

Funktionale Schleifen

Sven Eric Panitz

12. März 2019

Inhaltsverzeichnis

1	Schleifen in imperativen Sprachen	2
2	Eine Schleifenfunktion in Haskell	3
2.1	Eine Funktion für Schleifen	3
2.2	Beispielaufufe	4
2.3	Wie sieht es mit dem Stack aus?	5
2.4	Überlegungen zur Striktheit	6
3	Eine for-Methode in Java	7
4	Rekursionsschleife	9
4.1	Typische rekursive Funktionen in Haskell	9
4.2	Verallgemeinerung als Schleifenfunktion	11
4.3	Beispielaufufe	12
4.3.1	Fakultät	12
4.3.2	Länge einer Liste	12
4.3.3	Größtes Listenelement	12
4.3.4	Filter über eine Liste	13
4.3.5	Letztes Listenelement	13
4.3.6	Umkehrung einer Liste	13
4.3.7	Generierung einer unendlichen Liste	13
4.3.8	Generierung einer Liste mit Schranken	13
5	Schleifenfunktion in Standard-APIs	14
5.1	Faltungen in Haskell	14
5.2	Faltungen für Listen in Scala	15
5.3	Faltungen für Listen in Kotlin	16
5.4	Die reduce-Methode für Java Stream-Objekte	16
5.5	Die reduce-Funktion auf Python Listen	17
5.6	Die reduce-Funktion in Ruby	18

5.7 Die reduce-Funktion in Javascript	18
6 Lernzuwachs	18
7 Aufgaben	19

1 Schleifen in imperativen Sprachen

Eine der wichtigsten Anweisungen in imperativen Sprachen sind Schleifen. Sie dienen der Iteration, d.h. der wiederholten Ausführung eines Stück Codes. Rein funktionale Sprachen kennen überhaupt keine Anweisungen und damit auch keine Schleifen.

Die strukturierteste Schleifenanweisung ist in imperativen Sprachen der C-Familie die for-Schleife.

Betrachten wir einmal zwei typische for-Schleifen in Java, um dann im nächsten Abschnitt zu überlegen, wie solche Schleifen allgemein in Haskell umgesetzt werden können.

Folgende Java Methode berechnet die Fakultät einer Zahl und verwendet hierzu eine for-Schleife.

Java: Fakultät

```
class Fac{
    static long fac(int n){
        var result = 1L;
        for (int i = n;i>=0;i = i-1){
            result = result*i;
        }
        return result;
    }
}
```

Als zweites Beispiel für eine for-Schleife, eine Schleife, die über die Elemente einer Liste ganzer Zahlen iteriert und diese Zahlen in einer Ergebnisvariable aufaddiert.

Java: Summe

```
import java.util.*;
class L{
    static long sum(List<Integer> xs){
        var result = 0L;
        for (List<Integer> is=xs; !is.isEmpty(); is=is.subList(1,is.size())){
            result = result+is.get(0);
        }
        return result;
    }
}
```

Was haben solche Methoden gemeinsam? Sie haben alle den gleichen Aufbau. Sie bestehen alle aus 5 Teilen:

- einer Ergebnisvariablen `result` mit einem initialen Wert:
(`var result=1L`) bzw. (`var result = 0L`).
- einer Schleifenvariablen auch mit einem initialen Wert:
(`int i = n`) bzw. (`(List<Integer> is=xs)`).
- einer Prädikatsfunktion, die für einen Wert der Schleifenvariablen testet, ob die Schleife ein weiteres Mal zu durchlaufen ist:
(`i>=0`) bzw. (`!is.isEmpty()`).
- einer Schrittfunktion, die den nächsten Wert der Schleifenvariable berechnet:
(`i = i - 1`) bzw. (`is=is.subList(1,is.size())`).
- einer Ergebnisfunktion, die angibt, was die aktuelle Schleifenvariable zum Ergebnis beiträgt:
(`result=result*i`) bzw. (`result = result+is.get(0)`).

2 Eine Schleifenfunktion in Haskell

In diesem Abschnitt soll gezeigt werden, wie man mit Funktionen höherer Ordnung, über eine ganze Methodenstruktur abstrahieren kann.¹

Haskell: Loop

```
> module Loop where
> import Data.Char
```

2.1 Eine Funktion für Schleifen

Es soll in Haskell eine Funktion entwickelt werden, die das Muster einer `for`-Schleife abstrahiert. Eine Funktion, die eine `for`-Schleife zur Berechnung eines Ergebnisses aus einem Iterationsbereich verwendet, braucht entsprechend die 5 im letzten Abschnitt identifizierten Argumente.

Die Funktion unterscheidet zwei Fälle. Es wird in Abhängigkeit von der Schleifenvariablen getestet, ob die Schleife ein weiteres Mal zu durchlaufen ist. Ansonsten wird die Ergebnisvariable zurück gegeben.

Ein weiterer Schleifendurchlauf wird durch die Rekursion realisiert. Im rekursiven Aufruf wird mit den entsprechenden Funktionen die Schleifenvariable weitergeschaltet und das neue Ergebnis für den nächsten Durchlauf errechnet.

¹Das Modul `Data.Char` wird für eine der Aufgaben benötigt und ist daher hier importiert.

Haskell: for

```
> for result test step calcResult v
> |test v = for (calcResult result v) test step calcResult (step v)
> |otherwise = result
```

2.2 Beispielaufufe

Die Funktion `for` kann nun verwendet werden, um Funktionen, die im Prinzip Schleifen verwenden, zu realisieren. Beginnen wir mit den zwei Funktionen aus dem ersten Abschnitt. Die Fakultät ist eine `for`-Schleife, die so lange läuft, wie die Schleifenvariable noch positiv ist. Die Schleifenvariable nach jedem Durchlauf um 1 verringert. In jedem Durchlauf wird schließlich die Schleifenvariable zum Ergebnis multipliziert.

Wir erhalten einen Aufruf der Funktion `for` mit den folgenden vier Argumenten:

Haskell: fac1

```
> fac1 = for 1 (\x-> x>0) (\x->x-1) (*)
```

Dem aufmerksamen Leser wird aufgefallen sein, dass hier der eigentliche Parameter der Schleifenvariable fehlt. Dieses liegt an dem Typsystem mit dem sogenannten Currying. Man hätte auch schreiben können:

Haskell: fac1

```
fac1 n = for 1 (\x-> x>0) (\x->x-1) (*) n
```

Oder auch, um deutlich zu machen, dass `fac1` eine Funktion mit noch einem Argument ist:

Haskell: fac1

```
fac1 = \n -> for 1 (\x-> x>0) (\x->x-1) (*) n
```

Die zweite Methode aus dem ersten Kapitel hatte unterschiedliche Typen für die Schleifenvariable und das Ergebnis. Die Schleifenvariable war eine Liste und das Ergebnis eine Zahl. Trotzdem ist die Funktion `for` in der Lage, dieses auszudrücken. Lässt man sich den inferierten Typ der Funktion anzeigen, so sieht man, dass die beiden Typen unterschiedliche Typen sind.

`haskell title="for Typisierung»`

```
for :: r -> (a -> Bool) -> (a -> a) -> (r -> a -> r) -> a -> r
```

`;/haskell;`

Um die Summe der Zahlen einer Liste zu berechnen, ist zu berücksichtigen, dass man beim Errechnen des Ergebnisses von der Schleifenvariablen, die ja die komplette noch zu bearbeitende Liste ist, das erste Element zu nehmen und zum Ergebnis zu addieren ist.

Haskell: sum1

```
> sum1 = for 0 (not.null) tail (\r xs -> r+head xs)
```

Noch einfacher ist auf diese Weise auch die Funktion, die die Länge einer Liste berechnet, zu realisieren.

Haskell: len1

```
> len1 = for 0 (not.null) tail (\r _ -> r+1)
```

Es lassen sich auch Funktionen schreiben, die eine neue Liste erzeugen. Die folgende Funktion erzeugt eine neue Liste, deren Elemente die Elemente der Eingabeliste in umgekehrter Reihenfolge darstellen.

Haskell: rev1

```
> rev1 = for [] (not.null) tail (\r xs -> head xs:r)
```

Wie man sieht, ist uns mit der Funktion `for` eine mächtige Abstraktion gelungen, mit deren Hilfe sich jeweils in einer Zeile ganz unterschiedliche Funktionen definieren lassen.

2.3 Wie sieht es mit dem Stack aus?

Aus imperativen Sprachen ist man gewöhnt, dass Rekursionen potentiell ineffizient sind oder Gefahr laufen, dass der Stack überläuft. Insbesondere gibt es keine unendlich laufenden Rekursionen, da bei jedem rekursiven Aufruf auf dem Stack für die Parameter neuer Speicherplatz eingeräumt wird und auch die Rücksprungadresse abzulegen ist, auf die Position, von der die Funktion aufgerufen wurde.

Es gibt aber spezielle rekursive Funktionen, bei denen für den rekursiven Aufruf für diese Information kein neuer Speicher bereit gestellt werden muss, sondern der Speicher verwendet werden kann, der bereits für den ersten Aufruf der Funktion reserviert wurde. Das ist immer dann der Fall, wenn nach der Rückgabe des rekursiven Aufrufs keine lokale Information des ursprünglichen Aufrufs mehr benötigt wird. Das erkennt man insbesondere daran, wenn das Ergebnis des rekursiven Aufrufs direkt als Endergebnis für den ursprüngliche Funktionsaufruf verwendet wird. In imperativen Sprachen ist das daran zu erkennen, dass der rekursive Aufruf direkt nach dem Schlüsselwort der `return`-Anweisung erfolgt.

Eine solche rekursive Funktion wird *tail recursive* bezeichnet. Ein Compiler kann eine solche Rekursion wegoptimieren, indem statt eines Funktionsaufrufs mit einem neuen

reservierten Speicherbereichs auf dem Stack ein Sprung zum Anfang der Funktion generiert wird. Zuvor sind die Werte der Argumente für den rekursiven Aufruf in den Speicherbereich der Argumente abzuspeichern.

Diese Optimierung ist relativ einfach durchzuführen. Sie wird als *tail call optimization* bezeichnet.

Da die Programmiersprache C noch einen Sprungbefehl hat, kann die Optimierung hier im Quelltext schön explizit beschrieben werden.

Wir geben als Beispiel eine tail-rekursive Funktion für die Fakultät in C:

C: fac

```
int fac(int result, int n){
    if (n==0) return result;
    return fac(result*n,n-1);
}
```

Der rekursive Aufruf kann durch einen Sprung zum Anfang der Funktion ersetzt werden. Dazu sind den Argumenten vor der Sprunganweisung die Werte für den nächsten Durchlauf, also dem rekursiven Aufruf, zuzuweisen.

C: fac2

```
int fac2(int result, int n){
    begin:
    if (n==0) return result;
    result = result * n;
    n = n-1;
    goto begin;
}
```

Die meisten Compiler nehmen diese Optimierung vor, zumindest dann, wenn man sie mit dem Flag `-O2` aufruft. So nimmt der gcc die Optimierung für C-Programme vor, wie scalac in Scala-Programmen und natürlich auch der ghc in Haskell-Programmen. Nur der Java-Compiler scheint diese Optimierung noch immer nicht im Repertoire zu haben.

2.4 Überlegungen zur Striktheit

Ein wachsender Stack ist also nicht das Problem in Haskell, weil in diesem Fall der Compiler diese Rekursion in einen Sprung transformiert. Dafür kann ein anderes Problem eintreten. Haskell ist eine Programmiersprache mit einem Auswertungsmodell, das als *lazy* bezeichnet wird. Das werden wir an anderer Stelle noch diskutieren. Es bedeutet kurz gesagt, dass keine Berechnung durchgeführt wird, wenn nicht ganz sicher ist, dass das Ergebnis der Berechnung zum Gesamtergebnis beiträgt. Es wird erst eine komplexe Struktur im Heap angelegt, in der gespeichert ist, welche Berechnungen durchzuführen sind. Für unsere Funktion `fac1` bedeutet das, dass die Multiplikation der Zahlen zunächst

nicht ausgeführt wird. Es wird zunächst im Heap die Berechnung $n * (n - 1) * (n - 2) * \dots * 2 * 1$ gespeichert. Erst wenn das Ergebnis der Berechnung benötigt wird, wird diese Rechnung auch ausgeführt. Wir bekommen in diesem Fall zwar keinen wachsenden Stack, aber dafür wird ein wachsender Heap-Bereich benötigt.

Das Standard-Prelude bietet in Haskell eine Möglichkeit mit dem Operator `$!` die Auswertung eines Parameters einer Funktionsanwendung zu erzwingen. Man spricht davon, die Funktion *strikt* auf den Parameter anzuwenden.

Wir können eine Variante der Funktion `for` schreiben, in der bei jedem rekursiven Aufruf das Argument für das Ergebnis direkt errechnet wird.

Haskell: `for2`

```
> for2 result test step calcResult v
> |test v = (for2$(calcResult result v)) test step calcResult (step v)
> |otherwise = result
```

Verwenden wir diese Version für die Fakultät, wird die Multiplikation jeweils direkt ausgeführt und nicht erst die ganze Berechnung im Heap als Datenstruktur aufgebaut.

Haskell: `fac3`

```
> fac3 = for2 1 (\x-> x>0) (\x->x-1) (*)
```

Wann es sinnvoll ist, die normale Auswertung von Haskell zu durchbrechen und Argumente direkt auszuwerten, als strikt auszuwerten, ist eine selbst für Experten manchmal schwer zu entscheidene Frage.

3 Eine `for`-Methode in Java

Mit diesem Wissen können wir zurück einen Abstecher in die Programmiersprache Java machen. Seit Java 8 kann man Java auch als eine funktionale Sprache bezeichnen und ist es somit möglich auch Funktionen höherer Ordnung zu schreiben.

Java: `ForLoop`

```
import java.util.function.*;
import java.util.*;
class ForLoop{
```

Wir schreiben eine generische statische Funktion, die eins zu eins die Funktion `for` aus Haskell in Java realisiert. Sie ist generisch über zwei Typen. Den Typen der Schleifenvariable und den Typen des Ergebnisses. Sie hat 5 Parameter:

- das Ergebnis für den terminierenden Fall.
- eine Testfunktion, die anzeigt, ob die Schleife ein weiteres Mal zu durchlaufen ist.

- eine Funktion, die die Schleifenvariable für den nächsten Durchlauf neu setzt.
- eine Funktion, die zum bisherigen Ergebnis mit der aktuellen Schleifenvariable das nächste Ergebnis berechnet.
- der Startwert für die Schleifenvariable.

Java: forLoop Signatur

```
static <A,R> R forLoop
  ( R result
  , Predicate<? super A> p
  , Function<? super A,? extends A> step
  , BiFunction<? super R,? super A,? extends R> calc
  , A v){
```

Die Implementierung lässt sich direkt aus der Haskellimplementierung übersetzen. Es wird das Prädikat zum Testen verwendet, ob ein weiteres Mal die Schleife zu durchlaufen ist. Der rekursive Aufruf der Schleife findet mit akkumuliertem Ergebnis und einen Schritt weiter geschalteter Schleifenvariable statt. Im terminierenden Fall wird die Ergebnisvariable direkt zurück gegeben.

Java: forLoop Implementierung

```
if (p.test(v))
  return
  forLoop(calc.apply(result, v),p,step,calc,step.apply(v));
return result;
}
```

Da Java für Lambda-Ausdrücke eine kleine Typinferenz durchführt, können wir die beiden Beispielmethode vom Beginn wieder direkt umsetzen. Zunächst die Schleifenlösung für die Fakultät:

Java: fac

```
static long fac(int n){
  return forLoop(1L, x -> x>0, x -> x-1, (r,x) -> r*x, n);
}
```

Und auch die Schleife, die die Elemente einer Liste aufsummiert, lässt sich über die allgemeine Methode für Schleifen realisieren. Der Startwert, also das Ergebnis für leere Liste ist 0, der Schleifentest, prüft, dass die Schleifevariable noch nicht die leere Liste ist, die Weiterschaltung der Schleifenvariable, nimmt die Teilliste vom zweiten Element an und zum Ergebnis wird im jeden Schleifendurchlauf das erste Element der Schleifenvariable hinzuaddiert.

Java: sum

```
static long sum(List<Integer> xs){
    return forLoop
        (0
        ,x->!x.isEmpty()
        ,x->x.subList(1,x.size())
        ,(r,x)->r+x.get(0)
        ,xs);
}
```

Mit einer kleinen `main`-Methode können wir uns davon überzeugen, dass die Methoden die gewünschten Berechnungen durchführen.

Java: main

```
public static void main(String[] args){
    System.out.println(fac(5));
    var xs = List.of(1,2,3,4,5,6,7,8,9,10);
    System.out.println(sum(xs));
}
```

Wir hätten natürlich in Java auf die rekursive Lösung zugunsten einer eigentlichen Schleifenlösung verzichten können. Dieses ist insbesondere in Java empfehlenswert, weil der Java-Compiler noch immer keine tail-call-Optimierung durchführt und somit bei jedem Schleifenaufruf über Rekursion einen kompletten Stack-Frame anlegen muss.

Java: forLoop2 Implementierung

```
static <A,R> R forLoop2
    ( R result
    , Predicate<? super A> p
    , Function<? super A,? extends A> step
    , BiFunction<? super R,? super A,? extends R> calc
    , A v){
    for (;p.test(v);v=step.apply(v)) result = calc.apply(result, v);
    return result;
}
}
```

4 Rekursionsschleife

4.1 Typische rekursive Funktionen in Haskell

Das erste rekursive Programm, das man zumeist schreibt, hat wenig mit einer Schleife zu tun. Es ist in der Regel die Fakultätsfunktion, die aus der mathematischen Definition

direkt in eine rekursive Funktion umgesetzt wird.

In Haskell erhält man dann die folgende Funktionsdefinition:

Haskell: fac2

```
> fac2 n
> |n<=0 = 1
> |otherwise = n*fac (n-1)
```

Auch die Umsetzung einer Funktion, die die Summe aller Zahlen einer Liste berechnet, wird im ersten Entwurf das erste Element auf das Ergebnis des rekursiven Aufrufs für die Restliste aufaddieren:

Haskell: sum2

```
> sum2 [] = 0
> sum2 (x:xs) = x+sum xs
```

Eine Vielzahl von unterschiedlichen Funktionen können nach diesem Muster entwickelt werden. Manchmal ist das erste Ergebniselement kein fester Wert, sondern abhängig von dem Wert der Schleifenvariablen im terminierenden Fall.

Die folgende Funktion berechnet das größte Element einer Liste. Der terminierende Fall ist die einelementige Liste und bei dieser ist das alleinige Element der Liste das Ergebnis.

Haskell: max2

```
> max2 [x] = x
> max2 (x:xs) = max x (max2 xs)
```

Besonders interessant sind Funktionen die eine neue Liste erzeugt, die aus den Elementen der Argumentliste gebildet wird. Folgende Funktion filtert alle Element der Eingabeliste, für die eine Prädikatsfunktion wahr ergibt. Die Elemente der Ergebnisliste behalten die Reihenfolge der Liste bei.

Haskell: fi2

```
> fi2 _ [] = []
> fi2 p (x:xs)
> |p x = x:fi2 p xs
> |otherwise = fi2 p xs
```

Alle diese vier Funktionen gehen nach demselben Muster, über das wir in einer Funktion höherer Ordnung abstrahieren können.

4.2 Verallgemeinerung als Schleifenfunktion

Anders als im Beispiel mit der `for`-Schleife geben wir uns ein wenig mehr Mühe damit, exemplarisch die Typisierung auszuformulieren. Es ist eigentlich in Haskell nicht notwendig, weil der Compiler die Typen inferiert. Trotzdem ist es natürlich bei der Entwicklung eines APIs auch in Haskell gut, die Funktionen aus Dokumentationsgründen explizit zu typisieren.

In Haskell kann man sprechende Synonyme für Typen einführen. Was hier für die Typen der Parameter der Schleifenfunktion durchgeführt ist.

Haskell: Typesynonyme

```
> type TestFunction a = a -> Bool
> type BaseResult a r = a -> r
> type ResultFunction a r = a -> r -> r
> type StepFunction a = a -> a
```

Neu ist, dass für den Basisfall, wenn die Funktion terminiert kein konstanter Wert genommen wird, sondern eine Funktion, die das Endergebnis aus der Schleifenvariablen berechnet.

Mit diesen Typsynonymen erhält die Schleifenfunktion folgende Typ:

Haskell: Typisierung loop

```
> loop ::
>   TestFunction a
>   -> BaseResult a r
>   -> ResultFunction a r
>   -> StepFunction a
>   -> a
>   -> r
```

Die Implementierung ist die Verallgemeinerung aus den obigen vier Beispielen rekursiver Funktionen.

Haskell: Implementierung loop

```
> loop p baseResultF resultF step v
> |p v = baseResultF v
> |otherwise = resultF v (loop p baseResultF resultF step (step v))
```

Die Unterschiede zu der Funktion `for` sind:

- es lassen sich auch Funktionen realisieren, die keinen festen Wert für den terminierenden Fall haben, wie zum Beispiel die Funktion, die das Maximum einer Liste berechnet.

- die Listenelemente werden rechtsassoziativ verknüpft.
- die Funktion `loop` ist nicht *tail recursive* und die Rekursion kann nicht weg optimiert werden.

4.3 Beispielaufufe

Um zu zeigen, wie vielseitig die Funktion `loop` ist geben wir acht Beispielaufufe.

4.3.1 Fakultät

Hier die klassische rekursive Fakultät die tatsächlich $n*((n-1)*((n-2)*(\dots*(3*(2*1)\dots)))$ rechnet.

Haskell: `fac`

```
> fac = loop ((>=) 0) (const 1) (*) (\x->x-1)
```

4.3.2 Länge einer Liste

Die Berechnung der Länge einer Liste. Die Funktion zum Berechnen des Ergebnisses für jeden Schleifendurchlauf ignoriert die Schleifenvariable, was durch die Wildcard Variable im Lambda-Ausdruck zu erkennen ist.

Die Funktion für den terminierenden Fall ist die per Currying erzeugte Funktion mit der Funktion `const` aus dem Standardprelude.

Haskell: `len`

```
> len = loop null (const 0) (\_ y -> y+1) tail
```

4.3.3 Größtes Listenelement

Die Funktion, die das größte Listenelement errechnet. Hier ist der terminierende Fall interessant. Dieser ist nicht die leere, sondern die einelementige Liste. Deren `head` ist dann das gesuchte Ergebnis.

Haskell: `ma`

```
> ma :: Ord a => [a] -> a
> ma = loop (null.tail) head (\xs m -> max (head xs) m) tail
```

4.3.4 Filter über eine Liste

Hier die Funktion, die durch einen Filter eine neue Liste aus der Eingabeliste erzeugt.

Haskell: fi

```
> fi p = loop null id (\x y -> if p(head x) then (head x:y) else y) tail
```

4.3.5 Letztes Listenelement

Auch zur Berechnung des letzten Elements braucht es als terminierenden Fall, die elementige Liste.

Haskell: la

```
> la = loop (null.tail) head (\_ y -> y) tail
```

4.3.6 Umkehrung einer Liste

Das Umdrehen einer Liste geht komplexer als mit der Funktion `for`. Hier benötigen wir jetzt die Listenkonkatenation (`++`).

Haskell: re

```
> re = loop null id (\x r -> r++[head x]) tail
```

4.3.7 Generierung einer unendlichen Liste

Wir können auch eine unendliche Liste erzeugen. Hierzu ist der terminierende Fall immer auf `False` gesetzt. Die folgende Funktion generiert eine unendliche Liste aufsteigender Zahlen.

Haskell: from

```
> from = loop (\x->False) (const []) (\x rs -> x:rs) ((+) 1)
```

4.3.8 Generierung einer Liste mit Schranken

Wenn man den Terminierungstest in der Funktion `from` auf einen maximalen Zahlenwert setzt, dann können auch endliche Zahlenlisten erzeugt werden.

Haskell: fromTo

```
> fromTo frm to = loop (\x->x>to) (const []) (\x rs -> x:rs) ((+) 1)frm
```

Auch die Funktion `loop` erweist sich als mächtige Abstraktion, die es erlaubt die unterschiedlichsten Funktionen in einer Zeile zu definieren.

5 Schleifenfunktion in Standard-APIs

Das Prinzip der Funktionen höherer Ordnung, die über Schleifen bzw. Rekursionen abstrahieren, hat Eingang in die Standard-APIs der entsprechenden Sprachen gefunden. Wir werfen einen kurzen nicht vollständigen Blick auf die relevanten Funktionen.

In den meisten Programmier-APIs finden sich keine allgemeinen Schleifenfunktionen, wie wir sie gerade entwickelt haben, sondern spezielle Funktionen, mit denen man über die Elemente einer spezielle Listestruktur iterieren kann und die Elemente mit einer Operation verknüpft werden können.

5.1 Faltungen in Haskell

Im Haskell-Prelude werden die Schleifenfunktionen als Faltungen bezeichnet. Ursprünglich waren die Faltungen für Listen als Schleifenvariable implementiert. Dieses wurde verallgemeinert in eine Typklasse `Foldable`.

Wem Typklassen noch unbekannt sind, der stelle sich statt des allgemeinen Typs `(t a)` mit `Foldable t` den Listentyp `[a]` vor.

Wir geben einen Ausschnitt aus den Funktionen, die für `Foldable`-Typen definiert sind.

Haskell: Foldable

```
class Foldable t where
  foldr :: (a -> b -> b) -> b -> t a -> b
  foldl :: (b -> a -> b) -> b -> t a -> b
  foldr1 :: (a -> a -> a) -> t a -> a
  foldl1 :: (a -> a -> a) -> t a -> a
```

Die ersten beiden Funktionen, sind Schleifenfunktionen, die über die Elemente einer Struktur iterieren, diese mit einem Operator verknüpfen und ein neutrales Element dieser Operation benötigen, das insbesondere bei einer leeren Struktur, wenn also kein Element enthalten ist, als Ergebnis verwendet wird.

Die beiden Funktionen `foldr` und `foldl` unterscheiden sich darin, dass erstere die Operation rechtsassoziativ geklammert auf die Elemente anwendet, und letztere linksassoziativ.

Die beiden Versionen mit der 1 am Funktionsnamen, gehen davon aus, dass mindestens ein Element in der Struktur enthalten ist. Damit braucht es kein zusätzliches neutrales Element für die leere Liste. Für einelementige Listen übernimmt dieses ein Element dann die Rolle des Startwerts. Wir haben hierfür schon das Beispiel der Funktion, die das Maximum einer Liste errechnet, gesehen.

In der Dokumentation des Standard-Prelude sind Implementierungen der Listeninstanz dieser Funktionen gegeben (auch wenn diese nicht der eigentlich verwendete Implementierungen sind).

Dieses ist zum Beispiel die dort angegebene Implementierung der Funktion `foldr` für Listen:

Haskell: foldr Implementierung

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Viele Listenfunktionen sind im Standard-Prelude mit diesen Faltungen implementiert. Beispielsweise die Funktion, die das Minimum einer Liste von Elementen mit einer Ordnung errechnet:

Haskell: minimum Implementierung

```
minimum :: (Ord a) => [a] -> a
minimum [] = errorEmptyList "minimum"
minimum xs = foldl1 min xs
```

5.2 Faltungen für Listen in Scala

Das API für Listen der Programmiersprache Scala stellt drei Methoden für Schleifen zur Verfügung. Diese haben wie in Haskell auch `fold` im Namen.

Scala: fold API

```
def fold[A1 >: A](z: A1)(op: (A1, A1) => A1): A1

def foldLeft[B](z: B)(op: (B, A) => B): B

def foldRight[B](z: B)(op: (A, B) => B): B
```

Wie man sieht gibt es Versionen für Aufrufe zur linksassoziativen und rechtsassoziativen Verknüpfung der Elemente. Die allgemeine Version lässt offen, in welcher Klammerung die Operation angewendet wird.

Hier ein paar Beispielaufrufe:

Scala: fold Aufrufe

```
scala> List(1,2,3,4).fold(0)((x,y)=>x+y)
res5: Int = 10
scala> List(1,2,3,4).fold(0)((x,y)=>x-y)
res6: Int = -10
scala> List(1,2,3,4).foldLeft(0)((x,y)=>x-y)
res7: Int = -10
scala> List(1,2,3,4).foldRight(0)((x,y)=>x-y)
res8: Int = -2
```

Die Methoden werfen eine Fehler für leere Listen. Sie entsprechen also der Haskell-Versionen die im Namen auf 1 enden.

Scala: fold Aufrufe

```
scala> List().foldRight(0)((x,y)=>x+y)
<console>:12: error: value + is not a member of Nothing
      List().foldRight(0)((x,y)=>x+y)
```

5.3 Faltungen für Listen in Kotlin

Für Listen in der Programmiersprache Kotlin gibt es eine Methode `fold`.

Kotlin: fold API

```
abstract fun <R> fold(
    initial: R,
    operation: (R, Element) -> R
): R
```

Laut Dokumentation ist diese linksassoziativ geklammert.

Wir geben einen Beispielaufruf:

Kotlin: fold Aufrufe

```
listOf(1, 2, 3, 4).fold(1, {x,y->x*y})
```

5.4 Die reduce-Methode für Java Stream-Objekte

Mit Java 8 wurde durch die Einführung von Lambda-Ausdrücken Java nicht nur in den Sprachkonstrukten funktional, sondern erhielt auch Standard-APIs, die diesem Rechnung tragen. Hierzu wurde eine neue Schnittstelle entwickelt: **Stream**.

Diese stellt eine funktionale Erweiterung der Idee der Schnittstelle `Iterable` dar. In der Schnittstelle `Stream` befinden sich Methoden, die Schleifen über die Elemente eines Stream-Objekts ermöglichen, in denen die Objekte durch eine Operation verknüpft werden.

Die Schleifen-Methode für Stream-Objekte heißt `reduce` und ist dreifach überladen:

Java: reduce API

```
Optional<T> reduce(BinaryOperator<T> accumulator);

T reduce(T identity, BinaryOperator<T> accumulator);

<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
    ↪ BinaryOperator<U> combiner);
```

- Die erste Version verknüpft die Elemente mit einer binären Operation. Sie hat ein als Ergebnis ein Objekt der Klasse `Optional`. Dieses ist für den Fall, dass im Stream-Objekt gar kein Element enthalten ist.
- Die zweite Version hat als zusätzlichen Parameter ein neutrales Element für die Operation. Dieses dient insbesondere auch als Ergebnis für den Fall, dass kein Element im Stream-Objekt enthalten ist.
- Die allgemeinste Version ist die dritte, in der das Ergebnis nicht vom selben Typ sein muss, wie die Elemente des Stream-Objekts. Zusätzlich zum neutralen Element für ein leeres Stream-Objekt und der Funktion, die das Ergebnis akkumuliert, wird ein Operator als Argument benötigt, der Teilergebnisse zusammenführt. Dieses ist notwendig, wenn die Stream-Objekte die Elemente parallel verarbeiten.

5.5 Die reduce-Funktion auf Python Listen

Auch die dynamisch getypte Sprache Python bietet eine Funktion `reduce`, um die Elemente einer Liste mit einer Operation zu verbinden.

Hier zwei Beispielaufrufe dieser Funktion:

- einmal einer, in dem die Elemente einer Liste aufaddiert werden:

Python: reduce Aufruf

```
>>> xs = [1,2,3,4,5,6,7,8]
>>> reduce(lambda x,y:x+y,xs,0)
36
```

- und ein Aufruf, der das Maximum einer Liste berechnet:

Python: reduce Aufruf

```
>>> xs=[1,2,3,2,3,2,43,4,4,3,2]
>>> reduce(lambda x, y: y if x is None else max(x,y),xs,None)
43
```

5.6 Die reduce-Funktion in Ruby

Auch das Ruby API bietet eine Funktion mit Namen `reduce` an.

Wir gehen nicht ins Detail und geben zwei Beispielaufrufe:

Ruby: reduce Aufruf

```
pry(main)> [1,2,3,4].reduce(0) { |sum, num| sum + num }
=> 10
[4] pry(main)> [1,2,3,4].reduce(1) { |sum, num| sum * num }
=> 24
```

5.7 Die reduce-Funktion in Javascript

Selbst Javascript bietet eine Funktion `reduce` an.

Wir geben nur einen kurzen Beispielaufruf:

Javascript: reduce Aufruf

```
> [1,2,3,4].reduce( function(x,y){return x+y;})
10
```

Ein kleines Tutorial hierzu gibt es auf [Pit17].

6 Lernzuwachs

Folgende Erkenntnisse sollte sich nach Studium dieses Kurses gesammelt haben.

- Schleifen lassen sich als Rekursion umsetzen.
- Funktionen höherer Ordnung ermöglichen es, komplexe Kontrollstrukturen wie Schleifen in Funktionen allgemein zu schreiben.
- Rekursive Funktionen mit einer *tail recursion* können von einem Compiler zu einem Sprung optimiert werden. Somit wird es zur Laufzeit keinen Überlauf des Stack geben.

- Alle Sprachen, die Funktionen höherer Ordnung haben und in denen anonyme Funktionen über eine Lambda-Abstraktion erzeugt werden können, bieten in den Standardbibliotheken Funktionen an, die es erlauben über die Elemente aus Containern zu iterieren und die Elemente mit einem Operator zu verknüpfen. Diese Funktionen heißen zumeist *fold* oder *reduce*.
- Manche dieser Faltungsfunktionen verknüpfen die Elemente linksassoziativ andere rechtsassoziativ. Bei manchen wird vorausgesetzt, dass für die Operation das Assoziativgesetz gilt und somit die Klammerung keine Rolle spielen darf.

7 Aufgaben

- Im Haskell API gibt es Funktion für die rechtsassoziative und die linksassoziative Verknüpfung der Elemente. Wie sieht das in Java aus? Lesen Sie die Dokumentation der Methode `reduce` im Java API. Was wird über die Reihenfolge der Verknüpfung ausgesagt?
- Starten Sie in Emacs den Lisp-Interpreter mit `M-x ielm`. Definieren sie dort eine Faltungsfunktion über die Elemente einer Liste und machen Sie einen Beispielaufruf zum Summieren von Listenelemente.
- Schreiben Sie unter Verwendung der Funktion `for` eine Funktion, die die Quersumme des Arguments berechnet. Sie dürfen von einem Argument größer oder gleich 0 ausgehen.

Haskell: quersumme

```
> quersumme n = 0
```

- Schreiben Sie unter Verwendung der Funktion `for` eine Funktion, die aus einem String von Ziffern ein Integer Zahl einliest. Sie dürfen von einem Argument das nur Ziffern und mindestens eine Ziffer enthält, ausgehen.

Haskell: readNumber

```
> readNumber = const 0
```

- Schreiben Sie unter Verwendung der Funktion `for` eine Funktion, die aus eine positive ganze Zahl einen String erzeugt, der diese im Dualsystem darstellt.

Haskell: toBinary

```
> toBinary = const ""
```

- Schreiben Sie mit Hilfe der Funktion `for` eine Funktion `contains`, die testet, ob ein Element in einer Liste enthalten ist

Haskell: contains

```
> contains o = const False
```

- g) Schreiben Sie mit Hilfe der Funktion `loop` eine Funktion, die `ma`, die die Standardfunktion `map` für Listen umsetzt.

Haskell: ma1

```
> ma1 f = const []
```

- h) Wir haben mit Hilfe der Schleifenfunktionen bereits die Funktion `fi` entwickelt, die mit einer Prädikatsfunktion über die Element einer Liste filtert. Desweiteren haben wir mit `from` eine Funktion definiert, mit der die unendliche Liste der Zahlen ab 2 generiert werden kann. Mit Hilfe dieser können wir auf einfache Weise die Liste aller Primzahlen definieren:

```
> sieb (x:xs) = x:sieb (fi (\y->y `mod` x /=0) xs)
> prim = sieb$from 2
```

Reimplementieren Sie die Funktion `sieb` als `sieb2`, indem Sie die Schleifenfunktion `loop` verwenden, um die Rekursion auszudrücken.

Haskell: sieb2

```
> sieb2 = const []
```

- i) Schreiben Sie eine Funktion `woerter`, die aus einem Eingabestring eine Liste von Wörtern berechnet. Verwenden Sie hierzu die Funktion `loop`. Wörter können durch eine beliebige Anzahl von Leerzeichen getrennt sein. Ob es sich bei einem Zeichen um ein Leerzeichen handelt, können Sie mit der Funktion `isSpace` des Moduls `Data.Char` erfragen.

Tipp:

Verwenden Sie Ergebnistyp der Funktion `loop` den Tupeltyp `(String,[String])`. Das erste Tupelelement stellt dabei das aktuell eingelesene Wort dar, das zweite Tupelelement die bereits erkannten Wörter.

Haskell: woerter

```
> woerter = snd.result " ".loop null (const ([,[])) result tail
> where
>     result (x:xs) (w,ws) = ([,[])
```

Literatur

- [Pit17] Josh Pitzalis. How JavaScript's Reduce method works, when to use it, and some of the cool things it can do. <https://medium.freecodecamp.org/reduce-f47a7da511a9>, 2017. Accessed: 2019-03-08.