Auswertung

Auswertung funktionaler Programme

- wie bei imperativen Sprachen, auch in FuPS verschiedene Auswertungsmöglichkeiten
- Bsp.:

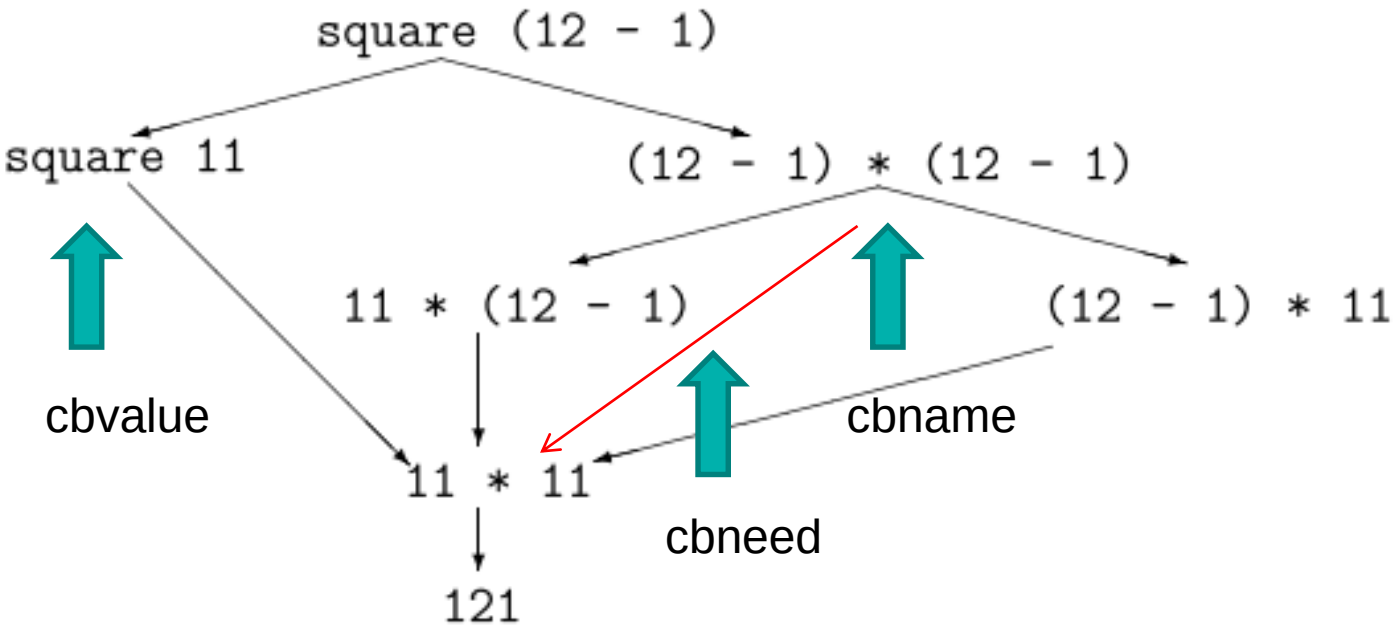
```
def square(x: Int) : Int = x * x
square(3)  ->          Int = 9
```

- spannend erst, wenn Parameter ein Ausdruck ist

```
square(12-1)  ->       Int = 121
```

- wann und wie oft wird `12-1` ausgewertet?
- hängt von Auswertungsstrategie ab
  - call-by-value
  - call-by-name
  - call-by-need

Überblick über Auswertungsstrategien



Call-by-value Auswertung (cbv)
(strikt, eager evaluation, leftmost-innermost)

- alle Parameterausdrücke von links nach rechts ausgewertet
  hier: `square(12-1) -> square(11)`
- Ersetzung des Aufrufs durch rechte Seite
  dabei: Variablen durch die Werte der obigen Auswertung ersetzt
  hier: `square(11) -> 11 * 11`
- entspricht der Java-Auswertung oder funktional: **ML**-Auswertung
- alle Parameter **genau einmal** ausgewertet,
  auch solche, die nicht gebraucht werden auf rechter Seite
- Endloszyklus in Parameter führt immer zu Endlosrechnung

Call-by-name Auswertung (cbn)
(nicht-strikt, lazy?? evaluation, leftmost-outermost)

- Ersetzung des Aufrufs durch rechte Seite
  dabei: Variablen durch die Ausdrücke in den Parametern ersetzt
  hier: `square(12-1) -> (12-1) * (12-1)`
- Teilausdrücke von links nach rechts ausgewertet
  hier: `(12-1) * (12-1) -> 11 * (12-1) -> 11 * 11`
- gibt es normalerweise nicht in imperativen Sprachen
  findet man aber als **Haskell**-Auswertung
- alle Parameter **erst** ausgewertet, wenn auf rechter Seite gebraucht,
  dann aber ggf. **mehrmals** bei mehrmaligem Auftreten
- Endloszyklus in Parameter führt nur zu Endlosrechnung,
  wenn Parameter auf rechter Seite erscheint.

- **manchmal als lazy bezeichnet**
- **aber Begriff unterschiedlich verwendet**
- **-> siehe spätere Vorlesung**

In Scala beides möglich
(cbv & cbn)

- Definiere endlos laufende Funktion

```
def endlos(i:Int) : Int = endlos(i+1)
```

- Definiere Funktion mit 2 Parametern, wo nur einer (hier x) auf rechter Seite vorkommt **cbv**:

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

- Funktionsaufruf von endlos im 2. Argument:

```
firstByValue(3+4, endlos(5))
    ->  endlos
```

- Definiere Funktion mit 2 Parametern, wo nur einer (hier x) auf rechter Seite vorkommt **cbn**:

```
def firstByName(x:Int, y : => Int) : Int = x
```

- Funktionsaufruf von endlos im 2. Argument:

```
firstByName(3+4, endlos(5))
    ->  3+4  ->  Int = 7
```

Vor (+)- / Nachteile (-) der beiden Strategien

Strategie	Parameter ausgewertet, der nicht gebraucht	Parameter mehrmals ausgewertet
Call-By-Value	ja -> -	nein -> +
Call-By-Name	nein -> +	ja -> -

- gesucht: Auswertungsstrategie, die beide Vorteile hat:
- **call-by-need** (Manchmal auch als: Lazy Evaluation bezeichnet)
- Werte Parameter nur aus, wenn auf rechter Seite gebraucht
- Speichert dann den Wert
- Mehrmals gebraucht, gespeicherten Wert wieder nutzen:

```
square(12-1) -> (12-1) * (12-1) -> 11 * 11
```

Implementierung von Call-by-need

- **Stack-Basiert**
  durch Bereitstellung von Speicherplätzen für Parameterwerte, falls Parameter auf rechter Seite vorkommt
- **Graph-Basiert**
  durch Setzen mehrerer Zeiger auf den gleichen Parameter (shared expressions)

```
square (12 - 1)

          *
->      ( )
      (12 - 1)
```

Indermarksche One-Funktion zum Compilertest

```
def one(n:Int) : Int =
if (n==0) 1 else one(n-1) * one(n-1)
```