

Funktionale und objektorientierte Programmierkonzepte



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Präsenz-Sprechstunde B

Simon Hock, Nhan Huynh, Daniel Mangold



Organisatorisches

Lambda-Ausdrücke

Funktionen höherer Ordnung

filter
map
fold

Arbeitsphase

- 15.12.2021 Präsenzsprechstunde im Raum in Raum S103/**223**!



- Anonyme Funktion
- Keinen Namen/ Bezeichner



- Schlüsselwort `lambda` oder λ
- Argumente optional, werden mit Leerzeichen getrennt
 - ▣ Klammer müssen trotzdem notiert werden!
- Argumente an Lambda-Ausdruck übergeben: Lambda-Ausdruck wird mit runden Klammern umschlossen, in welchen zuerst der Lambda-Ausdruck steht, gefolgt von den Argumenten

- 1 (`lambda` (Argumente*) Operation)
- 2 (λ (Argumente*) Operation)

Beispiel: Normale Funktion

```
1 ;; Type: number number -> number
2 ;; Returns: the sum of x and y
3 (define (add x y)
4   (+ x y))
5
6 (add 5 6) ; Returns 11
```

Beispiel: Lambda-Ausdruck

```
1 ;; Type: number number -> number
2 ;; Returns: the sum of x and y
3 (lambda (x y) (+ x y))
4
5 ;; Type: number number -> number
6 ;; Returns: the sum of x and y
7 ((lambda (x y) (+ x y)) 5 6) ; Returns 11
8 ; Different lambda expression since they are anonymous functions!
9 ; Alternative: Storing the lambda expression as constant, not a function!
10
11 ;; Type: number number -> number
12 ;; Returns: the sum of x and y
13 (define fct (lambda (x y) (+ x y)))
14 (fct 5 6) ; Returns 11
```

Wir benötigen ein funktionales Interface, um Lambda-Ausdrücke erzeugen zu können

Information: Ein funktionales Interface ist ein Interface mit genau eine nicht implementierte Methode.

```
1 @FunctionalInterface
2 public interface FunctionalInterface {
3
4     void apply();
5 }
```



```
1 Parameter -> Operation;  
2 (Parameter, ...) -> {  
3   Operationen;  
4 };
```

- Falls es nur einen Parameter existiert, dann können die Klammern weggelassen werden.
- Falls es nur eine Anweisung existiert, dann können die geschweiften Klammern und Semikolon weggelassen werden.
- Parameter werden mit einem Komma getrennt.
- Anweisungen werden mit einem Semikolon getrennt und falls die die Methode etwas zurückgeben soll, dann ist die letzte Anweisung ein **return**-Anweisung.

- Funktion erhält zwei `int` Zahlen und gibt eine `int` Zahl zurück.

Interface IntFunction

```
1 @FunctionalInterface
2 public interface IntFunction {
3
4     int apply(int a, int b);
5 }
```

IntFunction Lambda

```
1 IntFunction fct;  
2  
3 fct = (a, b) -> a + b;  
4 fct = (a, b) -> return a + b;  
5 fct = (a, b) -> {  
6     return a + b;  
7 }:  
8  
9 fct.apply(5, 6); // Returns 11  
10  
11 fct = (a, b) -> a * 3 - b;  
12  
13 fct.apply(5, 6); // Returns 9
```

- Namenlose Klassen
- „on the fly“ Objektbildung

IntFunction Anonyme Klasse

```
1 IntFunction fct = new IntFunction {  
2  
3     @Override  
4     public int apply(int a, int b){  
5         return a + b;  
6     }  
7 }
```

- Lambda-Ausdrücke sind nichts anderes als anonyme Klassen
- Compiler übersetzt Lambda in eine anonyme Klasse

```
1 IntFunction fct = (a, b) -> a + b;  
2  
3 public class <<AnonymeClass>> implements IntFunction {  
4     @Override  
5     public int apply(int a, int b){  
6         return a + b;  
7     }  
8 }
```



Information: Funktionen höherer Ordnung sind Funktionen, die als Parameter eine Funktion erhalten und oder eine Funktion als Rückgabewert haben. Ihre Vorteile liegen darin, dass sie ihre eigene Funktion anhand des Parameters anpassen (größere Abstraktion) oder Funktionen anhand bestimmter Parameter erstellen können. Durch die Abstraktion ergibt sich eine breitere Verwendungsmöglichkeit, d.h. eine Anpassbarkeit an eine Vielzahl gleichartiger Aufgaben

- Die Funktion ist eine einstellige Funktion $f : X \rightarrow \mathbb{B}$
- Prädikat: Boolesche Funktion
- Die Funktion wird auf jedes Element in der Menge angewendet
- Falls das Prädikat zutrifft, so wird das Element in die Ergebnismenge aufgenommen
- Ansonsten wird das Element nicht in die Ergebnismenge aufgenommen
- Formaler Aufbau:

`(filter fct elements)`

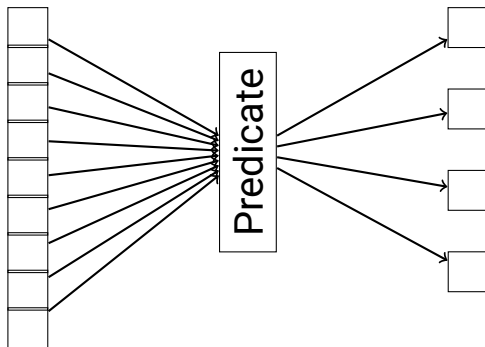


Abbildung: Abstrakte Visualisierung von `filter`

- Die Funktion bildet eine Menge anhand der Funktion ab
- Konkret wird die Funktion auf jedes Element der Menge angewendet
- Das Ergebnis ist eine Menge, die alle Ergebnisse der Funktion enthält
- Formaler Aufbau:

```
(map fct elements)
```

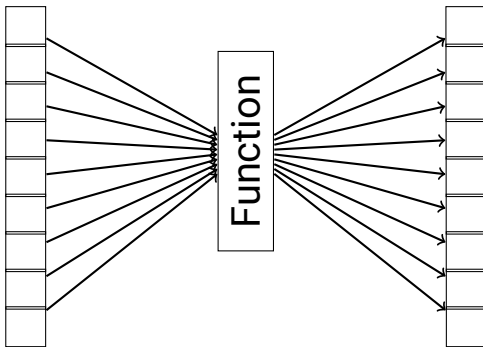


Abbildung: Abstrakte Visualisierung von `map`



Eine Funktion f ordnet jedem Element x einer Definitionsmenge D genau ein Element y einer Zielmenge Z zu.

$$f : \begin{cases} D \rightarrow Z \\ x \mapsto y \end{cases} \quad (1)$$

$$D \longrightarrow Z$$

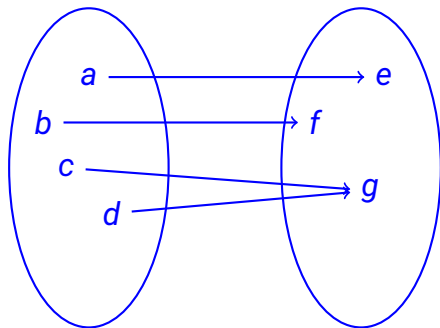


Abbildung: Visualisierung Mathematik Funktion

- Faltungsfunktion berechnet aus einer Menge einen einzelnen Wert
 - ▣ Typ des Wertes kann gleich dem Typ der Elemente in der Menge sein, muss es nicht
- Verknüpft das aktuelle Element mit dem momentanten Zwischenergebnis (Initialwert)
- Verwendet dieses Ergebnis als neuen Zwischenergebnis und wiederholt diesen Vorgang für jedes Element
- „Die Faltungsfunktion reduziert sozusagen eine Menge auf einen Wert“
- `foldl`: Ausführung von links nach rechts.
- `foldr`: Ausführung von rechts nach links.
- Formaler Aufbau:

```
(foldl fct initial elements)  
(foldr fct initial elements)
```

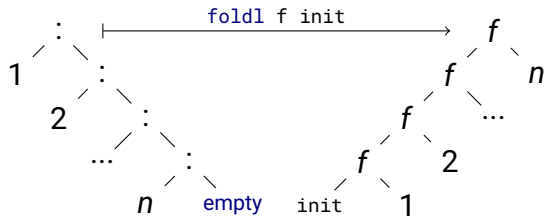


Abbildung: Visualisierung von foldl

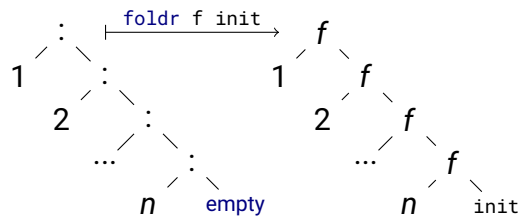


Abbildung: Visualisierung von foldr

Selbstständiges Arbeiten