Monaden

I/O ohne Nebeneffekte?

Erinnerung

Das Ergebnis einer Funktion darf nur von den Parametern abhängen.

Was heißt das für die folgenden Funktionen?

```
readLine ::                    String
putLine :: String ->                  ()
```

I/O ohne Nebeneffekte?

Erinnerung

Das Ergebnis einer Funktion darf nur von den Parametern abhängen.

Die Funktionen müssen die gesamte Welt als Parameter haben, da ihr Ergebnis davon abhängt.

```
readLine :: RealWorld ->                String
putLine :: String -> RealWorld ->                ()
```

I/O ohne Nebeneffekte?

Erinnerung

Das Ergebnis einer Funktion darf nur von den Parametern abhängen.

Und da die Welt verändert werden soll (ein Wert gelesen bzw. geschrieben), muss die „neue Welt“ zurückgegeben werden.

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           let (rw2, s2) = readLine rw1 in
           let output = s1 ++ s2 in
           putLine output rw2
```

Beobachtung: In den ersten beiden Zeilen wird die Welt aufwendig durchgeschleift. Für diese Aufgabe suchen wir einen geeigneten Kombinator.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           let (rw2, s2) = readLine rw1 in
           let output = s1 ++ s2 in
           putLine output rw2

bind f g rw = let (rw', s) = f rw in g s rw'
```

Neuer Kombinator: `bind`.

Versuche, beide Zeilen unter Verwendung von `bind` auszudrücken.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           let (rw2, s2) = readLine rw1 in
           let output = s1 ++ s2 in
           putLine output rw2

bind f g rw = let (rw', s) = f rw in g s rw'
```

Beobachtung: Muster von `bind` passt *fast*, aber nicht komplett.

Problem: Ausdruck `let output = s1 ++ s2 in putLine output` hat nicht die Form `g s`, die `bind` erwartet.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           let (rw2, s2) = readLine rw1 in (\s2 ->
           let output = s1 ++ s2 in
           putLine output) s2 rw2

bind f g rw = let (rw', s) = f rw in g s rw'
```

Lösung: Führe unbenannte Funktion ein und übergebe ihr direkt den benötigten Wert.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           let (rw2, s2) = readLine rw1 in (\s2 ->
           let output = s1 ++ s2 in
           putLine output) s2 rw2

bind f g rw = let (rw', s) = f rw in g s rw'
```

Jetzt passt das Muster genau.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           bind readLine (\s2 ->
           let output = s1 ++ s2 in
           putLine output) rw1

bind f g rw = let (rw', s) = f rw in g s rw'
```

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in
           bind readLine (\s2 ->
           let output = s1 ++ s2 in
           putLine output) rw1

bind f g rw = let (rw', s) = f rw in g s rw'
```

Führe Ersetzung ein zweites Mal durch.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in (\s1 ->
           bind readLine (\s2 ->
           let output = s1 ++ s2 in
           putLine output)) s1 rw1

bind f g rw = let (rw', s) = f rw in g s rw'
```

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = let (rw1, s1) = readLine rw0 in (\s1 ->
           bind readLine (\s2 ->
           let output = s1 ++ s2 in
           putLine output)) s1 rw1

bind f g rw = let (rw', s) = f rw in g s rw'
```

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main rw0 = bind readLine (\s1 ->
           bind readLine (\s2 ->
           let output = s1 ++ s2 in
           putLine output)) rw0

bind f g rw = let (rw', s) = f rw in g s rw'
```

Beachte: Die Funktion `bind` übernimmt die Verwaltung des RealWorld-Wertes vollständig! Mittels Unterversorgung kann jetzt sogar der Parameter `rw0` an `main` weggelassen werden.

Ein erstes IO-Programm

Aufgabe

Lies zwei Zeilen ein und gib deren Verknüpfung aus. Verwende dabei:

```
readLine :: RealWorld -> (RealWorld, String)
putLine :: String -> RealWorld -> (RealWorld, ())
```

```
main      = bind readLine (\s1 ->
            bind readLine (\s2 ->
            let output = s1 ++ s2 in
            putLine output))

bind f g rw = let (rw', s) = f rw in g s rw'
```

Beachte: Die Funktion `bind` übernimmt die Verwaltung des RealWorld-Wertes vollständig! Mittels Unterversorgung kann jetzt sogar der Parameter `rw0` an `main` weggelassen werden.

Jetzt wird `RealWorld` gar nicht mehr erwähnt.

Syntaktischer Zucker

Do-Notation und ihre Bedeutung

```
do x <- f                    do let x = y          do f
   g                            g
   ⇓                            ⇓                     ⇓
bind f (\x -> do g)          let x = y in do g     f
```

```
main = bind readLine (\s1 ->
       bind readLine (\s2 ->
       let output = s1 ++ s2
       in putLine output))
```

```
main = do s1 <- readLine
          s2 <- readLine
          let output = s1 ++ s2
          putLine output
```

- Die Abstraktion mit bind heißt *Monade*. Sie wird auch für Zustand, Nichtdeterminismus, Fehlerbehandlung, Sprünge usw. verwendet.
- Statt `RealWorld -> (RealWorld, a)` wird tatsächlich `IO a` verwendet. Der `RealWorld`-Typ ist für den Programmierer nicht sichtbar.
- So lassen sich in Haskell auch GUIs, Webanwendungen, Datenbankanwendungen, Spiele etc. entwickeln.