Lazy Evaluation

Lazy Evaluation
in Scala (in ScalaFiddle)

- **definieren Endlos-Loop**

```
def loop(x:Int) : Int = loop(x+1)
```

- **definieren Konstante mit Aufruf von loop**

```
val x = loop(3)          ->    Endlos-Loop
```

- **mache Konstante lazy mit Aufruf von loop**

```
lazy val x = loop(3)     ->    Abbruch
```

- **gebe Konstante aus**

```
x                        ->    Endlos-Loop
```

- **definiere Funktion, die nur 2. Argument ausgibt**

```
def f(x:Int,y:Int) = y
```

- **rufe mit Endlos-Loop im 1. Argument auf**

```
println(f(x,4))     ->   x lazy, Endlos-Loop
```

- **mache in Funktions-Definition 1. Argument lazy und obiger Aufruf**

```
def f(x : => Int,y : Int) = y    ->  4
```

Lazy in anderen Sprachen
Verwendung unendlicher Datenstrukturen

- **Wdh.:** Call-by-Value Standardauswertung in Scala
- **Was machen folgende Funktionen und der Ausdruck?**

```
def from(x: Int) : List[Int] = if (x<=0) Nil else x :: from(x+1)

def take(x: Int, xs: List[Int]) : List[Int] = xs match {
  case Nil => Nil
  case y :: ys => if (x <= 0) Nil else y :: take(x - 1, ys)
}

take(2,from(5))
```

- `from`: erzeugt endlose Liste beginnend von `x`
- `take`: liefert die ersten `x` Elemente einer Liste
- `take(2,from(5))`: Endlosschleife, da cbv, sollte `5::6::Nil` liefern

Lazy in anderen Sprachen
Verwendung von call-by-name

- **Ändere alle Parameter in call-by-name-Parameter in Scala**
- **Was machen folgende Funktionen und der Ausdruck dann?**

```
def from(x: =>Int) : List[Int] = if (x<=0) Nil else x::from(x+1)

def take(x: => Int, xs: => List[Int]) : List[Int] = xs match {
  case Nil => Nil
  case y :: ys => if (x <= 0) Nil else y :: take(x - 1, ys)
}

take(2,from(5))
```

- `take(2,from(5))`: Endlosschleife, auch bei call-by-name

- In **Haskell**: cbn Standard Auswertung
  -> Liste wird nur soweit ausgewertet, wie gebraucht.

- `take(2,from(5)): 5::6::Nil`

Lazy Evaluation
Analoges zu Haskell in Scala

- **Verwende Streams statt Listen**
- **Streams: Lazy Implementierung von Liste**
  - wird nur 1 Element erzeugt
  - Rest erst, wenn gebraucht
- **Dabei:**
  - `Nil  ->  Stream.Empty`
  - `::  ->  #::`

```
def from(x: Int) : Stream[Int] = if (x<=0) Stream.Empty
                                 else   x #:: from(x+1)
def take(x: Int, xs: Stream[Int]) : Stream[Int] = xs match {
  case Stream.Empty => Stream.Empty
  case y#:: ys      => if (x <= 0)  Stream.Empty
                       else y#::take(x - 1, ys)  }

take(2,from(5)).toList  -> List(5, 6)
```