Name: ______________________________

0.1 Multiple Choice

Ja Nein

☐ ☐ TODO

0.2 Funktionale Programmierung

(a) Ihnen sind die folgenden Funktionsdefinitionen gegeben, jedoch fehlen die Signaturen. Ergänzen Sie die Signaturen mit dem allgemeinstmöglichen Typ.

```
a ::

a x y z = (y x) && (y z)


b ::

b x y = b (x + head y) (tail y)


c ::

c x (y:z) = c [y] x ++ z


d ::

d x (y:z) = d (y x) z
```

(b) Implementieren Sie die Funktion *foldl*, die mit Hilfe einer Funktion und einem Startwert eine Liste von links ausgehend zu einem Ergebnis zusammenfaltet.

```
foldl ::
```

Implementieren Sie die Funktion *foldr*, die mit Hilfe einer Funktion und einem Startwert eine Liste von rechts ausgehend zu einem Ergebnis zusammenfaltet.

```
foldr ::
```

Implementieren Sie die Funktion *map*, die eine Funktion und eine Liste von Werten nimmt und als Ergebnis eine Liste mit den Funktionswerten liefert. Beispiele:

```
map ((+) 3) [0,1,2,3]             => [3,4,5,6]
map length [[], [2,3,4], [2,1]]   => [0,3,2]

map ::
```

0.3 Funktionale Programmierung

Gegeben sei der folgende Datentyp, der einen Binärbaum darstellt, dessen Knoten und Blätter mit Bruchzahlen markiert sind.

```
data FracTree = FracNode FracTree FracTree Integer Integer
              | FracLeaf Integer Integer
```

Der erste Integer entspricht dem Zähler, der zweite dem Nenner; d.h.

```
(FracLeaf 5 7)
```

entspricht dem Bruch: $\frac{5}{7}$

(a) Implementieren Sie eine Funktion *countN*, die zählt, wie viele Knoten eines *FracTree* natürliche Zahlen sind. Geben Sie auch eine sinnvolle Signatur an. Beispiele:

```
countN (FracNode (FracLeaf 2 5) (FracLeaf 2 1) 3 8) == 1
countN (FracNode (FracLeaf 12 3) (FracLeaf 14 7) 2 4) == 2
```

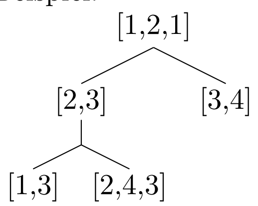
(b) Implementieren Sie eine Funktion *infty*, die testet, ob ein FracTree einen Bruch mit Nenner 0 enthält. Geben Sie auch eine sinnvolle Signatur an. Beispiele:

```
infty (FracNode (FracLeaf 2 5) (FracLeaf 2 1) 3 8) == False
infty (FracNode (FracLeaf 2 5) (FracLeaf 2 0) 3 8) == True
```

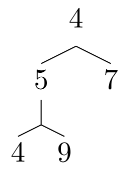
0.4 Funktionale Programmierung

Gegeben ist eine Datenstruktur *ListTree*, die einen Binärbaum darstellt, dessen Knoten und Blätter mit Listen von Integern markiert sind. Diese Datenstruktur verfügt über die Konstruktoren *ListNode ListTree ListTree [Integer]* und *ListLeaf [Integer]*. Außerdem ist die Datenstruktur *IntTree* mit Integer-Markierungen gegeben. Diese hat die Konstruktoren *IntNode IntTree IntTree Integer* und *IntLeaf Integer*.

(a) Schreiben Sie eine Funktion *sumUpToIntTree*, die einen ListTree bekommt und einen IntTree zurückgibt. Dabei soll an jeder entsprechenden Markierung des zurückgegebenen Baumes die Summe der Werte des Eingabebaums stehen. Vergessen Sie die Signatur nicht. Beispiel:



wird zu




```
sumUpToIntTree ::
```

0.5 Sprachen

Gegeben: Sprachdef. mit Syntaxdiagrammen Frage: Sind Wörter in Sprache

Def Grammatik, die nur das leere Wort enthält

TODO

0.6 Terminierung

(a) Geben Sie für folgende Funktionen je die größtmögliche Parametermenge an, sodass sie terminieren

```
f :: Int -> String -> Int
f 0 _ = 77
f x y = f (x - 1 + length y) (tail y)

g :: Int -> String -> Int
g 0 _ = 77
g _ "" = 88
g x y = g (x - 1 + length y) (tail y)

h :: Int -> String -> Int
h _ "" = 88
h 0 _ = 77
h x y = h (x - 1 + length y) (tail y ++ tail y)
```

(b) Geben Sie für die folgende Funktion den größtmöglichen Parameterbereich an und beweisen sie, dass sie terminiert:

```
t :: Int -> Int -> Int
t 0 0 = 0
t 0 x = t x + (x ‘div‘ 2)
t x y = y * (t (x - (x ‘div‘ 2)) (x + x ‘div‘ 2))
```

0.7 Funktionsgleichheit

Beweisen Sie, dass folgende Funktionen für alle $n \in \mathbb{N}$ das gleiche berechnen:

$f : N \to N$

TODO

0.8 Prozedurale Programmierung

Gegeben sind folgende Datenstrukturen:

```
class Vektor {
  double[] values;
}

class Matrix {
  double[][] values;
}
```

Schreiben Sie eine Prozedur *skalarMat(Vektor x, Vektor y, Matrix A)*, die das wie folgt definierte Skalarprodukt berechnet: $\langle x, y\rangle_A = x^T Ay$.

Dabei bezeichnet x^T den transponierten Vektor; die Matrixmultiplikation ist rechtsassoziativ.

```
  double skalarMat(Vektor x, Vektor y, Matrix A) {
```

0.9 Prozedurale Programmierung

TODO

0.10 Prozedurale Programmierung

TODO

0.11 Objektorientierte Modellierung

(a) Modellieren Sie folgendes Szenario als UML-Klassendiagramm mit Attributen und Methoden. Vergessern Sie die Multiplizitäten dabei nicht. Treffen Sie für nicht angegebene Informationen vernünftige Annahmen.

In einem Handballverein sind Mitglieder entweder Spieler oder Trainer. Alle Mitglieder haben einen Namen und eine Mitgliedsnummer. Spieler haben zusätzlich eine Spielernummer. Der Verein hat mehrere Mannschaften, die alle einen Trainer und einen Co-Trainer haben.

(b) Ergänzen Sie ihr UML-Diagramm um weitere Klassen gemäß der folgenden Beschreibung. Dabei brauchen Sie Attribute und Methoden nicht einzeichnen.

Wir betrachten jetzt den Handballverband. In ihm gibt es mehrere Ligen, in denen jeweils die Mannschaften mehrerer Vereine spielen. Dies tun sie in Hallen. Jedem Verein werden Hallen zur Verfügung gestellt, die dann in einem Spiel die Heimhalle sind.

0.12 Objektorientierte Programmierung - Implementierung

Sie haben in der Übung bereits gebräuchliche Datenstrukturen implementiert. Dabei fehlt bisher noch die *Menge*. Eine Menge hat folgende Eigenschaften:

1. Eine Menge besteht aus Elementen eines Typs T
2. Eine Menge kann leer sein
3. Eine Menge ist duplikatfrei
4. Mit der Methode *add(T t)* wird ein neues Element der Menge hinzugefügt
5. Mit der Methode *remove(T t)* wird das Element t - falls vorhanden - aus der Menge entfernt
6. Damit man die Daten einfach auch mit anderen Algorithmen benutzen kann, gibt die Methode *toArray()* alle (noch nicht gelöschten) Elemente der Menge zurück.