technische universität
dortmund

Klausurdeckblatt

Matrikel – Nr.:

Bitte tragen Sie ihre Matrikelnummer und ihren Namen in die dafür vorgesehenen Felder ein. Bitte in deutlicher Handschrift mit einem schwarzen Stift (nicht Bleistift)
Das Feld mit dem **Barcode ist unbedingt frei zu lassen.**
Danke.

Vorname:

Nachname:

Der Bereich unterhalb dieser Linie kann von der Fakultät frei gestaltet werden.

Klausur zur Vorlesung

Funktionale Programmierung

18.09.2024

Aufgabe	1	2	3	4	5	6	Σ
Punkte	10	10	10	10	10	10	60
erreicht							

In der Klausur sind insgesamt **60 Punkte** erreichbar.
Zum Bestehen sind mindestens **30 Punkte** (50 %) erforderlich.

Wir wünschen viel Erfolg!

Nach der Korrektur wird Ihre Note unter einem dreistelligen Pseudonym-Code veröffentlicht, den Sie von der Aufsicht erhalten. Tragen Sie diesen Code hier ein:

Hinweis: Wir als Klausurveranstalter sind organisatorisch nicht dazu in der Lage, vor bzw. während der Klausur zu überprüfen, ob Teilnehmer/-innen dazu berechtigt sind, die Klausur mitzuschreiben. Daher gilt folgendes: *Durch die Teilnahme an der Klausur erkennt der Teilnehmer bzw. die Teilnehmerin an, dass diese unter Vorbehalt stattfindet. Die Anerkennung der bei der Klausur erzielten Note hängt von der jeweils zuständigen Stelle ab und ist nicht automatisch durch die Teilnahme an der Klausur gegeben.*

Diese Seite bitte nicht beschriften!

Diese Seite bitte nicht beschriften!

Name, Vorname, Matrikelnummer | Bitte unbedingt leserlich ausfüllen!

Aufgabe 1 (Bäume) (3 + 3 + 4 = 10 Punkte)

Gegeben sei der folgende Haskell-Datentyp für binäre Bäume.

```
data Baum a = Leer | Knoten a (Baum a) (Baum a)
```

1. Definieren Sie eine Haskell-Funktion mit der Signatur
   `hoehe :: Baum a -> Int`,
   die die Höhe eines gegebenen Baums berechnet.

   Hinweis: Die Höhe eines Baums ist die maximale Anzahl der `Knoten`-Konstruktoren auf einem Pfad von der Wurzel zu einem `Leer`-Konstruktor. Der leere Baum hat Höhe 0.

2. Definieren Sie eine Haskell-Funktion mit der Signatur
   `preorder :: Baum a -> [a]`,
   die einen Binärbaum traversiert und die Elemente in preorder-Reihenfolge als Liste zurückgibt.

   Hinweis: Bei der preorder-Reihenfolge für einen inneren Knoten wird erst die Wurzel, dann der linke Teilbaum und schließlich der rechte Teilbaum ausgegeben.

3. Machen Sie `Baum` zu einer sinnvollen Instanz der Typklasse `Functor`.

Aufgabe 2 (Listenmonade) $(3 + 3 + 4 = 10$ Punkte)

1. Definieren Sie mittels Listenkomprehension eine unendliche Haskell-Liste `solutions :: [(Integer, Integer, Integer)]`, die genau die Tripel $(x, y, z) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ enthält, die eine Lösung der Gleichung $5x + y^2 + 10 = z$ sind.
   Die Liste muss produktiv sein.
   Das heißt, dass z.B. ein Aufruf `take 2 solutions` terminieren muss. Eine mögliche Ausgabe für `take 2 solutions` kann `[(0,0,10),(0,1,11)]` sein.

2. Gegeben sei die folgende Haskell-Funktion.

```haskell
mapMb :: Monad m => (a -> m b) -> [a] -> m [b]
mapMb f [] = return []
mapMb f (a:as) = f a >>= \b -> mapMb f as >>= \bs -> return $ b : bs
```

   Definieren Sie eine Haskell-Funktion mit der Signatur
   `mapM :: Monad m => (a -> m b) -> [a] -> m [b]`,
   indem Sie die `(>>=)`-Notation in `mapMb` in die do-Notation übersetzen. `mapMb` und `mapM` sollen für alle Eingaben dieselbe Ausgabe berechnen.

3. Definieren Sie eine Haskell-Funktion mit folgender Signatur:

```haskell
tryMap :: (a -> Maybe b) -> [a] -> Maybe [b]
```

   Analog zur Haskell-Funktion `map :: (a -> b) -> [a] -> [b]` soll `tryMap f xs` die Funktion `f` auf jedes element der Liste `xs` anwenden. Falls eine der Funktionsanwendungen `Nothing` ergibt, dann ist das Gesamtergebnis `Nothing`.
   Falls für `xs` = `[x1, x2, x3, ...]` gilt
   `[f x1, f x2, f x3, ...]` = `[Just y1, Just y2, Just y3, ...]`, dann ist das Gesamtergebnis `Just [y1, y2, y3, ...]`.

   Nutzen Sie zur Definition die do-Notation auf sinnvolle Weise.

Aufgabe 3 (Binärzahlen) $(3 + 3 + 2 + 2 = 10$ Punkte$)$

Gegeben sei der folgende Haskell-Datentyp für binär-kodierte natürliche Zahlen:

```
data Bin = LSB | Zero Bin | One Bin
```

Der Konstruktor `LSB` dient als „Markierung“ des niedrigsten Stellenwerts.
So entspricht z.B.

`LSB` der Zahl 0,
`Zero LSB` der Zahl 0,
`One LSB` der Zahl 1,
`One (Zero LSB)` der Zahl 2,
`One (Zero (Zero LSB))` der Zahl 4 und so weiter.

1. Definieren Sie eine Haskell-Funktion mit der Signatur
   `foldB :: a -> (a -> a) -> (a -> a) -> Bin -> a`,
   die der aus der Vorlesung bekannten Semantik einer Faltung des Datentypen `Bin` entspricht.

2. Definieren Sie eine Haskell-Funktion mit der Signatur
   `shift :: Bin -> Bin`,
   die den Wert eines Terms vom Typ `Bin` verdoppelt. `shift` soll einen Bitshift implementieren, der an der Stelle mit dem niedrigsten Stellenwert ein 0-Bit hinzufügt. Als Beispiele der Verdopplung, bzw. des Bitshifts können die obigen Beispielterme für die Zahlen $1, 2$ und 4 dienen.

   Nutzen Sie zur Definition die Faltung
   `foldB :: a -> (a -> a) -> (a -> a) -> Bin -> a`
   auf sinnvolle Weise.

3. Definieren Sie eine Haskell-Funktion mit der Signatur
   `rlz :: Bin -> Bin`,
   die alle führenden Nullen einer Binärzahl entfernt. `rlz` darf den Wert der Zahl nicht verändern.

4. Machen Sie `Bin` zu einer sinnvollen Instanz der Typklasse `Eq`. Unterschiedliche Repräsentationen einer Zahl sollen gleich sein. Die Funktion `rlz` darf als bekannt vorausgesetzt werden.

Aufgabe 4 (State-Monade) ($3 + 3 + 4 = 10$ Punkte)

Gegeben seien die aus der Vorlesung bekannten Haskell-Datentypen `Nat` und `State`, sowie die Instanzen der Typklassen `Functor`, `Applicative` und `Monad` für `State`.

```haskell

data Nat = Z | S Nat

newtype State s a = State {runS :: s -> (a, s)}

instance Functor (State s) where
  fmap f (State h) = State ((\(x,s) -> (f x, s)) . h)

instance Applicative (State s) where
  pure a = State (\s -> (a, s))
  State f <*> h = State ((\(g,s) -> let (x, s’) = runS h s in (g x, s’)) . f)

instance Monad (State s) where
  return a = State (\s -> (a, s))
  State h >>= f = State ((\(a,s) -> runS (f a) s) . h)
```

Im Folgenden wollen wir `State` nutzen, um eine Stack-Maschine zu implementieren.

Hierzu seien ebenfalls die folgenden zustandsbehafteten Varianten der Funktionen `push` und `pop` gegeben. Dabei entspricht der Zustand von `State` dem Stack.

```haskell
type Stack = [Int]

push :: Int -> State Stack ()
push i = State $ \ls -> ((), i : ls)

pop :: State Stack (Maybe Int)
pop = State $ \ls -> case ls of
  [] -> (Nothing, [])
  (x : xs) -> (Just x, xs)
```

1. Definieren Sie eine Haskell-Funktion mit der Signatur
   `clear :: State Stack Stack`,
   die den gesamten Stack zurückgibt und den leeren Stack als neuen Zustand setzt.

2. Definieren Sie eine Haskell-Funktion mit der Signatur
   `pushN :: [Int] -> State Stack ()`,
   die eine Liste als Argument nimmt und alle Elemente derart auf den Stack legt, dass die Reihenfolge aus der Liste erhalten bleibt und der Head der Liste als oberstes Element auf dem Stack liegt. Nutzen Sie zur Definition die do-Notation auf sinnvolle Weise.

3. Definieren Sie eine Haskell-Funktion mit der Signatur
   `popN :: Nat -> State Stack [Int]`,
   die ein `n` vom Typ `Nat` als Argument nimmt und die obersten `n` Elemente vom Stack entfernt und zurückgibt. Sollten weniger als `n` Elemente auf dem Stack liegen, so werden alle Elemente des Stacks zurückgegeben. Nutzen Sie zur Definition die do-Notation auf sinnvolle Weise.

Aufgabe 5 (Lambda Kalkül) (4 + 6 = 10 Punkte)

1. Beta-Reduzieren Sie den Lambda-Term $\Big(\lambda x.\ (\lambda y.\ y)\ \Big(\lambda z.\ z\ (\lambda a.\ x\ a)\ z\Big)\ (\lambda v.\ \lambda w.\ v)\Big)$ bis zur Normalform. Geben Sie alle Reduktionsschritte an.
2. Inferieren Sie den Typ des Lambda-Terms $\lambda y.\Big(x\ \Big(\lambda z.y\ (x\ y)\Big)\Big)$ in der Typumgebung $\{x : (a \rightarrow b) \rightarrow a\}$ im einfach getypten Lambda-Kalkül. Geben Sie den vollständigen Ableitungsbaum an und benennen Sie alle Regelanwendungen.

Aufgabe 6 (Church-Kodierungen) $(3 + 3 + 4 = 10$ Punkte$)$

Gegeben seien die folgenden Haskell-Datentypen für Wahrheitswerte und natürliche Zahlen.

```
data Bool = True | False

data Nat = Z | S Nat
```

Außerdem seien folgende Church-Kodierungen der Konstruktoren von `Bool` und `Nat` im ungetypten Lambda-Kalkül gegeben.

$$\mathit{true} = \lambda t.\lambda f.\ t$$
$$\mathit{false} = \lambda t.\lambda f.\ f$$

$$\mathit{zero} = \lambda z.\lambda s.\ z$$
$$\mathit{succ} = \lambda n.\lambda z.\lambda s.\ s\ (n\ z\ s)$$

1. Geben Sie einen Lambda-Term *even* im ungetypten Lambda-Kalkül an, der prüft ob eine Church-kodierte natürliche Zahl gerade ist. Appliziert auf eine Church-kodierte natürliche Zahl soll *even* also zu dem entsprechenden Church-kodierten `Bool` reduzieren.

2. Geben Sie eine Church-Kodierung für den Haskell-Datentyp der Binärzahlen

   ```
   data Bin = LSB | Zero Bin | One Bin
   ```

   im ungetypten Lambda-Kalkül an, indem Sie Lambda-Terme *lsb*, *zero* und *one* angeben, die den Church-Kodierungen der drei Konstruktoren `LSB`, `Zero` und `One` entsprechen.

3. Geben Sie einen Lambda-Term *shift* im ungetypten Lambda-Kalkül an, der eine Church-kodierte Binärzahl verdoppelt. Appliziert auf eine Church-kodierte Binärzahl soll *shift* also zu einer Church-kodierten Binärzahl reduzieren, die an der Stelle mit dem niedrigsten Stellenwert ein 0-Bit hinzugefügt hat.

Name, Vorname, Matrikelnummer

Bitte unbedingt leserlich ausfüllen!