3. Datenstrukturen

Lineare Datenstrukturen: Felder, Vektoren, Listen
Modelle: math. Folge $(a_i)_{i=1..}$ mit Basistyp T
oder: [T]

Nichtlineare Datenstrukturen:
Bäume
Modell(e): spezielle Graphen
Verwendung: effiziente Implementierung von Mengen
und vieles mehr (s.u)
Graphen:
Modell(e): Knoten und Kanten (zweistellige Relation über Knoten)

3.1 Folgen

Modell : [T] also Liste vom Basistyp T (funktional)
Invariante: z.B. Längenbeschränkung,
Wertebereichsbeschränkung
(z.B. "nur positive Zahlen")
Operationen:
Lesen, Ändern, Entfernen, Hinzufügen von Folgenelementen

Spezielle Folgen schon als ADT behandelt:
Keller (stack), Schlange (Queue), Prioritätsschlange

Implementierung: Feld, Vector (Java), Liste, Zahl (!)

Traversieren von Folgen

3.1.1 Traversieren von Folgen

Durchlaufen aller Folgenelemente:

```
for (each ai aus Folge (ai)i=1..n )
        op (ai)     // tue etwas
```

Funktional einfach!

Map

```
map :: (a -> b)  -> [a] -> [b]
map  op liste = [ op x |  x <- liste]
```


Funktionales Traversieren

Filtern

```
filter :: (a -> Bool) -> [a]  -> [a]
  filter p liste  = [ x | x <- liste , p x]
```

Beispiel: alle 'Null-Werte' aus Liste entfernen

```
x = filter p listevonZahlen
   where p x
        | x==0  = False
        | otherwise       = True
```

Funktionales Traversieren

Falten

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

Beispiel: Anzahl Zeichen eines Textes

```
textLength = foldl f 0 text
    where  f i c  = i+1
```


Traversieren und Bearbeiten von Folgen: Imperativ

Probleme

1. Wie abstrahiert man vom Bestimmen des nächsten Elements?

```
for (inti ..;..; i++)   {a[i] =....;...  }
     // Feldindex
      ....    this.next() ...
// nächstes Listenelement
// stark implementierungsabhängig
```

2. Wie schreibt man eine Funktion höherer Ordnung (in oo-Sprache wie java)?

Funktionen als Argumente?

```
public static Object[] map(Fct f,Object[] a)
  {
   Object[] res = new Object[a.length];
   for ( int i = 0; i < a.length; i++ )
     res[i]= f.call (a[i]);
   return res;
  }
```

Traversieren ohne Abstraktion (hier: Feldelemente durchlaufen)

Higher Order Function: Objekt übergeben, das die Funktion (Methode) besitzt.

Abstraktion von spezieller Funktion `f` mittels `interface`:

```
public interface Fct{
   public  Object call( Object arg )
        throws IllegalArgumentException;}
```

Beachte Signatur für einstellige Funktion:

```
f:: Object -> Object
```

Flexibilität der Signatur nicht durch parametrische Polymorphie (Typvariablen **`f:: a -> b)`** **sondern** durch Vererbung

Map

```
public class MapX{
  public static Object[] map(Fct f, Object[] a)
      throws IllegalArgumentException{
   Object[] res = new Object[a.length];
   for ( int i = 0; i < a.length; i++ )
     res[i]= f.call (a[i]);
   return res;
  }
}
```

Es wird Objekt übergeben, Funktion als Argument direkt nicht möglich

Imperative Liste entsprechend ohne Rückgabewert:

```
void map (....){ ....; f.call(a[i]);}
```


Implementierung einer Beispiel-"Funktion"

```
public class Twice implements Fct{
  public  Object call(Object arg)
     throws IllegalArgumentException {
    return
   new Integer(2*((Integer)arg).intValue());
  }
}
```

Noch eine Funktion

```
import Fct;
public class Reverse2 implements Fct{  public
  Object call( Object arg )
      throws IllegalArgumentException  {
    if ( !( arg instanceof StringBuffer ) )
      throw new IllegalArgumentException
       ("Falscher Argumenttyp: " + "ist "+
          arg.getClass().getName() + " ...");

  return ( (StringBuffer) arg ).reverse();
  // reverse is method of class Stringbuffer
 }
}
```


Anwendung

```
 import ...
 public class TestMap2{
   public static void main( String[] args )  {     ;
    if   ...         {  .....System.exit(0);
      }
    else {
    StringBuffer[] orig = new StringBuffer[l];
    for ( int i = 0; i < l; i++ )
         orig[i] = new StringBuffer( args[i] );
    Reverse2 reverse = new Reverse2();
    Object[] s = MapX.map( reverse, orig );
    for ( int i = 0; i < l; i++ )
      ... // print!
    }
 }
}
```