Übung 2 zu KMPS — Lösung

Aufgabe 8:
Beim InsertionSort wird immer das 1. Element einer Liste an der richtigen Stelle der sortierten Restliste eingefügt.

Implementieren Sie eine Funktion `insSort` in Scala unter Verwendung der Funktion `insert` aus dem Foliensatz und Pattern Matching über der internen Listenstruktur von Scala.

Lösung:
```
def insSort(xs:List[Int]) : List[Int] = xs match {
  case Nil => Nil
  case y :: ys => insert(y,insSort(ys))
}
```

Aufgabe 9:
Definieren Sie mit Hilfe von sog. Case Classes und Case Objects eine Datenstruktur „BinTree", die beliebig lange Binärbäume repräsentieren kann. Achten Sie hierbei darauf, dass der „leere Baum" später zur Laufzeit nur einmal vorliegen bzw. instanziiert sein darf.

Lösung:

```
abstract class BinTree
case object EmptyTree extends BinTree
case class Node(elem:Int,
                left:BinTree,
                right:BinTree) extends BinTree
```

Aufgabe 10:
Implementieren Sie eine Scala-Funktion `append`, die zwei Listen aneinanderhängt.
Jeweils einmal auf der Scala-internen Listenstruktur und einmal auf der selbst definierten Struktur mit Case Classes und Case Objects.

Lösung:
```
def append1(xs:List[Any],ys:List[Any]) : List[Any] = xs match {
  case Nil => ys
  case y::zs => y::append1(zs,ys)
}

append1(1::2::Nil,3::4::Nil)

def append2(xs:MyList,ys:MyList) : MyList = xs match {
  case Nil => ys
  case List(y, zs) => List(y,append2(zs,ys))
}

append2(List(1,List(2,Nil)),List(3,List(4,Nil)))
```

Übung 2 zu KMPS **Lösung**

Aufgabe 11:
Implementieren Sie eine Scala-Funktion `transList`, die eine beliebige Liste in der Scala-internen Listenstruktur in eine Liste in der selbst definierten Struktur mit Case Classes und Case Objects transformiert.

Lösung:
```
abstract class MyList
case object N extends MyList
case class L(head:Any,tail:MyList) extends MyList

def transList(xs:List[Any]) : MyList = xs match {
  case Nil => N
  case y::ys =>  L(y,transList(ys))
}

transList(Nil)
transList(1::2::3::Nil)
```

Aufgabe 12:
Implementieren Sie eine Scala-Funktion `präfix`, die überprüft, ob eine Liste `xs` mit einer Liste `ys` beginnt.
Verwenden Sie dabei die Scala-interne Listenstruktur und überlegen Sie sich eine möglichst große Testabdeckung.

Lösung:
```
def präfix(ys:List[Any],xs:List[Any]) : Boolean = ys match {
  case Nil => true
  case z::zs => xs match {
    case Nil => false
    case z1::z1s => if(z1!=z) false else präfix(zs,z1s)
  }
}
präfix(Nil,Nil)
präfix(1::2::3::Nil,Nil)
präfix(1::2::3::Nil,1::2::Nil)
präfix(Nil,1::2::Nil)
```

Aufgabe 13:
Implementieren Sie eine Scala-Funktion `attach`, die einen beliebigen Wert an das Ende einer Liste `xs` anhängt. Dabei dürfen Sie keine weiteren Funktionen verwenden.
Verwenden Sie dabei die Scala-interne Listenstruktur und überlegen Sie sich eine möglichst große Testabdeckung.

Lösung:
```
def attach(x:Any,xs:List[Any]) : List[Any] = xs match {
  case Nil => x::Nil
  case y::ys => y::attach(x,ys)
}

attach(1,Nil)
attach(3,1::2::Nil)
```

Übung 2 zu KMPS

Lösung

Aufgabe 14:
Implementieren Sie eine Funktion `anzKnoten`, die die Anzahl der Knoten eines Binärbaums berechnet. Der Binärbaum soll dabei gemäß der Darstellung in Aufgabe 9 definiert sein.
Überprüfen Sie die Funktion `anzKnoten`, indem Sie möglichst sinnvolle „Testdaten" als Eingabewerte verwenden.

Lösung:

```
abstract class BinTree
case object EmptyTree extends BinTree
case class Node(elem:Any,left:BinTree,right:BinTree) extends BinTree

def anzKnoten(xb : BinTree) : Int = xb match {
  case EmptyTree => 0
  case Node(elem, left, right) => 1 + anzKnoten(left) + anzKnoten(right)
}

anzKnoten(EmptyTree)
anzKnoten(Node(1,Node(2,EmptyTree,EmptyTree),EmptyTree))
```