Vergleich von Programmiersprachen

Am Beispiel Haskell - Java

von Nils Japke und Alexander Wollny & Reichenbach

Gliederung

1) Funktionale/ Imperative Programmierung
2) Beispiele in Haskell und Java
   a) Funktionen
   b) Rekursion
   c) Listen
   d) Listenbeschreibungen
   e) Funktionen höherer Ordnung
   f) Typvariablen
   g) Quicksort
3) Anwendung funktionaler Sprachen

Funktionale/ Imperative Programmierung

Funktional:

- Was?
- Keine Schleifen
- Variablen fest
- Strenge Typisierung
- Einfache Sprache
- Großer Speicheraufwand

Objektorientiert:

- Wie?
- Schleifen möglich
- Variablen veränderbar
- Polymorphie
- Maschinennähere Sprache
- Niedrigerer Speicheraufwand

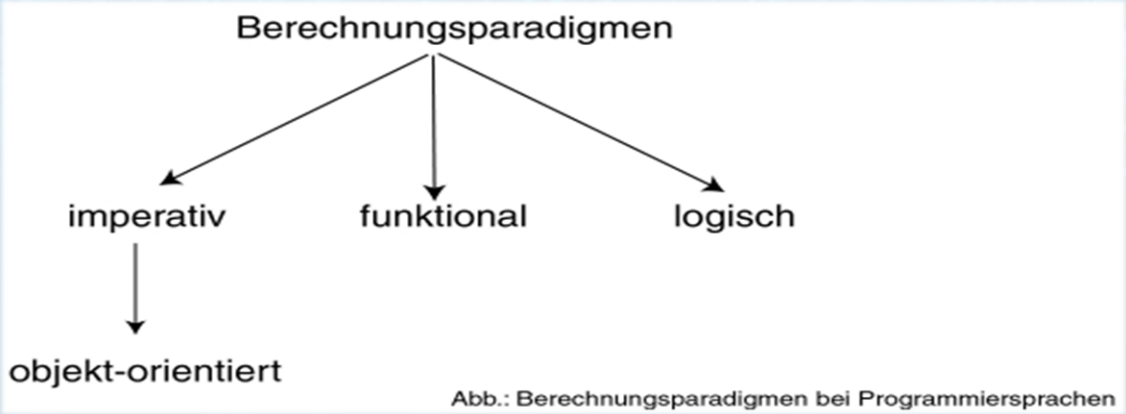

Abb.: Berechnungsparadigmen bei Programmiersprachen

Funktionen

Haskell:

```
plus5 :: [Int] -> [Int]
plus5 i = i+5
```

Java:

```
public class Test {
    public void plus5 (int x) {
          System.out.println(x+5);
    }
}
```

Rekursion

Haskell:

```
Fib :: [Float] -> [Int]
Fib 1 = 1
Fib 2 = 1
Fib n = Fib (n-1) + Fib (n-2)
```

oder

```
Fib :: [Float] -> [Int]
Fib n
 | n==1 || n ==2 = 1
 | otherwise = Fib (n-1) + Fib (n-2)
```

Java:

```
public int fib (int n) {
   if (n==1 || n == 2) {
        return 1;
   }
   else return (fib(n-1) + fib(n-2));
```

Listen

Haskell:

```
summe :: [Int] -> Int
summe [] = 0
summe (x:xs) = x + sum xs
```

Java:

```
public int summe(int[] n) {
    int sum = 0;
    for (int i; i<n.length; i++) {
          sum = sum + n;
    }
    return sum;
```

→ auch möglich mit ArrayList

Listenbeschreibungen

Haskell:

```
roundList :: [Float] -> [Int]
roundList i = [ round x | x <- i ]
```

Java:

```
interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}


Vector roundList (Enumeration i) {
  Vector res = new Vector();

  while ( i.hasMoreElements() ) {
    res = res.addElement( round(
   i.nextElement() ));
  }
  return res;
}
```

Funktionen höherer Ordnung

Haskell:

```
map :: (Float -> Int) -> [Float] ->
   [Int]
map f i = [ f x | x <- i ]
roundList = map round
```

Java:

```
Vector map (Function f, Enumeration
   i) {
  Vector res = new Vector();

  while ( i.hasMoreElements() ) {
    res = res.addElement (f.apply
          (i.nextElement() ));
  }
  return res;
}
interface Function {
  float apply( int i );
}
interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

Typvariablen

Haskell:

```
roundList = map round
    -- round vom Typ (Float -> Int)
upperList = map upper
    -- upper vom Typ (Char -> Char)
```

Java:

```
Vector construct (Function f,
    Enumeration i) {
  Vector res = new Vector();

  while ( i.hasMoreElements() ) {
    res = res.addElement( f.apply(
   i.nextElement() ));
  }
  return res;
}

interface Function {
  Object apply( Object o );
}
```

Typvariablen

Haskell:

```
roundList = map round
    -- round vom Typ (Float -> Int)
upperList = map upper
    -- upper vom Typ (Char -> Char)
```

Java:

```
Vector construct (Function f,
    Enumeration i) {
  Vector res = new Vector();

  while ( i.hasMoreElements() ) {
    res = res.addElement(f.apply
          (i.nextElement() ));
  }
  return res;
}

interface Function <T> {
  T apply( T o );
}
```

Quicksort

Haskell:

```
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
   quicksort [i | i <- xs, i <= x]
   ++ [x] ++
   quicksort [j | j <- xs, j > x]
```

Java:

```
Static void quicksort (int links,int rechts) {
    if (links<rechts) {
           int t = zerlege (links, rechts);
           quicksort (links, t-1);
           quicksort (t, rechts);}
    }

static int zerlege(int li, int re) {
    int l, r, trenn, x;
    l=li; r=re; trenn=feld.length/2;
    while (l<r) {
           if (feld[trenn]>feld[l]) {
                if (feld[trenn]<feld[r]) {
                     x=feld[r];
                     feld[r]=feld[l];
                     feld[l]=x; }
                else {r--;} }
           else {l++;} }
    return trenn;
```

Anwendung funktionaler Sprachen

Prototyping (Erstellen von Software Prototypen)

Vorteile von Haskell:

mathematische Schreibweise

leichtere Verständlichkeit diverser Problemstellungen

leichte Verständlichkeit des Codes

leichte Dokumentation des Codes

kurzer Code für diverse mathematische Probleme

=> leichteres Finden und Entfernen von Fehlern früh im Entwicklungszyklus

Quelle

- http://www.fh-wedel.de/~si/seminare/ss02/Ausarbeitung/1.paradigma/Paradig0.htm