

Arbeitsblatt: Lambda Ausdrücke

In Haskell können Funktionen auch als Ausdrücke anonym verwendet werden. Dies ist praktisch, wenn Sie eine Funktion nur einmal verwenden.

`\p -> e` ist eine anonyme Funktion, die einen Parameter namens `p` erwartet und als Resultat den Wert der Expression `e` zurück gibt. An der Stelle von `p` kann auch gleich ein Pattern stehen um das Argument zu zerlegen und den Bestandteilen einen Namen zu geben.

Beispiele:

```
\x -> x * 2
```

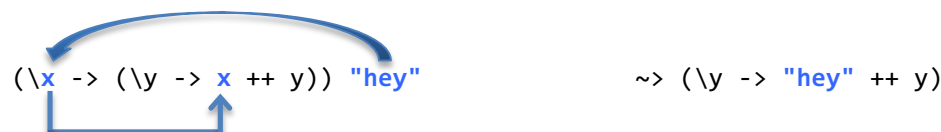
```
\((a,b),x:xs) -> a:x:b:xs
```

Die Expression `e` kann wiederum eine Lambda Expression sein. So lassen sich anonyme Funktionen mit "mehreren" Parametern realisieren. Der Funktionspfeil assoziiert nach rechts!

```
\x -> \y -> x ++ y           = (\x -> (\y -> x ++ y))
```

Die äussere Lambda Expression (mit dem `x` Parameter) gibt als Resultat die innere Lambda Expression (mit dem Parameter `y`) zurück.

Wenn Sie diese Lambda Expression auf "hey" anwenden, wird das "hey" an den Parameter `x` gebunden und zurück kommt eine Lambda Expression, die noch den `y` Parameter erwartet.

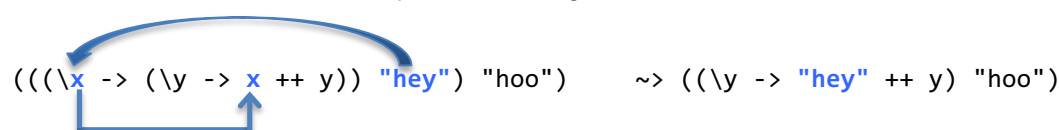


```
(\x -> (\y -> x ++ y)) "hey" ~> (\y -> "hey" ++ y)
```


Die Anwendung auf zwei Argumente ist nur Schein - tatsächlich werden die zwei Argumente nacheinander übergeben. Function Application assoziiert nach rechts!

```
(\x -> (\y -> x ++ y)) "hey" "hoo" = (((\x -> (\y -> x ++ y)) "hey") "hoo")
```

Wie "hey" auf die äussere Lambda Expression angewendet wird, haben wir oben schon gesehen. Trotzdem, weil es so schön ist, spielen wir das ganze durch:



```
(((\x -> (\y -> x ++ y)) "hey") "hoo") ~> ((\y -> "hey" ++ y) "hoo")
```



```
((\y -> "hey" ++ y) "hoo") ~> "hey" ++ "hoo" ~> "heyhoo"
```

Wie Sie wissen, gelten folgende Transformationsregeln:

```
f a = e      >>>   f = \a -> e
f a b = e    >>>   f = \a b -> e      >>>   f = \a -> \b -> e
```

Mit diesen Regeln und der obigen Erklärung wie solche Funktionen angewendet werden, haben Sie Haskell Funktionen total unter Kontrolle!

1. Aufgabe

Implementieren Sie folgende Funktionen mit Lambda Expressions.

a)

-- Erhöht den Wert von jedem Listenelement um eins:

`incAll :: [Int] -> [Int]`

b)

-- Addiert zu jedem Listenelement den ersten Parameter:

`addToAll :: Int -> [Int] -> [Int]`

c)

-- Entfernt alle Elemente deren Wert kleiner als 90 ist:

`keepOld :: [Int] -> [Int]`

d)

-- Entfernt alle Strings, die eine Länge von Eins haben.

`dropShort :: [String] -> [String]`

2. Aufgabe

Bestimmen Sie die Typen der folgenden Ausdrücke:

`(\x -> x > 9) 6` ::

`(\x -> tail x)` ::

`(\ (a,b) -> b ++ a)` ::

`(\t -> fst)` ::

`(\ (x:xs) -> x)` ::

`(\x y -> head y) 2` ::

`\ (a,b) -> fst a ++ b` ::