

Einführung in die Programmierung
WS 2018/19

Übungsblatt 6: Rekursion++

Besprechung: 03.12 - 07.12.2018

Aufgabe 6-1 *Syntaxdefinition: BNF-Satzform für arithmetische Ausdrücke*

Ein arithmetischer Ausdruck entsteht durch Verknüpfung von Zahlen mit arithmetischen Operatoren. Formal können arithmetische Ausdrücke folgendermaßen definiert werden:

- Jede Zahl ist ein arithmetischer Ausdruck.
 - Wenn A ein arithmetischer Ausdruck ist, dann ist auch (A) ein arithmetischer Ausdruck, d.h. ein korrekt geklammerter arithmetischer Ausdruck ist wiederum ein arithmetischer Ausdruck.
 - Wenn A_1 und A_2 arithmetische Ausdrücke sind, dann sind auch $A_1 + A_2$, $A_1 - A_2$, $A_1 * A_2$ sowie $A_1 \div A_2$ arithmetische Ausdrücke, d.h. auch die Summe, die Differenz, das Produkt und der Quotient von arithmetischen Ausdrücken sind arithmetische Ausdrücke.
- (a) Modellieren Sie in BNF-Satzform die syntaktische Variable $\langle \text{Ausdruck} \rangle$, welche alle Arten von arithmetischen Ausdrücken akzeptiert. Diese Variable ist also das sogenannte *Startsymbol*. Gerne dürfen Sie weitere Hilfs-Variablen einführen.
- (b) Geben Sie für den arithmetischen Ausdruck $((3 + 5) * 8) + 12$ die Ableitung an. Halten Sie sich dabei exakt an die von Ihnen definierten Syntaxregeln. Wenden Sie in jedem Ableitungsschritt nur genau eine Regel an.

Geben Sie die Lösung beider Teilaufgaben in einer Datei `ausdruck.txt` oder `ausdruck.pdf` ab.

Aufgabe 6-2 Zustände von Variablen

```
public class Zustand
{
    public static void main(String[] args) {
        final int DIV = 24;
        int variable;
        int counter = 1;
        {
            // *a*
            variable = counter++;
            int y = 12;
            variable += y;
            counter++;
            // *b*
        }
        final double d;
        {
            counter = 4;
            double a = 10.0;
            {
                d = a + ++counter;
                // *c*
            }
            counter = 3;
            while(counter > 0){
                counter--;
                a -= counter;
                // *d*
            }
        }
        variable = variable / DIV;
        // *e*
    }
}
```

Zu Aufgabe 6-2:

Geben Sie für jede der mit *a*, *b*, *c*, *d* und *e* gekennzeichneten Zeilen an, welche Variablen nach Ausführung der jeweiligen Zeile sichtbar sind („-“ = nicht sichtbar) und welchen Wert sie haben.

item
a									
b									
...									

Aufgabe 6-3 Mehr Rekursion

In dieser Aufgabe sollen Sie noch einmal Algorithmen mittels Rekursion in Java implementieren. Auch bei dieser Aufgabe dürfen Sie nur Basisoperationen verwenden, d.h. Sie müssen auch auf **Math** verzichten.

- Implementieren Sie eine rekursive Methode `long potenz(long x, int y)`, die zu zwei natürlichen Zahlen x, y den Wert x^y berechnet. Behandeln Sie auch eventuelle Spezialfälle.
- Implementieren Sie eine rekursive Methode `long spiegelzahl(long z)`, die zu einer gegebenen ganzen Zahl $z = z_1 z_2 \dots z_n$ ihre Spiegelzahl $z' = z_n z_{n-1} \dots z_1$ bestimmt, also die Zahl, die bei

umgedrehter Ziffernreihenfolge entsteht. Implementieren Sie die Funktion `int stellen(long z)`, welche gegeben einer Zahl `z`, die Anzahl an Stellen berechnet. Implementieren Sie außerdem eine Funktion `istPalindrom`, die zurückgibt, ob eine gegebene ganze Zahl vom Typ `long` eine Palindromzahl ist. Benutzen Sie hierzu KEINE anderen Klassen. Wir wollen die Problematik der führenden Nullen umgehen und definieren dazu, dass Vielfache von 10 niemals Palindromzahlen sein können. Bsp.: $1230 \neq \text{spiegelzahl}(0321) = 123$.

Aufgabe 6-4 *Noch mehr Rekursion*

Beim Backen von Keksen gilt, dass stets ein hoher Qualitätsanspruch bewahrt werden muss. Im Heim eines Informatikerprofessors geschieht dies durch eine ausgeklügelte Strategie, die man auch als Peer-Review kennt. Einzelne Kekse werden stichprobenartig ausgewählt und „getestet“. Es ist allerdings wichtig, dass bei stark wachsendem Keksinput die Testmenge nicht zu schnell mitwächst, um Unwohlsein zu vermeiden. Folgende Peer-Review-Strategie hat sich dabei entwickelt:

- Wenn kein Keks da ist, kann auch keiner probiert werden.
- Wenn es einen Keks gibt, sollte dieser auch getestet werden.
- Wenn es gerade viele Kekse gibt, teste 2. Die übrige Menge wird in zwei gleichgroße Haufen geteilt und nur ein Haufen wird weiter getestet.
- Wenn es ungerade viele Kekse gibt, dann testen wir einen und testen die übrige Menge wie zuvor.

Implementieren Sie eine Funktion `peer(int n)`, die zu einer gegebenen Menge Keksen die Anzahl der getesteten Kekse zurückgibt.

Aufgabe 6-5 *Rekursion vs. Iteration*

Sie kennen nun zwei wichtige Programmierparadigmen: Funktionale Programmierung und imperative Programmierung. Beide Paradigmen lösen gegebene Probleme auf einem anderen Abstraktionslevel, aber jeder Algorithmus lässt sich mehr oder weniger leicht in der anderen Weise umsetzen.

- (a) Die Summe der ersten n natürlichen Zahlen lässt sich sowohl imperativ als auch rekursiv berechnen. Implementieren Sie eine rekursive Funktion `summeRek(int n)`, die die Summe $1 + \dots + n$ berechnet.
- (b) Implementieren Sie außerdem `summeIt(int n)`, die das gleiche Ergebnis liefern sollte, aber ohne einen rekursiven Funktionsaufruf auskommt.
- (c) Auf dem letzten Übungsblatt haben Sie das Heron-Verfahren zum Wurzelziehen benutzt. Implementieren Sie nun eine iterative Version `wurzelIt(double x, int n)` basierend auf diesem Verfahren.
- (d) Fibonacci-Zahlen sind in den meisten Lehrbüchern als das Paradebeispiel für Rekursive Methoden angegeben. Implementieren Sie eine Funktion `fibIt(int n)`, die die n -te Fibonacci-Zahl

$$f_n = f_{n-1} + f_{n-2}$$

mit den Anfangswerten $f_1 = f_2 = 1$ iterativ berechnet.