Übung 3 zu KMPS — Lösung

Aufgabe 15:
Implementieren Sie eine Funktion `präorder`, die die Beschriftungen der Knoten eines Binärbaums berechnet und in der Scala-internen Listenstruktur zurückgibt. Der Binärbaum soll dabei gemäß der Darstellung in Aufgabe 9 definiert sein.
Sie dürfen die Funktion `append` aus Aufgabe 10 verwenden.

Überprüfen Sie die Funktion `präorder`, indem Sie möglichst sinnvolle „Testdaten" als Eingabewerte verwenden.

Lösung:
```
def präorder(xb: BinTree): List[Any] = {
  xb match {
    case EmptyTree => Nil
    case Node(e,l,r) => e::append(präorder(l),präorder(r))
  }
}

präorder(EmptyTree)
präorder(Node(1,Node(2,EmptyTree,EmptyTree),EmptyTree))
```

Aufgabe 16:
Überprüfen Sie, wie viele Blöcke auf den Funktionsstack gelegt werden, wenn Sie ggT aus Aufgabe 7 auswerten.

Lösung: Es wird bei jedem Funktionsaufruf jeweils ein Block auf den Stack gelegt. Da der Funktionsaufruf aber der letzte Befehl ist, kann der vorherige gelöscht werden, so dass immer nur ein Block gebraucht wird und der Platzaufwand konstant ist, falls der Compiler dieses Optimierungspotenzial nutzt.

Aufgabe 17:
Geben Sie eine tail-rekursive Lösung für die Fakultätsfunktion an und überprüfen Sie, wie viele Blöcke bei der Auswertung von `fak(5)` auf den Funktionsstack gelegt werden.
Hinweis: Gehen Sie von der iterativen Lösung aus und transformieren Sie diese in eine tail-rekursive.

Lösung:
```
def fakTR(n: Int) = fac(n,1)
def fac(n: Int, res: Int) : Int = if (n == 0) res else fac(n-1, res * n)
```

Bem.: ab Eingabe = 66 kommt immer 0 raus, da das Produkt wegen der Overflows 0 wird und dann alle weiteren Produkte auch.

Bei den Blöcken auf dem Stack verhält es sich analog zu Aufgabe 16, wobei hierbei noch der Block für `fakTR` unter dem Block für `fac` liegt.

Übung 3 zu KMPS — Lösung

Aufgabe 18:
Gegeben sei folgende Funktionsdefinition in Scala: `def e(i:Int) : Int = e(i)*e(i)`
Führen Sie die ersten 3 Schritte der Auswertung von `e(e(3+5))` aus gemäß der
a) Call-By-Value-Auswertungsstrategie
b) Call-By-Name-Auswertungsstrategie
c) Call-By-Need-Auswertungsstrategie
Unterstreichen Sie dabei den als nächstes auszuwertenden Ausdruck.
Unterscheidet sich call-by-name von call-by-need?

Wie oft wird der Ausdruck `3+5` bei der folgenden Funktionsdefinition `def e(i:Int) : Int = i * e(i)` ausgewertet, wenn Sie mit `e(3+5)` starten bei der
d) Call-By-Name-Auswertungsstrategie
e) Call-By-Need-Auswertungsstrategie
Illustrieren Sie Ihre Antwort, indem Sie die Rechnungen durchführen.

Lösung:
a) leftmost-innermost:
```
e(e(3+5)) =>  e(e(8)) => e(e(8)*e(8))
=>  e(e(8)* e(8) *e(8))
```

b) leftmost-outermost
```
e(e(3+5)) =>  e(e(3+5))*e(e(3+5))
=> e(e(3+5))*e(e(3+5)) *e(e(3+5))
=> e(e(3+5))*e(e(3+5)) *e(e(3+5)) *e(e(3+5))
```

c) call-by-need = call-by-name, da nie innen ausgewertet wird.

d) `e(3+5) => (3+5)* e(3+5) => 8*e(3+5) => 8 * (3+5) * e(3+5) => 8*8 *e(3+5) … mehrmals`
e) `e(3+5) => (3+5)* e(3+5) => 8*e(8) => 8 * 8 * e(8)  einmal`

Aufgabe 19:
Implementieren Sie jeweils eine first-order Funktion zur Summation der Zahlen, Quadrate und Zweier-Potenzen der Zahlen zwischen a und b.

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

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

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

Übung 3 zu KMPS **Lösung**

Aufgabe 20: Square Roots by Newton's Method
Implementieren Sie eine first-order Funktion `sqrt`, die die Quadratwurzel nach dem Newtonschen Verfahren berechnet.
Das Newtonsche Verfahren arbeitet zur Berechnung der Quadratwurzel von x wie folgt:
Man beginnt mit einem Startwert `y` (z.B. `y=1`).
Dann wird der bisher ermittelte Wert `y` verbessert, indem man als nächsten Wert den Mittelwert aus `y` und `x/y` verwendet.
Das Ganze soll an einem Beispiel erläutert werden:

y	x/y	(y + x/y)/2
1	2/1 = 2	1.5
1.5	2/1.5 = 1.3333	1.4167
1.4167	2/1.4167 = 1.4118	1.4142
1.4142 ...	...	

Lösung:

```
def sqrtIter(guess: Double, x: Double): Double =
      if (isGoodEnough(guess, x)) guess    //Abbruchkriterium
      else sqrtIter(improve(guess, x), x)  //nächster Schritt

def improve(guess: Double, x: Double) =  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =  abs(square(guess) - x) < 0.001

def sqrt(x: Double) = sqrtIter(1.0, x)
```