Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
24. Juni 2014

9. Übung zur Vorlesung
Programmierung und Modellierung

A9-1 *Hello World* Kreieren Sie mit GHC (nicht GHCI) eine ausführbare Datei, welche nach dem Start eine Begrüßung ausgibt und den Benutzer dazu auffordert, in je eine Zeile nacheinander zuerst Lieblingsfarbe und dann Lieblingstier eingeben. Danach soll das Programm noch einen String ausgeben, welcher zuerst Lieblingstier und dann Lieblingsfarbe wiedergibt:

```
> ghc helloTier.hs
...
> ./helloTier
Hi! Gib bitte zuerst Deine Lieblingsfarbe und dann
in die nächste Zeile Dein Lieblingstier ein:
Orange
Schildkröte
Psst, willst Du Schildkröte in Orange kaufen?
```

Hinweis: Das Beispiel zeigt eine Linux-Konsole. Je nach Betriebssystem kann der Aufruf einer ausführbaren Datei abweichen. Die Benutzereingabe waren "Orange" und "Schildkröte", der Rest wurde durch das Programm ausgegeben.

A9-2 *DO-Notation* Versuchen Sie, diese Aufgabe mit Papier und Bleistift zu lösen. Verwenden Sie GHC oder GHCI erst, wenn Sie nicht mehr weiter wissen. Was gibt das folgende Programm am Bildschirm aus? Wie oft wartet das Programm auf eine Benutzereingabe?

Hinweise: Suchen Sie sich aus, was Sie bei jeder Eingabeaufforderung jeweils eingeben – alle Ihre Eingaben sollten jedoch verschieden sein. Ignorieren Sie `hSetBuffering`, diese Aktion stellt lediglich sicher, dass das Programm auf allen Betriebssystemen gleich funktioniert.

```
import System.IO
main = do hSetBuffering stdout NoBuffering
          putStr "A: "
          a2 <- getLine
          b1 <- putStr "B: "
          let b2 = getLine
          let c1 = putStr "C: "
          c2 <- getLine
          putStr "D: "
          b2 <- b2
          putStrLn $ "A="++a2++" B="++b2++" C="++c2
```

A9-3 ***Input/Output***

a) Kreieren Sie mit GHC (nicht GHCI) eine ausführbare Datei, welche nach dem Start eine Begrüßung ausgibt und den Benutzer so lange nach dessen Namen fragt, bis dieser einen mit einem großen Buchstaben beginnenden Namen eingegeben hat. Danach beendet sich Ihr Programm.

Hinweis: Das Modul `Data.Char` könnte dazu nützliche Funktionen anbieten.

b) Erweitern Sie Ihr Programm aus der vorangegangenen Teilaufgabe, so dass nachdem der Namen akzeptiert wurde, das Programm den Benutzer seine weitere Funktionsweise durch eine Bildschirmausgabe erklärt – und dann natürlich dann auch so funktioniert:

- Wenn der Benutzer eine Zahl eingibt, erfolgt keine Ausgabe.
- Wenn der Benutzer keine Zahl eingibt, wird die Summe aller bisher eingegeben Zahlen ausgegeben.
- Wenn der Benutzer seinen Namen erneut eingibt beendet sich das Programm.

Hinweis: Zur Lösung reichen in Folien 11-28 und 11-30 vorgestellte IO-Funktionen aus. Verwenden Sie für die Summe einen Akkumulator, d.h. jede weitere Frage wird durch einen rekursiven Aufruf verursacht, welcher als zusätzlichen Funktionsparameter die bisher aufgelaufene Summe übergeben bekommt.

Als weitere Hilfestellung zur Vereinfachung können Sie die Funktion `readMaybe` aus dem Modul `Text.Read` der Standardbibliothek verwenden. Im Gegensatz zu `read` aus der Standardbibliothek liefert `readMaybe` anstatt einem echten Fehler lediglich `Nothing` zurück, falls der String nicht konvertiert werden kann.

c) *Zusatzaufgabe für Fortgeschrittene:* Implementieren Sie die vorangegangene Teilaufgabe erneut, ohne `DO` und ohne IO-Aktionen (`putStr`, `getLine`, ...) sondern nur mit einem einzigem Aufruf von `interact`.

H9-1 ***Maybe Monade*** **(0 Punkte)** (`.hs`-Datei als Lösung abgeben)
Vervollständigen Sie folgende Instanzdeklarationen! Versuchen Sie dabei so wenig wie möglich im Skript nachzuschlagen, da die Lösung dort praktisch drinsteht.

```
data Option a = None | Some a deriving (Show, Eq, Ord)

instance Monad Option where
  return = undefined                    -- TODO!
  (>>=)  = undefined                    -- TODO!

instance MonadPlus Option where
  mzero = undefined                     -- TODO!
  mplus = undefined                     -- TODO!
```

Hinweis: Falls `mplus` zwei `Some`-Werte erhält, so soll der erste bzw. linke Wert zurückgegeben werden.

H9-2 *Aktionskette* (5 Punkte) (`.hs`-Datei als Lösung abgeben)
Vervollständigen Sie in der Dateivorlage die Funktionen `chainAction1`, `chainAction2` und `chainAction3`, welche alle drei das gleiche tun sollen, so dass folgendes Beispiel in GHCI wie gezeigt abläuft:

```
> chainAction1 1 test1
1 -> 3
3 -> 4
4 -> 4
4 -> 9
9 -> 18
18
```

Die Dateivorlage ist dabei wie folgt:

```
import Control.Monad

chainAction1 :: Monad m => a -> [(a -> m a)] -> m a
chainAction1 = undefined -- !!! TODO !!!

chainAction2 :: Monad m => a -> [(a -> m a)] -> m a
chainAction2 = undefined -- !!! TODO !!!

chainAction3 :: Monad m => a -> [(a -> m a)] -> m a
chainAction3 = undefined -- !!! TODO !!!

tellOp :: (Show a, Show b) => (a -> b) -> a -> IO b
tellOp f x = let fx = f x in do
  putStrLn $ (show x) ++ " -> " ++ (show fx)
  return fx

test1 :: [Int -> IO Int]
test1 = map tellOp [(*3),(+1),(`mod` 7),(+5),(*2)]
```

a) Implementieren Sie `chainAction1` nur unter Verwendung von Rekursion und der DO-Notation, aber ohne Verwendung von Funktionen der Standardbibliothek! Lediglich `return` und `fail` sind erlaubt!

b) Implementieren Sie `chainAction2` wie in der vorangegangenen Teilaufgabe, aber jetzt ohne Verwendung der DO-Notation. Sie dürfen stattdessen alle Funktionen der Klasse `Monad` einsetzen, also `(>>)`, `(>>=)`, `return` und `fail`.

c) Implementieren Sie `chainAction3` noch ein drittes mal, dieses Mal jedoch mit umgekehrter Bedingung im Vergleich zu ersten Teilaufgabe: Sie dürfen weder direkte Rekursion, noch DO-Notation und auch keine Funktionen der Klasse `Monad` verwenden. Stattdessen dürfen Sie alle anderen Funktionen aus den Modulen `Prelude` und `Control.Monad` einsetzen!

H9-3 *Zustandsmonade* (3 Punkte) (`.hs`-Datei als Lösung abgeben)
In der zwölften Vorlesung wurde gezeigt, wie man eine Zustandsmonade zu Fuss implementieren kann, um einmal hinter die Monaden-Kulissen zu schauen. Dieser Code ist auf der Vorlesungshomepage verfügbar.

a) Erweitern Sie den vorhanden Code um eine Funktion
`modifyVar :: Name -> (Wert -> Wert) -> Env ()`
welche den Wert einer im Zustand gespeichert Variable gemäß einer als Parameter übergebenen Funktion abändert.

Hinweis:
Verwenden Sie den gegebenen Code als Vorlage. Sie dürfen neue Deklarationen hinzufügen, aber Sie dürfen keine bestehenden Definitionen abändern. Auch die Imports dürfen Sie nicht verändern!

Beispiele:

```
demo3 = do { putVar VarX 1; modifyVar VarX (+2); modifyVar VarX (*2) }
demo4 = forM [VarX,VarY,VarX,VarZ,VarY,VarX] (flip modifyVar succ)

> runState newState demo3
((),(6,0,0))

> snd $ runState newState demo4
(3,2,1)
```

b) Implementieren Sie `modifyVar` noch einmal zu Fuß, d.h. ohne die Verwendung der DO-Notation und ohne Verwendung von `(>>=)`, `(>>)`, etc. Die Funktionen `leseVar` und `schreibeVar` dürfen Sie verwenden.

Hinweis: Sie dürfen auch eine Lösung für beide Teilaufgaben abgegeben.

Abgabe: Lösungen zu den Hausaufgaben können bis Dienstag, den 1.07.2014, 11:00 Uhr mit UniworX abgegeben werden.

Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen.