionen verpackt und Funktionen könne auch als Input oder Output verwendet werden.

36. Welche Eigenschaften gelten als besondere Stärke funktionaler Programmierung? Welche möglicherweise als ihre Schwäche? Einfacher zu erlernen da weniger Grundkonzepte, höhere Produktivität da Programme viel kürzer, höhere Zuverlässigkeit (Korrektheitsüberlegungen einfacher da math. Fundiert)
geringere Performance, für inhärent Zustandbasierte Anwendungen und GUI nur schlecht geeignet

Teil II, Kapitel 2 ‘Vordefinierte Datentypen’

1. Welche vordefinierten Datentypen bietet Haskell?
Ganze Zahlen (Int, Integer), Gleitkommazahlen (Float, Double), Wahrheitswerte (Bool), Zeichenwerte (Char); Tupel, Listen, String ([Char])
2. Welche davon sind strukurierte Datentypen, welche unstrukturierte?
Unstruktruriert: elementare Typen (ganze Zahlen, Gleitkommazahlen, Wahrheitswerte, Zeichenwerte)
anderen sind strukturiert
3. Welche Datentypen gibt es in Haskell zur Darstellung von Zahlen? Wie unterscheiden sie sich? Ganze und Gleitkommazahlen, Präzision (Double, Float) bzw. unendlicher oder endlicher Bereich (Int/Integer)
4. Müssen die Namen von Datentypen mit einem Groß- oder einem Kleinbuchstaben beginnen? groß
5. Welche Werte enthält der Datentyp der Wahrheitswerte? True, False
6. Wann sprechen wir von einem Operator? Wann von einer Operation?
Operator ist das Werkzeug für die Durchführung der Operation
Bsp) Operation = Addition Operator = +
7. Welche vordefinierten Operatoren gibt es auf Gleitkommazahlen? Nennen Sie einige Beispiele. +, -, *, ^, - (Vorzeichenwechsel)
8. Wozu dienen die vordefinierten Funktionen `ord` und `char`? Welche Signatur haben diese Funktionen? ord gibt ASCII-Wert eines Zeichens wieder, char parsed und gibt den spezifizierten character zurück
9. Wie sind Zeichenkonstanten in Haskell dargestellt? Geben Sie einige Beispiele, die Ihre Antwort illustrieren. ‘a’ :: Char , ‘Z’ :: Char, ‘\t’ :: Char,...
10. Welches ist der vordefinierte Operator für Paare in Haskell? Welches der für Tripel?
Konstruktor: Paare (,), Tripel (,,)
11. Welches ist der vordefinierte Operator für Listen in Haskell? Konstruktor v1:(v2:(...))
12. Wie ist die leere Liste in Haskell dargestellt? Wie das leere Tupel? Liste [], Tupel ()
13. Wie kann man ein Element als neues Kopfelement an den Anfang einer Liste setzen?
x : list
14. Wie kann man ein Element als (neues) letztes Element an einer Liste anfügen?
list ++ [x]
15. Welchen Typ haben die folgenden Ausdrücke?
(a)`["sin","cos","tan"] :: [String]`
(b)`[sin,cos,tan] :: Floating a => [a -> a]`
(c)`[] :: [a]`

(d)("sin","cos","tan") :: (String, String, String)
(e)(sin,cos,tan) :: (a,a,a)
(f)() :: ()
(g)[("sin","cos","tan")] :: [(String, String, String)]
(h)[(sin,cos,tan)] :: (Floating a1, Floating a2, Floating a3) => (a1 -> a1, a2 -> a2, a3 -> a3)
(i)[()] :: [a]
(j)(["sin","cos","tan"]) :: [String]
(k)([sin,cos,tan]) :: Floating a => [a -> a]
(l)([]) :: [a]
16. Mit welchem Operator kann man auf ein bestimmtes Element in einer Liste zugreifen? !!
Bsp) [5,3,2] !! 1 ->> 3
17. Welchen Index hat das erste Element einer Liste? n-1
18. Was macht die Funktion `concat`? Verschmelzen von Listen
19. Ergänzen Sie die Signatur der Funktion `anzElem` zur Berechnung der Zahl der Elemente einer Liste ganzer Zahlen um die fehlende Implementierung:

```
anzElem :: [Int] -> Int
anzElem [] = 0
anzElem (x:xs) = 1 + anzElem xs
```

20. Gibt es in Haskell eine vordefinierte Funktion, die diese Aufgabe sogar in größerer Allgemeinheit leistet? Wenn ja, wie heißt sie und in welchem Sinn ist sie ‘allgemeiner’ ?
length, ist polymorph und kann auf Listen (bzw. Foldables) angewendet werden

21. Wie heißen die vordefinierten Funktionen in Haskell, um auf das Kopfelement und den Rest einer Liste ohne das Kopfelement zuzugreifen? Welchen Typ haben diese Funktionen?

head ([a] → a), tail ([a] → [a])

22. Wie heißen die vordefinierten Funktionen in Haskell, um auf erste bzw. das zweite Element eines Paars zuzugreifen? Welchen Typ haben diese Funktionen?
fst (a,b) → a, snd (a,b) → b
23. Gibt es vordefinierte Funktionen, mit denen man auf die Elemente von Tripeln und Quadrupeln zugreifen kann? Nein
24. Beschreiben Sie die Menge der geraden Zahlen von 500 bis 1000 (jeweils einschließlich) mithilfe eines
(a) Listenaufzählungsausdrucks. [500, 502..1000]
(b) Listenkomprehensionsausdrucks. [x | x <- [500, 502..1000]]
25. Welcher Name wird für den Typ `[Char]` in Haskell auch verwendet? String

Teil II, Kapitel 3 ‘Funktionen’

1. Was ist ein
(a) bedingter if n == 0 then 1 else 2
(b) bewachter | n == 0 = 1
| otherwise = 2
(c) Wächter-Ausdruck? Geben Sie je ein Beispiel an. otherwise, n==0

2. Was ist der stets erfüllte Wächter(ausdruck)? otherwise = True

3. Die Funktion `sumQuadrat` berechnet die Summe der ersten n Quadratzahlen ab 1 :

```
sumQuadrat :: Int -> Int
sumQuadrat n = if n == 1 then 1 else n*n + sumQuadrat (n-1)
```

Schreiben Sie `sumQuadrat` mithilfe
(a) bewachter Ausdrücke

```
sumQuadrat n
        | n == 1 = 1
        | otherwise = n*n + sumQuadrat (n-1)
```

(b) von Mustern
sumQuadrat 1 = 1
sumQuadrat n = n*n + sumQuadrat (n-1)
(c) eines case-Ausdrucks
sumQuadrat n = case n of 1 → 1
otherwise → n*n + sumQuadrat (n-1)
(d) einer anonymen λ-Abstraktion
sumQuadrat = \n -> if n == 1 then 1 else n*n + sumQuadrat (n-1)

4. Die Nutzung von Mustern in Funktionsdefinitionen macht Selektorfunktionen unnötig. Was ist damit gemeint? Geben Sie ein Beispiel an, das Ihre Antwort illustriert.
Es müssen vor der Verwendung nicht erst mittels einer Funktion die richtigen Elemente selektierte werden.

5. Welche Vorteile versprechen musterbasierte Funktionsdefinitionen?
Bessere Lesbarkeit, es können sprechende Ausdrücke verwendet werden, legen Struktur offen

6. Auf welche Weise(n) lassen sich in Funktionsdefinitionen lokale Deklarationen einführen und nutzen? Geben Sie (je) ein illustrierendes Beispiel an.
func n = maxi
where maxi = max n

func n = let maxi = max n
in maxi*3

8. Wozu dient das ‘;’in Haskell?
Das Semikolon kann verschiedene Variablen abtrennen
Bsp) let maxi=maximum n; mini = minimum n
in ...
9. Welche Bezeichnungen werden üblicherweise, konventionsgemäß für
(a) Zeichenwerte c, c’, d,...
(b) Wahrheitswerte b, b’,...
(c) Ganze Zahlen n, n’, m,..
(d) Werte beliebigen Typs x, x’, y, y’
und für Listen von
(a) Zeichenwerten cs,cs´,ds,...
(b) Wahrheitswerten bs,bs´,...
(c) ganzen Zahlen ns,ns´,ms,...
(d) Werten beliebigen Typs gewählt? xs,xs´,ys,ys´,...

11. Gegeben ist (ohne Implementierung) die Signatur der (Editor-) Funktion:
```
type Text = String
type Suchwort = String
type Ersatzwort = String
```

Rechtsassoziativ
```
suche_und_ersetze :: (Text -> (Suchwort -> (Ersatzwort -> Text)))
```

und der Funktionsterm:

```
text = "Sehr langer Text, sehr langer Text, sehr langer Text"
suchwort = "sehr"
ersatzwort = "noch viel laengerer"
Linksassoziativ
(((suche_und_ersetze text) suchwort) ersatzwort)
```

Klammern Sie die Funktionssignatur und den Funktionsterm vollständig, aber nicht überfl üssig.

12. Welche Stelligkeit haben Haskell-Funktionen? Illustrieren Sie Ihre Antwort anhand der Funktionsuche und `ersetze`. Einstellig, es wird immer nur ein Argument gleichzeitig konsumiert

13. Geben ist die vollständig, aber nicht überflüssig geklammerte Funktionsdefinition:
`(binom :: (Int -> (Int -> Int)))((binom n) k) = (div (fac n) ((fac k) * (fac (n-k))))`
Nutzen Sie Haskells Klammereinsparungsregeln für Funktionssignaturen und Funktionsterme aus, um so viele Klammern wie möglich wegzulassen, ohne dass sich die Bedeutung ändert.
binom :: Int -> Int -> Int
binom n k = div (fac n) (fac k * fac (n-k))

14. Ist die Funktion `binom` aus der vorigen Aufgabe curryfiziert oder uncurryfiziert deklariert?
curryfiziert
15. Geben Sie die jeweils fehlende Deklarationsvariante für `binom` an.
binom :: (Int → Int) → Int
16. Was spricht für eine curryfizierte Deklaration von `binom`? Was ggf für eine uncurryfizierte?
Uncurryfiziert: rein von der Signatur ist klar, wie viele Parameter die Funktion hat
Curryfiziert: Funktion ist partiell auswertbar

17. Wozu dienen die Funktionen `curry` und `uncurry`?
curry macht aus einer uncurryfizierten eine currifizierte
uncurry macht aus einer currifizierten eine uncurryfizierte Funktion

18. Was bedeutet Infix-, was Prä- und Postfixschreibweise für Operatoren?
Infix: 3+4 - dazwischen
Präfix: (+) 3 4 – davor
Postfix: 5!

19. Sind stets alle drei Schreibweisen für einen Operator anwendbar?
Nein, Postfix kommt eher selten vor. (in Haskell gar nicht, manchmal in der Mathematik)

20. Nennen Sie eins oder mehrere Beispiele für Operatoren, die üblicherweise als Präfix-, Postfix- oder Infixoperator gebraucht werden.
Siehe 18.

21. Gibt es in Haskell für jede dieser drei Operatorschreibweisen mindestens je ein Beispiel?
Nein, Postfix nicht sonst siehe 18.

22. Was versteht man in Haskell unter einem Operatorabschnitt?
Partiell ausgewertet Binäroperationen (*2), (2*)

23. Funktionen sind oft nur partiell definiert, z.B. die Fakultäts- und die Fibonacci-Funktion. Was sollte man bei der Definition partiell definierter Funktionen beachten?
Kommutativität

24. Geben Sie je ein Beispiel für eine Funktionsdefinition an, bei der Ihre Hinweise aus der vorigen Aufgabe beachtet worden sind bzw. nicht.
(-2) 1 = -1
(2-) 1 = 1
Bei Addition egal

25. Wozu dient die Abseitsregel in Haskell? Was besagt sie?
Block weiter links → neue Box eröffnet
weiter rechts → Zeile geht weiter

gleich auf → neue Deklaration in gleicher Box

Teil II, Kapitel 4 ‘Typsyonyme, Neue Typen, Typklassen’

1. Wozu dienen Typsynonyme? Wofür sind sie hilfreich?
Neue Namen für bestehende Typen, bieten ausdrucksstarke Funktionssignaturen

2. Was leisten Typsynonyme nicht?
Typsicherheit

3. Geben Sie ein je Beispiel an, die Ihre Antworten auf die vorigen beiden Fragen illustrieren.
type Nat = Int
Nat = -2 liefert keine Fehler

4. Wozu dienen Neue Typen in Haskell? Wie werden sie deklariert?
Erlauben Datentypen zu erstellen, die typsicher sind
newtype Name = Konstruktorname Datentyp

5. Neue Typen führen wie Typsynonyme zu höherer Typsicherheit. Richtig oder falsch? Begründen Sie Ihre Antwort.
Falsch, neue Typen führen zu Typsicherheit, Typsynonyme aber nicht.

6. Typ- und Datenkonstruktorname Neuer Typen dürfen übereinstimmen. Richtig oder falsch?
Richtig

7. Wozu dienen Typklassen in Haskell?
Neue typsichere Typen zu erstellen, indem bereits vorhandene Typen erweitert werden

8. Illustrieren Sie am Beispiel der Typklasse Eq den grundsätzlichen Aufbau von Typklassen in Haskell.
Liefert vordefinierte Operationen == und /=
instance Eq Bool where
    True /= False = False

9.Eq ist eine vordefinierte Typklasse in Haskell. Nennen Sie weitere Beispiele vordefinierter Typklassen.Wozu dienen sie? Show, Read, Ord, Num, Enum

10. Typklassen in Haskell bilden eine Hierarchie. Was ist damit gemeint?
Bestimmte Typklassen sind Untertypen anderer Typklassen

11. Wie ist die hierarchische Beziehung zwischen den Typklassen, die Sie in der vorvorherigen Aufgabe angegeben habe
???

12. Typklassen sehen für ihre Elemente oft sehr viele Funktionen vor. Müssen diese Funktionen immer alle vom Programmierer ausgeführt werden, wenn ein Typ zu einem Elemet der Typklasse gemacht werden soll?
Nein, es kann auch auf Protoimplementierungen zurückgeriffen werden bzw. die minimal complete implementation programmiert werden, der Rest wird automatisch vervollständigt. Für Eq, Ord, Num, Show, Enum, Bounded und Read kann auch mittels deriving eine Instanz gebildet werden, wo das Ausprogrammieren gänzlich entfällt, außer man braucht spezielle Implementierungen.

13. Machen Sie folgende Typen zu einer Instanz der Typklasse Eq:
(a)
```
newtype MyBool = MB Bool
instance Eq MyBool where
      (MB True) == (MB True) = True
      (MB False) == (MB False) = True
```

```
      (MB _) == (MB _) = False
```

(b)
```
newtype MyInt = MI Int
instance Eq MyInt
      (MI x) == (MI y) = x==(y-5) || x ==(y+5) || (x-5)==  || (x+5)==y
```

Dabei soll Gleichheit von `MyBool`-Werten analog zu der von `Bool`-Werten definiert sind;
Gleichheit von `myInt`-Werten jedoch schon gegeben sein, wenn sich zwei Zahlen um höchstens 5 oder weniger unterscheiden;
Ungleichheit, wenn sie sich um 6 oder mehr unterscheiden.

14. Wozu dienen `deriving`-Klauseln in Haskell?
Automatische Instanzbildung

15. Welche der Instanzbildungen aus der vor vorigen Aufgabe hätte man mit gleicher Bedeutung auch mit einer `deriving`-Klausel vornehmen können? Begründen Sie Ihre Antwort.
a), da hier keine speziellen Anforderungen an die Gleichheit gegeben waren

16. Wann lohnt es sich, die in Haskell vordefinierten Typklassen um neue, eigene Typklassen zu erweitern?
Wenn die Funktionalität von vordefinierten Typklassen auf einen konkreteren Typ angepasst werden soll.

17. Geben Sie ein Beipiel für eine selbst definierte Typklasse an, das Ihre Antwort auf die vorige Frage illustriert.
```
class (Eq a) => Aehnlich a where
  aehnlich :: a → a → Bool
```

nur Instanzen von Eq können Instanzen von Aehnlich werden

18. Was versteht man unter
(a) Vererbung
Wenn eine Typklasse eine andere erweitert (Bsp Ord und Eq) und die Implementierungen aus Eq erbt
(b) Überschreiben im Zusammenhang mit Typklassen?
Anpassung der Implementierungen im Untertyp

19. Illustrieren Sie Ihre Antwort auf die vorige Frage anhand eines oder mehrerer geeigneter Beispiele.
Beispielsweise kann in einem Untertyp von Eq auch etwas als gleich erachtet werden, wenn es nur um einen bestimmten Wert abweicht, dann muss die Implementierung überschrieben werden.

20. Vergleichen Sie Haskells Typklassen mit Klassen in Java oder anderen objektorientierten Sprachen. Welche Unterschiede gibt es?
Klassen in OOP dienenn der Strukturierung von Programmen und sind Blaupausen zur Generierung von Werten;
In Haskell dienen sie der Organisation und Verwaltung von Überladung, erhalten Typen durch explizierte Instanzbildung

21. Was verstehen wir unter überladenen Funktionen?
Funktionen die Funktionen von Typklassen verwenden

22. Wie lässt sich der Begriff Überladung weiter verfeinern?
Direkt und indirekt überladene Funktionen

23. Geben Sie Beispiele an, an denen diese Verfeinerung des Überladungsbegriffs gezeigt und erläutert werden kann.

(==) :: Eq a => a → a → Bool direkt
f :: (Num a, Waehrung a) => a → a → a indirekt

24. Woran erkennt man überladene Funktionen?
Am nichtleeren Signaturkontext

25. Woran erkennt man die verfeinerten Überladungsbegriffe für Funktionen
direkt überladene F. sind unmittelbar in einer Typklasse eingeführt
indirekt überladene F. stützen sich auf solche Funktionen ab

Teil II, Kapitel 5 'Algebraische Datentypdeklarationen'

1. Wie erklärt sich die Bezeichnung algebraische Datentypdeklarationen?
Algebraische Datentypen sind zusammengesetzte Datentypen. Diese können mittels data deklariert werden.

2. Warum wird neben dem Konzept Neuer Typen in Haskell mit dem Konzept algebraischer Typennoch eine weitere Möglichkeit geschaffen, selbstdefinierte Datentypen in Haskell-Programmen einzuführen?
Datentypen bieten die Möglichkeit mehrere Konstruktoren zu erstellen, was mehr Flexibilität ermöglicht.

3. Nennen Sie einige Beispiele vordefinierter Aufzählungstypen in Haskell.
Bool, Ordering, () = Trivialer Typ oder Nulltypeltyp

4. Wann spricht man von einem Aufzählungstyp, wann von einem Produkt-, wann von einem Summentyp?
Aufzählungstypen haben endlich viele Werte (vergl. Enums)
Produkttypen (Verbundtypen) haben möglicherweise endlich viele Elemente (vergl. Records in Datenbanken)
Summentypen (Vereinigungstypen) haben ebenfalls möglicherweise endlich viele Werte die sich aus Werten verschiedener Typen zusammensetzen

5. Warum sind die Wahrheitswert `True` und `False` in Haskell großgeschrieben?
Weil Typnamen in Haskell großgeschrieben werden

6. Definieren Sie einen Aufzählungstyp `Sternzeichen`, dessen Werte die 12 Sternzeichen des Jahreskreises.
data Sternzeichen = Wassermann | Fische | Widder | Stier | Zwillinge | Krebs | Löwe | Jungfrau | Waage | Skorpion | Schütze | Steinbock deriving (Eq, Ord, Bounded, Enum, Read, Show)

7. Geben Sie ein Beispiel für einen Produkttyp an.
data Person = P Vorname Nachname

8. Geben Sie ein Beispiel für einen Summentyp an.
data Maybe a = Nothing | Just a deriving …
data Either a b = Left a | Right b deriving …
data Baum = Wurzel Baum Int Baum | Blatt Int

9. Wie ist eine algebraische Datentypdeklaration aufgebaut? Erklären Sie den Aufbau anhand Ihres Beispiels für einen Summentyp aus der vorigen Aufgabe und benennen Sie die vorkommenden Teile.
data Name = Konstruktor1 rekursiverAufruf existierenderTyp rekursiverAufruf | Konstruktor2 existierenderTyp

10. Datenwertkonstruktoren sind Abbildungen. Was sind ihre Argumente, was ihr Resultat? Benutzen Sie ein Beispiel, um Ihre Antwort zu illustrieren.
Ihre Argumente sind der Konstruktorname und die Datentypen des Datenwerts.
data Buerger = B Person Wohnort

Konstruktorname B
Datentypen Person und Wohnort
Das Resultat ist ein Datentyp der Felder der entsprechenden Datentypen besitzt

11. Definieren Sie einen Typ `Baum`, dessen Werte folgende Eigenschaften haben sollen. Blätter sind mit einer Abbildung auf den ganzen Zahlen benannt; innere Knoten sind mit einer ganzen Zahl und einer Zeichenreihe benannt und haben genau drei Teilbäume.
data Baum = Blatt Int | Knoten Int String Baum Baum Baum

12. Geben Sie beispielhaft einige konkrete `Baum`-Werte an.
baum1 = Blatt 12
baum2 = Knoten 4 (Blatt 1) (Knoten 3 (Blatt 5) (Blatt 6) (Blatt 7)) (Blatt 8)

13. Algebraische Datentypen sind strukturierte Datentypen. Warum bietet sich für Funktionsdefinitionen auf algebraischen Datentypen besonders die musterbasierte Funktionsschreibweise an? Illustrieren Sie Ihre Antwort auch anhand eines aussagekräftigen Beispiels.
Nachdem verschiedene Konstruktoren vorhanden sein können, lassen sich diese am besten mit einem Abfragemuster auswerten.
FuncBaum Baum → String
funcBaum Blatt _ = “Blatt”
funcBaum Knoten _ = “Knoten”

14. Definieren Sie eine Funktion `infix`, die einen `Baum`-Wert in Infixordnung durchläuft (d.h. linker Teilbaum, Wurzel, rechter Teilbaum) und als Resultat die Liste der innere Knoten benennenden ganzen Zahlen in der Reihenfolge des Besuchs dieser Knoten liefert, d.h. eine Zahl kommt um so später in der Liste vor, je später ein Knoten von `infix` besucht wird.

infixBaum :: Baum → [Int]
infixBaum (Blatt n) = n
infixBaum (Knoten n s Baum Baum) = Baum ++ [n] ++ Baum

15. Ändern Sie Ihre Funktion aus der vorigen Aufgabe so ab, dass eine Zahl um so früher in der Resultatliste vorkommt, je später der mit ihr benannte innere Knoten besucht wird.
infixBaumLate :: Baum → [Int]
infixBaumLate b = reverse (infixBaum b)
Falls Hilfsfunktionen nicht erlaubt: InfixBaum in where

16. Welche Vorteile bietet Haskells Feldsyntax im Zusammenhang mit algebraischen Datentypen für einen Programmierer?
Geschenkte Selektorfunktionen, wesentlich einfacherer Verarbeitungsfunktionen, sprechende Typdeklarationen

17. Zeigen Sie diese Vorteile anhand eines geeigneten Beispiels auf.
data Meldedaten = Md { vorname :: String, nachname :: String } deriving (Eq, Ord, Show)

18. Welche Möglichkeiten bietet Haskell grundsätzlich, um Typdeklarationen für Leser eines Programms möglichst ‘sprechend’ zu machen?
Kommentierung, Typsynonyme, Feldsyntax (Verbundtypsyntax)

19. Illustrieren Sie Ihre Antwort auf die vorige Frage anhand eines aussagekräftigen Beispiels, falls erforderlich, mehrerer.

```
data Meldedaten = Md String -- Vorname
                     String – Nachname
                  deriving (Eq, Ord, Show)

type Vorname = String
type Nachname = String
```

data Meldedaten = Md Vorname Nachname deriving (Eq, Ord, Show)

ata Meldedaten = Md { vorname :: String, nachname :: String } deriving (Eq, Ord, Show)

20. Zur Wahl stehen folgende zwei Typvereinbarungen:
`type Kilogramm = Float`
`newtype Kilogramm = Kg Float`
Was spricht für, was gegen die Vereinbarung von `Kilogramm` als
(a) Typsynonym? Sprechende Bezeichnung, keine Typsicherheit, Funktionen von Float bereits vorhanden
(b) Neuer Typ? Sprechende Bezeichnung, typsicher, alle benötigten Funktionen müssen selbst implementiert werden

21. Wann immer eine
(a)`type`-Vereinbarung möglich ist, ist auch eine `newtype`-Vereinbarung möglich
i. und umgekehrt. Da newtype einen Konstruktor zulässt und auch bei Type nur ein Wert pro Namen vergeben werden kann.
ii. aber nicht umgekehrt.
(b) `newtype`-Vereinbarung möglich ist, ist auch eine `data`-Vereinbarung möglich
i. und umgekehrt.
ii. aber nicht umgekehrt. Da bei newtype nur ein Konstruktor erlaubt ist. Bei data ist dies schon erlaubt.
Richtig oder falsch? Begründen Sie jeweils Ihre Antwort.

22. Definieren Sie einen Datentyp `Haustier`, wobei jeder `Haustier`-Wert den Namen, den Lieblingsplatz (Sofa, Fensterbank, Bett, Futternapf, Feld-Wald-und-Wiese) und Hoert-auf (Zuruf, manchmal, garnicht) Information angibt mittels
(a)`type`
```
type HaustierName = String
type HaustierLieblingsplatz = String
type HaustierHoert-auf = String
type Haustier = (HaustierName, HaustierLieblingsplatz, HaustierHoert-auf)
```

(b)`newtype`
```
newtype HaustierName = N String
newtype HaustierLieblingsplatz = LP String
newtype HaustierHoert-auf = HA String
newtype Haustier = H (HaustierName, HaustierLieblingsplatz, HaustierHoert-
auf)
```

(c)`data`
```
type HaustierName = String
type HaustierLieblingsplatz = String
type HaustierHoert-auf = String
data Lieblingsplatz = Sofa | Fensterbank | Bett | Futternapf | Feld-Wald-und-Wiese
data Hoert-auf = Zuruf | manchmal | garnicht
data Haustier = H (Haustiername, Lieblingsplatz, Hoert-auf)
```

23. Welche Vor- und Nachteile haben die drei Vereinbarungsvarianten aus der vorigen Aufgabe?
type: nicht typsicher, aber einfach, sprechende Namen, alle Funktionen von String vorhanden
newtype: typsicher, unverwechselbare neue Identität, alle nötigen Funktionen müssen selbst implementiert werden
data: Funktionen müssen selbst implementiert werden, typsicher, sprechende Bezeichnungen, zusammenfassen von Werten möglich

24. Haben Sie für die `data`-Variante der `Haustier`-Deklaration von der Feldsyntax Gebrauch gemacht? Wenn ja, geben Sie eine entsprechende `data`-Deklaration ohne Feldsyntax an; wenn nein, dann mit.

25. Vergleichen Sie die feldsyntaxnutzende und -nichtnutzende `data`-Deklarationen für `Haustier` miteinander. Welche Vor- und Nachteile haben Sie im Vergleich zueinander

Teil II, Kapitel 6 'Muster und mehr'

1. Wie sieht der Musterausdruck (oder kürzer das Muster)
(a) der leeren Liste []
(b) einer nichtleeren Listeaus? (n:ns)

2. Welche Ausdrücke passen auf folgende Muster(ausdrücke)? Geben Sie einige Beispiele an:
(a)`[]`
(b)`(x:[])`
(c)`(x:xs)`
(d)`(x:y:[])`
(e)`(x:y:xs)`

3. Gibt es Muster(ausdrücke), auf die genau Listen folgender Werte passen, die Listen genau dieser Werte beschreiben? Listen mit mindestens einem Element, deren
(a) erstes (1:ns)
(b) letztes Element eine 1 ist? (ns:1:[]) für nur ein Element vor dem letzten, sonst gibt es keinen solchen Ausdruck, da nicht gesagt werden kann wie viele Elemente vor dem letzten vorhanden sind
Geben Sie den entsprechenden Musterausdruck an oder erläutern Sie, warum es keinen entsprechenden Musterausdruck gibt.

4. Schreiben Sie eine musterdefinierte Haskell-Rechenvorschrift `laenge` zur Berechnung der Länge von Listen des Typs:
`data Liste = Leer | Kopf (String,Int) Liste`
Geben Sie dabei auch die Signatur der Funktion `laenge` an.

laenge :: Liste → Int
laenge Leer = 0
laenge (Kopf (s, n) list) = n + (laenge list)

5. Welche Muster(ausdrücke) gibt es für die Beschreibung von
(a) Wahrheitswerten Konstanten:True, False Joker _
(b) Gleitkommazahlen Konstanten, Variablen, Joker
(c) Zeichen Konstanten, Variablen, Joker
(d) Tupelwerten Konstanten, Variablen, Joker, Mischung
Welche Werte sind von diesen Mustern jeweils beschrieben?

6. Welche Namens-/Bezeichnungskonventionen gelten für die Muster(ausdrücke) aus der vorigen Aufgabe?

7. Was sind Konstruktormuster? Illustrieren Sie Ihre Antwort auch anhand eines Beispiels.
(_:xs) Konstruktormuster (Klammern) für Liste
Oder data Ausdruck = Opd Zett | Add Ausdruck
Add und Opd Konstruktormuster

8. Was sind als-Muster? Wozu sind sie nützlich?
Das Pattern kann einfacher ausgedrückt werden indem mit dem Schlüsselwort @ ein Name für einenlangen Ausdruck erzeugt wird, machen Ausdrücke übersichtlicher und einfacher

9. Welches Symbol dient zur Bezeichnung von als-Mustern?
@

10. Geben Sie einige konkrete Beispiele für als-Muster an.
paar@(x,y)
tripel@(x,y,z)

11. Welche Definitionen der ersten 10 Haskell-Beispiele aus Kapitel 1.1.1 machen von Mustern Gebrauch?
(a) Listen Sie diese Beispiele und die verwendeten Muster auf.
Fakultät (iii) - Integerwerte, Längenberechnung von Listen – Listenwerte, Umkehren von Zeichenreihen – Strings, Transformieren von Listen – Listenwerte, Binomialkoef. - Integerwerte, Sieb des Eratosthenes - Listenwerte
(b) Beschreiben Sie informell mit Worten, welche Werte von diesen Mustern beschrieben werden,welche Werte auf sie passen? Siehe oben

12. Was haben Muster und Selektorfunktionen miteinander zu tun?
Selektoren und Muster haben viel gemeinsam
Beispielsweise liefern für ein Paar (a,b)
das Muster (_,b) = b und der Selektor snd das gleiche Ergebnis

13. Illustrieren Sie den Zusammenhang von Mustern und Selektorfunktionen aussagekräftig anhand eines Beispiels.
Siehe 12.

14. Was sind i.a. Vorteile musterbasierte Definitionen?
Knappe gut lesbare Spezifikationen

15. Was sind mögliche Nachteile musterbasierter Definitionen, wenn Programme weiterentwickelt werden, wenn sich Parameter von Funktionen ändern, neu hinzukommen oder bisherige wegfallen?
In diesen Fällen müssen oft alle Muster überarbeitet werden und neue Fälle hinzugenommen werden was schlecht wartbar ist. Kann auch zu subtilen Fehlern führen.

16. Welcher Schreibweise und welchem Bildungsprinzip aus der Mathematik ist Listenkomprehension nach gebildet?
Mengenbildungsoperation

17. Beschreiben Sie als Listenkomprehensionen: Die Liste der
(a) geraden ganzen Zahlen. [x | x ← [2,4..]]
(b) geraden ganzen Zahlen, die eine 2er-Potenz sind. [2^x | x ← [1,2..]]
(c) Nachfolger der geraden ganzen Zahlen, die eine 2er-Potenz sind. [2^x +1| x ← [1,2..]]
(d) Quadrate der Nachfolger der geraden ganzen Zahlen, die eine 2er-Potenz sind. [(2^x +1)^2| x ← [1,2..]]

18. Was versteht man im Zusammenhang mit Listenkomprehension unter einem
(a) Generator <-
(b) Transformator [id n | n ← [1,2..]] id
(c) Filter [x | x ← head [1,2,3]] head
(d) Test [x | x ← [1,2,3], x < 2] <
Illustrieren Sie Ihre Antworten anhand aussagekräftiger Beispiele.

19. Sind
(a)`(:) Konstruktor`
(b)`(++) Listenoperator`
Listenkonstruktoren oder Listenoperatoren?

20. Was unterscheidet Listenkonstruktoren und Listenoperatoren?

Es gibt nur einen einzigen Listenkonstruktor (:), aber mehrere Operatoren. Konstruktoren führen zu eindeutigenn Darstellungen, Operatoren nicht

21. Listenkonstruktoren und Listenoperatoren dürfen in gleicher Weise in Mustern verwendet werden. Richtig oder falsch? Begründen Sie Ihre Antwort.
Nein, nur Konstruktoren dürfen in Musterausdrücken verwendet werden, da Operatoren nicht eindeutig sind.

22. Wie werden das
(a) Variablenmuster n = irgendwas
(b) Jokermuster _ = irgendwas
in Haskell geschrieben?

23. Welche Werte eines Typs beschreiben das
(a) Variablenmuster? ausgewählte
(b) Jokermuster? Alle

24. Wie unterscheiden sich Variablen- und Jokermuster in der Verwendbarkeit?
Jokermuster lässt alle Werte eines Typs zu, wohingegen das Variablenmuster auf einen bestimmten Wert abzielt.

25. Warum soll man sich für das Joker- und nicht das Variablenmuster entscheiden, wenn beide anwendbar sind?
Da das Jokermuster Polymorphie unterstützt und flexibler ist.

Präludium, Teil I

1. Gegeben ist die Funktion:
f :: Int -> Int
f n = if n == 1 then 1 else (f (n-1)) + n^2
(a) Welchen Wert hat der Aufruf f 3? 14
(b) Wie verhält sich f beim Aufruf mit dem Wert (−3)? non exhaustive pattern error
(c) Was berechnet f für Argumentwerte größer oder gleich null? Summe aller Quadratwerte der Zahlen von 1 bis n
(d) Wie verhält sich f für Aufrufe mit echt negativen Argumentwerten? Non exhaustive pattern error
(e) Werten Sie den Aufruf f 2 Schritt für Schritt aus. Ein Schritt ist eine Expansion, eine Operatorauswertung oder die Wahl des Ausdrucks im dann- bzw. sonst-Fall der Fallunterscheidung nach Auswertung deren Bedingung zu wahr bzw. falsch. Die Reihenfolge der Rechenschritte ist nicht vorgegeben.
f 2 = if 2 == 1 then 1 else (f(2-1)) + 2^2
f 2 = if 2 == 1 then 1 else (f(2-1)) + 4
f 2 = if False then 1 else (f(2-1)) + 4
f 2 = if False then 1 else (f(1)) + 4
f 2 = if False then 1 else (f 1 = if 1 == 1 then 1 else (f(1-1)) + 2^1) + 4
f 2 = if False then 1 else (f 1 = if True then 1 else (f(1-1)) + 2^1) + 4
f 2 = if False then 1 else (f 1 = if True then 1 else (f(1)) + 2^1) + 4
f 2 = if False then 1 else (f 1 = if True then 1 else (f(1)) + 2) + 4
f 2 = if False then 1 else 1) + 4
f 2 = 1+ 4
f 2 = 5

(f) Wenden Sie eine andere Reihenfolge der Rechenschritte als in der vorigen Aufgabe an, um den Wert des Ausdrucks f 2 zu berechnen.
f 2 = if 2 == 1 then 1 else (f(2-1)) + 2^2
f 2 = if False then 1 else (f(2-1)) + 2^2
f 2 = if False then 1 else (f(2-1)) + 4
f 2 = if False then 1 else (f(1)) + 4
f 2 = if False then 1 else (f 1 = if 1 == 1 then 1 else (f(1-1)) + 2^1) + 4
f 2 = if False then 1 else (f 1 = if True then 1 else (f(1-1)) + 2^1) + 4

f 2 = if False then 1 else 1) + 4
f 2 = 1 + 4
f 2 = 5

2. Was passiert bei der Ausführung eines
(a) imperativen – Befehle werden hintereinander ausgeführt, Anweisungen ändern Ausdrücke
(b) funktionalen Programms? Funktionen legen Ausdrücke fest

3. Was sind wichtige Kennzeichen imperativer/objektorientierter und funktionaler Programmierung?
Anweisungen legen Ausdrücke fest, Seiteneffekte erwünscht

7. In präskriptiven Sprachen steht das ‘wie’ im Vordergrund, in deklarativen Sprachen das ‘was’. Was ist damit gemeint?
Präskriptive Sprachen legen fest wie es passieren soll. (Speicher festlegen, auf bestimmte Speicherstelle schreiben,...)
Deklarative Sprachen sagen was passieren muss aber nichts über das wie (Speicher,...) wird abstrahiert.

8. Welche große Gruppen gibt es innerhalb der Familien
(a) präskriptiver – objektorientiert - Java
(b) deklarativer Sprachen? Logisch und funktional, Prolog und Haskell
Was sind typische Beispiele von Sprachen dieser Gruppen?

9. Das Haskell-Vokabular, die Zahl seiner Schlüsselwörter ist sehr gering. Richtig oder falsch? Wie viele ungefähr?
Richtig, 20 ?

11. Was ist der sog.Standard Prelude von Haskell? Was beinhaltet er?
Standard module das alle wichtigen Standardfunktionalitäten von Haskell beinhaltet.

12. In seinem Artikel Haskell: A Language for Modern Timesstellt Mihai Maruseac fest:“LearningHaskell opens one’s mind [...], which might produce clearer and shorter implementations.”Wie gelangt er zu dieser Einschätzung?
Funktionale Programmierung braucht nicht so viel Code da nur das wie beschrieben werden muss.

13. Welche Sprachkonstrukture sind für imperative, objektorientierte Sprachen zentral, unverzichtbar,fehlen aber in funktionalen Sprachen? Schleifen

Teil III, Kapitel 7 ‘Rekursion’

1. Warum ist Rekursion für funktionale Sprachen (so viel) wichtiger als z.B. für objektorientierte Sprachen? Weil es keine Schleifen gibt und sonst keine Möglichkeit Wiederholungen umzusetzen.

2. Worum geht es im Problem der Türme von Hanoi? Scheiben auf eine andere Stange umzuschichten

4. Welche Rekursionsmuster unterscheiden wir auf
(a) mikroskopischer – repetitive Rekursion (pro Zweig max. 1 rekursiver Aufruf als äußerster Aufruf), lineare Rekursion (pro Zweig max. 1 rekursiver Aufruf nicht als äußerste Operation), baumartige (kaskadische) Rekursion (pro Zweig mehrere Aufrufe nebeneinander), geschachtelte Rekursion (rekursive Aufrufe enthalten wieder Aufrufe)
(b) makroskopischer Ebene – indirekte Rekursion zwei oder mehrere Funktionen rufen sich wechselseitig auf

5. Welchen Rekursionstyp haben die Implementierungen von:
(a) quickSort (Kap. 7.1.1) - baumartig/kaskadenartig

(b) hanoi (Kap. 7.1.2) - baumartig/kaskadenartig
(c) binom, binom′ - gar nicht bezogen binom, binom′′ - baumartig/kaskadenartig(Kap. 1.1.1)
(d) zip (Kap. 1.1.1) – nicht in 1.1.1???
(e) ggt (Kap. 1.1.1) - repetitive
(f) mod (Kap. 1.1.1) – repetitive

6. Was ist Rechnen auf Parameterposition? Wozu dient es?
Ergebnis wird hierbei in einem oder verteilt über mehrere zusätzlich Parameter sukzessive akkumuliert – Beispiel fibonacci (merkt sich vorherige Werte)
→ effizienter Berechnung

7. Was ist ein Aufrufgraph? Wozu sind Aufrufgraphen nützlich?
Enthält für jede in S deklarierte Rechenvorschrift einen Knoten mit Namen als Beschriftung, Kanten von f nach g wenn f g aufruft
hilft Struktur von Systemen von Rechenvorschriften übersichtlich zu veranschaulichen

8. Wie sind Aufrufgraphen aufgebaut, woraus bestehen sie?
Siehe 7

9. Geben Sie den Aufrufgraph des folgenden Haskell-Skripts an:

```
fibSchritt :: (Integer,Integer) -> (Integer,Integer)
fibSchritt (m,n) = (n,m+n)

fibPaar :: Integer -> (Integer,Integer)
fibPaar n =
        | n == 0 = (0,1)
        | otherwise = fibSchritt (fibPaar (n-1))

fib' :: Integer -> Integer
fib' n = fst (fibPaar n)

fst :: (a,b) -> a
fst (x,y) = x
```

10. Was kann man am Aufrufgraphen aus der vorigen Teilaufgabe ablesen?
Wechselseitige Rekursion

11. Gibt es weitere Eigenschaften, die man aus Aufrufgraphen ablesen kann, aber nicht anhand des obigen Beispiels? Welche?
Allgemein:
- direkte Rekursion (Selbstreferenz)
- wechselseitige Rekursion (Kreis)
- direkte hierarchische Abbildung (Kante von g nach f aber nicht umgekehrt)
- indirekte hierarchische Abstützung (Pfad von g nach f aber nicht umgekehrt)
- indirekte wechselseitige Abstützung (Pfad von g nach f und f nach g)
- Unabhängigkeit

12. Geben Sie ein Beipiel an, in dem sich diese Eigenschaften im Aufrufgraphen ablesen und demonstrieren lassen. Siehe 11.

13. Was sind obere Schranken einer Funktion? Was untere?
Werte die die Laufzeit mit einfachen Ausdrücken abschätzen

14. Sind obere und untere Schranken einer Funktion eindeutig? Begründen Sie Ihre Antwort.
Nein, es beispielsweise kann sowohl n^2 als auch n^3 eine Schranke sein

15. Wie kann man sich obere und untere Schranken von Funktionen graphisch veranschaulichen? Als Funktionsgraphen

17. Wie, mit welchen Symbolen drückt man diese Beziehungen zweier Funktionen f und g aus? Klein und groß O, Omega, Teta

18. Wie heißen diese Symbole? Nach wem sind sie benannt? Landausymbole
19. Was ist eine Komplexitätsklasse?
Gruppen in mit deren der Aufwand von Problemen klassifiziert wird

20. Geben Sie einige Beispiele wichtiger Komplexitätsklassen an.
P, NP, EXPTime, EXPSpace, NL, polynomiell, exponentiell

21. Wie verändert sich für Funktionen dieser Komplexitätsklassen der Rechenaufwand, wenn sich die Eingabe um den Faktor
(a) 10
(b) 1.000
(c) 1.000.000 vergrößert?
→ in beliebige polynomielle oder expontielle Funktionen einsetzen

22. Sie haben für ein Problem ein exponentielles Lösungsverfahren. Empfiehlt es sich, den schnellsten am Markt erhältlichen Rechner zu kaufen? Begründen Sie Ihre Antwort.
Nein, schnellerer Algorithmus bringt mehr

24. Nennen Sie je ein Beispiel für eine Funktion mit
(a) konstanter n = 1
(b) linearer func n = n + func n-1
(c) quadratischer n = n + func n-1 + func n-1
(d) exponentieller
Komplexität.

25. Warum betrachtet man asymptotische Schranken als Maß für die Berechnungskomplexität von Funktionen anstatt deren konkrete und deshalb exakten Aufwandsfunktionen
Weil auf große Eingabegrößen konstante Faktoren kaum Unterschiede machen und diese durch bessere Hardware oder verschiedene Hardware oft größer sind.

Teil III, Kapitel 8 ‘Auswertung einfacher Ausdrücke’

1. Was ist mit
(a) Expandieren – das Einsetzen der Funktionsbeschreibung für den Funktionsnamen; der Ausdruck wird dadurch länger (expandiert) – Einsetzen von Funktionstermen, -Aufrufen
(b) Simplifizieren von Ausdrücken gemeint? Ausrechnen/Auswerten von Ausdrücken, wodurch Ausdrücke zusammengefasst werden können – Erstellen funktionsfreier Ausdrücke

2. Geben Sie je ein Beispiel für einen
(a) Expansions- : f 1 = 3 + 1 – 7 für f n = 3 + n - 7
(b) Simplifikationsschritt: 4+5 = 9
eines Ausdrucks an.

3. Was ist ein Funktionsterm?
Ein Funktionsterm ist eine Funktion mit den für die Auswertung benötigten Parameterwerten

4. Welche Freiheitsgrade gibt es bei der Auswertung von Funktionstermen?
Argumentauswertung vor (applikativ) oder nach der Expansion (normal)

5. Illustrieren Sie diese(n) Freiheitsgrad(e) an (je) einem Beispiel.

2 * fun (2-1)
applikativ: 2 * fun 1
normal 2 * if (2-1) == 0 then 1 else 7

6. Was legen der
(a) applikative – Argumente werden vor Expansion ausgewertet
(b) normale Auswertungsstil – Argumente werden nach Expansion ausgewertet
für die Auswertung von Funktionstermen fest?

7. Vervollständigen Sie die
(a) applikative
S: (natSum 1 + 2) + 3
E: ((if 1 == 0 then 0 else natSum (1-1) + 1) + 2) + 3
S: ((if False then 0 else natSum (1-1) + 1) + 2) + 3
S: ((natSum (1-1) + 1) + 2) + 3
S: ((natSum 0 + 1) + 2) + 3
E: (((if 0 == 0 then 0 else natSum (0-1) + 0) + 1) + 2) + 3
S: (((if True then 0 else natSum (0-1) + 0) + 1) + 2) + 3
S: ((0 + 1) + 2) + 3
S: (1 + 2) + 3
S: 3 + 3
S: 6

(b) normale Auswertung des Funktionsterms `natSum 3` aus Beispiel 5 in Kapitel 8.2
(E): (if ((3-1)-1) == 0 then 0 else natSum ((3-1)-1)-1 + ((3-1)-1)) + (3-1) + 3
(S): (if (2-1) == 0 then 0 else natSum ((3-1)-1)-1 + ((3-1)-1)) + (3-1) + 3
(S): (if 1 == 0 then 0 else natSum ((3-1)-1)-1 + ((3-1)-1)) + (3-1) + 3
(S): (if 1 == 0 then 0 else natSum ((3-1)-1)-1 + ((3-1)-1)) + (3-1) + 3
(S): (if False then 0 else natSum ((3-1)-1)-1 + ((3-1)-1)) + (3-1) + 3
(S): natSum (((3-1)-1)-1) + ((3-1)-1) + (3-1) + 3
(E): (if (((3-1)-1)-1) == 0 then 0 else natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if ((2-1)-1) == 0 then 0 else natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if (1-1) == 0 then 0 else natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if 1 == 0 then 0 else natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if False then 0 else natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): natSum (((3-1)-1)-1)-1) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(E): (if (((3-1)-1)-1)-1) == 0 then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if ((2-1)-1)-1) == 0 then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if ((1-1)-1) == 0 then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if (1-1) == 0 then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if 0 == 0 then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): (if True then 0 else natSum ((((3-1)-1)-1)-1)-1) + (((3-1)-1)-1)-1)) + ((3-1)-1)-1)) + ((3-1)-1) + (3-1) + 3
(S): 0 + ((3-1)-1)-1 + ((3-1)-1) + (3-1) + 3
(S): 0 + ((2-1)-1) + ((3-1)-1) + (3-1) + 3
(S): 0 + (1-1) + ((3-1)-1) + (3-1) + 3
(S): 0 + 0 + ((3-1)-1) + (3-1) + 3
(S): 0 + 0 + (2-1) + (3-1) + 3
(S): 0 + 0 + 1 + (3-1) + 3
(S): 0 + 0 + 1 + 2 + 3
(S): 0 + 1 + 2 + 3
(S): 1 + 2 + 3

(S): 3 + 3
(S): 6

8. Gegeben sind die musterbasierten Definitionen der Funktionen `fun91` und `fun91´`:

```
fun91 :: Integer -> Integer
fun91 n
      | n > 100 = n – 10
      | n <= 100 = fun91 (fun91 (n+11))

fun91’:: Integer -> Integer
fun91’ n =if n > 100 then n - 10 else fun91’ (fun91’ (n+11))
```

Werten Sie die vier Funktionsterme

(a)`fun91 101`

applikativ:

```
fun91 101
(E)   | 101 > 100 = 101 – 10
      | 101 <= 100 = fun91 (fun91 (101+11))
(S)   | True = 101 – 10
      | False = fun91 (fun91 (101+11))
(S)   101 – 10
(S)   91
```

normal:

```
fun91 101
(E)   | 101 > 100 = 101 – 10
      | 101 <= 100 = fun91 (fun91 (101+11))
(S)   | True = 101 – 10
      | False = fun91 (fun91 (101+11))
(S)   101 – 10
(S)   91
```

(b)`fun91 100`

applikativ:

```
fun91 100
(E)   | 100 > 100 = 100 – 10
      | 100 <= 100 = fun91 (fun91 (100+11))
(S)   | False = 101 – 10
      | True = fun91 (fun91 (100+11))
(S)   fun91 (fun91 (100+11)
(S)   fun91 (fun91 111)
(E)   fun91 (fun91 111
            | 111 > 100 = 111 – 10
            | 111 <= 100 = fun91 (fun91 (111+11))
)
(S)   fun91 (fun91 111
            | True = 111 – 10
            | False = fun91 (fun91 (111+11))
)
(S)   fun91 (fun91 111
            | True = 111 – 10
            | False = fun91 (fun91 (111+11))
)
(S) fun91 (111 – 10)
(S) fun91 101
weiter siehe oben
```

(c)`fun91´101`

normal/applikativ:

```
fun91’ 101 =
(E) if 101 > 100 then 101 - 10 else fun91’ (fun91’ (101+11))
(S) if True then 101 - 10 else fun91’ (fun91’ (101+11))
(S) 101 – 10
(S) 91
```

(d)fun91′100

normal:

```
fun91’ 101 =
(E) if 100 > 100 then 100 - 10 else fun91’ (fun91’ (100+11))
(S) if False then 100 - 10 else fun91’ (fun91’ (100+11))
(S) fun91’ (fun91’ (100+11))
(E) fun91’ (if (100+11) > 100 then (100+11) - 10 else fun91’ (fun91’
(100+11)+11))
(E) if (if (100+11) > 100 then (100+11) - 10 else fun91’ (fun91’
(100+11)+11)) > 100 then (if (100+11) > 100 then (100+11) - 10 else fun91’
(fun91’ (100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100 then
(100+11) - 10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if (if 111 > 100 then (100+11) - 10 else fun91’ (fun91’ (100+11)+11)) >
100 then (if (100+11) > 100 then (100+11) - 10 else fun91’ (fun91’
(100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100 then (100+11) -
10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if (if True then (100+11) - 10 else fun91’ (fun91’ (100+11)+11)) > 100
then (if (100+11) > 100 then (100+11) - 10 else fun91’ (fun91’ (100+11)+11))
– 10 else fun91’ (fun91’ ((if (100+11) > 100 then (100+11) - 10 else fun91’
(fun91’ (100+11)+11))+11))
(S) if ((100+11) - 10) > 100 then (if (100+11) > 100 then (100+11) - 10 else
fun91’ (fun91’ (100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100
then (100+11) - 10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if (111 - 10) > 100 then (if (100+11) > 100 then (100+11) - 10 else
fun91’ (fun91’ (100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100
then (100+11) - 10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if 101 > 100 then (if (100+11) > 100 then (100+11) - 10 else fun91’
(fun91’ (100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100 then
(100+11) - 10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if True then (if (100+11) > 100 then (100+11) - 10 else fun91’ (fun91’
(100+11)+11)) – 10 else fun91’ (fun91’ ((if (100+11) > 100 then (100+11) -
10 else fun91’ (fun91’ (100+11)+11))+11))
(S) if 111 > 100 then (100+11) - 10 else fun91’ (fun91’ (100+11)+11)) – 10)
(S) if True then (100+11) - 10 else fun91’ (fun91’ (100+11)+11)) – 10)
(S) (100+11) – 10
(S) 111 – 10
(S) 101
```

für je einige wenige (!) Expansionsschritte(i) applikativ(ii) nomal aus.

9. Werte den Funktionsterm fac 2

(i) applikativ

```
fac 2 =
(E) 2 * fac (2-1)
(S) 2 * fac 1
(E) 2 * (1 * fac (1-1))
(S) 2 * (1 * fac 0)
(E) 2 * (1 * 1)
(S) 2 * 1
(S) 2
```

(ii) normal

```
fac 2 =
(E) 2 * fac (2-1)
```

(E) 2 * (fac (2-1) == 0 = 1
fac True = (2-1) * fac ((2-1)-1)
)

für die musterbasierte Defintion der Fakultätsfunktion aus:

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n-1)
```

10. Führen Sie auch je einige Berechnungsschritte durch, die zur Lösung von
(a) Übungsaufgabe 8.2.2
(b) Übungsaufgabe 8.2.3
(c) Übungsaufgabe 8.2.6 aus Kapitel 8.2 nötig sind.

11. Was ist eine maximale Berechnungsfolge?
Eine nicht mehr verlängerbare Folge von Simplifikations- und Expansionsschritten

12. Sind maximale Berechnungsfolgen stets endlich? Begründen Sie Ihre Antwort.
Nein, da es Berechnungen gibt, die nicht endlich sind. Wird für eine solche Berechnung versucht eine maximale Berechnungsfolge zu erstellen, endet dieser Versuch nie.

13. Was sagt das nach Alonzo Church und John Barkley Rosser benannte Theorem über terminierende maximale Berechnungsfolgen aus?
Jede terminierende maximale Berechnungsfolge endet mit demselben Wert.

14. Warum ist das Theorem von Church und Rosser für die Implementierung funktionaler Programmiersprachen wichtig?
Weil dadurch die Wohldefinierbarkeit der Semantik eine Grundlage erhält.

15. Warum ist das Theorem von Church und Rosser auch für die Implementierung funktionaler (und auch nichtfunktionaler!) Programmiersprachen wichtig?
Siehe 14.

Teil III, Kapitel 9 ‘Programmentwicklung, Programmverstehen’

2. Welches Vorgehen schlägt Graham Hutton bei der Entwicklung von Programmen vor? Erläutern Sie knapp, aber präzise die verschiedenen Phasen seines Vorgehensvorschlags.
1) Lege Datentypen fest
2) Führe relevante Fälle auf
3) Lege die Lösung für die einfachen Basisfälle fest
4) Lege die Lösung für die übrigen Fälle fest
5) Verallgemeinere und vereinfache das Lösungsverfahren

7. Gegeben ist die Deklaration:

```
mapWhile :: (a -> b) -> (a -> Bool) -> [a] -> [b]
mapWhile f p [] = []
mapWhile f p (x:xs)
      | p x = f x : mapWhile f p xs
      | otherwise = []
```

Was ist die Bedeutung von `mapWhile`? Erklären Sie knapp, aber gut nachvollziehbar, was sich aus den einzelnen Zeilen der Deklaration von `mapWhile` herauslesen lässt und wie die Auswertung der Funktion vorgeht.
MapWhile wendet die Funktion f solange auf das erste Element der Liste an, solange p x wahr ist.

8. Das erste und zweite Argument von `mapWhile` sind funktional, sind Funktionen. FÜr das zweite Argument gibt es noch eine genauere Bestimmungsbezeichnung als Funktion. Welchen? Gibt es noch weitere Bezeichnungen für diesen Begriff?

Wahrheitswertfunktion

9. Gegeben ist die Deklaration:

```
filterMap :: [a] -> (a -> Bool) -> (a -> b) -> [b]
filterMap [] _ _ = []
filterMap (x:xs) p f
      | p x = (f x) : filterMap xs p f
      | True = filterMap xs p f
```

Was ist die Bedeutung von `filterMap`? Erklären Sie knapp, aber gut nachvollziehbar, wie die Auswertung der Funktion vorgeht.
FilterMap nimmt eine Liste und fügt ein Element auf das die Funktion f angewendet wird zu r Ergebnisliste hinzu falls p x True liefert und fährt mit dem Rest der Liste gleich fort. Ansonsten wird mit dem Rest der Liste fortgefahren.

10. Welche Laufzeitkomplexität (linear, quadratisch, logarithmisch,... in der Zahl, in der Größe von...) haben
(a) `mapWhile`? linear
(b)`filterMap`? linear

Teil I – II, Verschiedene Kapitel

1. Wodurch unterscheiden sich die Typen `Int` und `Integer` voneinander?
Int hat ist auf einen bestimmten Bereich definiert. Integer bietet die Möglichkeit potentiell unendliche Ganzzahlen zu speichern

2. Welche vordefinierten Operatoren gibt es auf Wahrheitswerten? Nennen Sie einige Beispiele.
&&, ||, not

3. Tupelwerte sind i.a. heterogen, Listenwerte homogen. Was bedeutet das?
In Tupeln können Werte unterschiedlicher Typen zusammengefasst werden, in Listen nur Werte gleicher Typen

4. Welche Liste repräsentiert der Ausdruck`[1,3..10]`?
Liste aller ungeraden Zahlen kleiner 10

5. Welches ist die Grunddarstellung des ‘syntaktisch gezuckerten’ Ausdrucks`[1..4]`?
1 : (2 : (3 : (4 : [])))

6. Welchen Wert haben folgende Ausdrücke?
(a) `map (+1) [-2..2] -> [-1..3]`
(b) `map (1+) [-2..2] -> [-1..3]`
(c) `map inc [-2..2] where inc n = n+1 -> [-1..3]`
(d) `map inc [-2..2] where inc =\n -> n+1 -> [-1..3]`
(e) `map inc [-2..2] where inc = (+1) -> [-1..3]`

7. Welche der folgenden Ausdrücke sind gültig, welche nicht?
(a)`["sin","cos","tan"] = gültig`
(b)`["sin",cos,"tan"] = nicht gültig (nicht homogen)`
(c)`[sin,cos,tan] = kann gültig sein (je nach dem welchen Typ sin,cos,tan haben)`
(d)`["sin",[],"tan"] = gültig, da [] als leere Charliste interpretiert wird`
(e)`["sin","","tan"] = gültig`
(f)`["sin"," ","tan"] = gültig`
(g)`["sin",‘ ’,"tan"] = ungültig da ‘ ’ kein String ist bzw. kein [Char]`
(h)`[‘sin’,‘ ’,‘tan’] = ungültig weil sin bzw. tan kein Char`
Geben Sie für gültige Ausdrücke den Typ an; für nichtgültige Ausdrücke eine knappe Begründung, warum sie nicht gültig sind.

8. Was sind Listenaufzählungsausdrücke? Illustrieren Sie Ihre Antwort mit einigen Beispielen.
Generieren Listen über Typen geordneter aufzählbarer Werte automatisch
[2..10], [‘a’,’b’..’z’],...

9. Wie sehen die Muster(ausdrücke) aus, die Listen genau der folgenden Werte beschreiben, die ‘daraufpassen’ ?
(a) Listen mit genau
i. keinem - []
ii. einem - (x:[])
iii. Zwei – [x,y]
iv. drei – [x,y,z]
Element(en)?
(b) Listen mit mindestens
i. einem – (x:xs)
ii. zwei – (x:y:xs)
iii. Drei – (x:y:z:xs)
Element(en)?

10. Gibt es Muster(ausdrücke), auf die genau Listen folgender Werte passen, die Listen genau dieser Werte beschreiben?
(a) Listen mit höchstens
i. einem
ii. zwei
iii. Drei
Element(en). - Geht nicht
(b) Listen mit einer beliebigen Zahl von Elementen - _
Geben Sie den entsprechenden Musterausdruck an oder erläutern Sie, warum es keinen entsprechenden Musterausdruck gibt.

11. Wozu dienen `let`- und `where`-Deklarationen? Wodurch unterscheiden Sie sich? Geben Sie je einillustrierendes Beispiel an.
ermöglichen Funktionsdefinitionen in anderen Funktionen zu erstellen. Let erstellt die Funktion vor der restlichen Deklaration where danach

```
func n = maxi n
        where maxi = maximum x

func n = let maxi  = maximum x
                 in maxi n
```

12. Zeigen Sie, dass die linke und rechte Seite der definierenden Gleichung von `curry` denselben Typ besitzen:
`curry :: ((a,b) -> c) -> a -> b -> c`
`curry f x y = f (x,y)`
f x y
f :: ((a,b) → c), x :: a, y :: b; = f (x,y) :: (a,b) :: c

13. Ergänzen Sie Klammern in der obigen Definition von `curry` so, dass Signatur und definierende Gleichung jeweils vollständig, aber nicht überflüssig geklammert sind.
`curry :: ((a,b) -> c) -> a -> b -> c`

14. Wie sind `newtype`-Deklarationen aufgebaut? Erklären Sie die einzelnen Bestandteile einer `newtype`-Deklaration und wie sie heißen?
newtype name = Konstruktor Feldtyp

15. Algebraische Datentypen in Haskell erlauben neue numerische Typen einzuführen, z.B. für komplexe Zahlen. Um andere, nicht vordefinierte Datentypen einzuführen, braucht man Haskells Konzept der Neuen Typen. Richtig oder falsch. Begründen Sie Ihre Antwort.

Nein, algebraische Datentypen lassen auch komplett neue Typen zu,

18. Was meint Rechnen mit Funktionen?
Funktionen erzeugen mit Funktionen höhrerer Ordnung neue Funktionen

21. Was sind Protoimplementierungen einer Typklasse? Wozu sind sie nützlich?
Sind Defaultimplementierungen der Funktionen einer Typklasse, wenn Instanzen von Typklassen gebildet werden müssen nicht alle Funktionen implementiert werden

22. Wofür gibt es eine Protoimplementierung in der Typklasse Eq?
(==), (/=)

23. Grundsätzlich kann jeder Typ in Haskell zur Instanz einer Typklasse gemacht werden, gleich ob mit `type`, `newtype` oder `data` eingeführt. Richtig oder falsch? Falls falsch, was genau?
Falsch, für type ist das nicht immer möglich, da hier vordefinierte Typen nur umbenannt werden und diese nicht einfach erweiter werden können. Fü newtype und data ist dies möglich.

24. Machen Sie den Typ:
`data NaturalNumber = Null | Succ NaturalNumber`
unter bestmöglicher Ausnutzung von Protoimplementierungen zu einer Instanz der Typklassen:
(a)Eq

```
instance Eq NaturalNumber where
      Null == Null = False
      Null == _ = False
      _ == Null = False
      Succ n == Succ m = m == n
```

(b)Show

```
toInt :: NaturalNumber -> Int
toInt Null = 0
toInt (Succ x) = 1 + toInt x

instance show NaturalNumber where
      show Null = “null”
      show (Succ x)= “ “ ++ toInt (Succ n)
```

(c)Ord

```
instance Ord NaturalNumber where
      (<=) Null Null = True
      (<=) (Succ x) (Succ y) = toInt (Succ x) <= toInt (Succ y)
      (<=) _ _ = False
```

(d)Num
(e)Enum

```
instance Enum NaturalNumber where
      fromEnum Null = 0
      fromEnum (Succ x) = toInt (Succ x)
      toEnum n = Succ n
      toEnum 0 = Null
```

Überlegen Sie sich, wie Sie möglichst sinnvoll bei der Implementierung von Funktionen vorgehen, die auf den ganzen statt natürlichen Zahlen negative Ergebnisse liefern würden. Die Funktion `show` soll in üblicher Weise als Folge von Dezimalziffern ausgeben.

25. Wiederholen Sie die vorige Aufgabe für den Typ:
`data IntegerNumber = Null | Succ IntegerNumber | Pred IntegerNumber`
Wie gehen Sie mit der Nichteindeutigkeit der Darstellung ganzer Zahlen als `IntegerNumber`-Werte bei den Instanzbildungen um? Benutzen Sie eine 'Normaldarstellung' für `IntegerNumber`-Werte für Funktionsergebnisse? Wie sieht Ihre Normaldarstellung aus? Wie ist sie definiert?

26. Wenn Sie eine Normaldarstellung benutzen, schreiben Sie eine Wahrheitsfunktion:
`is_nf :: IntegerNumber -> Bool`
die überprüft, ob ein `IntegerNumber`-Wert in Normalform ist oder nicht.

27. Modifizieren Sie Ihre Instanzbildungen aus der vorvorausgehenden Aufgabe so, dass Funktionen angewendet auf Werte in Normalform wieder Werte in Normalform liefern, ansonsten mit einer Fehlermeldung abbrechen.

28. Wenn Sie eine Normaldarstellung benutzen, schreiben Sie eine Transfomrationsfunktion:
`to_nf :: IntegerNumber -> IntegerNumber`
die einen `IntegerNumber`-Wert in seine Normalform überführt.

29. Ändern Sie die ursprünglichen Instanzbildungen fürr `IntegerNumber` so ab, dass Funktionen ihre Argumente in Normalform überführen und ihre Ergebnisse ebenfalls in Normalform liefern.

30. Was ist Haskells Feldsyntax? Wozu ist sie nützlich
Ist eine einfacherer Möglichkeit data-Typen zu erstellen, nachdem Konstruktor können Feldnamen und Typen aufgeführt werden.

Teil IV, Kapitel 10 'Funktionen höherer Ordnung'

1. Woran erkennt man eine Funktion
(a) erster – liefern einen Wert zurück, nehmen Wert als Input
(b) höherer Ordnung? - liefern eine Funktion zurück oder nehmen Funktion als Input

2. Welche kürzere Sprechweise gibt es für Funktionen höherer Ordnung?
Funktionale

3. Geben Sie einige Beispiele in Haskell vordefinierter Funktionen höherer Ordnung an auf
(a) numerischen Typen: +, - , *
(b) Listentypen: map, filter, zipWith
(c) Wahrheitswerten: ||, &&

5. Was ist mit dem Schlagwort von Funktionen als first class citizens gemeint?
Funktionen können als Input und Output von Funktionen verwendet werden

6. Was versteht man unter funktionaler Abstraktion?
1. Stufe: nichtfunktionale Operanden als Parameter von Operanden
höherer Stufe: Funktionen höherer Ordnung, Verknüpfungsvorschriften werden zu funktionalen Parametern

7. Was ist das Pendant zu funktionaler Abstraktion in
(a) prozeduralen - Prozeduren
(b) objektorientierten Programmiersprachen? - Methoden

8. Was geschieht bei funktionaler Abstraktion
Siehe 6
(a) erster
(b) höhererStufe?

9. Illustrieren Sie funktionale Abstraktion
(a) erster
f :: (Int,Int,Int) -> Intf (a,b,c) = (a * b + c) * (b + a * c)
(b) höherer anhand je eines geeigneten Beispiels.
fac n | n==0 = 1| n>0 = n * fac (n-1)

10. Wie spielen Funktionen höherer Ordnung und Curryfizierung einander in die Hände?
Nachdem curryfizierte Ausdrücke einzeln konsumiert werden können, kann eine partiell ausgewertete Funktion zurückgegeben werden.

11. Funktionale Abstraktion unterstützt Wiederverwendung.
(a) In welcher Weise? Operatoren wie (kleiner, größer) können als Parameter verwendet werden.
(b) Wovon? Berechnungsvorschriften

12. Welche Art von Wiederverwendung bzw. von was unterstützt funktionale Abstraktion
(a) erster – einzelner Operationen
(b) höherer Stufe? gleicher Operationsstrukturen

13. Welche funktionalen Argument oder/und Resultate bzw. Argument oder/und Resultattypen haben die Funktionen `zip` und `zipWith`?

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
Argumente: Liste von Typ a, Liste von Typ b
Resultat: Liste von Tupeln von Typ a und b

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []
Argumente: Funktion die mittels a und b auf c abbildet, Liste vom Typ a,
Liste vom Typ b
Resultat: Liste vom Typ c
```

14. Implementieren Sie die Funktion `zip` mithilfe von `zipWith`.
typlify :: a -> b -> (a,b)
typlify a b = (a,b)

```
zip :: [a] -> [b] -> [(a,b)]
zip as bs = zipWith tuplify as bs
```

15. Nennen Sie einige Beispiele in Haskell vordefinierter Funktionen mit funktionalen
(a) Argumenten: filter, map, foldr, foldl
(b) Resultat: partielle Auswertung

16. Was ist mit λ-lifting gemeint?
Bildung konstanter Funktionen mittels λ-Ausdruck

17. In welcher Weise spielen Curryfizierung und partielle Auswertung von Funktionen einander in die Hände?
Nachdem curryfizierte Ausdrücke einzeln konsumiert werden können, kann eine partiell ausgewertete Funktion zurückgegeben werden.

18. Was ist Funktionskomposition, was Funktionsapplikation? Erklären Sie den Unterschied.
Funktionskomposition = Zusammenfügen von Funktionen
Funktionsapplikation = Anwenden von Funktion auf bestimmte Eingabeparameter

19. Woran erkennt man
(a) Funktionskomposition – (f . g) x

(b) Funktionsapplikation - (f g) x
in Haskell (syntaktisch)?

20. Wozu dienen die Funktionen–`foldl`–`foldr`? Wie bzw. worin unterscheiden sie sich?
Aggregieren alle Listenelemente mittels Verknüpfungsoperation; Auswertung von linke (foldl) und Auswertung von rechts (foldr)

21. Was berechnen bzw. welchen Wert haben folgende Ausdrücke?
(a)`foldl (^) 2 [1,2,3] = 64`
(b)`foldr (^) 2 [1,2,3] = 1`

22. Im Standard-Präludium ist die Funktion `concat` wie folgt definiert:
```
concat :: [[a]] -> [a]
concat = foldr (++) []
```
(a) Wozu dient `concat`, welche Bedeutung hat es? Zusammenschmelzen von Listen
(b) Wäre eine Implementierung von `concat` mit `foldl` anstelle von `foldr` genauso möglich? Wäre es sinnvoll? Begründen Sie Ihre Antwort.
Ja, könnte auch so umgesetzt werden. Es spricht nichts dagegen.

23. Was ist mit applikativem Rechnen gemeint? Was mit funktionalem?
Rechnen mit elementaren Typen, Rechnen mit Funktionen

25. Übertragen Sie die Idee applikativen und funktionalen Rechnens im Algorithmus von Euklid auf die Berechnung des kleinsten gemeinsamen Vielfachen zweier natürlicher Zahlen m und n mit m,n≥1

Teil IV, Kapitel 11 'Polymorphie'

1. Gegeben sind die Funktionen `change` und `mapfun`:
```
change :: Eq a => (a -> b) -> a -> b -> (a -> b)
change f x y = g where g = \z -> if z == x then y else f z

mapfun :: Eq a => (a -> b) -> [(a,b)] -> (a -> b)
mapfun f [] = f
mapfun f ((x,y) : zs) = mapfun (change f x y) zs
```
(a) Warum muss bei `change` eine Kontexteinschränkung für die Typvariable a vorgenommen werden, nicht aber für b?
Weil x mit z verglichen wird und daher Eq implementieren muss, mit b wird nicht verglichen
(b) Warum gilt das in gleicher Weise für `mapfun`?
Weil mapfun change aufruft

2. Was ist die Bedeutung der Funktion change, was von `mapfun`? Erklären Sie knapp, aber gut nachvollziehbar, wie die beiden Funktionen arbeiten.
Change: gibt eine Funktion zurück die falls der Eingabewert gleich dem Eingabeparameter x von change ist den Eingabeparameter y von change zurück gibt ansonsten die Funktion f auf z anwendet.
Mapfun: nimmt eine Funktion und eine Liste als Input, falls die Liste leer ist wird die Liste retourniert ansonsten wird die Funktion f auf das erste Tupel der Liste angewandt und die Funktion rekursiv aufgerufen.

3. Sind die Funktionen `change` und `mapfun` curryfiziert oder uncurryfiziert?
Uncurryfiziert

4. Was ist die
(a) curryfizierte - `change :: Eq a => a -> b -> a -> b -> a -> b`
(b) uncurryfizierte - `mapfun :: Eq a => a -> b -> [(a,b)] -> a -> b`
Lesart von `change` und `mapfun`?

5. Geben Sie einige Beispiele in Haskell vordefinierter
(a) parametrisch – fst, snd, length, curry, uncurry, id, flip
(b)ad hoc polymorpher Funktionen an. (+) , (-), (*), show, (==), ...

6. Von welchem Polymorphietyp sind folgende Funktionen aus Standard-Präludium bzw. Kapitel 10:
(a)`(+) :: Num a => a -> a -> a: direkt überladene Ad-Hoc-Polymorphie`
(b)`rekSchema :: (Num a, Ord a) => b -> (a -> b -> b) -> a -> b: indirekt überladenen Ad-Hoc-Polymorphie`
(c)`flip :: (a -> b -> c) -> (b -> a -> c) – echte Polymorphie, Parametrische Polymorphie`
Geben Sie den Polymorphietyp (und seine Synonyme) möglichst genau an.

7. Ist `f :: a -> b` sinnvoll als Typdeklaration einer polymorphen Funktion? Begründen Sie Ihre Antwort. Überlegen Sie, ob sich die Deklarationszeile zu einer vollständigen Implementierung ergänzen lässt.
Ja, nachdem weder a noch b eingeschränkt sind können beliebige Typen verwendet werden
Bsp) f a = “Hallo” ++ a

8. Ändert sich Ihre Antwort, wenn wir statt `f` folgende Funktion `g`:
`g :: a -> a` betrachten? Wenn ja, wie?
Bei g muss der Eingabetyp gleich dem Ausgabetyp sein, es ist schwierig eine polymorphe Funktion hierfür zu finden, da Berechnungen nur auf Num Typen durchführbar sind und andere Operationen nur auf anderen Typen

9. Ergänzen Sie (möglichst allgemein) den Typ der Funktionskomposition:
```
(.) :: ...
(.) f g = \x -> f (g x)

(.) :: (b -> c) -> (a -> b) -> a -> c
```

10. Was ist Überladung?
Das erneute Schreiben einer Funktionsdefinition in einem Untertyp

11. Wodurch unterscheidet sich direkte und indirekte Überladung?
Direkt überladenen Funktionen sind direkt in Typklassen eingefügt, indirekt überladene Funktionen stützen sich auf andere überladene Typklassen ab

12. Was erlaubt Polymorphie auf Datentypen wiederzuwenden?
Datenstrukturnamen, Konstruktionsweisen und Aufbau von Datenwerten

13. Was unterscheidet Überladung und echte Polymorphie?
Echte Polymorphie hat keine Einschränkung bezüglich der Typkontexte der Parameter, unechte schon

14. Insert ‘some’ and ‘any’ appropriately:
A parametric polymorphic function works for some data type,an ad hoc polymorphic function works for any data type.

15. Gegeben ist folgende Funktion:
```
f [] = ([],[])
f ((x,y):zs) = (x:xs,y:ys) where (xs,ys) = f zs
```
(a) Welches ist der allgemeinste Typ von `f`?
F :: [(a,b)] -> ([a], [b])
(b) Von welchem Polymorphietyp ist `f`, wenn es polymorph ist?
Echte Polymorphie
(c) Was ‘macht’ `f`? Beschreiben Sie knapp, aber gut nachvollziehbar, was `f` ‘macht’, welche Bedeutung es hat.
F erstellt aus einer Liste von Tupeln ein Tupel von Listen

16. Was erlaubt
(a) echte – Funktionsnamen und Funktionsrümpfe
(b) unechte Polymorphie auf Funktionen - Funktionsnamen wiederzuverwenden?

17. Woran erkennt man eine
(a) echte – keine Typvariable in der Funktionssignatur ist kontexteingeschränkt
(b) unecht polymorphe Funktion? - zumindest eine Typvariable in der Funktionssignatur ist kontexteingeschränkt

18. Unechte Polymorphie auf Funktionen lässt sich in direkte und indirekte unechte Polymorphie scheiden; echte Polymorphie nicht. Richtig oder falsch?
Richtig

19. Wie spielen Typklassen und unechte Polymorphie einander in die Hände?
Unechte Polymorphie basiert auf Typklassen

20. Was ist damit gemeint, dass Typklassen voneinander erben können?
Durch die Instanzbildung kann eine Typklasse Funktionen einer anderen Typklasse gemacht werden

21. Typklassen in Haskell dienen der Organisation von Überladung. Wie ist das zu verstehen? Was ist damit gemeint?
Mit Typklassen können Spezifikationen für mehrere Funktionen bereit gestellt werden und Protoimplementierungen bereitgestellt werden

22. Überladung von Funktionen für neue Typen lässt sich manchmal automatisch vornehmen.
(a) Welche Funktionen kommen dafür (ausschließlich) infrage? Funktionen der vordefinierten Typklassen Eq, Ord, Read, Bounded, Enum, Show
(b) Wo liegen die Grenzen der automatischen Überladung dieser Funktionen?
Sie bieten nur Standardimplementierungen, welche nicht bei jedem Typ passen
(c) Wie kann die Überladung erreicht werden, wenn der Automatismus dafür nicht (mehr) infrage kommt?
Implementierung mit instance
(d) Welche Haskell-Sprachkonstrukte dienen zur automatischen bzw. nicht automatischen Überladung?
Deriving, instance
(e) Gibt es generelle Grenzen von Überladung, d.h. gibt es Typen, für die eine grundsätzlich überladungsfähige Funktion nicht überladen werden kann? Begründen Sie Ihre Antwort.
Ja, bei Unentscheidbarkeit (bspw. ob etwas gleich ist) kann auch Überladen nicht helfen. Beispielsweise ist auch Vollständigkeit und Konsistenz ein Problem.

23. Für echt polymorphe Funktionen gilt: Ein Funktionsname, eine Implementierung. Was gilt entsprechend für unecht polymorphe Funktionen?
Ein Funktionsname mehrere Implementierungen

24. Polymorphie verspricht höhere
(a) Transparenz, Lesbarkeit
(b) Verlässlichkeit, Wartbarkeit von Programmen. Warum? Begründen Sie Ihre Antwort.
Sprechende Namen

25. Wie wirkt Polymorphie der Knappheit guter Namen entgegen?
Namen können durch echte und unechte Polymorphie wiederverwendet werden

Teil I – III, Verschiedene Kapitel

1. Wie unterscheiden sich die Typen `Float` und `Double` voneinander?

Präzision

2. Was ist ein Relator? Was ist eine Relation?
==, /=
eine Funktion die einen Relator verwendet

3. Was sind Listenkomprehensionsausdrücke? Illustrieren Sie Ihre Antwort mit einigen Beispielen.
Listen die durch die Verwendung der mathematischen Mengenoperation gebildet werden
[x | x <- [1,2,3]]

4. Woran erkennt man:
(a) schlicht – nur ein Selbstaufruf als äußerster Aufruf
(b) linear – nur ein Selbstaufruf nicht als äußerster Aufruf
(c) baumartig – mehrere Selbstaufrufe nebeneinander
(d) geschachtelt rekursive Funktionen? - rekursive Aufrufe beinhalten rekursive Aufrufe

5. Welche synonyme Bezeichnungen gibt es für die Rekursionsarten aus der vorigen Teilaufgabe?
Schlicht – repetitiv
baumartig – kaskadisch

6. Geben Sie ein Beispiel an für zwei (oder mehr) wechselweise rekursive Funktionen.
Func1 func2
func2 = func

7. Welche Begriffe sind statt wechselweiser Rekursion auch üblich?
Indirekte, verschränkte

8. Funktionen können in Haskell mithilfe verschiedener Sprachmittel definiert werden.
(a) Nennen Sie möglichst viele.
Bedingte Ausdrücke, bewachte Ausdrücke, case-Ausdrücke, lokale Deklarationen (let, where) anonyme Funktionen, Muster
(b) Welches Sprachmittel werden dabei am häufigsten benutzt? Warum?
Bewachte Ausdrücke bzw. Muster, da diese auch für verschiedene Datentypen geeignet ist

9. Welches Symbol erlaubt es, mehr als eine Gleichung in eine Zeile zu schreiben? ;

10. Welche Klammererung gilt in Haskell implizit für
(a) Funktionssignaturen – rechtsassoziativ (Klammern werden rechts geschlossen)
(b) Funktionsterme – linksassoziativ (Klammern werden links geschlossen)

11. Was versteht man unter curryifzierten Funktionen? Was unter uncurryfizierten? Illustrieren Sie Ihre Antwort anhand geeigner Beispiele.
Uncurrifiziert – mit Klammerung – Tupel
Curryfiziert – ohne Klammerung – ohne Tupel

12. Geben Sie die Signatur und Implementierung der vordefinierten Funktionen `curry` und `uncurry` an.

curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = g where g x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) → c)
uncurry g = f where f (x,y) = g x y

13. Gegeben ist die Deklaration:
```
foldFrom :: (a -> b -> a) -> a -> [b] -> (a -> Bool) -> a
foldFrom f e [] _ = e
```

```
foldFrom f e xxs@(x:xs) p
      | p a = foldl f e xxs
      | True = foldFrom f e xs p
```

Was ist die Bedeutung von `foldFrom`? Erklären Sie knapp, aber gut nachvollziehbar, was sich aus den einzelnen Zeilen der Deklaration von `foldFrom` herauslesen lässt und wie die Auswertung der Funktion vorgeht.

FoldFrom nimmt eine Funktion die ein Element von Typ a nimmt und mittels einem Element von Typ b zu einem Element von Typ a macht, ein Element vom Typ a, eine Liste von Elementen vom Typ b und eine Funktion die Elemente vom Typ a auf Bool abbildet und liefert ein Element von Typ a zurück
Genauer falls die Wahrheitswertfunktion gleich a ist wird die Funktion f auf den Inputparameter e und alle Elemente der Inputliste an ansonsten auf wird die Funktion rekursiv aufgerufen

14. Sind
(a)`xs @ (x : (y : (z : zs)))` – gültig as Pattern von einer Liste in Standarddarstellung
(b)`xs @ (ys ++ zs)` – ungültig da ++ nur für Listenelemente und ys und zs keine Listen
gültige Muster(ausdrücke)? Begründen SIe Ihre Antwort.

16. Wofür können in Haskell ‘sprechende’ Namen selbst gewählt werden?
Typen, Variablen

17. Was ist hier falsch? Markieren und verbessern Sie alle Fehler, so dass sich das Programm interpretieren und übersetzen lässt:

```
type Verwandt = Bool
type Bekannt = Bool
data Wohnform = EFH, ZFH, MFH, DH, RH, HH, PH Deriving (Eq, Show)
newtype Bürger = B Person Wohnform Nachbarn Freunde deriving (Eq, Show)
newtype Nachbarn = N [(Bürger,Verwandt,Bekannt)] deriving (Eq, Show)
newtype Freunde = F [Bürger] deriving (Eq, Show)

verwandteNachbarn :: Bürger -> [Person]
verwandteNachbarn (B _ _ (N ls) _)
 = (p : (B p _ _ _;verwandt;_) <- ls; verwandt = true]

verwandteNachbarnNamen :: Bürger -> ([Nachname,Vorname])
verwandteNachbarnNamen (B _ _ (N ls) _) =
[(nn,vn) || (B (P vn nn _) _ _ _ vw _) <= ls && not vw <> true]
```

Überprüfen Sie mit ghc oder Hugs, ob Sie alle Fehler gefunden haben.

18. Definieren Sie einen Datentyp, der es erlaubt, für verschiedene Modelle von Autoherstellern verschiedener Marken wesentliche Eigenschaften wie Antriebsart, Abmessungen, Motorleistung und Listenpreis abzulegen
type Antriebsart = String
type Abmessungen = String
type Motorleistung = Float
type Listenpreis = Float
data Modell = M Antriebsart Abmessungen Motorleistung Listenpreis

19. Geben Sie möglichst genau an, was für einen Datentyp Sie in der vorigen Aufgabe gewählt haben. Geben Sie auch an, warum Sie den Datentyp so und nicht anders gewählt haben.
Einen Produkttyp mit 4 Datenfeldwerten welche mittels type benannt wurden

20. Machen Sie Ihren Datentyp (wenn möglich) aus der vorvorgehenden Aufgabe zu Instanzen der Typklassen
(a)Eq
instance Eq Modell where
(==) (M aa am m l) (M aa2 am2 m2 l2) = aa == aa2 && am == am2 && m == m2 && l == l2

(b)Show
`instance Show Modell where`
`show (M aa am m l) = “Modell:” ++ aa ++ am ++ m ++ l`
mithilfe expliziter `instance`-Deklarationen.

21. Falls Sie Ihren Datentyp aus der Voraufgabe nicht zu einer Instanz der von `Eq` und `Show` machen können, was ist der Grund dafür? Was müssten Sie ändern, um die Instanzbildungen vornehmen zu können?
Wurde gemacht

22. Implementieren Sie die Funktion zwei-hoch, in Zeichen 2:

$$2^{\cdot} : \mathbb{N} \to \mathbb{N}$$
$$\forall n \in \mathbb{N}.\ 2^n = \begin{cases} 1 & \text{falls } n = 0 \\ 2 * 2^{n-1} & \text{falls } n > 0 \end{cases}$$

in möglichst vielen syntaktischen Varianten in Haskell (mit Fallunterscheidungen, bewachten Ausdrücken, Mustern, Listenkomprehensionen, anonymen λ-Abstraktionen,...).

Bewachte Ausdrücke:
```
type Nat = Integer
zweiHoch :: Nat -> Nat
zweiHoch
        | n == 0 = 1
        | n > 0 = 2* zweiHoch (n-1)
        | otherwise = error “Number is negative”
```

case:
```
zweiHoch n = case n of 0 = 1
                       otherwise 2 * zweiHoch (n-1)
```

23. Typklassen wie `Ord` sehen vor, dass für ihre Instanzen sehr viele Funktionen implementiert sein müssen. Bei einer konkreten Instanzbildung sind tatsächlich aber oft nur sehr wenige zu implementieren.Warum?
Weil Typklassen Protoimplementierungen bereitstellen auf welche zurückgeriffen werden kann

24. Welche Funktionen reichen oft aus, bei Instanzbildungen für die Typklasse `Ord` implementiert zuwerden?
(<=) ist die minimal complete implementation

25. Wann ist es nötig, mehr als die Funktionen aus der Voraufgabe zu implementieren? Illustrieren Sie Ihre Antwort mit einem passenden Beispiel.
Falls automatische Instanzbildung mittels Prototypen nicht den Spezialanforderungen erfüllt.

Teil V, Kapitel 12 ‘λ-Kalkül’

1. Wofür ist der λ-Kalkül erdacht worden? Auf wen geht er zurück?
Um Fragen der Berechenbarkeit formalisieren zu können, Alonso Church

2. Was bedeutet intuitiv berechenbar, intuitive Berechenbarkeit?
Es gibt eine irgendwie machbare mechanische Methode die in endlich vielen Schritten für gültige Argumentwerte einen Funktionswert liefert und für nicht gültige Argumentwerte einen Fehlerwert liefert oder nie abbricht

3. Was unterscheidet intuitive von formaler Berechenbarkeit?
Intuitive Berechenbarkeit ist nicht präzise behandelbar

4. Was besagt die Churchsche These?
Etwas ist intuitiv berechenbar gdw es in Lambda-Kalkül berechenbar ist.

5. Ist die Churchsche These verifizierbar? Ist sie falsifizierbar? Begründen Sie Ihre Antwort.
Die These ist nicht verifizierbar, gilt aber solange als richtig bis sie falsifiziert ist. Es könnte immer sein, dass ein Gegenbeweis gefunden wird, außer es wird bewiesen, dass die These immer stimmt.

6. Welche Bedeutung hat Wilhelm Ackermann für die Berechenbarkeitstheorie?
Hat mit der Ackermannfunktion eine Funktion geliefert, die berechenbar, aber nicht primitiv-rekursiv-berechenbar ist.

7. Wird intuitive Berechenbarkeit infrage gestellt? Wenn ja, warum und in welcher Weise?
Ja, da es eine vielleicht noch mächtigere Berechnungsmethode geben kann

8. Wodurch zeichnet sich der λ-Kalkül gegen über anderen Berechnungsmodellen aus?
Hat größere Praxisnähe (da Basis funktionaler Programmiersprachen)

9. Was ist mit dem Bonmot der λ-Kalkül sei die Assembler-Sprache funktionaler Programmierung gemeint? Dass das Lambdakalkül die Grundlage der funktionalen Programmiersprachen bietet, da diese nur stark angereicherte angewandte Lambdakalküle darstellen.

10. Was unterscheidet den reinen von sog. angewandten λ-Kalkülen?
Syntaktisch angereichtert und damit praxis- und programmiersprachennäher

11. Wie ist der reine λ-Kalkül syntaktisch aufgebaut? Welche Konstrukte gibt es?
Abstraktionen: (λ x.(x x)) – wobei (x x) Rumpf
Applikationen: ((λx.(x x)) y) – wobei (λx.(x x)) Rator und y Rand

12. Welche Regeln, Konventionen gelten im reinen λ-Kalkül zur Einsparung von Klammern?
Rechtsassoziativität für λ-Sequenzen in Abstraktionen also (λ x.e) = λ x. e, (λ x.(x x)) = λ x. (x x)
Linksassoziativität für Applikationssequenzen also e1 e2 statt (e1 e2)

13. Markieren Sie in den folgenden λ-Ausdrücken alle freien und gebundenen Vorkommen von Namen, bei gebundenen Vorkommen zusätzlich, woran sie gebunden sind:

(a) *λ f. λ g. λ h. f (g h)) λ g. ((λ g. g) g))*
(b) *((λ x. λ y. x y) ((((λ x. λ y. x y) a) b)) c*

14. Markieren Sie in den folgenden λ-Ausdrücken die Bindungs- und Gültigkeitsbereiche aller definierenden Vorkommen von Namen:

(a) *λ f. λ g. λ h. f (g h)) λ g. ((λ g. g) g))*
(b) *((λ x. λ y. x y) (((λ x. λ y. x y) a) b)) c*

15. Was sind Ratoren in λ-Ausdrücken? Was Randen?
In einer Applikatiion ist der Rand das was in den Rator eingesetzt werden soll

16. Was unterscheidet Konversions- und Reduktionsregeln des reinen λ-Kalküls?
Konversionsregeln sind ein Überbegriff für Reduktions- und Abstraktionsregeln
(Reduktion – Anwendung von links nach Rechts; Abstraktion – Anwendung von rechts nach links)

17. Was bezeichnet man mit syntaktischer Substitution?
Ist die dreistellige Abbildung zur bindungsfreien Ersetzung der freien Vorkommen von x durch Ausdruck e in einen Ausdruck e'

18. Warum kann syntaktische Substitution naiv angewendet zu Bindungsfehlern führen?
Weil wenn in den Gültigkeitsbereich einer Variable mit dem gleichen Namen eingesetzt wurde, darf eigentlich nicht ersetzt werden

19. Wie vermeidet syntaktische Substitution bei korrekter, nicht naiver Anwendung Bindungsfehler?
Durch Umbenennung gleichnamiger Variablen

20. Was versteht man unter einem Reduktionsschritt? Unter einer Reduktionsfolge?
Ein Reduktionsschritt ist das Anwenden der Beta- oder Nü-Reduktion, eine Reduktionsfolge ist eine (un)endlicheFolge von Beta-, Nü-Reduktionen und Alpha-Konversionen

21. Woran erkennt man, dass ein λ-Ausdruck in Normalform vorliegt?
Wenn sich der Ausdruck mit Beta- und Nü-Reduktionen nicht weiter reduzieren lässt

22. Welcher Zusammenhang besteht zwischen den Begriffen normale, applikative, innerste, äußerste Reduktionsordnung?
Normale und äußerste Reduktionsordnung, applikative innerste Reduktionsordnung

23. Was legen die Begriffe normale, applikative, innerste, äußerste Reduktionsordnung für die Vereinfachung von λ-Ausdrücken fest? Was lassen sie offen?
Ob innen oder außen angefangen wird zu reduzieren, lässt offen ob links, rechts oder mittig angefangen wird bei gleichwertigen Ausdrücken

24. Welche Reduktionsordnungen legen fest, was die Reduktionsordnungen aus der vorigen Aufgabe noch offenlassen? In welcher Weise treffen sie diese Festlegung(en)?
Linksnormale Reduktion, linksapplikative Reduktion

25. Welche Beinamen haben die sog. Church/Rosser-Theoreme? Was besagen sie?
Konfluenz-, Diamant- oder Rauteneigenschaft bzw. Standardisierung;
Wenn e1, e2 ineinander konvertierbar sind dann gibt es einen gemeinsamen Ausdruck zu dem e1, e2 reduziert werden können (Wenn eine Normalform existiert ist diese eindeutig)

Standardisierung: Wenn e1 und e1 mit einer endlichen Reduktionsfolge reduzierbar siind (also in NF) dann führt die normale Reduktionsfolge von e1 nach e2. (Normale Reduktion terminiert häufigstmöglich)

26. Wie ist die Semantik von λ-Ausdrücken definiert?
e = λ-Ausdruck; falls NF existiert ist diese die Bedeutung und der Wert von e

27. Kann die Semantik eines λ-Ausdrucks undefiniert sein? Begründen Sie Ihre Antwort.
Ja, falls keine NF existiert, also die Reduktionsfolge nie terminiert

28. Woran erkennt man einen Kombinator im λ-Kalkül?
Der Rator ist gleich dem Rand, die Funktion wird auf sich selbst angewandt

29. Welche Bedeutung hat der Y-Kombinator im λ-Kalkül?
Damit kann die Rekursion umgesetzt werde (Rekursion durch Kopie)

30. Reduzieren Sie den λ-Ausdruck

((λ x. λ y. x y) (((λ x. λ y. x y) a) b)) c

(a) normal

(λ y. (((λ x. λ y. x y) a) b) y) c
(((λ x. λ y. x y) a) b) c
((λ y. a y) b) c
(a b) c

(b) applikativ

((λ x. λ y. x y) ((λ y. a y) b)) c
((λ x. λ y. x y) (a b)) c
(λ y. (a b) y) c
(a b) c
zur Normalform.

Teil V, Kapitel 13 'Auswertungsordnungen'

1. Was wird durch eine Auswertungsordnung festgelegt?
Die Reihenfolge der Expansions- und Simplifikationsschritte

2. Rechtsnormale und rechtsapplikative Auswertung sind die dualen Gegenstücke linksnormaler und linksapplikativer Auswertung. Beschreiben Sie das Vorgehen

(a) rechtsnormaler
Ausdrucksauswertung nach Expansion, Start mit rechtestem Element
(b) rechtsapplikativer
Ausdrucksauswertung vor Expansion, Start mit rechtestem Element
Auswertung.

3. Werten Sie den Term:
fac :: Integer -> Integerfac n = if n == 0 then 1 else (n * fac (n – 1))

fib :: Int-> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)

2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+fib((5+7)*2)
(a) rechtsnormal
2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+fib((12)*2)
2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+fib(24)
2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+(fib (24-2) + fib (24-1))
2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+(fib (24-2) + fib (23))
2^3+fac(fib(square(2+2)))+3*5+fib(fac(7*(5+3)))+(fib (24-2) + (fib (21) + fib (22)))
usw.

(b) rechtsapplikativ
aus.

4. Warum (vermutlich) sind rechtsnormale und rechtsapplikative Auswertung in (vermutlich) keinem Übersetzer und Interpretierer für eine funktionale Sprache implementiert zu finden?
Weil sie längere Auswertungsreihenfolgen bewirken

5. Was bedeutet

(a) normale – Auswertung von Ausdrücken nach Expansion
(b) applikative – vor Expansion
Funktionstermauswertung?

6. In Haskell werden Funktionsterme standardmäßig normal ausgewertet. Richtig? Falsch? Ja, aber? Begründen Sie Ihre Antwort.

Nein, in Haskell wird eine Spezialform verwendet (lazy evaluation), welche den Ausdruck mittels Transformation von Graphen auswertet, da normale Auswertung jeden Ausdruck sonst so oft auswertet, wie er vorkommt.

7. In imperativen Sprachen spricht man von call by value-Auswertung; in funktionalen Sprachen von applikativer Auswertung. Vervollständigen Sie den Satz entsprechend.

8. In funktionalen Sprachen spricht man von normaler Auswertung; in imperativen Sprachen von call-by-name Auswertung. Vervollständigen Sie den Satz entsprechend.

9. In imperativen Sprachen spricht man von call by need-Auswertung; in funktionalen Sprachen von lazy evaluation/ späte Auswertung. Vervollständigen Sie den Satz entsprechend.

10. Linksnormale und linksapplikative Auswertung liefern stets dasselbe Ergebnis. Richtig? Falsch? Ja, aber? Begründen Sie Ihre Antwort.
Falls, es kann sein dass die linksapplikative Auswertung nicht terminiert, während die linksnormale terminiert. Sonst schon

11. Was unterscheidet das Vorgehen von
(a) applikativer, normaler
siehe oben
(b) früher, später
beide Werten Ausdruck maximal bzw. genau einmal aus; aber bei später Auswertung werden Ausdrücke erst nach Expansion ausgewertet, bei früher davor Auswertung?

12. Wie oft wird ein Argument eines Funktionsterms bei
(a) applikativer – genau einmal
(b) normaler – so oft wie es vorkommt
(c) früher – genau einmal
(d) später – maximal. einmal
Auswertung ausgewertet?

13. Warum spricht man statt von applikativer auch von strikter Auswertung?
Weil für strikte Argumente immer frühe Auswertung angewandt werden darf, da für strikte Argumente gilt, dass der Wert von f nicht definiert ist, falls der Wert eines Parameters nicht definiert ist

14. Für Haskell ist die späte Auswertungsordnung für Ausdrücke festgelegt. Etwas anderes ist nicht möglich. Richtig oder falsch? Begründen Sie Ihre Antwort.
Falsch, frühe Auswertung kann mit $! erzwungen werden

15. Etwas salopp kann man sagen, dass die normale Auswertungsordnung am häufigsten terminiert. Was ist damit gemeint?
Dass falls es eine terminierende Auswertungsreihenfolge gibt, diese durch die normale Auswertung gefunden wird. Applikative Auswertung terminiert nicht immer

16. Frühe und applikative Auswertungsordnung bezeichnen dasselbe; späte und normale Auswertungsordnung ebenso. Richtig oder falsch? Begründen Sie Ihre Antwort.
Falsch, späte und normale Auswertung unterscheiden sich in der Häufigkeit der Funktionstermauswertung

17. Innerste und applikative Auswertungsordnung bezeichnen dasselbe; äußerste und normale Auswertungsordnung ebenso. Richtig oder falsch? Begründen Sie Ihre Antwort.
Richtig

18. In keiner Auswertungsordnung werden Funktionstermargumente seltener ausgewertet als bei später Auswertungsordnung. Was heißt das genau und wie wird es technisch erreicht und sichergestellt?
Durch die Darstellung von Ausdrücken als Graphen und Berechnung mittels Graphentransformation wird sichergestellt, dass jeder vorkommende Ausdruck genau einmal ausgewertet wird; gleiche Ausdrücke nur einmal

19. Warum hat man sich in einigen funktionalen Sprachen für frühe, in anderen für späte Auswertung entschieden? Was spricht für frühe, was für späte Auswertung? Was spricht umgekehrt jeweils dagegen?
Vorteile späte Auswertung:
Terminiert häufigstmöglich, Auswertung nur einmal notwendig, Umgang mit potentiell unendlichen Konstrukten
Nachteile späte Auswertung:
Implementierung anspruchsvoll

Vorteile frühe Auswertung:
einfach Implementierung, einfach Integration imperativer Konzepte, mathematisch natürlicher, geringe Laufzeit

Nachteil frühe Auswertung:
terminiert nicht so oft, Auswertung für jeden Ausdruck einmal, unendliche Konzepte schwierig umsetzbar

20. Was versucht eine Striktheitsanalyse über ein Programm herauszufinden?
Ob eine späte, faule Auswertung durch eine frühe ersetzt werden kann, ohne dabei das Risiko einzugehen, dass die Auswertung nicht terminiert

Teil V, Kapitel 14 ‘Typprüfung, Typinferenz’

1. Wann spricht man von
   (a) stark – prüfen zur Übersetzungszeit
   (b) schwach – prüfen zur Laufzeit
   getypten Sprachen?

2. Was spricht für die Wahl einer
   (a) stark – verlässlicher Code, effizienterer Code, Typinfo ist Doku
   (b) schwach – dynamisches Binden und dynamische Typprüfungen, Reflexion
   getypten Programmiersprache?

3. Was unterscheidet Typprüfung und Typinferenz?
Typprüfung kontrolliert den Typ mit dem erwarteten Typ, Typinferenz leitet aus den gegebenen Information den Typ her

4. Was unterscheidet monomorphe und polymorphe Typen? Wofür stehen, was repräsentieren sie?
Monomorphe Typen: haben eindeutig bestimmten konkreten Typ
Polymorphen Typen: einen oder mehrere (bis unendlich) viele konkrete Typen

5. Wie kann das grundsätzliche Vorgehen bei Typprüfung und Typinferenz zusammengefasst und beschrieben werden?
Bei Typprüfung wird durch Kontextauswertung geprüft ob die Typung korrekt ist
Typinferenz wertet die Typumgebung aus mit Hilfe von Axiomen und Unifikation und bestimmt einen gemeinsamen Typs

6. Was bedeutet es für einen Ausdruck wohlgetypt zu sein?
Dass er einen wohldefinierten Typ besitzt

7. Welche Rolle spielt Unifikation für polymorphe Typprüfung und Typinferenz?
Findet die allgemeinsten gemeinsamen Instanz einer Menge von Typausdrücken

8. Für monomorphe Typprüfung und Typinferenz wird Unifikation nicht gebraucht. Richtig oder falsch? Begründen Sie Ihre Antwort.
Richtig, da nur ein Typ in Frage kommt

9. Inferieren Sie händisch den allgemeinsten Typ der Funktion foldl:

```
foldl f e []     = e
foldl f e (x:xs) = foldl f (f e x) xs
```

(a -> b -> a) -> a -> b -> a

10. Was sind Typsprachen? Wozu dienen sie?
Typsprachen werden durch eine Typgrammatik erzeugt, wobei jeder Ausdruck der Sprach einen Typ bestimmt; dienen für die Umsetzung von Typinferenzalgorithmen

11. Was sind Typsysteme? Wozu dienen sie?
Ein System mit Axiomen und Reglen, welche ebenfalls für den Typinferenzalgorithmus benötigt werden

12. Warum kann man bei erfolgreicher Typprüfung, Typinferenz für ein Programm von einem teilweisen Korrektheitsbeweis für dieses Programm sprechen?
Weil Fehler zur Laufzeit aufgrund von Typfehlern ausgeschlossen sind

13. Was versteht man unter
(a) der Instanz eines Typausdrucks – den konkreten Typ eines Ausdrucks
(b) einer gemeinsamen Instanz einer Menge von Typausdrucken – ein konkreter Typ der für mehrere Typausdrücke erfüllt ist
(c) der allgemeinsten Instanz einer Menge von Typausdrucken?
Der allgemeinste konkrete Typ der mehrere Typausdrücke erfüllt

14. Warum ist der Ausdruck:

```
length ([]++[True]) + length ([]++[1,2,3])
```

in Haskell wohlgetypt, der Ausdruck:

```
length (xs++[True]) + length (xs++[1,2,3])
```

aber nicht?
Weil xs hier als Bool und Integerliste verwendet wird, die leere Liste aber keinen Typ vorgibt

15. Was bedeutet das Axiom VAR in Worten? Was die Regel COND?

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

$$\text{COND} \quad \frac{\Gamma \vdash exp : Bool \quad \Gamma \vdash exp_1 : \tau \quad \Gamma \vdash exp_2 : \tau}{\Gamma \vdash \text{ if } exp \text{ then } exp1 \text{ else } exp2 : \tau}$$

Teil I – IV, Verschiedene Kapitel

1. Operatoren und Relatoren sind syntaktische Begriffe; Operationen und Relationen semantische. Erklären Sie den Unterschied anhand vordefinierter Operatoren und Relatoren in Haskell.
Operatoren und Relatoren (+), (==),... Sind Rechenzeichen
Die Berechnung davon ist semantisch und lässt bspw. 4 == 4 zu True auswerten zu, aber nur falls == auf dem Typ definiert ist sonst kann die Syntaktik nicht in Semantik umgewandelt werden

2. Geben Sie je einige gültige Werte folgender Typen an:
   (a) [Int -> Int] - 2*n mit n Int
   (b) [Float -> Float] - 2 *n mit n Float
   (c) [Char -> Int] - ord c
   (d) [Int -> IChar – ungültig ??
   (e) [String -> String] - “hello” ++ s
   (f) [String -> Char]
   (g) [String -> Int]
   (h) [Num a => a -> a -> a]
   (i) [Ord a => a -> a -> Bool]

3. Welchen allgemeinsten Typ haben folgende Ausdrücke und Teilausdrücke und woran erkennt man das jeweils?
   (a) [] ++ [True] - [Bool]
   (b) [x ++ y] - [[a]]
   (c) x : [y,‘z’] - [Char]
   (d) [x * y] – [Num a => a -> a-> a]
   (e) ... - Da hat jemand keinen Bock mehr gehabt

4. Welchen allgemeinsten Typ haben folgende Funktionen und woran erkennt man das?
   (a) f x y = (x + y) * (x - y) - Num a => a -> a -> a
   (b) g k x y = k (x,y) – ((a,b) -> c) -> a -> b -> c
   (c) h [] = []
       h (x:xs) = (h xs) ++ [x]
       [a] -> [a]

5. Mit welchen vordefinierten Funktionen in Haskell stimmen f, g und h aus der vorigen Aufgabe überein?
a) ??? Hoogle kennt keine
b) curry
c) reverse

6. Welche der Funktionen f, g und h aus Aufgabe 4 sind rekursiv? Welche nicht? Welches Rekursionsmuster liegt bei den rekursiven Funktionen vor.
c ist rekursiv; b könnte wechselseitig rekursiv sein, abhängig davon wie k definiert ist
c – repetitive Rekursion

7. Was versteht man unter Rechnen auf Parameterposition? Illustrieren Sie Ihre Antwort mit einem Beispiel.
Zusätzliche Parameter für die Rekursion mitgeben → effizienter manchmal

8. Beschreiben Sie die Liste [‘a’,‘e’,‘i’,‘o’,‘u’] mithilfe einer Listenkomprehension.
[x | x <- [‘a’,‘e’,‘i’,‘o’,‘u’]]

9. Was besagt der Hauptleitsatz fkt. Programmierung?

10. Geben Sie je ein Beispiel an für eine:
   (a) schlicht -
   ggt m n

```
                | n == 0 = m
                | m >= n = ggt (m-n) n
                | m < n = ggt (n-m) m
```

(b) linear

```
func 0 = 0
func n = func (n-1) * n
```

(c) baumartig

```
func 0 = 1
func n = func(n-1) * func(n-2)
```

(d) geschachtelt

```
func n
                | n > 3 = n – 4
                | otherwise = func (func (n+3))
```

rekursive Funktion.

11. Was ist Rekursion auf
    (a) mikroskopischer
    betrachtet einzelne Rechenvorschriften und deren Selbstaufruf
    (b) makroskopischer
    betrachtet das Zusammenspiel wenn sich Funktionen gegenseitig aufrufen
    Ebene?

12. Sie haben für ein Problem ein exponentielles Lösungsverfahren. Ist es sinnvoller, einen neuen, schnelleren Rechner zu kaufen oder ein alternatives komplexitätsmäßig günstigeres Lösungsverfahren zu finden? Begründen Sie Ihre Antwort.
Alternativeres Lösungsverfahren, da exponentielle Verfahren mit der Eingabegröße sehr schnell wachsen, was ein neuer Rechner für kleine Eingabegrößen lösen kann aber nicht für größere

13. Sie haben für ein Problem ein Lösungsverfahren einer bestimmten Komplexität. Ist es ein Königsweg, ein Lösungsverfahren geringerer Komplexität zu suchen, um schneller zu Ergebnissen zu gelangen? Begründen Sie Ihre Antwort.
Nein, es gibt (derzeit) nicht für Probleme eine poly. Lösung, man kann aber oft Näherungsverfahren verwenden.

14. Zeigen Sie, dass die linke und rechte Seite der definierenden Gleichung von uncurry denselben Typ besitzen:

```
uncurry :: (a -> b -> c)-> ((a,b) -> c)
uncurry g (x,y) = g x y
```

15. Was sind gültige Operatorabschnitte? Was nicht? Welche Bedeutung haben die gültigen Operatorabschnitte?
(a) (*10) - gültig
(b) (10*) - gültig
(c) ((*) 10) – gültig, aber äußerste Klammer kann weggelassen werden
(d) (10 (*)) - ungültig, da (*) nicht als Präfix verwendet wird und somit auch nicht partiell ausgewertet werden kann, 10 ist somit nirgends zugehörig
(e) (div 2) - gültig
(f) (2 div) – Ausdruck kann wieder nicht partiell ausgewertet werden da der 2 vorher steht und div ein Präfixoperator ist
(g) (‘div‘ 2) – gültig
(h) (2 ‘div‘) - gültig
(i) (-1) – ungültig, wird als -1 als Wert interpretiert und nicht als Funktion
(j) (1-) - gültig
(k) ((-)1) - gültig
(l) (1(-)) - ungültig, weil Präfix wie Infix verwendet wird

16. Warum liefern mit dem Schlüsselwort type eingeführte Namen keine höhere Typsicherheit? Weil nur Namensgebung

17. Was versteht man unter einer Minimalvervollständigung einer Typklasse? Wofür ist eine Minimalvervollständigung gut?
Die Funktionen die implementiert werden müssen, damit alle Funktionen der Typklasse verwendet werden können. Sparen das schreiben aller Funktionsimplementierungen

18. Was ist automatische Typklasseninstanzbildung? Welche Voraussetzungen müssen vorliegen? Welche Grenzen oder Einschränkungen gibt es dafür?
Mittels deriving; Bildet Funktionsimplementierungen für den Untertyp automatisch
Kann nur für Untertypen von Enum, Bounded, Read, Show, Eq und Ord gemacht werden
Es dürfen keine Spezialimplementierungen benötigt werden

19. type und newtype-Deklarationen leisten dasselbe, fuhren aber zu unterschiedlich performanten Implementierungen. Richtig oder falsch. Begründen Sie Ihre Antwort.
Falsch newtype bringt Typsicherheit, type nicht

20. Aufzählungs- und Produkttypen sind Spezialfälle von Summentypen. In welcher Weise?
Summentypen können mehrere Konstruktoren mit mehreren Feldern verwenden, bei Aufzählungstypen sind nur Nullstellige Konstruktoren erlaubt, bei Produkttypen sidn genau 1 mehrstelliger Konstruktor erlaubt

21. Für die Implementierung des Datentyps Haustier stehen folgende zwei Typvereinbarungen zur Wahl:

```
type Name = String
data Lieblingsplatz = Sofa | Fensterbank | Futternapf | Bett | FeldWaldundWiese
deriving Eq
type HoertAuf = Zuruf | Manchmal | Garnicht deriving Eq

type Haustier = (Name,Lieblingsplatz,Hoertauf)
newtype Haustier = H (Name,Lieblingsplatz,HoertAuf)
```

Was spricht für, was gegen die Vereinbarung von Haustier als
(a) Typsynonym? Sprechende Namen, keine Typsicherheit
(b) Neuer Typ? Typsicher, aber Funktionen von Tripel können nicht verwendet werden

22. Was ist der allgemeinste Typ von f?

```
f [] _ _ = []
f (x:xs) p g
 | p x  = (g x) : f xs p g
 | True = f xs p g

f :: [a] -> (a -> Bool) -> (a -> c) ->  [c]
```

23. Listenkomprehensionen stützen sich i.a. auf Generatoren, Transformatoren, Filter und Tests. Was davon wird in der Definition von quickSort genutzt?

```
quickSort :: Ord a => [a] -> [a]
quickSort []     = []
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                   ++ [x]
                   ++ quickSort [y | y <- xs, y > x]
```

Generatoren und Tests

24. Kann die Definition von quickSort vereinfacht werden zu?

```
quickSort :: Ord a => [a] -> [a]
quickSort []     = []
```

```
quickSort (x:xs) = quickSort [y | y <- xs, y <= x]
                   ++ quickSort [y | y <- xs, y > x]
```

Begründen Sie Ihre Antwort.

Teil VI, Kapitel 15 'Interaktive Programme: Ein- und Ausgabe'

1. Referentielle Transparenz ist eine grundlegende Eigenschaft rein funktionaler Programme.
(a) Was bedeutet referentielle Transparenz? Das ein Ausdruck immer den gleichen Wert liefert, egal wann er aufgerufen wird.
(b) Welche Probleme stellen sich für referentielle Transparenz im Zusammenhang mit Ein- und Ausgabe? Nachdem durch Eingabe die Werte verändert werden können ist referentielle Transparenz nicht mehr zwangsläufig gegeben (Seiteneffekte!!)

2. Wie werden in Haskell diese Probleme gelöst/vermieden?
Rein funktionaler Kern und imperative Dialogschicht; Kernfunktionen bleiben unbeeinflusst

3. Warum spielt der ansonsten weitgehend uninteressante Nulltupeltyp im Zusammenhang mit Ein- und Ausgabe eine bedeutsame Rolle?
Weil dieser der IO Typ von allen Schreibaktionen darstellt, wird als formal erforderliches Ergebnis verwendet

4. Was unterscheidet Werte des vordefinierten polymorphen Datentyps IO von Werten aller anderen Typen in Haskell?
Sie bewirken Lese- oder Schreibseiteneffekte

5. Die do-Notation in Haskell ist syntaktischer Zucker. Wofür?
Für (>>=) bzw. (>>) also die Festlegung der Reihenfolge von I/O-Operationen

6. Erklären Sie knapp, aber gut nachvollziehbar wie die Auswertung folgenden Ein-/Ausgabeprogramms vor sich geht:

```
summiere :: IO Int
summiere
 = do n <- getInt -- n = Int
      if n == 0 -- Bool
       then return 0
       else (do m <- summiere
                return (n + m)) -- m = Int

getInt :: IO Int
getInt = do line <- getLine -- line IO String
            return (read line :: Int)
```

Summiere liest ein Int ein und vergleicht die Zahl mit 0 falls sie null ist wird 0 retourniert ansonsten wird summiere rekursiv aufgerufen um einen weiteren Wert einzulesen, der mit dem aktuellen addiert wird → Addiert alle Zahlen die eingelesen werden

getInt liest einen String von der Eingabe ein und retourniert diesen als Int

Annotieren Sie den Typ aller (Teil-) Ausdrücke.

7. Schreiben Sie die Programme summiere und getInt aus der vorigen Aufgabe ohne do-Notation.

```
summiere :: IO Int
summiere = getInt >>= (\n -> if n == 0 then return 0 else summiere >>= (\
m -> return (n + m)))
```

```
getInt :: IO Int
getInt = getLine >>= (\line -> return (read line :: Int))
```

8. Warum mag man folgendes Programm iterativartig nennen, auch wenn es nicht wirklich iterativ ist, keine while-Schleife darstellt, wie der Name suggerieren mag?

```
while :: IO Bool -> IO () -> IO ()
while bedingung aktion
  = do b <- bedingung
       if b
        then
         do aktion
            while bedingung aktion
        else
         return ()
```

Weil es die Befehle imperativ abarbeitet
der Variable b wird der Wert von Bedingung einem IO Bool-Wert zugewiesen. Falls b True ist wird die mitgelieferte aktion ausgeführt und while rekursiv mit der aktion und der bedingung wieder aufgerufen. Falls die Bedingung nicht zu True auswertet wird die Rekursion beendet und der Nulltupeltyp retourniert

Beschreiben Sie auch hier wieder knapp, aber gut nachvollziehbar, wie die Auswertung des Programms vor sich geht. Geben Sie auch hier wieder den Typ aller (Teil-) Ausdrucke an.

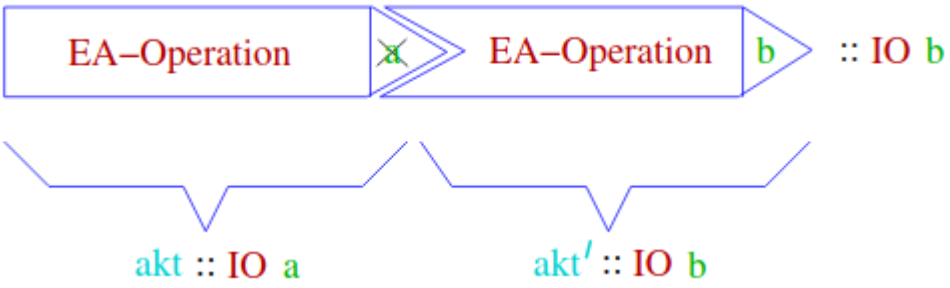
9. Woran erkennt man Ein-/Ausgabeausdrucke in Haskell?
An den IO-Tyoen als Eingabe- bzw. Rückgabetyp

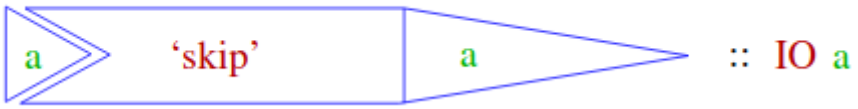
10. Warum bezeichnet man Ein-/Ausgabeausdrucke in Haskell auch als Aktionen?
Weil nicht einfach Werte zurückgeliefert werden, sondern auch Seiteneffekte bewirkt werden

11. Wie lassen sich Ein-/Ausgabeaktionen zu Sequenzen zusammensetzen, komponieren?
(>>=) bzw. mit do

12. Erklären Sie informell die operationelle Bedeutung von:
(a) (>>=) :: IO a -> (a -> IO b) -> IO b – setzt Eingabe- und Ausgabeoperationen zusammen, zeitliche Anordnung
(b) (>>) :: IO a -> IO b -> IO b – setzt Eingabe- und Ausgabeoperationen zusammen, wobei das Argument von a “vergessen wird”
(c) return :: a -> IO a
Ist eine Abbildung die den a-Wert auf eine Aktion mit einem a-Wert als Ergebnis abbildet

13. Veranschaulichen Sie die Bedeutung der Abbildungen aus der vorigen Aufgabe auch mithilfe graphischer Darstellungen.





14. Erläutern Sie den Unterschied von return in Haskell zu gleichbenannten Routinen aus anderen Sprachen, die Sie kennen.
Return in Haskell reicht eine funktional reine Komponente an die Ausgabe als Aktionsergebnis weiter (IO a); die Ausführung der restlichen Aktionssequenz wird dadurch nicht beeinträchtigt

15. Wie hängen die Abbildungen (>>) und (>>=) zusammen?
Bei >> wird das Ergebnis der ersten Aktion ignoriert, bei >>= ist das Ergebnis der Vorgängeraktion wichtig

16. Wie hängen unbenannte Bindungen in do-Ausdrücken und die Abbildung (>>) zusammen?
Da bei >> auch das Ergebnis "vergessen" wird hat es den gleichen Effekt, wie wenn eine unbenannte Bindung erstellt wird, da auf den Wert dieser auch nicht mehr zugegriffen werden kann.

17. In welchem Sinn kann man return und <- (etwas salopp) als zueinander revers verstehen?
Weil <- das Ergebnis einer Aktion "abspeichert" und somit als funktionalen Ausdruck zugänglich macht

18. Lassen sich lokale Wertvereinbarungen in do-Ausdrucken einführen? Wenn ja, wie? Geben Sie ggf. ein illustrierendes Beispiel an.
Ja, mit benannten Wertvereinbarungen
Bsp) w1 <- akt1 bzw mit let für rein funktionale Ausdrücke

19. Was ist das funktionale, was das prozedurale Verhalten eines Ein-/Ausgabeausdrucks?

20. Wie unterscheiden sich die Bedeutungen von
(a) <- (in do-Ausdrucken) – für nicht rein funktionale Ausdrücke, einfache Wertzuweisung
(b) let (in do-Ausdrucken) – nur für rein funktionale Ausdrücke, einfache Wertzuweisung
(c) := (Wertzuweisungsoperator in imperativen Sprachen) , mehrfache Wertzuweisung voneinander?

21. Schreiben Sie jeweils

(a) mit

```
askForChar :: IO ()
askForChar = do putStrLn "Please enter a char"

out :: IO String
out = do getMore
         putStrLn "komischer hinweis: "

getMore :: IO String
getMore = do x <- getChar
             if x == 's' then return [x] else getMore return (x : y)
```

(b) ohne

```
askForChar2 :: IO ()
askForChar2 = putStrLn "Please enter a char"
```

```
out :: IO String
out = getMore >>= (\x -> return $ "komischer hinweis: " ++ x)

getMore :: IO String
getMore = getChar >>= (\x -> if x == 's' then return [x] else getMore
>>= (\y -> return $ x : y))
```

do-Notation ein Ein-/Ausgabeprogramm, dass
– zur Aufgabe (?? was zur Hölle soll das heißen – Eingabe?? weil Ausgabe macht keinen Sinn?!) eines Zeichens auffordert
- ein Zeichen einliest
und diesen Zyklus so lange wiederholt, bis schließlich das Zeichen ‘s‘ für *stopp* eingegeben wird. Mit Ausnahme dieses Stoppzeichens ‘s‘ sollen anschließend alle gelesenen Zeichen zusammen mit einem kurzen die Ausgabe erklärenden Hinweis ausgegeben werden.

22. Ändern Sie Ihre Programme aus der vorigen Aufgabe so ab, dass nach Beendigung der Ausgabe statt aller gelesenen Zeichen nur die gelesenen Kleinvokale ausgegeben werden.
23. Ändern Sie Ihre Programme aus den beiden vorigen Aufgaben so ab, dass die Ein- und Ausgabe nicht mehr vom und auf den Bildschirm erfolgen, sondern von und in eine (andere) Datei.

24. Andern Sie Ihre Programme aus den vorigen Aufgaben so ab, dass aus dem Eingabestrom von Zeichen die Ziffern herausgefiltert werden und nach Beendigung der Eingabe der Zahlwert dieser Ziffernfolge zusammen mit einem erklärenden Hinweis ausgegeben wird.
25. Ergänzen Sie für ein oder zwei der Programme aus der Voraufgabe zur Überprüfung der Typkorrektheit den Typ aller vorkommenden (Teil-) Ausdrücke.

Teil VI, Kapitel 16 ‘Robuste Programme: Fehlerbehandlung’

1. Welche
(a) Ziele sollte eine systematische Fehlerbehandlung verfolgen?
(b) Eigenschaften sollte sie dabei erfüllen?

2. Sind diese Ziele und Eigenschaften stets gleichzeitig zu erreichen oder stehen sie sich zum Teil im Wege? Begründen Sie Ihre Antwort.

3. Warum ist ein Weiterrechnen mit einem falschen/unsinnigen Wert oft gefährlicher als ein Fehlerabbruch?
Weil dadurch simuliert wird, das alles in Ordnung ist und UserIn meint das Ergebnis sei korrekt.

4. Was ist mit dem Panikmodus als Fehlerbehandlung gemeint?
Mit Error-Meldung abbrechen

5. Wie lässt sich eine Fehlerbehandlung im Panikmodus in Haskell am einfachsten umsetzen?
error :: String -> a

6. Illustrieren Sie diese Umsetzung anhand eines passenden Beispiels.
error “Fehler aufgetreten”

7. Wo liegen die Stärken, Schwächen einer Fehlerbehandlung im Panikmodus?
Einfach, generell und schnell; Nachteile: radikal, jede Info geht verloren

8. Ein anderer Vorschlag zur Fehlerbehandlung empfiehlt die Verwendung von Auffangwerten.
(a) Was ist damit gemeint?
Falls etwas fehlschlägt einen vordefinierten Wert retournieren

(b) Welche Varianten können möglicherweise genauer unterschieden werden?
Funktionsspezifischer Auffangwert, aufrufspezifischer Aufffangwert

9. Wie lässt sich eine Fehlerbehandlung mithilfe von Auffangwerten in Haskell umsetzen?
otherwise = -1 oder mit Inputwert n : otherwise = n

10. Illustrieren Sie diese Umsetzung anhand passender Beispiele, ggf. je eines für verschiedene Varianten.
Siehe 9
11. Wo liegen die Stärken, Schwächen einer Fehlerbehandlung mithilfe von Auffangwerten?
Es gibt nicht immer passende Auffangwerte, intransparent da keine Fehlermeldung

12. Illustrieren Sie die Stärken und Schwächen einer Fehlerbehandlung mithilfe von Auffangwerten anhand geeigneter Beispiele.
Für Rückgabezahlen aus R gibt es keinen sinnvollen Auffangwert

13. Welche weitere Möglichkeit bietet sich in Haskell an, eine systematische Fehlerbehandlung durchzufuhren?
Fehlertypen (Maybe)

14. Welches sind die Stärken und Schwächen dieser Möglichkeit?
Fehler können erkannt, abgefangen, weitergereicht und behandelt werden (mit check if Nothing)

15. Illustrieren Sie diese Möglichkeit wieder anhand treffender Beispiele.
```
func n
        | n >= 0 = Just (n*2)
        | otherwise = Nothing
```

16. Können Sie anhand Ihrer Beispiele auch die Stärken und Schwächen aufzeigen?

17. Was ist mit *Verschleierung* eines Fehlers durch Fehlerbehandlung gemeint?
Dass das Programm weiterläuft obwohl ein Fehler passiert ist

18. Warum ist eine Verschleierung von Fehlern gefährlich?
Dass man mit Rückgabewert weiterrechnet

19. Lässt sich eine Verschleierung von Fehlern stets vermeiden? Begründen Sie Ihre Antwort.
Mit Totalabbruch ja, allerdings ist das bei sicherheitskritischen Dingen nicht machbar

20. Welcher Typ ist in Haskell als Fehlertyp gemeint? Maybe

21. Wie viele Fehlerarten lässt dieser Typ unterscheiden? Just (kein Fehler), Nothing Fehler

22. Definieren Sie einen eigenen Fehlertyp in Haskell, der für Int-Werte zwischen den Fehlerarten *Überlauf*, *Unterlauf*, *Sonstiges* zu unterscheiden erlaubt.
data IntError = keinFehler | Überlauf | Unterlauf | Sonstiges

23. In welcher Lesart kann man die Bedeutung der Funktion
```
        may_Maybe :: (a -> b) -> Maybe a -> Maybe b
```
als ein ‘Typ-Lifting’ verstehen?

24. Welche andere Lesart gibt es für map_Maybe noch?

25. Wozu ist map_Maybe nützlich?

Teil VI, Kapitel 17 ‘Programmierung im Großen: Module’

1. Was bezeichnen die Begriffe
   (a) Kohäsion – innerer Zusammenhalt von Modulen
   (b) Koppelung – äußerer Zusammenhang von Modulen
   im Zusammenhang mit Modulen? Was beschreiben, wozu dienen sie?

2. Was haben Kohäsion und Koppelung mit *guter* Modularisierung zu tun?
Schwache Koppelung und hohe Kohäsion

3. Welche Perspektiven können wir einnehmen, um die *Güte* einer Modularisierung zu beurteilen?

4. Welche Eigenschaften sind unter diesen Perspektiven wichtig, um die *Güte* einer Modularisierung zu beurteilen?

5. Was sind Indizien einer wenig(er) gut gelungenen Modularisierung?
Es muss nicht viel an anderen Modulen geändert werden, wenn ein Modul geändert wird

6. Um *gute* Modularisierung technisch überhaupt zu ermöglichen, muss das Modularisierungskonzept einer Sprache das *Geheimnisprinzip* erfüllen. Was ist damit gemeint?
Alles was in ein Modul gehört und nicht nach außen sichtbar sein muss soll nicht exportiert werden

7. Haskells Modulkonzept unterstutzt keinen *automatischen Reexport*. Was ist damit gemeint?
Dinge die importiert werden müssen beim export extra deklariert werden, da sie sonst nicht exportiert werden

8. Wie ist in Haskell vorzugehen, wenn ein Reexport nötig ist?
module M2 (module M1, blabla) where import M1

9. Was verstehen wir unter einem
   (a) selektiven – nur bestimmte Modulteile importieren / exportieren, oder nur bestimmte Module nicht importieren (hiding) / exportieren
   (b) nicht selektiven – alles importieren /exportieren
   Import, Export?

10. Auf welche Weisen sind in Haskell ein
   (a) selektiver
   Import:
   module M2 where
   import M1

   Export:
   module M2 (module M2,…) where
   import M1

   (b) nicht selektiver
   Import:
   import M2
   Export: automatisch bei global sichtbaren Bezeichnern
   Import, Export möglich?

11. Werden beim Export algebraischer und Neuer Typen in Haskell nur die Typnamen oder auch deren Datenkonstruktoren mit exportiert?
Nur Namen, Geheimnisprinzip

12. Wie ist ein Modul in Haskell grundsätzlich aufgebaut?

modul name where
Imports
Deklarationen

13. Wann liegt ein Namenskonflikt beim Import von Modulen vor?
Wenn lokale Namen und Namen aus den importierten Modulen gleich sind

14. Welche Möglichkeiten gibt es in Haskell mit Namenskonflikten beim Import von Modulen umzugehen?
Es können lokale Namen für importiertes Modul vergeben werden mit qualified
Bsp) import qualified M1
Zugriff mit M1.konfliktname

15. Welche Konventionen sollen beim Schreiben von Modulen in Haskell eingehalten werden?
Pro Datei ein Modul, Modul und Dateiendung (hs lhs) stimmen überein, alle Deklarationen beginnen mit module

16. Was ist die Idee hinter dem Konzept abstrakter Datentypen?
AnwenderIn muss die Implementierung nicht kennen, nur verwenden

17. Welche Herausforderungen sind bei der Definition eines abstrakten Datentyps zu meistern?
Gesetze wählen so das Verhalten eindeutig

18. Warum benötigt man für die Implementierung eines abstrakten Datentyps doch (auch) einen konkreten Datentyp?
Weil Haskell kein dezidiertes Sprachkonstrukt für die Umsetzung bietet

19. Inwiefern ist Haskells Modulkonzept nützlich für die Implementierung abstrakter Datentypen in Haskell?
Weil es erlaubt das Geheimnisprinzip umzusetzen

20. Spezifizieren Sie nach dem Vorbild des abstrakten Datentyps Warteschlange mit first-in/first-out (FIFO) Verhalten den abstrakten Datentyp Stapel mit last-in/first-out (LIFO) Verhalten.

21. Implementieren Sie ebenfalls nach Vorbild des abstrakten Datentyps Warteschlange auch den abstrakten Datentyp Stapel mithilfe verschiedener konkreter Datentypen.

22. Wie ist die Implementierung abstrakter Datentypen in Haskell von einem programmiertechnischen Standpunkt aus zu beurteilen? Gibt es Schwächen?
Ja, Funktionssignaturen und Gesetze können AnwenderInnen nur umständlich und unsicher mittels Kommentaren kommuniziert werden

23. Was wäre eine bessere Möglichkeit, wenn wir die Sprache Haskell erweitern, verändern durften?
Neues Sprachkonstrukt einführen

24. Auf wen gehen grundlegende Arbeiten zum Konzept abstrakter Datentypen zurück?
John V. Guttag

25. Was sind generelle Vorteile, Gewinne aus der Möglichkeit, Programme zu modularisieren?
Bessere Strukturierung, Wiederverwendbarkeit, unterstützt arbeitsteilige Programmierung

Teil I – V, Verschiedene Kapitel

1. Lösen Sie die Aufgabe des Umkehrens von Zeichenreichen

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```
(a) repetitiv rekursiv.

```
reverse :: [a] -> Int -> [a]
reverse [] _ = []
reverse l n= reverse ((!!) ((length l) – (n+1)) l) : (take ((length l)-2*n) l) : ((!!) n l))
```

(b) ohne Verwendung des Operators (++).

```
reverse :: [a] -> Int -> [a]
reverse [] _ = []
reverse l n= last l : reverse (take (length – (n+1)) l)
```

3. Welche der folgenden Ausdrucke sind gültig? Welche nicht? Was ist der Wert der gültigen Ausdrücke?

(a) fst (2,3) - gültig, 2
(b) fst (2,3,5) - ungültig da nur für Paare definiert
(c) fst [2,3] – ungültig nicht auf Listen
(d) fst [2,3,5] - ungültig
(e) head (2,3) – ungültig nicht auf Tupel definiert
(f) head (2,3,5) - ungültig
(g) head [2,3] - gültig, 2
(h) head [2,3,5] – gültig, 2
(i) tail (2,3) - üngültig, nicht auf Tupel
(j) tail (2,3,5) - ungültig
(k) tail [2,3] - 2
(l) tail [2,3,5] – [3,5]
(m) last (2,3) – ungültig nicht auf Tupel
(n) last (2,3,5) - ungültig
(o) last [2,3] - 3
(p) last [2,3,5] – 5

4. Illustrieren Sie die Aussage des *Hauptleitsatzes fkt. Programmierung* anhand geeigneter Beispiele.

5. Welche Performanzfallen lauern bei unbedachter Verwendung von Rekursion?
Exponentielle Funktionen

6. Geben Sie ein Beispiel an, wo es eine solche Falle gibt.
Fibonacci wenn unbedacht implementiert

7. Wie lassen sich diese Fallen (oft) vermeiden?
Meistens ja, außer bei NP-Problemen, wobei hier Näherungen verwendet werden können

8. Wenden Sie die Vermeidungsmethode auf Ihr Beispiel aus der vorausgehenden Aufgabe an.
Werte von Fibonacci können sich gemerkt werden und mittels Parameterposition mitgegeben werden

9. Welche Berechnungskomplexität haben folgende Funktionen:

(a) Fakultät

```
fac :: Int -> Int
fac n
 | n == 0 = 1
 | n >= 1 = n * fac (n-1)
```
linear

(b) Fibonacci

```
fib :: Int -> Int
fib n
 | n == 0 = 0
 | n == 1 = 1
 | n >= 2 = fib (n-1) + fib (n-2)
```
exponentiell

(c) Bionimialkoeffizieten

```
binom :: Int -> Int -> Int
binom n k = div (fac n) (fac k * fac (n-k))
 | n == 0 = 1
 | n >= 1 = n * fac (n-1)
```

10. Einige der folgenden Auswertungsstrategien bedeuten dasselbe. Welche?
(a) Normal
(b) Applikativ
(c) Linksnormal
(d) Linksapplikativ
(e) Rechtsnormal
(f) Rechtsapplikativ
(g) Linkestinnerst
(h) Linkestäußerst
(i) Rechtestinnerst
(j) Rechtestäußerst
(k) Faul
(l) Fleißig sie für die Auswertung?

Fleißig = applikativ, linksapplikativ = linkestinnerst, linkestäußerst = linksnormal, selbiges für die Begriffe mit rechts

11. Zeigen Sie, dass die linke und rechte Seite der definierenden Gleichung von flip denselben Typ besitzen:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x
```

12. Auf welche Weise können üblicherweise als Infixoperatoren gebrauchte Operatoren in Haskell als Präfixoperatoren gebraucht werden und umgekehrt? Illustrieren Sie Ihre Antwort an je einem Beispiel.
Infix nach Präfix durch Klammerung; umgekehrt durch schreiben unter Hochkommata `

13. Welche Eigenschaft muss ein Operator haben, damit man in Haskell einen Operatorabschnitt bilden kann?
Es muss currifiziert sein und somit partielle Auswertung zulassen und mehr als ein Argument verarbeiten

14. Listen sind in Haskell schon vordefiniert. Definieren Sie induktiv einen eigenen parametrisch polymorphen Listentyp, für den folgendes gilt:
(i) LL bezeichnet die leere Liste.
(ii) Ist l eine Liste und w ein Wert vom Listenelementtyp, so vereinigt Cons Liste l und Element w zu einer neuen Liste, in der w als letztes Element an l angefugt ist.

15. Für die Implementierung des Datentyps Haustier stehen folgende zwei Typvereinbarungen zur Wahl:

```
type Name = String
data Lieblingsplatz = Sofa | Fensterbank | Futternapf | Bett | FeldWaldundWiese
deriving Eq
type HoertAuf = Zuruf | Manchmal | Garnicht deriving Eq
```

newtype Haustier = H (Name,Lieblingsplatz,HoertAuf)
data Haustier = H (NameLieblingsplatz,HoertAuf)

Was spricht für, was gegen die Vereinbarung von ¨ Haustier als

(a) Neuer Typ? Bieten hier geringere Laufzeitkosten als data sonst hier beide Arten gleich sinnvoll

(b) algebraischer Typ?

18. Was haben alle Funktionen, die durch λ-Lifting entstehen, gemeinsam?
Sind Funktionen die mittels anonymer Funktion gebildet werden

20. Warum sind in Haskell zwei Funktionale zum Falten von Listen vordefiniert?
Weil je nach Anwendungsfall die eine oder die andere Variante nicht zielführend und oder ineffizient ist

21. Angewendet auf die gleichen Argumente liefern die Funktionen foldl, foldr stets dasselbe Resultat. Richtig oder falsch? Begründen Sie Ihre Antwort.
Falsch, Beispielsweise macht es bei nicht kommutativen Operationen einen Unterschied oder auch beim Potenzieren

22. Was besagt der *Hauptleitsatz funktionaler Programmierung*?

23. Funktionen höherer Ordnung und Polymorphie versprechen eine höhere Programmiereffizienz. Warum? Begründen Sie Ihre Antwort.
Nicht nur Funktionen können wiederverwendet werden sonder auch Strukturen von Funktionen (Extremwertfunktion in die nur mehr Operator eingesetzt werden muss)

24. Was sind typische Fragen, die in der Berechnungstheorie gestellt und untersucht werden? Ist ein Problem entscheidbar (berechenbar) in n Schritten

25. Was ist hier falsch? Markieren und verbessern Sie alle syntaktischen Fehler:

```
data Tree = Leaf Int Int | Root Tree Int Tree

depth :: Tree -> Int
depth (Leaf _ _)     = 0
depth (Root l _ r) = (max (depth l) (depth r)) + 1

sumtree :: Tree -> Int
sumtree (Leaf x y)  = x+y
sumtree (Root l _ r) = sumtree l + sumtree r

flatten :: (Int -> Int -> Int) -> Tree -> [Int]
flatten f (Leaf x y)   = f x y : []
flatten f (Root l z r) = (flatten f l) ++ [z] ++ (flatten f r)
```

Überprüfen Sie mit ghc oder Hugs, ob Sie alle Fehler gefunden haben.

Teil VI, Kapitel 18 ‘Funktionale und allgemeine Programmierprinzipien’

1. Was ist die Idee reflexiven Programmierens nach Simon Thompson?
2. Beschreiben Sie knapp, aber prägnant die verschiedenen Phasen reflexiver Programmierung.
3. Welche Schlüssel- und Leitfragen sind für diese Phasen typisch? Nennen Sie je einige Beispiele. ¨
4. In welche Phase gehört die Frage “ob sich das Problem verallgemeinern und so möglicherweise einfacher lösen lässt?”
5. Vergleichen Sie die Idee reflexiven Programmierens mit den Entwurfsprozessen nach

(a) Graham Hutton

(b) Norman Ramsey

aus Kapitel 9. Welche Gemeinsamkeiten, Unterschiede fallen Ihnen auf?

6. Was sind *nichtfunktionale* Eigenschaften eines Programms im Unterschied zu *funktionalen*? Nennen Sie einige Beispiele typischer nichtfunktionaler Eigenschaften von Programmen.
7. In welche Phase gehört die Frage “ob das Programm auch nichtfunktionale Eigenschaft gut erfüllt”?
8. Was bedeutet es, dass ein Problem
    (a) überspezifiziert
    (b) unterspezifiziert
ist? Warum muss beides vermieden werden?
9. In welche Phase gehört die Frage “ob das Problem über- oder unterspezifiziert ist”?
10. Warum ist *reflexives Programmieren* ein allgemeines Programmierprinzip, nicht nur ein funktionales?
11. Was ist mit *Kapseln algorithmischen Vorgehens* gemeint?
12. Illustrieren Sie die Idee anhand eines Beispiels.
13. Welches Sprachmittel funktionaler Sprachen ist für die Kapselung entscheidend? ¨
14. Warum ist das Teile-und-Herrsche als Berechnungsverfahren für die Berechnung der Fibonacci- ¨ Zahlen nicht geeignet?
15. Wie geht *mergeSort* zum Sortieren von Listen vor? Implementieren Sie das Verfahren in Haskell. Lassen Sie sich ggf. von der Darstellung auf http://idea-instructions.com/merge-sort leiten.
16. Wenden Sie das Vorgehen von Teile-und-Herrsche nach dem Vorbild von *quickSort* auf *mergeSort* an.
17. Wie unterscheiden sich die Implementierungen aus den vorigen beiden Teilaufgaben voneinander? Welche scheint Ihnen konzeptuell und implementierungstechnisch einfacher zu sein? Warum?
18. Welche anderen Probleme haben Sie schon kennengelernt, die sich mittels Teile-und-Herrsche gut lösen lassen? Wo haben Sie diese Probleme kennengelernt?
19. Lösen Sie einige dieser Probleme mithilfe des gekapselten Teile-und-Herrsche-Verfahrens aus Kapitel 18.
20. Wenn Sie einige dieser Probleme in anderen Lehrveranstaltungen schon gelöst haben, wie unterscheiden sich diese Lösungen von ihren Lösungen aus der vorigen Aufgabe?
21. Warum ist die beschriebene Idee zur Kapselung ein funktionales Programmierprinzip, aber kein allgemeines? Z.B. kein objektorientiertes?
22. Was versteht man unter einem *Strom*, was unter *Stromprogrammierung*?
23. Was sind *generiere/anpass*-Modularisierungen?
24. Welche Eigenschaft funktionaler Sprachen ist dafür entscheidend? Warum?
25. Illustrieren Sie am Beispiel des Siebs des Eratosthenes eine generiere/selektiere-Modularisierung.
26. Was unterscheidet *generiere/anpass*-Modularisierungen konzeptuell von ‘gewöhnlichen’ Modularisierungen?
27. Zeigen Sie anhand von Beispielen wie ein
    (a) *generiere*-Modul mit vielen *selektiere*-Modulen
    (b) *selektiere*-Modul mit vielen *generiere*-Modulen
kombiniert werden kann.
28. Ist der Unterschied zwischen einem selektiere- und einem filtere-Modul klar definiert? Wie könnte der Unterschied pragmatisch definiert werden?
29. Illustrieren Sie die Idee der *generiere/selektiere*-Modularisierung graphisch.
30. Warum ist Terminierung für *generiere/anpass*-Modularisierungen auch in Sprachen wie Haskell mit später Auswertung eine relevante Frage? Illustrieren Sie Ihre Antwort anhand eines Beispiels.
31. Geben Sie ein Beispiel für eine *generiere/transformiere*-Modularisierung an.
32. Was ist informell mit dem *Münchhausen*-Prinzip gemeint? Illustrieren Sie die Idee anhand eines Beispiels.
33. Illustrieren Sie die Idee der *generiere/transformiere*-Modularisierung graphisch.
34. Geben Sie einige Beispiele für Modularisierungen an, wo mehrere Generatoren, Selektoren, Filter, Transformatoren zum Einsatz kommen.
35. Keiner der drei Schritte des *Siebs des Eratosthenes* terminiert:
    (i) Schreibe alle (unendlich vielen!) natürlichen Zahlen ab 2 hintereinander auf.

(ii) Die kleinste nicht gestrichene Zahl ist eine Primzahl. Streiche alle (unendlich vielen!) Vielfachen dieser Zahl.

(iii) Wiederhole den vorigen Schritt (unendlich oft!) mit der jeweils nächstkleinsten noch nicht gestrichenen Zahl.

Dennoch kann aufbauend auf dieser Idee ein effektives Haskell-Programm zur Berechnung z.B. der ersten 1.000 Primzahlen geschrieben werden. Warum? Ist das in gleicher Weise in jeder funktionalen Programmsprache möglich? Begründen Sie Ihre Antwort.

36. Warum sind *generiere/anpass*-Modularisierungen ein funktionales Programmierprinzip, aber kein allgemeines? Z.B. kein objektorientiertes?

37. Geben Sie einen Haskell-Ausdruck an, dessen Wert der Strom der Potenzen von 2 ist.

38. Passen Sie den Haskell-Ausdruck aus der vorigen Aufgabe so an, dass sein Wert die 2er Potenzen zwischen 1.000 und 100.000 sind.

39. Haben Sie die vorigen Aufgaben mit einer Listenkomprehension gelöst? Wenn nein, probieren Sie es mit einer Listenkomprehension. Wenn ja, probieren Sie es ohne.

40. Welche der Lösungen aus den beiden Voraufgaben erscheint Ihnen angemessener? Warum?

Teil VII, Kapitel 19 ‘Ruckschau, Ausschau’

1. Durch welche Charakteristika und Eigenschaften unterscheiden sich

(i) funktionale

(ii) imperative

Programmierung voneinander?

2. Peter Pepper schreibt in seinem Buch über funktionale Programmierung in Opal, ML, Haskell ¨ und Gofer: “Die Fülle an Möglichkeiten [in fkt. Sprachen] erwächst aus einer kleinen Zahl von elementaren Konstruktionsprinzipien”. An welche Prinzipien denkt Pepper dabei? Im Zshg. mit

(a) Funktionen?

(b) Datenstrukturen?

3. Funktionen sind *erstrangige* Sprachelemente in funktionalen Sprachen. Was ist damit gemeint?

4. Konrad Hinsen verspricht in seinem Artikel über *The Promises of Functional Programming* die Versprechen funktionaler Programmierung aufzudecken. Welche sind das? Welche deckt er auf?

5. Welche Versprechen überlässt Hinsens Artikel der darauf aufbauenden Arbeit von Konstantin Läufer und George Thiruvathukal?

6. Wie wurden Sie die Frage von John W. Backus aus seiner Rede aus Anlass der Verleihung des *Turing Award* im Jahr 1977 beantworten, ob *“die Programmierung vom von Neumann-Stil befreit werden könne?”* Begründen Sie Ihre Antwort.

7. In welchen weiteren regelmäßig angebotenen Lehrveranstaltungen an der TU Wien stehen Fragen funktionaler Programmierung im Vordergrund?

8. Welches sind die großen Themenkreise dieser Veranstaltungen?

9. Nach Edsger W. Dijkstra, *Turing Awar*d Preisträger von 1972, ist eine hervorragende Zukunft funktionaler Programmierung höchstens durch eine Sache behindert. Welche? Stimmen Sie ihm zu? Begründen Sie Ihre Antwort.

10. Was ist der *Turing Award*? Wer verleiht ihn? Wie oft? Wofür? Womit wird der Preis oft verglichen und gleichgesetzt?

11. Wurde jemandem der *Turing Award* für Beiträge zur oder mit Bezug zur funktionalen Programmierung verliehen?

12. Peter Pepper schreibt in *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*, SpringerV., 2. Auflage, 2003: “*Functional programming is fun*“, weil funktionale Programmierung etwas von der Eleganz der Mathematik in die Programmierung bringt.” Stimmen Sie zu? Begründen Sie Ihre Antwort.

13. Mihai Maruseac schreibt in *Haskell: A Language for Modern Times*, Crossroads, the ACM Magazine for Students 24(1):64-66, 2017: “Learning Haskell opens one’s mind to new programming paradigms, which might produce clearer and shorter implementations.” Stimmen Sie zu? Begründen Sie Ihre Antwort wieder.

14. Yaron Minsky schreibt in OCaml for the Masses, Communications of the ACM 54(11):53-58, 2011: “[...] why the next language you learn should be functional!” Stimmen Sie zu? Begründen Sie Ihre ¨ Antwort auch hier.

15. Sehen Sie die Versprechen funktionaler Programmierung gegeben von Konrad Hinsen in *The Promises of Functional Programming*, Computing in Science and Engineering 11(4): 86-90, 2009, und Konstantin Läufer und George K. Thiruvathukal in *The Promises of Typed, Pure, and Lazy Functional Programming: Part II*, Computing in Science and Engineering 11(5): 68-75, 2009, als erfüllt an? Begründen Sie Ihre Antwort.

Teil I – VI, Verschiedene Kapitel

1. Was ist mit Rechnen mit Funktionen gemeint? Illustrieren Sie Ihre Antwort auch anhand einiger aussagekräftiger Beispiele.
2. Was unterscheidet applikative von funktionaler Programmierung?
3. Ist applikative Programmierung im engeren Sinn in einer vollausgebildeten fkt. Sprache wie Haskell überhaupt möglich? Begründen Sie Ihre Antwort.
4. Betrachten Sie die beiden Auswertungen des Terms 3 * 5 + square (2 + 3) + 5 - 3

```
3 * 5 + square (2 + 3) + 5 - 3
  ->> 15 + square (2 + 3) + 5 - 3
  ->> 15 + (2 + 3) * (2 + 3) + 5 - 3
  ->> 15 + 5 * (2 + 3) + 5 - 3
  ->> 15 + 5 * 5 + 5 - 3
  ->> 15 + 25 + 5 - 3
  ->> 40 + 5 - 3
  ->> 45 - 3
  ->> 42

3 * 5 + square (2 + 3) + 5 - 3
  ->> 3 * 5 + square (2 + 3) + 2
  ->> 3 * 5 + square 5 + 2
  ->> 3 * 5 + 5 * 5 + 2
  ->> 3 * 5 + 25 + 2
  ->> 3 * 5 + 27
  ->> 15 + 27
  ->> 42
```

über der Funktion

```
square :: Int -> Int
square n = n * n
```

(a) Welche Auswertungsstrategie ist jeweils angewendet worden?
(b) Woran erkennt man das jeweils?

5. Gegeben ist der Term

```
(2 + 3) * 5 + (fac (5-3) + 3) * 7 + (-5) + 1
```

über der Fakultätsfunktion fac:

```
fac :: Nat0 -> Nat1
fac n = if n == 0 then 1 else n * fac (n-1)
```

Wählen Sie einige der nachstehenden Strategien und werten Sie den Term entsprechend der gewählten Strategien aus:

(i) Normal
(ii) Linksnormal
(iii) Rechtsnormal
(iv) Applikativ
(v) Linksapplikativ
(vi) Rechtsapplikativ

6. Sind

(i) Mittignormal
(ii) Mittigapplikativ

auch sinnvolle Auswertungsstrategien? Begründen Sie Ihre Antwort.

7. *Rechnen auf Parameterposition* erlaubt lineare Rekursion durch einen anderen Rekursionstyp zu ersetzen. Welchen? Warum ist diese Ersetzung wichtig?
8. Auf welchen Säulen ruhen Stärke und Eleganz fkt. Programmierung?
9. Wie passt das Aristoteles-Zitat *“Das Ganze ist mehr als die Summe seiner Teile”* auf funktionale Programmierung?

10. Was legen folgende Auswertungsstrategien fest? Welche Freiheitsgrade lassen sie für die Auswertung?

(a) Normal
(b) Applikativ
(c) Linksnormal
(d) Linksapplikativ
(e) Rechtsnormal
(f) Rechtsapplikativ
(g) Linkestinnerst
(h) Linkestäußerst
(i) Rechtestinnerst
(j) Rechtestäußerst
(k) Faul
(l) Fleißig

11. Gegeben sind folgende Listentypen:

```
data Liste a  = LL  | Cons (Liste a) a
data Liste' a = LL' | Cons' a (Liste' a)
```

Schreiben Sie zwei Haskell-Rechenvorschriften:

(a) L2L'
(b) L'2L

die Werte vom Typ Liste in ihre entsprechende Liste'-Werte überführt und umgekehrt. Geben Sie dabei auch die vollständigen Typsignaturen für L2L' und L'2L an. Nehmen Sie Typklasseninstanzbildungen für Liste und Liste' vor, wenn für L2L' und L'2L nötig; aber auch nur dann.

12. Sind die Funktionen L2L' und L'2L monomorph oder polymorph? Falls polymorph, welche Art von Polymorphie liegt vor? Welche anderen Bezeichnungen sind für diesen Polymorphiebegriff üblich?

13. Was ist der allgemeinste Typ von f?

```
f g p [] = []
f g p (x:xs)
 | p x       = g x : f g p xs
 | otherwise = []
```

14. Welche Eigenschaft beschreibt den

(a) inneren
(b) äußeren

Zusammenhang von Modulen?

15. Welche Berechenbarkeitsmodelle können Sie nennen? Mit welchen davon haben Sie sich in dieser oder anderen Lehrveranstaltungen schon genauer beschäftigt?
Turingmaschinen, λ-Kalkül, Markov-Algorithmen, allgemein rekursive Funktionen, μ-rekursive Funktionen, Registermaschinen

16. Warum sind angewandte λ-Kalküle in der Berechenbarkeitstheorie von geringer Bedeutung?
Weil es zusätzliche Syntax bietet, die Beweise erschwert

17. Warum ist die α-Konversion im λ-Kalkül unverzichtbar?
Weil sonst kein Einsetzen in andere Funktionen möglich wäre

18. Warum stellen die Church-/Rosser-Theoreme die Grundlage für die Definition der Semantik des reinen λ-Kalküls dar?
Weil die Normalformen die Semantik der λ-Ausdrücke repräsentieren

19. Worin liegt die Bedeutung des λ-Kalküls für die funktionale Programmierung? Worin für Haskell?
Ist Basis für Berechnungen; Haskell und andere funktionale Programmiersprachen sind mit weiterer Syntaktik angereicherte λ-Kalküle.

20. Warum ist die linksnormale Auswertungsordnung ohne implementierungspraktische Bedeutung?
Weil sie auf Grund der mehrfachen Auswertung gleicher Ausdrücke sehr ineffizient ist

21. Wann werden Typfehler in Programmen einer
(a) stark – Übersetzungszeit
(b) schwach – Laufzeit
getypten Sprachen aufgedeckt?

22. Haskell ist eine stark getypte Sprache mit automatischer Typinferenz. Was bedeutet das? Typprüfung zur Übersetzungszeit, Typen können auch automatisch erkannt werden ohne explizit angegeben werden zu müssen

24. In prozeduralen Sprachen bereitet Ein-/Ausgabe konzeptuell keine Probleme, in funktionalen hingegen schon. Warum?
Weil referenzielle Transparenz verletzt wird

25. Von welchem Rekursionstyp ist die Ackermann-Funktion?
geschachtelte