4 Programmieren in Java III

4.1 Rekursive Funktionen und Prozeduren

Ein Unterprogramm kann sich selbst aufrufen. Dabei sind, in einer korrekt entworfenen Programmiersprache, die lokalen Variablen der verschiedenen Aufrufebenen strikt voneinander getrennt.

Um einen unendlichen Abstieg zu vermeiden, muss sorgfältig auf Abbruchbedingungen geachtet werden.

Beispiel Fakultät: $F(n) = n!$ (s. Übung)

$$F(n) = \begin{cases} 1 & \text{für } n = 0, \\ n \cdot F(n-1) & \text{für } n > 0. \end{cases}$$

Doppelfakultät: $D(n) = n!!$

$$D(n) = \begin{cases} 1 & \text{für } n = 0, 1, \\ n \cdot D(n-2) & \text{für } n > 1. \end{cases}$$

Anwendung Approximation von π durch das Wallis-Produkt

$$\begin{aligned} \frac{\pi}{2} &= \prod_{n=1}^{\infty} \left(\frac{2n}{2n-1} \cdot \frac{2n}{2n+1} \right) = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdots \\ &= \lim_{n \to \infty} \left(\frac{D(2n)}{D(2n-1)} \right)^2 \cdot \frac{1}{2n+1} \end{aligned}$$

4.2 Größter gemeinsamer Teiler – euklidischer Algorithmus

Definition: Seien $a, b \in \mathbb{Z}$ ganze Zahlen. Die Zahl $d = \mathrm{ggT}(a, b)$ ist dann der größte gemeinsame Teiler von a und b, wenn gilt

i) d ist Teiler, $d|a$ und $d|b$, und

ii) es gibt keinen größeren Teiler, jeder weitere gemeinsamen Teiler teilt auch d, d.h. für $c \in \mathbb{Z}$ mit $c|a$ und $c|b$ gilt schon $c|d$.

Lemma: Es gilt $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-b,b)$.

Jeder gemeinsame Teiler von a und b teilt auch $a-b$, und jeder gemeinsame Teiler von $a-b$ und b teilt auch $(a-b)+b=a$.

Folgerung: Es gilt $\mathrm{ggT}(a,b) = \mathrm{ggT}(a-q\cdot b,b)$, und es gibt Faktoren $q \in \mathbb{Z}$ mit

$$-|b| < a - q\cdot b < |b|.$$

Einfache Rekursion zur Bestimmung des größten gemeinsamen Teilers

$$\mathrm{ggT}(a,b) = \begin{cases} \mathrm{ggT}(|a|,|b|), & \\ a & \text{wenn } b=0, \\ \mathrm{ggT}(b,a) & \text{wenn } b>a, \\ \mathrm{ggT}(a-b,b) & \text{sonst .} \end{cases}$$

```
  static int ggT(int a, int b){
   System.out.println("ggT(\t"+a+",\t"+b+"\t)");
   if(a<0) a=-a;
   if(b<0) b=-b;
   if(b==0) return a;
   if(b>a) return ggT(b,a);
   return ggT(a-b,b);
 }
```

Produziert die Liste von Aufrufen:

```
ggT(    37,     23      )
ggT(    14,     23      )
ggT(    23,     14      )
ggT(    9,      14      )
ggT(    14,     9       )
ggT(    5,      9       )
ggT(    9,      5       )
ggT(    4,      5       )
```

```
ggT(    5,      4       )
ggT(    1,      4       )
ggT(    4,      1       )
ggT(    3,      1       )
ggT(    2,      1       )
ggT(    1,      1       )
ggT(    0,      1       )
ggT(    1,      0       )
```

Es gibt viele Wiederholungen an der zweiten Stelle, d.h. dasselbe b wird mehrfach abgezogen.

Durch Benutzung von Division mit Rest

```
  static int ggT(int a, int b){
   System.out.println("ggT(\t"+a+",\t"+b+"\t)");
   if(a<0) a=-a;  if(b<0) b=-b;
   if(b==0) return a;
   int q = a/b;
   return ggT(b,a-q*b);
 }
```

ergibt sich die kondensierte Aufrufliste

```
ggT(    37,     23      )
ggT(    23,     14      )
ggT(    14,     9       )
ggT(    9,      5       )
ggT(    5,      4       )
ggT(    4,      1       )
ggT(    1,      0       )
```

4.3 Lindenmayer–Systeme

nach Aristide Lindenmayer (1925–1989)

Ein L-System ist ein Termersetzungssystem. Es besteht aus einem Alphabet oder Zeichenvorrat Z, der wiederum in Variable V und Konstante K aufgeteilt ist. Aus den Zeichen können beliebige Worte gebildet werden. Ein solches Wort wird als Startwort S vermerkt. Die Dynamik eines L-Systems wird durch die Ersetzungsregeln R definiert. Dabei wird jeder Variablen ein Wort zugewiesen.

Die Entwicklung eines L-Systems erfolgt in Generationswechseln. Jede Generation ist ein Wort. Die erste Generation ist das Startwort S. Bei einem Generationswechsel werden im aktuellen Wort alle Variablen durch die Zeichenkette ihrer Ersetzungsregel ersetzt. Konstante Zeichen werden einfach kopiert.

Beispiel: Wachstum von Fadenalgen.

```
V: A,B,   K: (leer), S: A
R: A -> AB, B -> A

Die ersten Generationen
1: A                          |  1 Zeichen
2: AB                         |  2 Zeichen
3: AB A                       |  3 Zeichen
4: AB A AB                    |  5 Zeichen
5: AB A AB AB A               |  8 Zeichen
6: AB A AB AB A AB A AB       | 13 Zeichen
```

Die Zeichenanzahl entwickelt sich entsprechend der Fibonacci-Folge. A kann als erwachsene Alge, B als Knospe bzw. die Kombination AB als sich teilende Zelle interpretiert werden.

```
V: F, K: +,-, S: F
R: F->+F--F+

Die ersten Generationen
1: F
2: +F--F+
3: +[+F--F+]--[+F--F+]+
4: ++[+F--F+]--[+F--F+]+--+[+F--F+]--[+F--F+]++
```

Das Wort einer Generation kann als Folge von Steuerungsanweisungen für einen Roboter interpretiert werden. Z.B. eine Zeichenturtle (Stiftplotter). Hier ergibt
F: einen Schritt vorwärts (forward),
+: Drehen nach links (left) und
-: Drehen nach rechts (right), jeweils um 45^o,
ein interessantes Muster. Schrittweite und Winkel kann variiert werden.

Dabei bietet es sich an, die Variablen–Zeichen als rekursive Prozeduren zu realisieren. Auf der Tiefe des Steuerwortes führen sie dann den Zeichenbefehl aus, in geringeren Tiefen wird das Ersetzungswort aufgerufen.

```
  /* + */ static void P(){ ari.left(angle); }
  /* - */ static void M(){ ari.right(angle); }

  /* F */ static void F(){ level++;
    if(level==depth){ ari.forward(step); }
    else { P();F();M();M();F();P(); }
    level--; }
```

```
V: F,  K: +, −,  S: F
R: F -> F+F--F+F
  mit Winkel 60° ergibt die Koch-Kurve

V: F,  K: +, −,  S: F
R  : F -> F+F−F−F+F
   mit Winkel 90° ergibt eine Variante der Koch-Kurve

V: R, L,  K: +, -,  S: R
R: R -> +R--L+, L -> -R++L-
  mit Winkel 45° ergibt das Drachenfraktal.
  Alternativ dazu mit kürzeren Regeln

V: R, L,  K: F, +, −,  S: FR
R: R -> R+LF, L -> FR-L
  Winkel 90°, R und L werden nicht als Zeichenbefehl interpretiert.
```

java/LSystem_Dragon.java

```
import ch.aplu.turtle.Turtle;

public class LSystem_Dragon /* Aristide Lindenmayer */
{
  static Turtle ari;
  static double size, step, angle;
  static int depth, level;

  static void init(int dep){
    ari = new Turtle();
    size = 120;
    angle = 90;
```

```
    depth = dep; level=0;
    step = 2.4*size/Math.pow(2.0,depth/2.0);
    if(depth>4) ari.hideTurtle();
    ari.setPos(-0.8*size,0);
    ari.right(45*(depth+1));
  }

  static void P(){ ari.left(angle); }
  static void M(){ ari.right(angle); }
  static void F(){ ari.forward(step); }

static void R(){
  level++;
  if(level<depth){ /*R+LF*/ R();P();L();F(); }
  level--;
}

static void L(){
  level++;
  if(level<depth){ /*FR-L*/ F();R();M();L(); }
  level--;
}


  static void run(){ F();R(); }

  public static void main(String[] args)
  {

    init((args.length>0)?Integer.parseInt(args[0]):4);
    run();
  }
}
```