Prüfung Funktionale Programmierung

Datum	:	04.02.2016, 08:30 Uhr
Bearbeitungszeit	:	90 Minuten
Prüfer	:	Prof. Dr. Oliver Braun
Hilfmittel	:	Keine
Erreichbare Punkte	:	93

Name: _____

Vorname: _____

Matrikelnummer: _____ Studiengruppe: _____

Hörsaal: _____ Platz Nr.: _____

Unterschrift: _____

Bitte kontrollieren Sie, ob Sie eine vollständige Angabe mit 5 Aufgaben auf 8 Seiten erhalten haben.

Aufgabe	1	2	3	4	5	Summe
max. Punkte	16	18	21	20	18	93

Anmerkungen:

- Sie müssen als Antworten **keine** kompletten Programme schreiben, sondern nur den explizit verlangten Teil eines Programms.
- Schreiben Sie die Lösungen in die dafür vorgesehenen Kästchen. Sollte Ihnen der Platz dabei nicht reichen, benutzen Sie die Rückseite **und vermerken Sie das im dazugehörigen Kästchen!**
- Die Rückseiten können Sie ansonsten für Nebenrechnungen etc. nutzen.

Aufgabe 1 (16 Punkte)

Ergänzen Sie die Typsignaturen für die folgenden Funktionen:

Hinweis: Denken Sie daran, dass in Haskell Literale überladen sein können, z.B.
`1 :: Num a => a`

(a) (4)

```
fun1 ::
fun1 x y z = return $ x ++ show y ++ z
```

(b) (4)

```
fun2 ::
fun2 a c d = do
    b <- a
    if b then a else Just c
```

(c) (4)

```
import Control.Applicative ((<$>), (<*>))
```

```
fun3 ::
fun3 a b = case a of
    Nothing -> a
    _       -> (++) <$> a <*> Just b
```

(d) (4)

```
fun4 ::
fun4 a b = \c -> foldr a 0 $ replicate c b
```

Aufgabe 2 (18 Punkte)

Eine Dualzahl soll als Liste aus den Ziffern 0 und 1 dargestellt werden. Dazu wird ein eigener Datentyp und ein Typsynonym wie folgt definiert:

```
data Dualdigit = Zero | One
type Dual = [Dualdigit]
```

(a) Implementieren Sie die Funktion (8)

```
dual2Int :: Dual -> Int
```

die eine Dualzahl in eine Dezimalzahl umwandelt. Nutzen Sie dazu mindestens eine Higher Order Function.

(b) Implementieren Sie die Funktion (10)

```
int2dual :: Int -> Dual
```

so dass gilt `int2dual . dual2Int == id` und `dual2Int . int2dual == id`.

Aufgabe 3 (21 Punkte)

Gegeben sei folgender Datentyp:

```
data Games a = Running [a] | GameOver
```

für eine Spiele-Monade. Solange alle Spiele laufen, sollen `Running`-Werte miteinander verknüpft werden können. Der Wert `GameOver` dient sowohl als Spielende-Zustand als auch als Fehlerzustand. (Tipps: Denken Sie an `Maybe` und nutzen Sie aus, dass Listen auch Funktoren, applikative Funktoren und Monaden sind).

(a) Geben Sie für den Datentyp `Games` eine Instanz der Klasse `Functor` an. (7)

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

(b) Geben Sie für den Datentyp `Games` eine Instanz der Klasse `Applicative` an. (7)

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

(c) Geben Sie für den Datentyp `Games` eine Instanz der Klasse `Monad` an. (7)

```
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    fail :: String -> m
    fail msg = error msg
```

Sie können für die Definition von bind die folgende Funktion verwenden:

```
flattenGames :: [Games a] -> Games a
flattenGames (GameOver : _) = GameOver
flattenGames (game : []) = game
flattenGames (Running xs : games) = case flattenGames games of
  GameOver -> GameOver
  Running xss -> Running $ xs ++ xss
```

Aufgabe 4 (20 Punkte)

Geben Sie das Ergebnis der folgenden Ausdrücke an. Sie können Ihre Nebenrechnungen im entsprechenden Kästchen machen. Unterstreichen Sie in dem Fall die Lösung.

(a) `foldr (+) 12 [1,4..21]` (2)

(b) `drop 7 $ concat [ map (+1) x | x <- map (:[]) [1..10]]` (6)

(c) `product $ take 123 $` (6)
`(\xs -> zipWith (-) (filter odd xs) (filter even xs)) [1..]`

(d) `foldr (+) 12 . map (foldr (*) 1) . replicate 10 $ [12..12]` (6)

Aufgabe 5 (18 Punkte)

Gegeben seien folgende Datentypen und Typsynonyme für Pizzen:

```
data Pizza = Pizza
    { size :: Size
    , toppings :: [Topping]
    }
data Topping = Topping
    { name :: String
    , price :: Cent
    }
type Size = Int
type Cent = Int
```

(a) Definieren Sie eine Funktion (8)

`pizzaPrice :: Pizza -> Cent`

den Preis einer Pizza in Cent berechnet. Der Preis setzt sich zusammen aus einem Preis für die Größe, den die, bereits implementierte, Funktion `size2Price :: Size -> Cent` berechnet, sowie der Summe der Preise für die zusätzlichen Toppings.

(b) Gegeben seien außerdem (10)

```
type Seconds = Int
data Pizzeria = Pizzeria
    { noOfOvens :: Int
    , pizzaQueue :: [Pizza]
    }
type Distance = Int
data Customer = Customer
    { distance :: Distance
    }
```

Definieren Sie eine Funktion

```
timeToDelivery :: Pizza -> Customer -> Pizzeria -> Seconds
```

die die Lieferzeit der bestellten Pizza berechnet. Diese setzt sich zusammen aus der Vorbereitungszeit der Pizza und aller Pizzen die noch in der Queue sind, der Backzeit für alle Pizzen und der Auslieferungszeit zum Kunden. Für eine Pizza egal welcher Größe wird eine Vorbereitungszeit von 10 Minuten ohne Toppings berechnet. Für jedes Topping werden 20 Sekunden mehr benötigt. Die Backzeit beträgt für jede Pizza 10 Minuten. In einem Ofen kann immer nur eine Pizza gebacken werden. Hat die Pizzeria ‘n‘ Öfen, können ‘n‘ Pizzen gleichzeitig gebacken werden. Bei einem Kunden wird die Entfernung in Kilometer gespeichert. Pro Kilometer wird eine Lieferzeit von 3 Minuten berechnet.