Übung 6 zu KMPS — Lösung

Aufgabe 34:
a) Machen Sie aus der Funktion `foldr` aus Aufgabe 33 eine polymorphe Funktion, so dass sie möglichst allgemein verwendbar ist.

b) Wie muss man die Funktion aus a) aufrufen, um die Subtraktion der Listenelemente vom Startwert zu erhalten?

Lösung:
a)
```
def foldrGen[A,B](f:(A,B) => A, start: A, xs: List[B]) : A =
  xs match {
    case Nil => start
    case y::ys => f(foldrGen(f,start,ys),y)
  }
```
b)
```
foldrGen[Int,Int]((x,y) => x-y,0,1::2::3::4::Nil)
```

Aufgabe 36:
Geben Sie eine **Objekt-Funktionale** Implementierung der `filter`–Methode an, die auf Folie ObjektFunktionalMutableImmutable 4 unten verwendet wird in der Klasse `List`. Dabei ist nur die Angabe der Methode erforderlich und Sie können davon ausgehen, dass man die Klasse `List` anpassen kann.

Geben Sie hierzu jeweils eine Lösung an, die wie folgt arbeitet:

a) immutable

b) mutable

c) Welche der beiden Variationen wird auf Folie ScalaBesonderheiten 6 verwendet?

d) Was würde sich ergeben, wenn Sie die andere Variante verwenden würden?

Lösung:
Zuerst a) und b) wie gewünscht nur in Auszügen:

```
// Sei list ein Attribut der Klasse List
var list: List[Int] = 1::2::3::4::Nil

// Müssen neue filter-Funktionen definieren, da schon in List
```

a)
```
def filterImmutable(func: Int => Boolean): List[Int] =

    // Listen sind in Scala immutable
    def filter_help(toFilter: List[Int]): List[Int] =
        toFilter match {
            case Nil => Nil
            case head::tail =>
                if (func(head))        head::inner(tail)
                else                   inner(tail)
        }
```

Übung 6 zu KMPS Lösung


```
    filter_help(list)
```

b)

```
def filterMutuable(func: Int => Boolean): Unit =
    // setze Attribut auf Ergebnis von filterA.
    list = filterImmutable(func)
```

Jetzt weitere vollständige Lösungen mit einigen Scala-Feinheiten:
Weil man nicht von der Klasse List erben kann muss man seine eigene Klasse schreiben, die eine Liste als Member enthält. Das **implicit def** sorgt dafür, dass eine Liste bei Bedarf einfach in diese Klasse umgewandelt wird.
Die Funktion wurde myfilter genannt, weil filter bereits in der Klasse List existiert.
Bei den Immutable-Varianten benötigt man einen 2. Parameter, den man für die rekursiven Aufrufe braucht. Da man myfilter allerdings nur mit dem Prädikat als Parameter aufrufen möchte, werden 2 Varianten vorgestellt, die diesen Konflikt behandeln.

a) **Immutable Variante 1 (Default-Argument): (filtert Membervariable, gibt gefilterte Kopie zurück)**

```
implicit def listToList_with_FilterIm1[T](l: List[T]) = new List_with_FilterIm1[T](l)

class List_with_FilterIm1[T](list: List[T]) {
  def myfilter(condition: T => Boolean, l: List[T] = this.list): List[T] = l match {
    case head :: tail => condition(head) match {
      case true => head :: myfilter(condition, tail)
      case false => myfilter(condition, tail)
    }
    case Nil => Nil
  }
}
```

Aufruf:

```
var listIm : List_with_FilterIm1[Int] = List(1,2,3,4,5,6,7,7,7,8,8,9)
listIm.myfilter(x => x % 2 == 0)
```

a) **Immutable Variante 2 (Hilfs-Methode): (gleiches Verhalten wie Variante 1)**

```
implicit def listToList_with_FilterIm2[T](l: List[T]) = new List_with_FilterIm2[T](l)

class List_with_FilterIm2[T](list: List[T]) {
  def filter_recursive[T](l: List[T], condition: T => Boolean): List[T] = l match {
    case head :: tail => condition(head) match{
      case true => head :: filter_recursive(tail, condition)
      case false => filter_recursive(tail, condition)
    }
    case Nil => Nil
  }

  def myfilter(condition: T => Boolean): List[T] = filter_recursive(list, condition)
}
```

Aufruf:

```
var listIm2 : List with FilterIm1[Int] = List(1,2,3,4,5,6,7,7,7,8,8,9)
listIm2.myfilter(x => x % 2 == 0)
```

b) **Mutable: (filtert Membervariable, modifiziert Membervariable, gibt nichts zurück)**

Übung 6 zu KMPS **Lösung**


```
implicit def listToList_with_Filter[T](l: List[T]) = new List_with_Filter[T](l)

class List_with_Filter[T](var list: List[T]) {
  def myfilter(condition: T => Boolean): Unit = list match {
    case head :: tail => condition(head) match {
      case true =>
        this.list = tail
        myfilter(condition)
        this.list = head :: this.list
      case false =>
        this.list = tail
        myfilter(condition)
    }
    case Nil => //do nothing
  }
}
```

Aufruf:

```
var list : List_with_Filter[Int] = List(1,2,3,4,5,6,7,7,7,8,8,9)
list.myfilter(x => x % 2 == 0)
print(list.list)
```

c) Es wird die Immutable-Variante verwendet, damit die ursprüngliche Liste `xs` nicht verändert wird, da sie in den nächsten Parametern noch in der ursprünglichen Version gebraucht wird.

d) Das `filter` im Parameter 1 würde `xs` so verändern, dass nur noch die Elemente enthalten sind, die kleiner als das erste sind.
-> Im 2. und 3. Parameter würden leere Listen erzeugt.
Der rekursive Aufruf im 1. Argument würde dann iteriert als Ergebnis eine Liste liefern, die höchstens das ursprünglich kleinste Element enthält, falls man irgendwann auf eine Liste der Länge 1 kommt. Sonst erhält man die leere Liste.

Aufgabe 37:
Die Fibonacci-Funktion `fib` ist wie folgt definiert : 1 -> 0, 2 -> 1, n -> fib(n-1) + fib(n-2)

a) Implementieren Sie eine Funktion `fibonacci`, die einen unendlichen Lazy-Stream der Fibonaccizahlen erzeugt.

b) Implementieren Sie eine Funktion `nth`, die die n-te Zahl in einem Stream liefert.

c) Implementieren Sie die Fibonacci-Funktion `fib` unter Verwendung der beiden Funktionen aus a) und b).

d) Welchen Zeit- und Platz-Aufwand benötigt `fib`?

Lösung:

```
a)
def fibonacci: Stream[Int] = {
  def loop(a: Int, b: Int): Stream[Int] = a #:: loop(b, b + a)
  loop(0, 1)
}

b)
def nth(n:Int,xs:Stream[Int]): Int = xs match {
```

Übung 6 zu KMPS

Lösung


```
   case y#:: ys   => if (n == 1)  y  else nth(n - 1, ys)
}

c)
def fib(n:Int) = nth(n,fibonacci)

d)
```

Beides Linear, da man immer n Einträge der Liste erzeugen muss mit n rekursiven Aufrufen von loop.

Aufgabe 38:
Geben Sie jeweils eine Implementierung der folgenden beiden Higher-Order-Prädikate an, ohne Verwendung von Pattern Matching, indem Sie lediglich die `filter`-Funktion richtig einsetzen:

a) `exists`, das überprüft, ob es zu einem gegebenen Prädikat ein Listenelement gibt, für das das Prädikat gilt.

b) `forall`, das überprüft, ob ein gegebenes Prädikat für alle Listenelemente gilt.

Lösung:
a)

```
def exists[A](p:A=>Boolean, xs:List[A]) : Boolean = !(filter(xs,p) == Nil)


exists[Int](x=>x%2==0,1::3::Nil)
```

b)

```
def forall[A](p:A=>Boolean,xs:List[A]) : Boolean = (filter(xs,not(p)) == Nil)
//mit:
def not[A](f: A => Boolean): A => Boolean = !f(_)


forall[Int](x=>x%2==0,2::4::Nil)
```

Alternativ:

```
def forall[A](p:A=>Boolean, xs:List[A]) : Boolean = filter(xs,p) == xs
```

Aufgabe 39:
a) Geben Sie einen Lambda-Ausdruck an, der die Verdoppelung einer Zahl berechnet.

b) Wenden Sie Ihren Ausdruck aus a) auf die Zahl 5 an und zeigen Sie, dass der Ausdruck den Wert 10 liefert.

Lösung:
a) $\lambda x.(x+x)$

b) $\lambda x.(x+x)$ 5 -> 5+5 -> 10

Übung 6 zu KMPS **Lösung**

Aufgabe 40:

a) Implementieren Sie die `filter`-Funktion aus Praktikum 3 Aufgabe 2 in Java unter Verwendung von Lambda-Ausdrücken. Dabei darf die `filter`-Methode der Java-Bibliothek nicht verwendet werden.

b) Wenden Sie die `filter`-Funktion aus a) auf die Testeingabe aus Praktikum 3 Aufgabe 2 an.

Lösung:
a) rekursiv:

```
public static <T> List<T> filter(List<T> list, Function<T, Boolean> condition) {
  if(list.isEmpty()) return Collections.<T> emptyList();
  T first = list.get(0);
  ArrayList<T> result = new ArrayList<T>();
  if(condition.apply(first)) result.add(first);
  result.addAll(filter(list.subList(1, list.size()), condition));
  return result;
}
```

imperativ:

```
public static <T> List<T> filter_imp(List<T> list, Function<T, Boolean> condition) {
  for(int i = list.size() - 1; i >= 0; i--) {
    if(!condition.apply(list.get(i))) list.remove(i);
  }
  return list;
}
```

Beispiel Anwendung:

```
ArrayList<Integer> list2 = new ArrayList<Integer>(Arrays.asList(1,2,3,4,5,6));
List<Integer> filtered2 = filter(list2, (x)->x%2==0);
System.out.println(filtered2);
```