Prof. Martin Hofmann, PhD
Dr. Steffen Jost

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

9. Musterlösung 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.

LÖSUNGSVORSCHLAG:

```
main = do
  putStrLn "Hi! Gib bitte zuerst Deine Lieblingsfarbe und dann"
  putStrLn "in die nächste Zeile Dein Lieblingstier ein: "
  farbe <- getLine
  tier  <- getLine
  let ausgabe = "Psst, willst Du " ++ tier ++ " in " ++
                farbe ++ " kaufen?"
  putStrLn ausgabe
```

Es ist dabei unerheblich, ob der Eingangstext in einer Zeile oder in zwei verschiedenen ausgegeben wird. Es ist auch nicht von Bedeutung, ob die Ausgabe erst mit einem let-Ausdruck unter einem lokalen Bezeichner abgelegt wird oder direkt als Argument an `putStrLn` übergeben wird.

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
```

LÖSUNGSVORSCHLAG:

Der Benutzer wird drei Mal aufgefordert, etwas einzugeben. Wir geben der Reihe nach die Zahlen 1,2 und 3 ein:

```
> main1
A: 1
B: 2
D: 3
A=1 B=3 C=2
```

Die ersten beiden Zeilen sollten inzwischen klar sein: Es wird ein String ausgegeben und eine Zeile eingelesen, d.h. bei Eingabe 1 gilt danach `a2="1"`

Die dritte Zeile gibt erneut einen String aus, jedoch wird das Ergebnis des Funktionsaufrufs `putStr "B: "` an die Variable `b1` gebunden. Die Funktion `putStr` liefert jedoch immer nur `()` zurück.

In der vierten und fünften Zeile werden Ausdrücke an lokale Bezeichner `b2` und `c1` gebunden. Das `let` ist eine rein funktionale Abkürzung, d.h. es ist außerhalb der IO-Monade. Weder `getLine` noch `putStr` werden dadurch ausgeführt!

Zeile 6 führt die zweite Eingabeaufforderung aus, und bindet das Ergebnis an den lokalen Bezeichner `c2`. Bei Eingabe 2 gilt danach also `c2="2"`.

Zeile 7 führt nun den an `b2` gebundenen Ausdruck innerhalb der Monade aus, d.h. es kommt zu einer Eingabeaufforderung. Das Ergebnis der Eingabe wird an den lokalen Bezeichner `b2` gebunden. Die vorangegangene Definition von `b2` wird dadurch überschattet. Der Rückpfeil der DO-Notation ist im Gegensatz zu `let` nicht rekursiv!

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`.

LÖSUNGSVORSCHLAG:

Eine Lösung zur Teilaufgabe a) erhält man als Teil der Teilaufgabe b), z.B. für Variante 2):

```
main = readAction
```

Wir stellen drei verschiedene Lösungsmöglichkeiten vor:

1) Die hässlichste Lösung, welche wir nur zeigen, weil vermutlich viele Studenten mit Kenntnissen in imperativen Sprachen diesen Weg gewählt haben werden. Der Code ist ein großer Block, ohne klare Trennung, ohne Wiederverwendungsmöglichkeiten.

2) Modulare Lösung. Dies ist vermutlich die empfehlenswerte lesbare Lösung, auch wenn der Unterschied zu Variante 1) gar nicht so groß ist! Dank dem Einsatz der Funktion rreturnïst diese Variante sehr modular. Es gibt getrennte Funktion mit klaren Aufgaben.

3) Rein Funktionale Lösung: Ein einziger Aufruf von `interact` reicht aus; damit ist sofort klar, das dieser Code nicht die Festplatte löscht (vgl. Lösung 2: jede IO-Funktion, z.B. auch `calcAux`, könnte die Festplatte löschen!); wir brauchen keine DO-Notation; wenig Code. Allerdings könnte man darüber streiten, ob der Code nicht so gut lesbar ist wie in Variante 2).

Variante 1)

```
import Data.Char

main :: IO ()
main = do -- stage1
  putStrLn "Wie heissen Sie?"
  name <- getLine
  if not (startsWithUpper name)
    then do
      putStrLn "Ihr Name _muss_ mit einem Großbuchstaben beginnen!"
      main
    else do
      putStrLn "Zahlen werden addiert, Ihr Name beendet, alles andere zeigt Summe"
      stage2 name 0

stage2 :: String -> Int -> IO ()
stage2 name acc = do
    input <- getLine
    case readMaybe input of
      (Just v) -> stage2 name (acc + v)
      Nothing  -> do
        putStrLn ("Summe: " ++ (show acc))
        if input == name
            then putStrLn "Auf Wiedersehen!"
            else stage2 name acc


-------------------------------------
-- Rein funktionale Hilfsfunktionen:

startsWithUpper :: String -> Bool
startsWithUpper (h:_) = isUpper h
startsWithUpper  _    = False
-- Alternative tolerante Version
readMaybe :: Read a => String -> Maybe a
readMaybe s | (x,_):_ <- filter (null.snd) (reads s) = Just x
            | otherwise = Nothing
```

Variante 2)

```
import Data.Char

main :: IO ()
main = do
  name <- readAction
  calcAction name
  putStrLn "Auf Wiedersehen!"

readAction :: IO String
readAction = do
  putStrLn "Wie heissen Sie?"
  name <- getLine
  if startsWithUpper name
    then return name
    else do
      putStrLn "Ihr Name _muss_ mit einem Großbuchstaben beginnen!"
      readAction

calcAction :: String -> IO Int
calcAction name = do
    putStrLn "Zahlen werden addiert, Ihr Name beendet, alles andere zeigt Summe."
    calcAux 0 -- Hilfsfunktion mit Akkumulator
  where
    calcAux :: Int -> IO Int
    calcAux acc = do
        input <- getLine
        case readMaybe input of
          (Just v) -> calcAux $ acc + v
          Nothing  -> do
            putStrLn $ "Summe: " ++ (show acc)
            if input == name then return acc
                             else calcAux acc

--------------------------------------
-- Rein funktionale Hilfsfunktionen:

startsWithUpper :: String -> Bool
startsWithUpper (h:_) = isUpper h
startsWithUpper  _    = False

-- Akzeptiert nur, wenn Einlesen eindeutig ist.
readMaybe :: Read a => String -> Maybe a
readMaybe s | [(x,"")] <- reads s = Just x
            | otherwise           = Nothing
```

Variante 3)

```
import Data.Char
import Data.Maybe

main :: IO ()
main = interact $ unlines . lineActions . lines
  where
    lineActions :: IOLineProcess
    lineActions = readName
                $ calculator
                $ const ["Auf Wiedersehen!"]

---------------------------
-- Rein Funktionaler Code!
-- Für Modularität übergeben wir jeweils eine Funktion,
-- welche die Interaktion fortführt, mit dem Typ:
type IOLineProcess = [String] -> [String]

readName :: (String -> IOLineProcess) -> IOLineProcess
readName continue l = "Bitte Namen eingeben: " : checkName l
  where
    checkName :: [String] -> [String]
    checkName (name:input)
      | startsWithUpper name = continue name input
      | otherwise = "Ihr Name _muss_ mit einem Großbuchstaben beginnen!"
                  : "Bitte ordentlichen Namen eingeben: "
                  : checkName input

calculator :: IOLineProcess -> String -> IOLineProcess
calculator continue name l = "Zahlen werden addiert, Ihr Name beendet, alles andere zeigt Summe:"
                           : calculatorAux 0 l
  where
    calculatorAux acc (h:t)
      | (Just v) <- readMaybe h = calculatorAux (acc+v) t
      | h == name = "Endsumme: " : show acc
                  : continue t
      | otherwise = "Summe: "    : show acc
                  : calculatorAux acc t

-- type IOLineProcess a = [String] -> (a,[String])

{- Wäre auch ein interessante Alternative! Vergleiche dies mit der Definition für IO a von
Folie 11-15! Hier könnten wir also wieder eine Monade einsetzen, aber eine spezialisierte!
Während uns die IO Monade erlaubt, z.B. die Festplatte zu löschen, würde uns diese
spezialisierte Monade nur noch String-Ein/Ausgabe erlauben.
Je genauer die Schnittstelle festgelegt ist, desto besser! -}
-------------------------------------
-- Rein funktionale Hilfsfunktionen:
startsWithUpper :: String -> Bool
startsWithUpper = (any isUpper) . (take 1)
-- Alternative tolerante Version, benötigt aber "import Data.Maybe" am Anfang:
readMaybe :: Read a => String -> Maybe a
readMaybe s = listToMaybe [ p |(p,"") <- reads s]
```

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.

LÖSUNGSVORSCHLAG:
Der Datentyp `Option` entspricht ganz genau dem Datentyp `Maybe`, lediglich mit umbenannten Konstruktoren.

```
instance Monad Option where
  return = Some
  None   >>= k = None
  Some a >>= k = k a

instance MonadPlus Option where
  mzero = None
  mplus None     m = m
  mplus (Some a) m = Some a
```

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!

LÖSUNGSVORSCHLAG:

```
chainAction1 :: Monad m => a -> [(a -> m a)] -> m a
chainAction1 a1   []  = return a1
chainAction1 a1 (h:t) = do a2 <- h a1
                           chainAction a2 t
```

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`.

LÖSUNGSVORSCHLAG:

```
chainAction2 :: Monad m => a -> [(a -> m a)] -> m a
chainAction2 a1   []  = return a1
chainAction2 a1 (h:t) = h a1 >>= \a2 -> chainAction a2 t
```

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!

LÖSUNGSVORSCHLAG:

```
chainAction2 :: Monad m => a -> [(a -> m a)] -> m a
chainAction2 = foldM (\acc f -> f acc)
```

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.

LÖSUNGSVORSCHLAG:

Wir müssen lediglich `getVar` und `putVar` zu einer monadischen Aktion verkleben:

```
modifyVar1 :: Name -> (Wert -> Wert) -> Env ()
modifyVar1 name f = do
  wert <- getVar name
  putVar name (f wert)
```

Äquivalent dazu ist natürlich auch

```
modifyVar2 :: Name -> (Wert -> Wert) -> Env ()
modifyVar2 name f = getVar name >>= \wert -> putVar name $ f wert
```

Um die volle Punktzahl zu erhalten, müssen wir die Aufgabe zur Übung ohne die Verwendung des monadischen bind lösen. Dies hat keinen tieferen Sinn, außer der Vertiefung des Verständnis.

```
modifyVar3 :: Name -> (Wert -> Wert) -> Env ()
modifyVar3 name f = Env $ \state0 ->
  let wert0  = leseVar name state0
      wert1  = f wert0
      state1 = schreibeVar name wert1 state0
  in ((),state1)
```

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.