Higher-Order Functions

Higher-Order Programmierung

- **bisher: First-Order Programmierung**
  Funktionen nur außen, weder als Parameter noch als Wert
  ähnlich zu Prädikaten in First-Order Logic (Prolog)
- **jetzt: Higher-Order Programmierung**
  Funktionen als Parameter bzw. Werte
- **Motivation**
  äußere Funktion verarbeitet innere immer gleich
- **Bsp.:** Summation von Funktionswerten im Bereich von a bis b
  Berechne $\sum_{n=a}^{b} f(n)$ für spezielle f.

Higher-Order Programmierung
Beispiel

- **Summation von Funktionswerten im Bereich von a bis b**

1. Summation der Integerwerte zwischen a und b

```
def sumInts(a: Int, b: Int): Int =
    if (a > b) 0 else a + sumInts(a + 1, b)
```

2. Summation der Quadratwerte der Zahlen zwischen a und b

```
def sumSquares(a: Int, b: Int): Int =
    if (a > b) 0 else square(a) + sumSquares(a + 1, b)
```

3. Summation der Zweierpotenzen der Zahlen zwischen a und b

```
def sumPOT(a: Int, b: Int): Int =
    if (a > b) 0 else pot(a) + sumPOT(a + 1, b)
```

- **Extrahiere gleichen Anteile zu einer Higher-Order Funktion**

```
def sum(f: Int => Int, a: Int, b: Int): Int
=  if (a > b) 0 else f(a) + sum(f,a + 1, b)
```

Higher-Order Programmierung
Spezielle Funktionen durch Instanzierung

```
def sumInts(a: Int, b: Int): Int
      = sum(id, a, b)
def sumSquares(a: Int, b: Int): Int
      = sum(square, a, b)
def sumPOT(a: Int, b: Int): Int
      = sum(pot, a, b)
```

mit

```
def id(x: Int): Int             = x
def square(x: Int): Int         = x * x
def pot(x: Int): Int =
      if (x == 0) 1
      else 2 * pot(x - 1)
```

Anonyme Funktionen

- **bisher: jede Funktion separat definiert**
- werden aber jeweils nur einmal gebraucht: Einwegfunktionen, darum
- **jetzt: als anonyme Funktionen definieren (Syntactic Sugar)**
- analog zu: anonymen Klassen in Java (Listener)
- **Schreibe Funktionsdefinition direkt in Aufruf:**

```
def sumInts(a: Int, b: Int): Int
                  = sum((x: Int) => x, a, b)
def sumSquares(a: Int, b: Int): Int
                  = sum((x: Int) => x * x, a, b)
```

- **Typ von x kann weggelassen werden**

```
def sumInts(a: Int, b: Int): Int
                  = sum(x => x, a, b)
```

- durch Typ von `sum` klar
- **Später bei Lambda-Ausdrücken gebraucht!!!**

Higher-Order Funktionen
weitere Motivation & Beispiel

- **bei Listenbearbeitung häufig eins der drei Muster:**
  - Filter spezielle Elemente aus Liste : `filter`
  - Verknüpfe Listenelemente : `fold`
  - und: `map`
- **gesucht:**
  Funktion `map`, die eine Funktion `f` auf alle Listenelemente anwendet
- **Bsp.: für square** 1::2::3::4::*Nil* -> 1::4::9::16::*Nil*
- **Lösung:**

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

Aufruf: `map(1::2::Nil,x=>x*x) -> 1::4::Nil`

Higher-Order Funktionen
weiteres Beispiel

- **gesucht:** Funktion `filter`

  liefert zu Integerliste, Liste aller Elemente, die Bedingung erfüllen

- **Typ (Signatur):** mathematisch:

  List[Integer] x (Integer -> Boolean) -> List[Integer]

- **Implementierung:**

```
def filter(xs: List[Int], f : Int => Boolean) : List[Int]
 =  xs match {
     case Nil    => Nil
     case y::ys => if (f(y)) y::filter(ys,f) else filter(ys,f)
    }
```

- **Aufruf:** alle Elemente einer Integerliste, die gerade sind

```
filter(1::2::3::4::Nil, x => x%2==0)
   ->   2::4::Nil
```