Übungsaufgabe 3

Ziel

In dieser Übungsaufgabe geht es darum, die Erstellung eigene Datentypen und die Implementierung von Typklassen zu üben.

Diese Übung lässt sich nicht mehr durch Recherche auf Stackoverflow lösen. Wenn Du die vorherigen Aufgaben nicht durch nachdenken gelöst hast, wird sich das hier bitter rächen.

Intervall Map

Ziel dieser Aufgabe ist es eine Intervall Map zu implementieren. Eine Intervall Map hat zwei Typparameter, einmal für den Schlüssel (k) und einmal für den Wert (v):

```
data IntervalMap k v = …
```

Im Gegensatz zu einer normalen Map, wo nur für einzelne Schlüssel ein Wert definiert ist, gibt eine Intervall Map für jeden Schlüssel auch einen Wert zurück. Deshalb wird für die Initialisierung einer Intervall Map auch immer ein Startwert benötigt. Hierfür muss die Funktion singleton implementiert werden, die folgende Signatur hat:

```
singleton :: v -> IntervalMap k v
```

Eine neue Intervall Map von Int auf Char mit dem Wert „a“ lässt sich dann durch folgenden Ausdruck erzeugen:

```
a = singleton 'a' :: IntervalMap Int Char
```

Diese Intervall Map würde nun für jeden möglichen Schlüssel immer das Zeichen „a“ zurückgeben. Für die Abfrage der Map ist der ! Operator zu implementieren. Dieser hat folgende Signatur:

```
(!) :: Ord k => IntervalMap k v -> k -> v
```

Der erste Parameter ist somit die Intervall Map die abgefragt werden soll und der zweite Parameter ist der Schlüssel, der verwendet wird. Würden wir unsere Intervall Map a abfragen, erhalten wir folgendes Ergebnis:

```
> a ! 5
'a'
```

Zum Testen empfiehlt es sich Teile der Intervall Map abzufragen und mittels List Comprehension Tupel zu generieren, die Schlüssel und Wert beinhalten, z. B.

```
[(x, a ! x) | x <- [1..25]]
```

Einzelne Teile der Intervall Map werden durch die Funktion insert verändert. Diese Funktion hat folgende Signatur:

```
insert :: Ord k => k -> k -> v -> IntervalMap k v -> IntervalMap k v
```

Der erste Parameter ist hierbei der erste Schlüssel ab dem ein neuer Wert gesetzt werden soll, der zweite Parameter ist das Ende des Bereichs, der dritte Parameter ist der Wert für diesen Bereich und der vierte Parameter ist die Intervall Map, die verändert werden soll. Die veränderte Intervall Map wird anschließend zurückgegeben. Wichtig ist hierbei zu beachten, dass sich teilweise überschriebene Intervalle anschließend fortsetzen. Nehmen wir an, ich habe eine Intervall Map, die

mit ‚a' initialisiert ist, dann würde der Bereich 1 – 10 die Zeichenkette „aaaaaaaaaa" sein. Mache ich nun ein Insert im Bereich 3 bis 10 würde der Bereich 1 – 10 folgende Zeichnkette ergeben: „aabbbbbbba". Führe ich anschließend noch ein Insert in den Bereich 5 – 7 durch ergibt sich folgende Zeichenkette für den Bereich 1 – 10 „aabbccbbba".

Im Folgenden ein paar Definitionen zu Testen:

```
a = singleton 'a' :: IntervalMap Int Char
b = insert 10 20 'b' a
c = insert 9 21 'c' b
d = insert 5 15 'd' c
e = insert 14 22 'e' d
f = insert 10 19 'f' e
```

Werden nun die ersten 25 Werte abgefragt sollten sich folgende Zeichenketten ergeben:

```
> [a ! x| x <- [1..25]]
"aaaaaaaaaaaaaaaaaaaaaaaaa"
> [b ! x| x <- [1..25]]
"aaaaaaaaabbbbbbbbbbaaaaaa"
> [c ! x| x <- [1..25]]
"aaaaaaaaccccccccccccaaaaa"
> [d ! x| x <- [1..25]]
"aaaaddddddddddcccccaaaaa"
> [e ! x| x <- [1..25]]
"aaaaddddddddeeeeeeeeaaaa"
> [f ! x| x <- [1..25]]
"aaaaddddfffffffffeeeaaaa"
```

Intervall Map zum Funktor machen

Ein Funktor hat genau einen Typparameter. Legen wir den ersten Typparameter unserer Intervall Map fest, dann können daraus einen Funktor machen. Dies macht auch Sinn, weil es dann das Mapping über die Werte erlaubt. Mache also im zweiten Teil der Aufgabe die Intervall Map zu einem Funktor:

```
instance Functor (IntervalMap k) where
```

Anschließend sollten folgende Tests funktionieren:

```
g = fmap fromEnum f
h = "Hello" <$ g
> [(x,g ! x)| x <- [1..25]]
[(1,97),(2,97),(3,97),(4,97),(5,100),(6,100),(7,100),(8,100),(9,100),(10,10
2),(11,102),(12,102),(13,102),(14,102),(15,102),(16,102),(17,102),(18,102),
(19,101),(20,101),(21,101),(22,97),(23,97),(24,97),(25,97)]
> [(x,h ! x)| x <- [1..25]]
[(1,"Hello"),(2,"Hello"),(3,"Hello"),(4,"Hello"),(5,"Hello"),(6,"Hello"),(7
,"Hello"),(8,"Hello"),(9,"Hello"),(10,"Hello"),(11,"Hello"),(12,"Hello"),(1
3,"Hello"),(14,"Hello"),(15,"Hello"),(16,"Hello"),(17,"Hello"),(18,"Hello")
,(19,"Hello"),(20,"Hello"),(21,"Hello"),(22,"Hello"),(23,"Hello"),(24,"Hell
o"),(25,"Hello")]
```

Intervall Map zum applikativen Funktor machen

Im letzten Schritt soll die Typklasse Applicative für die Intervall Map implementiert werden:

```
instance Ord k => Applicative (IntervalMap k) where
```

Zum Test sollen drei Intervall Maps erstellt, wobei eine Intervall Map Operatoren beinhaltet und die anderen beiden Zahlenwerte:

```
i = insert 5 10 110 $ insert 10 15 90 $ singleton 100 :: IntervalMap Int
Int
j = insert 5 10 (-) $ insert 10 15 (*) $ singleton (+) :: IntervalMap Int
(Int -> Int -> Int)
k = insert 3 18 2 $ singleton 10 :: IntervalMap Int Int
```

Anschließend sollen die Operatoren in j auf die Zahlen in i und k angewendet werden.

```
l = j <*> i <*> k
```

Das Ergebnis sollte wie gefolgt aussehen:

```
> [(x,l ! x)| x <- [1..20]]
[(1,110),(2,110),(3,102),(4,102),(5,108),(6,108),(7,108),(8,108),(9,108),(1
0,180),(11,180),(12,180),(13,180),(14,180),(15,102),(16,102),(17,102),(18,1
10),(19,110),(20,110)]
```