Lambda-Ausdrücke

Lambda-Ausdrücke
Motivation

- Funktionale Konzepte spielen immer stärkere Rolle

-> Einführung von Funktionalen Konzepten in Programmiersprachen

- Meistens als Lambda-Ausdrücke bezeichnet
- Funktionen als Parameter und Rückgabetyp in Funktionen
- ähnlich zu anonymen Funktionen : `def sumInts = sum(x=>x)`
- **x=>x** anonym entspricht: `def id(x:Int):Int = x`
- im Lambda-Kalkül alles anonym: `λx.x`
- Anwendung auf 5: `(λx.x)(5) -> 5`
- http://www.inf.fu-berlin.de/lehre/WS12/ALP1/lectures/V19_ALPI_Lambda_Kalkuel_2013.pdf
- http://www.betoerend.de/dasLandHinterDemEndeDesSinns/lambda/welcome.html

Lambda-Kalkül
in a NutShell

- formale Sprache zur Untersuchung von Funktionen
- von Church & Kleene in 30ern eingeführt
- Grundlage vieler funktionalen Sprachen
- 2 grundlegende Bausteine:
  - Funktionsabstraktion `λx.A` `(x => A)` `(linksassoziativ)`
  - Funktionsapplikation `F A` `(F(A))` `(rechtsassoziativ)`
- Beispiele:
  - Identität `λx.x` `(Typ: Any -> Any)`
  - Funktion, die jede Funktion auf die Identitätsfunktion abbildet
    `λy.(λx.x) (Typ: Any -> (Any -> Any))`
  - Applikation: Identität angewandt auf sich selbst
    `λx.x (λy.y) -> λy.y`
  - Funktion, die eine Funktion zweimal auf was anderes anwendet
    `(λf.(λx.f(f x)))` angewendet auf `square 3 -> 81`
    `-> (λx.square(square x)) 3 -> (square(square 3))`

Lambda-Ausdrücke
in Java

- Funktionen als Parameter und Rückgabetypen in Funktionen
- bis Java 8 nur über Interfaces möglich
- ab Java 8 Lambda-Ausdrücke
- Bsp.:
  - `x -> x + 1  // Bezeichner links nur allein ohne Typ`
  - `(Integer i) -> list.add(i)`
  - ```
    (Integer a, Integer b) -> {
      if (a < b) return a + b;
      return a;                 }
    ```
  - `() -> System.out.println("Hallo World!")`
- Syntax:

```
LambdaExpr ::= LambdaPar '->' LambdaBody
LambdaPar  ::= Identifier | '(' ParameterList ')'
LambdaBody ::= Expression | Block
```
```

Lambda-Ausdrücke
in Java: weitere Beispiele

```
(int x) -> x + 1  // ok: Parliste mit 1 Parameter
int x -> x + 1    // falsch: Da Partyp ->  Klammern
(x) -> x + 1      // ok: fehlender Typ wird deduziert
x -> x+1          // ok: einzelner Bezeichner ohne Typ
(int x, int y) -> x+y // ok: Parliste mit 2 Parametern
int x, int y -> x+y // falsch: siehe 2. Beispiel
(x,y) -> x+y      // ok: siehe 3. Beispiel
x,y -> x+y        // falsch, da 2 Par. -> Klammern
(x, int y) -> x+y // falsch: Par. nicht mit und ohne Typ
() -> 42          // ok: Parliste darf leer sein

(int x) -> return x+1 // falsch, da rechts kein Ausdruck
(int x) -> { return x+1; } // o.k., da Anweisung
                                    geklammert
```

Lambda-Ausdrücke
in Java : Einsatz

- schönere Formulierung anstelle anonymer Klassen bei EventListener:

```
handler.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        // Auszuführender Code
    }
 } )
```

folgende Darstellung:

```
h.addActionListener((e) -> {Auszuführender Code}  )
```

- **Idee:** mache aus Interface mit Methode anonyme Funktion: Parameter -> Ausdruck oder Anweisungsblock

Lambda-Ausdrücke
in Java : Erweiterung von Java um Interfaces

Wie Funktionstypen in Parametern oder als Rückgabe definieren?

Hierzu im Package `java.util.function`:

- `Function<T,R> // Funktion T -> R`

```
Function<Integer, Integer> doppelt = (n) -> 2*n;
doppelt.apply(3); // liefert 6
```

- `Predicate<T> // Prädikat T -> boolean`

```
Predicate<Integer> isEven = (n) -> n % 2 == 0;
isEven.test(2); // liefert true
```

- `..`

zur Ausführung: `apply/test`

Lambda-Ausdrücke
in Java : als Parameter

```
Integer applyNurWennP(  Predicate<Integer> p,
                        Function<Integer,Integer> f,
                        Integer i) {
      if (p.test(i)) return f.apply(i);
      else return i;
}

applyNurWennP(x -> x%2 == 0, x -> x*x, 6))
 -> 36
applyNurWennP(x -> x%2 == 0, x -> 2*x, 6))
 -> 12
applyNurWennP(x -> x%3 == 0, x -> x/3, 6))
 -> 2
```

Lambda-Ausdrücke
in Java : als Rückgabetyp

```
Function<Integer,Integer> func() {
      return (x -> 2*x);
}

func().apply(3)
-> 6
```