Nachname: _____

Vorname: _____

Matrikelnummer: _____

Aufgabe	Points	Score
Programmanalyse mit Modulen und I/O	20	
Programmieren mit Listen	32	
Datatypen und Funktionen Höherer Ordnung	26	
Auswertung und Typen	12	
$\sum$	90	

In der wirklichen Klausur haben folgende Regeln gegolten:

- Sie haben 90 Minuten Bearbeitungszeit.
- Die Klausur besteht aus 4 Aufgaben, in denen Sie 90 Punkte erreichen können
- Die erreichbaren Punkte sind am Rand notiert.
- Entfernen Sie nicht die Heftklammer.
- Verwenden Sie keinen roten Stift.
- Antworten können in deutscher oder englischer Sprache verfasst werden.

Sie können bereits jetzt fast alle Aufgaben lösen.

- Bei der Bearbeitung von Aufgabe 1 sollten Sie noch auf Vorlesung 10 warten.
- Aufgabe 4(c) können Sie zum Zeitpunkt der Abgabe noch nicht mit dem Vorlesungswissen lösen.

Die Bearbeitung der Klausur ist freiwillig und wird nicht bewertet. Sie können die bearbeitete Klausur jedoch bis zum **7. Januar 2026** in OLAT hochladen (z.B. durch Anfertigung von Fotos der einzelnen Seiten, die dann in einem PDF zusammengefasst werden und dann als 1 PDF hochgeladen werden). Alle hochgeladenen Klausuren werden vollständig korrigiert und im Proseminar am 14. Januar zurückgegeben. An diesem Termin wird dann auch die Lösung besprochen.

Aufgabe 1: Programmanalyse mit Modulen und I/O 20

Betrachten Sie folgendes Programm.

```
1  import Text.Read(readEither)
2
3  data Expr = Div Expr Expr | Num Double deriving Read
4
5  eval :: Expr -> IO Double
6  eval (Num x) = return x
7  eval e@(Div e1 e2) = do
8    x1 <- eval e1
9    x2 <- eval e2
10   if x2 /= 0
11     then return (x1 / x2)
12     else let message = "div-by-0 error in expression " ++ show e
13          in putStrLn message
14
15 main :: IO ()
16 main = do
17   putStrLn "enter expression:"
18   s <- getLine
19   case readMaybe s of
20     Nothing -> main
21     Just e -> do
22       let result = eval e
23       putStrLn $ "the result is " ++ show result
```

Das Programm enthält vier Fehler, die zu einem Compile-Fehler führen.

- Identifizieren Sie die Fehler durch Angabe der Zeilennummern,
- durch eine kurze Erklärung, warum einer Fehler vorliegt, und
- geben Sie an, wie man den Fehler beheben kann.

Beachten Sie, dass alle vier Fehler unabhängig von einander sind.
Beachten Sie weiters, dass `readMaybe :: Read a => String -> Maybe a` vom Modul `Text.Read` exportiert wird.

(a) Fehler #1 (5)

(b) Fehler #2 (5)

(c) Fehler #3 (5)

(d) Fehler #4 (5)

Aufgabe 2: Programmieren mit Listen 32

Ein Wort w ist ein Palindrom, wenn es vorwärts und rückwärts gelesen gleich ist. Die Wörter "hannah", "refer", und "a" sind Palindrome, aber "paul" und "valid" sind es nicht.

Ein Palindrom kann man für beliebige Listen definieren, z. B. ist $[1, 2, 7, 2, 1]$ auch ein Palindrom, aber $[1, 8, 9, 1]$ nicht.

Für die folgenden Aufgaben außer Aufgabe (b) dürfen Sie beliebige Prelude Funktionen nutzen, z.B., `map`, `length`, `take`, `drop`, `words`, `unwords`, `[i .. j]`, und so weiter.

(a) Definieren Sie eine Haskell-Funktion `palindrome` die bestimmt, ob eine Eingabe-Liste ein Palindrom ist. Geben Sie auch den Typ von `palindrome` an, der so allgemein wie möglich sein sollte. (4)

Beispiele:

- `palindrome "kayak" && palindrome "" && palindrome [1,2,7,2,1]` sollte zu `True` auswerten.
- `palindrome "paul" || palindrome [1,2]` sollte zu `False` auswerten.

(b) Definieren Sie eine Funktion `partition :: (a -> Bool) -> [a] -> ([a], [a])` mit folgender Funktionalität. Wenn `partition p xs = (ys, zs)`, dann enthält `ys` die Elemente von `xs`, die das Prädikat `p` erfüllen, und `zs` enthält all anderen Elemente von `xs`. (8)
Zum Beispiel sollte `partition (> 5) [4,10,7,3,2]` zu `([10,7], [4,3,2])` auswerten.

Für diese Aufgabe ist es nicht erlaubt, irgendwelche vordefinierten Funktionen auf Listen zu verwenden, außer die Listen-Konstruktoren. List-Comprehension ist ebenfalls nicht erlaubt.

(c) Definieren Sie eine Haskell-function `magicSentence :: String -> Bool` die bestimmt, ob ein Satz *magisch* ist, d.h., ob mindestens die Hälfte der Wörter in dem Satz Palindrome sind. (8)

- The Eingabe ist ein Satz, der als Haskell `String` repräsentiert wird, und die Wörter innerhalb des Satzes sind mit Leerzeichen getrennt.
- Jedes Vorkommen eines Wortes wird einzeln gezählt, d.h. `"a bob is a fast vehicle"` ist ein Satz mit 6 Worten, und er ist magisch, da (mindestens) 3 davon Palindrome sind: `"a"`, `"bob"` and `"a"`.
- `"malayalam is a nice language"` ist nicht magisch, da es nur 2 Palindrome gibt, aber 5 Wörter.

Anmerkung: Natürlich dürfen Sie `palindrome` und `partition` verwenden, auch wenn Sie die entsprechenden Aufgaben nicht gelöst haben.

(d) Definieren Sie eine Haskell-Funktion `subPalindromes`, so dass `subPalindromes xs` eine Liste aller nicht-trivialen Palindrome erzeugt, die als Teil-Listen von `xs` vorkommen. (12)

- ein nicht-triviales Palindrom hat eine Länge von mindestens 3.
- eine Teil-Liste von `xs` bekommt man, indem ein beliebiges Anfangsstück und Endstück von `xs` entfernt.

Beispiel: `subPalindromes "hello to otto and hannah"` sollte eine Liste erzeugt, die genau die Strings `"to ot"`, `"o o"`, `" otto "`, `"otto"`, `"hannah"` und `"anna"` enthält, wobei die Reihenfolge egal ist.

Tipp: List-Comprehensions könnten sinnvoll sein.

Aufgabe 3: Datatypen und Funktionen Höherer Ordnung 26

Betrachten Sie folgendes Programm.

```
import Data.List(nub, sort)
-- nub :: Eq a => [a] -> [a]
-- "nub" removes all duplicates from the given list
-- sort :: Ord a => [a] -> [a]
-- sum :: Num a => [a] -> a
-- "sum" computes the sum of all elements in a list of numbers
-- map :: (a -> b) -> [a] -> [b]
data Tree a = Tree a [Tree a]

node (Tree x _) = x
subtrees (Tree _ ts) = ts
mapTree f (Tree x ts) = Tree (f x) (map (mapTree f) ts)
foldTree f (Tree x ts) = f x (map (foldTree f) ts)
```

(a) Geben Sie den allgemeinsten Typ von `node`, `subtrees` und `mapTree` an. (4)

(b) Angenommen, wir möchten eine Funktion `sumTrees :: [Tree Int] -> Int` definieren, die die Summe aller Knoten in einer Liste von Integer-Bäumen berechnen soll. (4)

Beispiel: `sumTrees [Tree 3 [], Tree 2 [Tree 1 [], Tree 4 []]] = 3 + 2 + 1 + 4 = 10`

Genau eine der folgenden Gleichungen stellt eine geeignete Implementierung dar. Identifizieren Sie diese. (4 Punkte für die korrekte Antwort, 1 Punkt für keine Antwort, 0 Punkte für eine falsche Antwort)

- ☐ `sumTrees = sum . subtrees`
- ☐ `sumTrees = sum . map node`
- ☐ `sumTrees = sum . map (mapTree id)`
- ☐ `sumTrees = sum . map (foldTree (\ x xs -> x + sum xs))`

(c) Wir möchten eine Funktion `cumulativeSum :: Tree Integer -> Tree Integer` definieren, die jeden Knoten in einem Integer-Baum durch die Summe aller Zahlen in dem entsprechenden Teilbaum ersetzt. (6)

Beispiel: `cumulativeSum (Tree 1 [Tree 1 [], Tree 1 [Tree 1 [], Tree 1 []]]) = Tree 5 [Tree 1 [],Tree 3 [Tree 1 [],Tree 1 []]]`

Nehmen Sie an, die Implementierung soll über `foldTree` erfolgen.

```
cumulativeSum = foldTree undefined
```

Ersetzen Sie `undefined` durch einen entsprechenden λ-Ausdruck oder erklären Sie, warum `cumulativeSum` nicht mittels `foldTree` definiert werden kann.

(d) Nun betrachten Sie die Funktion `set :: Ord a => Tree a -> [a]`, die einen Baum in eine sortierte Liste von Knoten überführt, wobei alle doppelten Vorkommen entfernt werden sollen. (12)

Beispiel: `set (Tree 1 [Tree 1 [], Tree 2 [Tree 1 [], Tree 1 []]]) = [1,2]`. Betrachten Sie drei mögliche Versuche, `set` zu implementieren:

```
set1 = sort . nub . foldTree (\x -> concat)
set2 = nub . sort . foldTree (\x ts -> x : concat ts)
set3 = sort . nub . mapTree id
```

Geben Sie für jede der Funktionen `set1`, `set2` und `set3`, an, ob Sie eine korrekte Implementierung von `set` sind oder nicht; für die fehlerhaften Funktionen sollen Sie zudem kurz das Problem beschreiben.

Aufgabe 4: Auswertung und Typen 12

Zu jeder Frage gibt es genau eine richtige Antwort. Das Ankreuzen dieser Antwort ergibt 4 Punkte; kein Kreuz zu machen ergibt 1 Punkt; das Ankreuzen einer falschen oder mehrerer Antworten ergibt 0 Punkte.

Betrachten Sie folgendes Programm.

```
foo = bar 0
bar _ [] = []
bar x (y:ys) = (x + y) : bar (x + y) ys
```

(a) Was ist der allgemeinste Typ von `foo`? (4)

- ☐ `[Int] -> Int`
- ☐ `Num a => [a] -> [a]`
- ☐ `[Int] -> [Int]`
- ☐ `[a] -> [a]`

(b) Was ist das Ergebnis der Auswertung von `foo [1,2,3,4,5]`? (4)

- ☐ `[1,3,6,10,15]`
- ☐ `[0,1,3,6,10]`
- ☐ `15`
- ☐ Keine der obigen Antworten.

(c) **Nicht mit dem Wissen aus Vorlesungen 1–10 lösbar!** (4)

Angenommen, `foo xs` wird für eine endliche Liste `xs :: [Int]` aufgerufen.

Welche Aussage ist korrekt?

- ☐ Keine der folgenden Antworten.
- ☐ Der Speicherverbrauch ist konstant, sowohl für die Innermost-Strategie und für Lazy-Evaluation.
- ☐ Der Speicherbedarf wird bei der Innermost-Strategie kontinuierlich wachsen, da die Auswertung mit Innermost-Strategie zu einer unendlichen Berechnung führt.
- ☐ Der Speicherbedarf ist konstant mit Lazy-Evaluation, aber steigt linear in der Länge von `xs` bei Innermost-Strategie.