Einführung in rekursives Programmieren

mit numerischen und graphischen Beispielen (fraktale Kurven)

von Werner Rockenbach, Simmern

Nikolaus Wirth definiert die Rekursion wie folgt:
Ein Objekt heißt rekursiv, wenn es sich selbst als Teil enthält oder mithilfe von sich selbst definiert ist.
Rekursionen spielen in den Programmiersprachen eine wichtige Rolle. Sie sind elegant und scheinbar leicht zu durchschauen machen aber große Schwierigkeiten bei der Analyse bzw. bei der Umsetzung in ein lauffähiges Programm.
Rekursion ist Wiederholung durch Schachtelung, die Vergehensweise ist „top-down“, d.h. vom allgemeinen zum besonderen. Die Iteration ist Wiederholung durch Aneinanderreihung, die Vorgehensweise ist „bottom-up“, d.h. vom besonderen zum allgemeinen. Jede rekursive Prozedur kann in eine gleichwertige Iteration umgewandelt werden und umgekehrt.
Damit die Rekursion nicht zu einer Endlos-Wiederholung wird, bedarf es eines Rekursionsanfangs. Der Rekursionsanfang entspricht der Abbruchbedingung bei Schleifen (Iteration).

Mathematische Rekursionen

1. Fakultät

a) Iteration

Die Fakultät ist wie folgt definiert: $n! = 1 \cdot 2 \cdot 3 \cdot ..... \cdot n$

Beispiel: $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$

```
function Fakultaet(n : integer) : integer;
var     k, fak: integer;
begin
        fak := 1;
        for k : = 1 to n do fak := fak * k;
        result := fak
end;
```

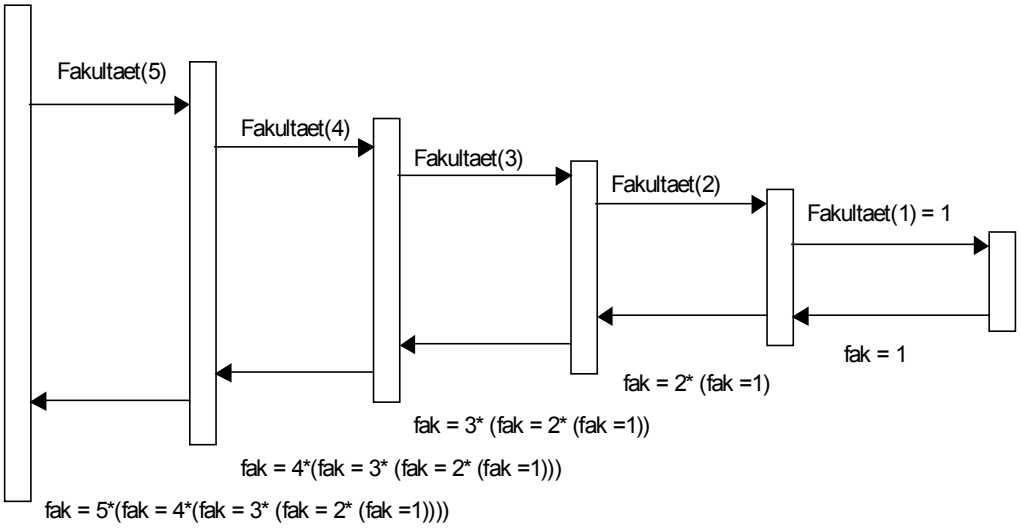
b) Rekursion

Die Fakultät ist wie folgt definiert:

$$n! = \begin{cases} 1 & \text{falls } n = 1 \\ n * (n-1)\,! & \text{sonst} \end{cases}$$

Beispiel: $5! = 5 \cdot 4! = 5 \cdot 4 \cdot 3! = 5 \cdot 4 \cdot 3 \cdot 2! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Das UML-Sequenzdiagramm verdeutlicht nochmals die Rekursion.




```
function Fakultaet(n : integer) : integer;
var fak : integer;
begin
        if n = 1 then fak := 1
                else fak := n*Fakultaet(n-1);
        result := fak
end;
```

2. Summe der ersten natürlichen Zahlen

a) Iteration

Die Summe ist wie folgt definiert: Summe(n) = 1 + 2 + 3 + ... + n

```
function Summe(n : integer) : integer;
var     k, s: integer;
begin
        s := 0;
        for k : = 1 to n do s := s  + k;
        result := s
end;
```

b) Rekursion

Rekursive Definition:

$$summe(n) = \begin{cases} 0 & \text{falls } n = 0 \\ n + summe(n-1) & \text{sonst} \end{cases}$$

```
function Summe(n : integer) : integer;
var s : integer;
begin
        if n = 0 then s := 0
                else s := n + summe(n-1);
        result := s
end;
```

3. Binomialkoeffizient

a) Iteration

Definition: $\binom{n}{k} = \frac{n!}{(n-k)!\,k!}$ für n, k ε N und n ≥ k

Bei der Programmierung kann auf die Fakultät Funktionen zurückgegriffen werden.

```
function Binomialkoeffizient(n, k : integer) : integer;
begin
        result := fakultaet(n) div (fakultaet(n-k) * fakultaet(k))
end;
```

b) Rekursion

(1) Definition $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

Diese Definition ergibt sich unmittelbar aus dem Pascalschen Dreieck.

```
function Binomialkoeffizient(n, k : integer) : integer;
begin
        if  (k = 0) or  (k = n)
        then result := 1
        else result := Binomialkoeffizient(n - 1, k - 1) + Binomialkoeffizient(n - 1, k = 1)
end;
```

(2) Definition $\binom{n}{k} = \frac{n}{k} \cdot \binom{n-1}{k-1}$

```
function Binomialkoeffizient(n, k : integer) : real;
begin
        if  (k = 0) or  (k = n)
        then result := 1
        else result := n/k * Binomialkoeffizient(n - 1, k - 1)
end;
```

4. Weitere Rekursionen

4.1. Fibonacci-Folge

Ein zur Zeit 0 geborenes Kaninchenpaar erzeugt von zweiten Monat seiner Existenz an in jedem Monat ein weiteres Kaninchenpaar, und auch die Nachkommen befolgen das selbe Vermehrungsgesetz. Wie viele Kaninchenpaare sind nach n Monaten vorhanden?

Monat	0	1	2	3	4	5	6	7	8
Anzahl der Paare	1	1	2	3	5	8	13	21	34

Bildungsgesetz:

$$\text{fibonacci}(n) = \begin{cases} 1 & \text{für } n \leq 1 \\ \text{fibonacci}(n-1) + \text{fibonacci}(n-2) & \text{sonst} \end{cases}$$

```
function fibonacci(n : integer) : integer;
begin
        if n <= 1 then result := 1
                  else  result := fibonacci(n – 1) + fibonacci(n-2)
end;
```

4.2. Größter gemeinsamer Teiler ggt(a,b)

Definition:

$$\text{ggt}(a,b) = \begin{cases} b & \text{wenn a mod b = 0} \\ \text{ggt}(b, a \bmod b) & \text{sonst} \end{cases}$$

Beispiel: ggt(58, 18) = ggt(18, 4) = ggt(4, 2) = 2

```
function ggt(a, b : integer) : integer;
begin
        if a mod b = 0 then result := b
                       else result := ggt(b, a mod b)
end;
```

4.3. Ganzzahliger Rest

Definition:

$$\text{Rest}(a, b) = \begin{cases} a & \text{wenn a < b} \\ \text{Rest}(a-b, b) & \text{sonst} \end{cases}$$

Beispiel: Rest(58, 16) = Rest(42, 16) = Rest(26, 16) = Rest(10, 16) = 10

```
function rest(a, b : integer) : integer;
begin
        if a < b then a
                 else result  := rest(a-b, b)
end;
```

4.4. Quadratzahlen

Definition:

$$n^2 = \begin{cases} 1 & \text{für } n = 1 \\ (n-1)^2 + 2n - 1 & \text{sonst} \end{cases}$$

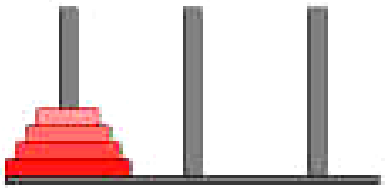
```
function quadratzahl(n : integer) : integer;
begin
        if n = 1 then result :=1
                else result := quadratzahl(n-1) + 2*n –1
end;
```

5. Türme von Hanoi

Nach einer Legende standen einmal drei goldene Säulen vor einem Tempel in Hanoi. Auf einer der Säulen befanden sich 100 Scheiben, jedes mal eine kleinere auf einer größeren Scheibe. Ein alter Mönch bekam die Aufgabe, den Scheibenturm von Säule 1 nach Säule 2 unter folgenden Bedingungen zu transportieren:

(1) Es darf jeweils nur die oberste Scheibe von einem Turm genommen werden.
(2) Es darf niemals eine größere Scheibe auf einer kleinern liegen.

Wenn der Mönch die Arbeit erledigt habe, so berichtet die Legende, dann werde das Ende der Welt kommen.

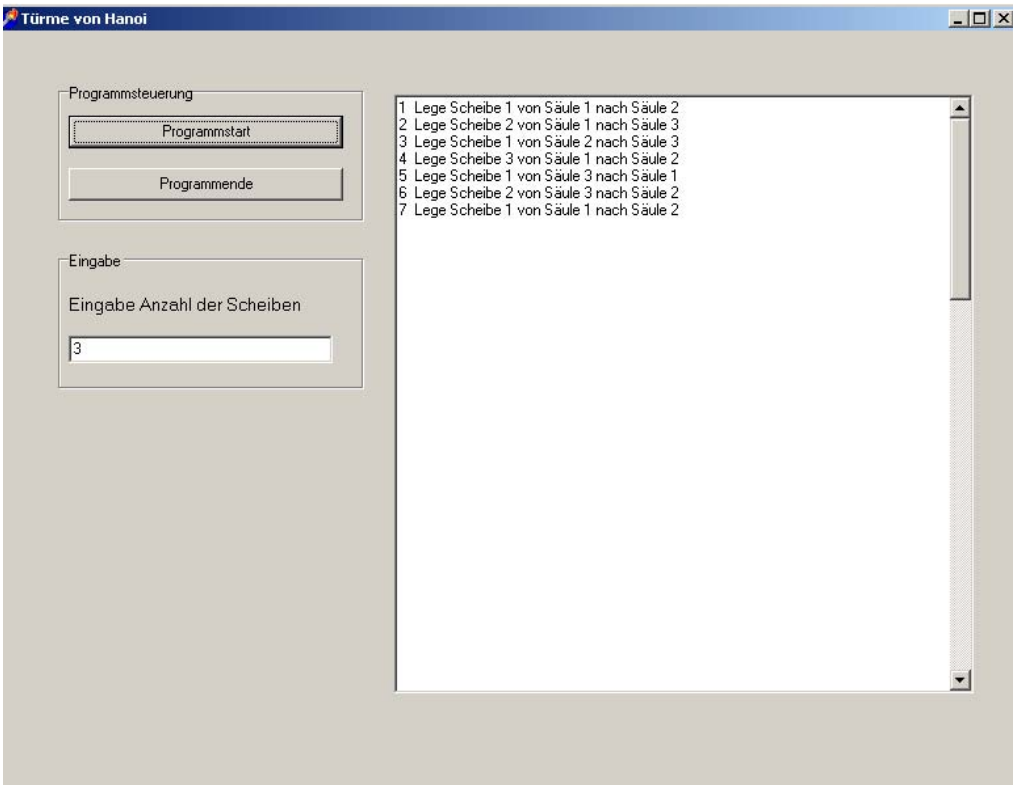


Rekursive Lösung:

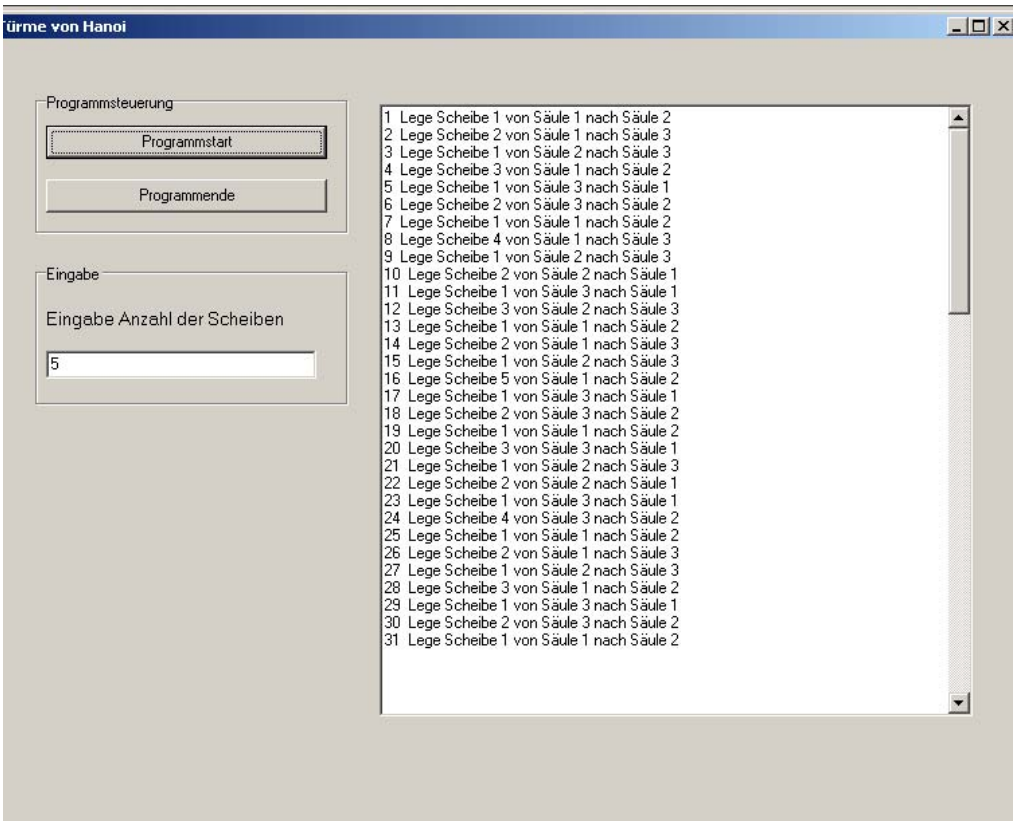
(1) Transportiere n-1 Scheiben von Säule S1 nach Säule S3
(2) Transportiere die n-te Scheibe von Säule S1 nach Säule S2.
(3) Transportiere n-1 Scheiben von Säule S3 nach Säule S2

```
procedure Turm(n, S1, S2, S3 : integer);
begin
        if n > 0 then begin
                        Turm(n-1, S1, S3, S2);
                        Schreibe “Lege Scheibe”, n, “von Säule”, S1, “nach Säule”, S2
                        Turm(n-1, S3, S2, S1)
                      end
end;
```

Programmdurchlauf für n = 3



Programmdurchlauf für n = 5



6. Primitiv rekursive Funktionen

Definition: Eine Funktion f heißt primitiv rekursiv, wenn sie entweder eine der elementaren Funktionen (Nullfunktion, Nachfolgerfunktion, Projektionsfunktion) ist oder aus diesen durch Rekursion oder Verkettung erhalten werden kann.

a) Nullfunktion N(x) = 0

```
function null(n : integer) : integer;
begin
        result := 0
end;
```

b) Nachfolgerfunktion N(x) = x + 1

```
function nachfolger(n : integer) : integer;
begin
        result := n + 1
end;
```

c) Addition $f_1(x, y) = x + y$

Die Addition kann rekursiv mit Hilfe der Nachfolgerfunktion definiert werden.

$$\text{Addition}(x,y) = \begin{cases} x & \text{für } y = 0 \\ \text{Nachfolger}(\text{Addition}(x,y-1)) & \end{cases}$$

```
Additon(4,3)  = Nachfolger(Addition(4,2))
              = Nachfolger(Nachfolger(Addition(4,1)))
              = Nachfolger(Nachfolger(Nachfolger(Addition(4,0))))
              = Nachfolger(Nachfolger(Nachfolger(4)))
              = Nachfolger(Nachfolger(5))
              = Nachfolger(6)
              = 7
```

```
function f1(x, y : integer) : integer ;
begin
        if y = 0 then result := x
                 else result := nachfolger(f1(x, y-1))
end;
```

d) Multiplikation $f_2(x,y) = x * y$

Die Multiplikation kann rekursiv mit Hilfe der Addition definiert werden.

$$\text{Multiplikation}(x, y) = \begin{cases} 0 & \text{für } y = 0 \\ \text{Addition}(x, \text{Multiplikation}(x, y-1)) & \text{sonst} \end{cases}$$

```
Multiplikation(3,3)   =   Addition(3, Multiplikation(3,2))
                      =   Addition(3, Addition(3, Multiplikation(3,1)))
                      =   Addition(3, Addition(3, Addition(3, Multiplikation(3,0))))
                      =   Addition(3, Addition(3, Addition(3, 0)))
                      =   Addition(3, Addition(3, 3))
                      =   Addition(3, 6)
                      =   9
```

```
function f2(x, y : integer) : integer;
begin
        if y = 0 then result :=0
                 else  result := f1(x, f2(x, y – 1))
end;
```

e) Potenz $f_3(x, y) = x^y$

Die Potenz kann mit Hilfe der Multiplikation definiert werden.

$$\text{Potenz}(x, y) = \begin{cases} 1 \text{ für } y = 0 \\ \text{Multiplikation}(x, \text{Potenz}(x, y - 1)) \text{ sonst} \end{cases}$$

```
Potenz(4,3)   =   Multiplikation(4, Potenz(4, 2))
              =   Multiplikation(4, Multiplikation( 4, Potenz(4, 1)))
              =   Multiplikation(4,  Multiplikation(4 , Multiplikation(4 , Potenz(4, 0))))
              =   Multiplikation(4,  Multiplikation(4 , Multiplikation(4 , 1)))
              =   Multiplikation(4,  Multiplikation(4 , 4))
              =   Multiplikation(4,  16)
              =   64
```

```
function f3(x, y : integer) : integer;
begin
        if y = 0 then result := 1
                 else result := f2(x, f3(x, y – 1))
end;
```

7. Rekursive Funktionen

a) Ackermannfunktion

Die Ackermannfunktion ist eine, 1926 von Wilhelm Ackermann erfundene, extrem schnell wachsende Funktion, mit deren Hilfe in der theoretischen Informatik Grenzen von Computer- und Berechnungsmodellen aufgezeigt werden können.
1926 vermutete David Hilbert, dass jede berechenbare Funktion primitiv-rekursiv ist. Vereinfacht gesprochen bedeutet dies, dass jede Funktion, die durch einen Computer berechnet werden kann, sich aus einigen wenigen sehr einfachen Regeln zusammensetzen lässt und die Dauer der Berechnung sich im Vorfeld abschätzen lässt. Dies trifft auf nahezu alle in der Praxis vorkommenden Funktionen zu. (Die Ausnahmen sind vorwiegend im Bereich des Compilerbaus zu finden.) Im gleichen Jahr konstruierte Ackermann eine Funktion, die diese Vermutung widerlegt, und veröffentlichte sie 1928. Diese Funktion wird heute ihm zu Ehren **Ackermannfunktion** genannt. Die Ackermannfunktion kann also von einem Computer in endlicher Zeit ausgewertet werden, ist aber nicht primitiv-rekursiv.

Definition: $Ackermann(x,y) = \begin{cases} y + 1 & \text{für } x = 0 \\ Ackermann(x-1, 1) & \text{für } y = 0 \\ Ackermann(x-1, Ackermann(x,y-1)) & \text{sonst} \end{cases}$

```
function ackermann(x, y : integer) : integer;
begin
        if  x = 0 then result := y + 1
                if y = 0 then result := ackermann(x – 1, 1)
                        else  result := ackermann(x – 1, ackermann(x, y – 1));
end;
```

Beispiel: Ackermann(2,3) = ?

Ackermann(1,0) = Ackermann(0,1) = 2

Ackermann(1,1) = Ackermann(0, Ackermann(1,0))
= Ackermann(0,2)
= 3

Ackermann(1,2) = Ackermann(0, Ackermann(1,1))
= Ackermann(0,3)
= 4

Ackermann(2,0) = Ackermann(1,1)
= 3

Ackermann(1,3) = Ackermann(0, Ackermann(1,2))
= Ackermann(0,4)
= 5

Ackermann(2,1) = Ackermann(1, Ackermann(2,0))
= Ackermann(1,3)
= Ackermann(0,Ackermann(1,2))
= Ackermann(0,4)
= 5

Ackermann(2,3) = Ackermann(1,Ackermann(2,2))
= Ackermann(1, Ackermann(1,Ackermann(2,1)))
= Ackermann(1, Ackermann(1,5))
= Ackermann(1, Ackermann(0, Ackermann(1,4)))
= Ackermann(1, Ackermann(0, Ackermann(0, Ackermann(1,3))))
= Ackermann(1, Ackermann(0, Ackermann(0,5)))
= Ackermann(1, Ackermann(0,6))
= Ackermann(1,7)
= Ackermann(0, Ackermann(1,6))
= Ackermann(0, Ackermann(0, Ackermann(1,5)))
= Ackermann(0, Ackermann(0, Ackermann(0, Ackermann(1,4))))
= Ackermann(0, Ackermann(0, Ackermann(0, Ackermann(0, Ackermann(1,3)))))
= Ackermann(0, Ackermann(0, Ackermann(0, Ackermann(0,5))))
= Ackermann(0, Ackermann(0, Ackermann(0, 6)))
= Ackermann(0, Ackermann(0,7))
= Ackermann(0,8)
= **9**

Ackermannfunktion

	x = 0	x = 1	x = 2	x = 3	x = 4
y = 0	1	2	3	5	13
y = 1	2	3	5	13	
y = 2	3	4	7	29	
y = 3	4	5	9	61	
y = 4	5	6	11	125	

Bei größeren Zahlen reicht der Stack nicht aus und das System hängt sich auf. Mit x = 4 und y =1 meldet das System Stack-Überlauf.

b) Ulam-Funktion

Eine weitere rekursive Funktion ist die ULAM-Funktion.
Stanislaw Marcin Ulam (*1909 , +1984) polonischer Mathematiker. Bekannt durch die „Monte-Carlo-Methode“ in der Statistik und durch verschiedene Zahlenspielereien.

$$\text{Definition:} \quad \text{Ulam}(x) = \begin{cases} 1 & \text{für } x = 1 \\ \text{Ulam}(x/2) & \text{für } x > 1 \text{ und } x \text{ gerade} \\ \text{Ulam}(3*x+1) & \text{für } x > 1 \text{ und } x \text{ ungerade} \end{cases}$$

```
function Ulam(x : integer) : integer;
begin
        if x = 1 then result := 1
                 else if (x > 1) and (x mod 2 = 0)
                      then result := Ulam(x div 2)
                      else if (x > 1) and (x mod 2 <> 0)
                           then result := Ulam(3*x + 1)
end;
```

c) Hofstadterfunktion

Eine weitere recursive Funktion, die nach einem bekannten Mathematiker benannt wurde, ist die Hofstadterfunktion.
Douglas R. Hofstadter (* 1945), Mathematiker und Physiker. Nobelpreisträger für Physik.

$$\text{Definition:} \quad \text{Hof}(n) = \begin{cases} 1 & \text{für } n = 1 \text{ oder } n = 2 \\ \text{Hof}(n - \text{Hof}(n-1)) + \text{Hof}(n - \text{Hof}(n-2)) & \text{für } n > 2 \end{cases}$$

```
function Hof(n : integer) : integer;
begin
        if (n = 1 ) or (n = 2) then result :=1
                                else result := Hof(n – Hof(n-1)) + Hof(n – Hof(n-2))
end;
```

8. Rekursive Grafiken

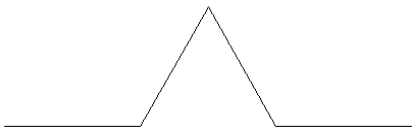
8.1. Koch-Kurve

Die Kochkurve gehört zur Gruppe der Monsterkurven. Monsterkurven sind Kurven die eine Fläche mit endlichem Inhalt und einem Umfang unendlicher Länge beinhalten. Mit Hilfe der Turtle-Grafik lassen sich solche selbstähnliche Kurven erzeugen.

Ausgang Strecke der Länge 1.

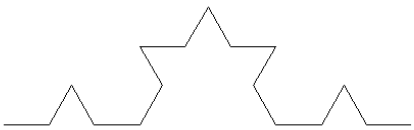
Nummer 1

Die Figur der Nummer 1 nennt man Initiator. Die Strecke wird nun in drei gleiche Teile geteilt und das Mittelstück durch zwei Seiten eines gleichseitigen Dreiecks ersetzt.



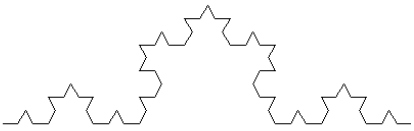
Nummer 2

Die Figur der Nummer 2 ist der sogenannte Generator, d.h. er erzeugt die Kurve. Im nächsten Schritt wird jede der vier Strecken des Generators durch den Generator selbst, allerdings auf ein Drittel seiner Größe reduziert, ersetzt.



Nummer 3

Im nächsten Schritt wird wiederum jede Strecke durch den erneut um ein Drittel verkürzten Generator ersetzt. Und so geht es weiter.



Nummer 4

Initiator (entspricht dem Rekursionsanfang)

Zeichne1(Länge : real)
Vorwärts(Länge)

Generator (entspricht dem Rekursionsschritt)

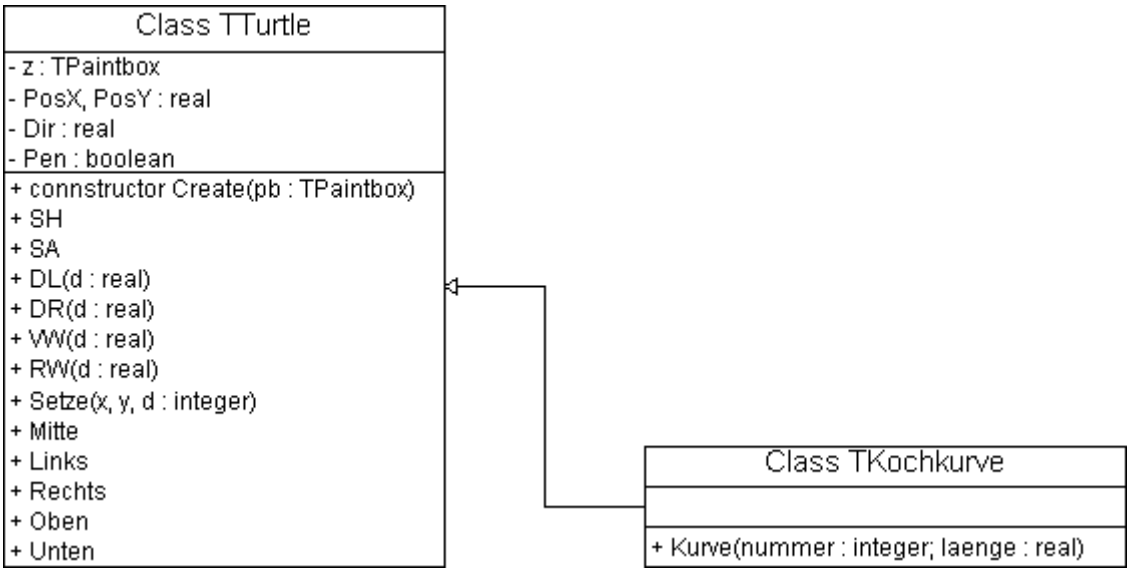
Zeichne2(Länge : real)
Zeichne1(Länge/3)
Drehe links (60)
Zeichne1(Länge/3)
Drehe rechts (120)
Zeichne1(Länge/3)
Drehe links(60)
Zeichne1(Länge/3)

Um das allgemeine Bildungsgesetz der rekursiven Prozedur auszudrücken, wird an Stelle der Anhängseln 1, 2,... ein Parameter namens Nummer gesetzt.

Allgemeine Methode

Zeichne(Nummer : integer; Länge : real)	
Nummer = 1 ja	nein
Vorwärts(Länge)	Zeichne(Nummer-1, Länge/3)
	Drehe links (60)
	Zeichne(Nummer-1, Länge/3)
	Drehe rechts (120)
	Zeichne(Nummer-1, Länge/3)
	Drehe links (60)
	Zeichne(Nummer-1, Länge/3)

UML-Klassendiagramm (Kochkurve)



Die Klasse TTurtle steht in der Programmierumgebung von Delphi nicht zur Verfügung. Sie kann selbständig als Unit erstellt oder aber im Unterricht bereitgestellt werden.

PosX	x-Koordinate Turtle
PosY	y-Koordinate Turtle
Dir	Richtung Turtle in Grad
Pen	Stift

SH	Methode Stifthoch
SA	Methode Stiftab
DL(d)	Methode Drehe links um Betrag d
DR(d)	Methode Drehe rechts um Betrag d
VW(d)	Methode Vorwärts um Betrag d
RW(d)	Methode Rückwärts um Betrag d
Setze(x,y,d)	Methode Turtle auf Position x, y mit Richtung d setzen
Mitte	Methode Turtle in die Mitte setzen, Richtung nach oben
Links	Methode Turtle mittig an den linken Bildschirmrand setzen
Rechts	Methode Turtle mittig an den rechten Bildschirmrand setzen
Oben	Methode Turtle mittig an den oberen Bildschirmrand setzen
Unten	Methode Turtle mittig an den unteren Bildschirmrand setzen

Unit U_Turtle

```
UNIT U_Turtle;

interface

uses ExtCtrls, Graphics;

type
    TTurtle= class
            private
                 z: TPaintBox;
                 PosX, PosY: real;
                 Dir: real;
                 Pen: boolean;
            public
                 constructor Create(pb: TPaintBox); { Initialisiert die Turtle mit der zugehörigen Zeichenfläche }
                 procedure SH;                  { Stift hoch }
                 procedure SA;                  { Stift ab }
                 procedure DL(d: real);           { Drehe links }
                 procedure DR(d: real);           { Drehe rechts }
                 procedure VW(d: real);           { Vorwärts }
                 procedure RW(d: real);           { Rückwärts }
                 procedure Setze(x, y, d: integer); { Turtle auf eine Position setzen }
                 procedure Mitte;               { Turtle in die Mitte setzen }
                 procedure Links;               { Turtle mittig an den linken Bildschirmrand setzen }
                 procedure Rechts;              { Turtle mittig an den rechten Bildschirmrand setzen }
                 procedure Oben;                { Turtle mittig an den oberen Bildschirmrand setzen }
                 procedure Unten;               { Turtle mittig an den unteren Bildschirmrand setzen }
          end;

implementation
{ ---------------------- mathematische Hilfsfunktionen --------------------------- }
function arcsin(x: real): real;
begin
   arcsin:= arctan(x/sqrt(1-sqr(x)))
end;

function arccos(x: real): real;
begin
  arccos:= arctan(sqrt(1-sqr(x))/x);
end;

{ ---------------------- Turtle-Methoden --------------------------- }

procedure TTurtle.Mitte;
begin
  PosX:= z.Width div 2;
  PosY:= z.Height div 2;
  Dir:= 90;
end;

procedure TTurtle.Links;
begin
  PosX:= 0;
  PosY:= z.Height div 2;
  Dir:= 0;
end;
```

```
procedure TTurtle.Rechts;
begin
  PosX:= z.Width;
  PosY:= z.Height div 2;
  Dir:= 180;
end;

procedure TTurtle.Oben;
begin
  PosX:= z.Width div 2;
  PosY:= 0;
  Dir:= 270;
end;

procedure TTurtle.Unten;
begin
  PosX:= z.Width div 2;
  PosY:= z.Height;
  Dir:= 90;
end;

Constructor TTurtle.Create(pb: TPaintBox);
begin
  z:= pb;
  PosX:= 0;
  PosY:= 0;
  Dir:= 0;
  Pen:= true;
  Mitte;
  z.Refresh;
end;

procedure TTurtle.SH;
begin
  Pen:= false;
end;

procedure TTurtle.SA;
begin
  Pen:= true;
end;

procedure TTurtle.DL(d: real);
begin
  Dir:= Dir+d;
  if Dir>360 then while Dir>360 do Dir:= Dir-360
             else while Dir<0 do Dir:= Dir+360;
end;

procedure TTurtle.DR(d: real);
begin
  DL(-d);
end;

procedure TTurtle.VW(d: real);
begin
  z.Canvas.MoveTo(trunc(PosX),trunc(PosY));
  PosX:= PosX+cos(Dir/180*pi)*d;
  PosY:= PosY-sin(Dir/180*pi)*d;
  if Pen then
    z.Canvas.LineTo(trunc(PosX),trunc(PosY));
end;
```

```
procedure TTurtle.RW(d: real);
begin
  VW(-d);
end;

procedure TTurtle.Setze(x, y, d: integer);
begin
  PosX:= x;
  PosY:= y;
  Dir:= d;
end;

end.
```

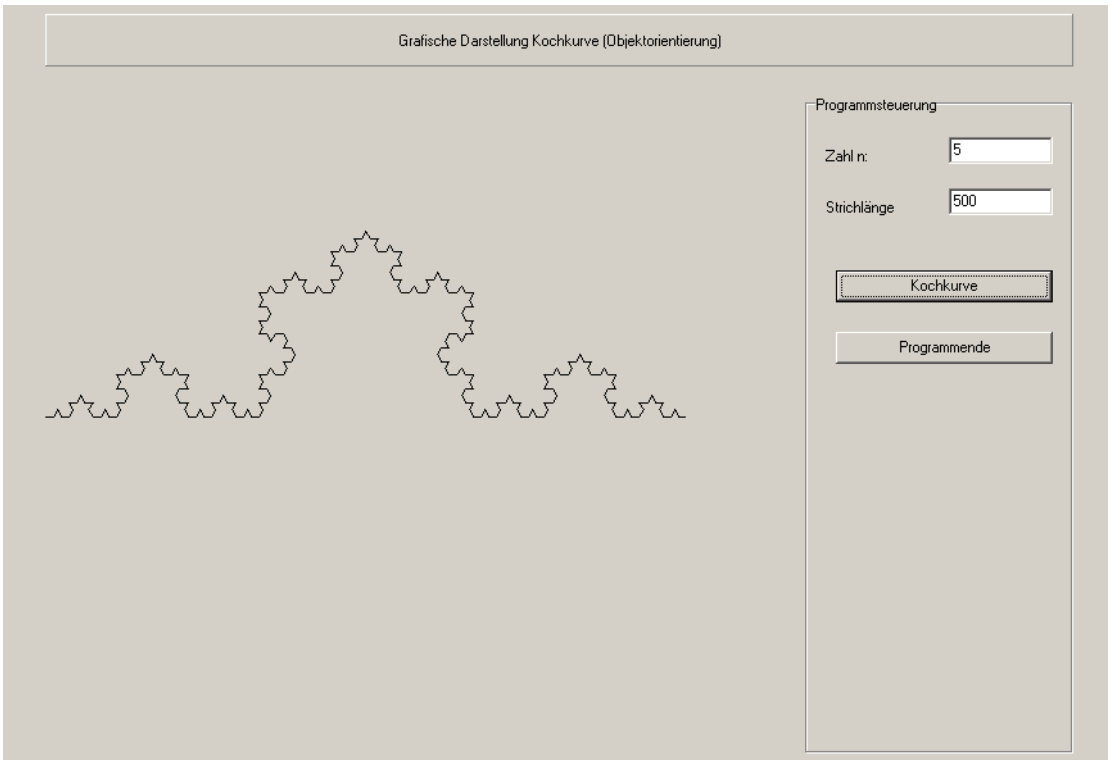
Unit mKochkurve

```
UNIT mKochkurve;

interface

uses ExtCtrls, Graphics, U_Turtle;


type
   TKochkurve = CLASS(TTurtle)
              public
                  procedure kurve(nummer:integer; laenge : real);
              end;

implementation


procedure TKochkurve.kurve(nummer:integer; laenge:real);
begin
  if nummer=1 then VW(laenge)
            else begin
                   kurve(nummer-1, laenge/3);
                   DL(60);
                   kurve(nummer-1, laenge/3);
                   DR(120);
                   kurve(nummer-1, laenge/3);
                   DL(60);
                   kurve(nummer-1, laenge/3)
                 end;
end;

end.
```

UNIT UKochkurve

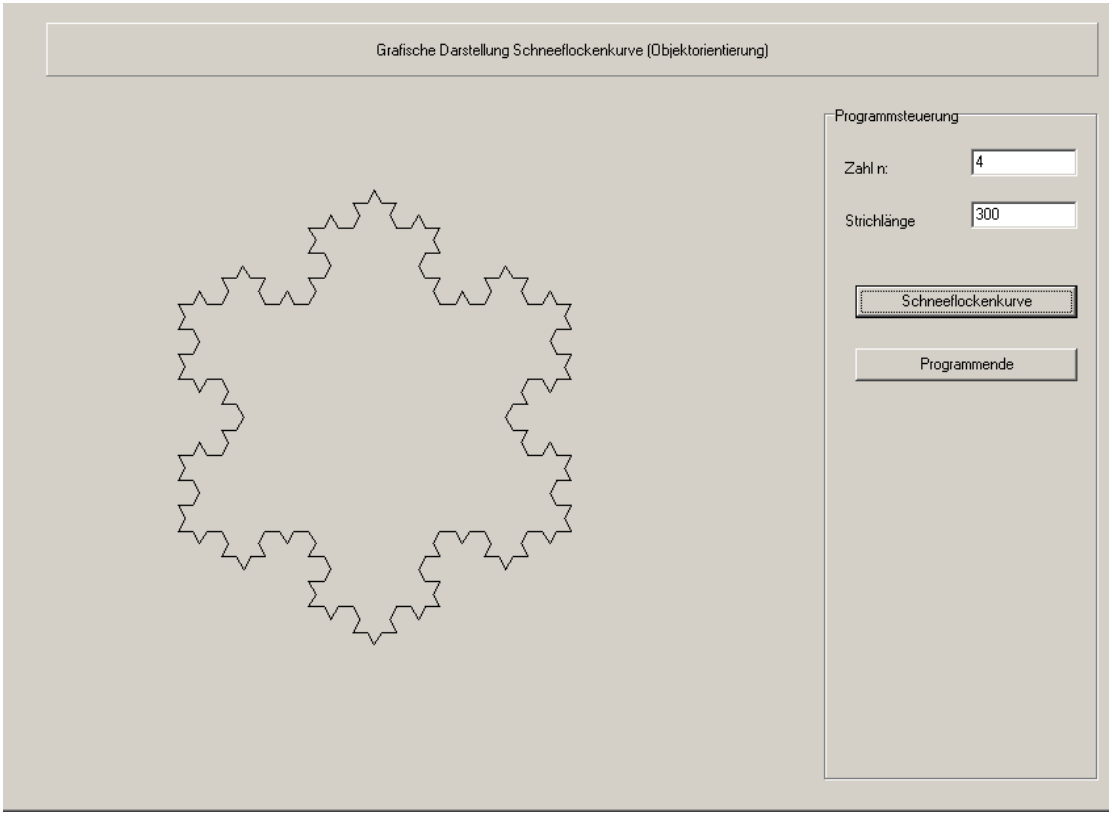
```
unit UKochkurve;

(* Werner Rockenbach  15. August 2004 *)

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, ExtCtrls,U_Turtle, mKochkurve;

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    GroupBox1: TGroupBox;
    Label1: TLabel;
    AnzahlEdit: TEdit;
    Button1: TButton;
    Button2: TButton;
    PaintBox1: TPaintBox;
    Label2: TLabel;
    laengeEdit: TEdit;
    procedure Button2Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  end;

var
  Form1: TForm1;
  Kochkurve: TKochkurve;

implementation

{$R *.DFM}

procedure TForm1.Button2Click(Sender: TObject);
begin
     close
end;

procedure TForm1.Button1Click(Sender: TObject);
var n : integer;
    laenge : real;

procedure Eingabe(var n: integer; var laenge : real);
begin
     n:=StrToInt(AnzahlEdit.text);
     laenge:=StrToFloat(LaengeEdit.text);
end;

begin
     Eingabe(n,laenge);
     Kochkurve:=TKochkurve.Create(Paintbox1);
     Kochkurve.Links;
     Kochkurve.SA;
     Kochkurve.Kurve(n,laenge);
end;

end.
```

Bildschirmkopie für n =5 und Strichlänge 500



8.2. Schneeflockenkurve

Bildet man die Rekursion für die drei Seiten eines gleichseitigen Dreiecks, so erhält man eine geschlossene Kurve. Dabei wird die Rekursion nacheinander für die drei Seiten des Dreiecks ausgeführt.



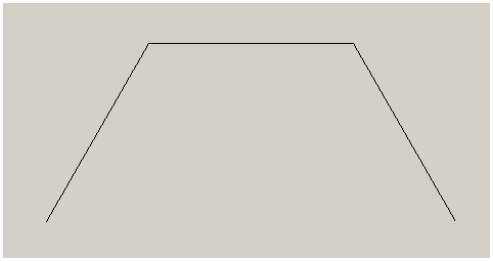
8.3. Pfeilspitzenkurve

Initiator Strecke der Länge 1

Nummer 1

Zeichne1(Länge : real)
Vorwärts(Länge)

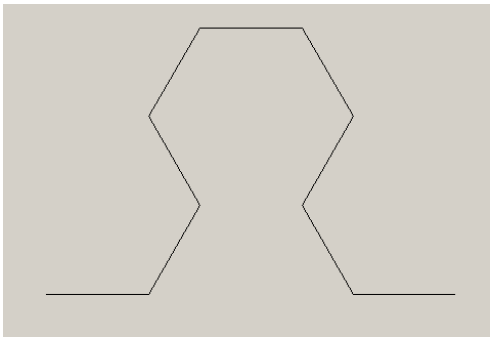
Generator



Nummer 2

Zeichne2(Länge : real)
Drehe links (60)
Zeichne1(Länge/2)
Drehe rechts (60)
Zeichne1(Länge/2)
Drehe rechts (60)
Zeichne1(Länge/2)
Drehe links (60)

Pfeilspitzenkurve Nummer 3



Die Pfeilspitzenkurve erhält man mit dem Generator als Linkskurve gefolgt vom Generator als Rechtskurve und einem anschließenden Generator als Linkskurve.
In die Prozedur wird für die verschiedenen Drehrichtungen ein Parameter Orientierung eingeführt., der abwechselnd den Wert +1 und –1 annimmt.

Algorithmus Pfeilspitzenkurve Nummer 2 mit Parameter Orientierung

Zeichne2(Länge : real; Orientierung : integer)
Drehe links (60 * Orientierung)
Zeichne1(Länge/2)
Drehe rechts (60 * Orientierung)
Zeichne1(Länge/2)
Drehe rechts (60 * Orientierung)
Zeichne1(Länge/2)
Drehe links (60 * Orientierung)

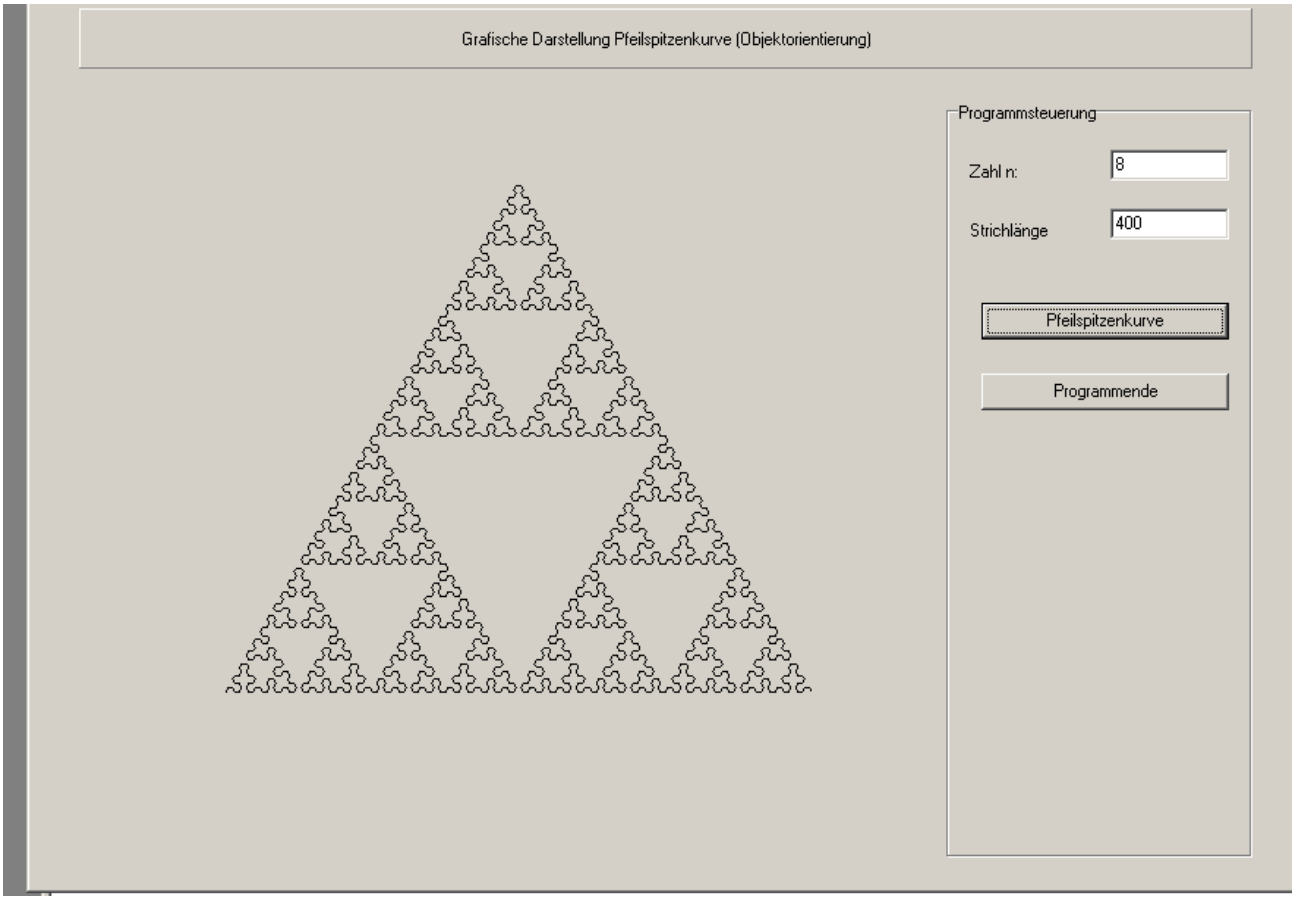
Algorithmus Pfeilspitzenkurve Nummer 3 mit Parameter Orientierung

Zeichne3(Länge : real; Orientierung : integer)
Drehe links (60 * Orientierung)
Zeichne2(Länge/2)
Drehe rechts (60 * Orientierung)
Zeichne2(Länge/2)
Drehe rechts (60 * Orientierung)
Zeichne2(Länge/2)
Drehe links (60 * Orientierung)

Allgemeine Methode

Zeichne(Nummer : integer; Länge : real; Orientierung : integer)	
Nummer = 1	
ja	nein
Vorwärts(Länge)	Drehe links (60 * Orientierung)
	Zeichne(Nummer-1, Länge/2, -Orientierung)
	Drehe rechts (60 * Orietierung)
	Zeichne(Nummer-1, Länge/2, Orientierung)
	Drehe rechts (60 * Orientierung)
	Zeichne(Nummer-1, Länge/2, -Orientierung)
	Drehe links (60 * Orientierung)

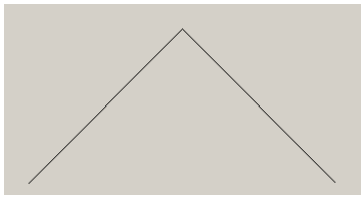
Pfeilspitzenkurve für n = 8 und Strichlänge 400



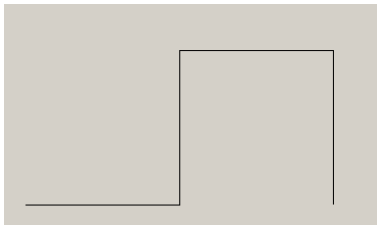
8.4. Drachenkurve

Die Drachenkurve hat als Initiator die Strecke der Länge 1
Nummer 1

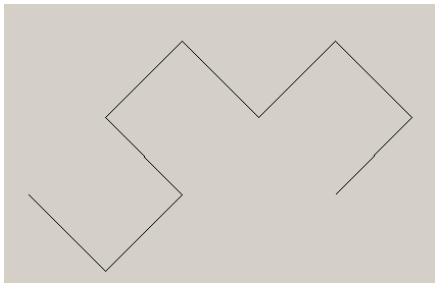
Der Generator besteht aus einem rechten Winkel
Nummer 2



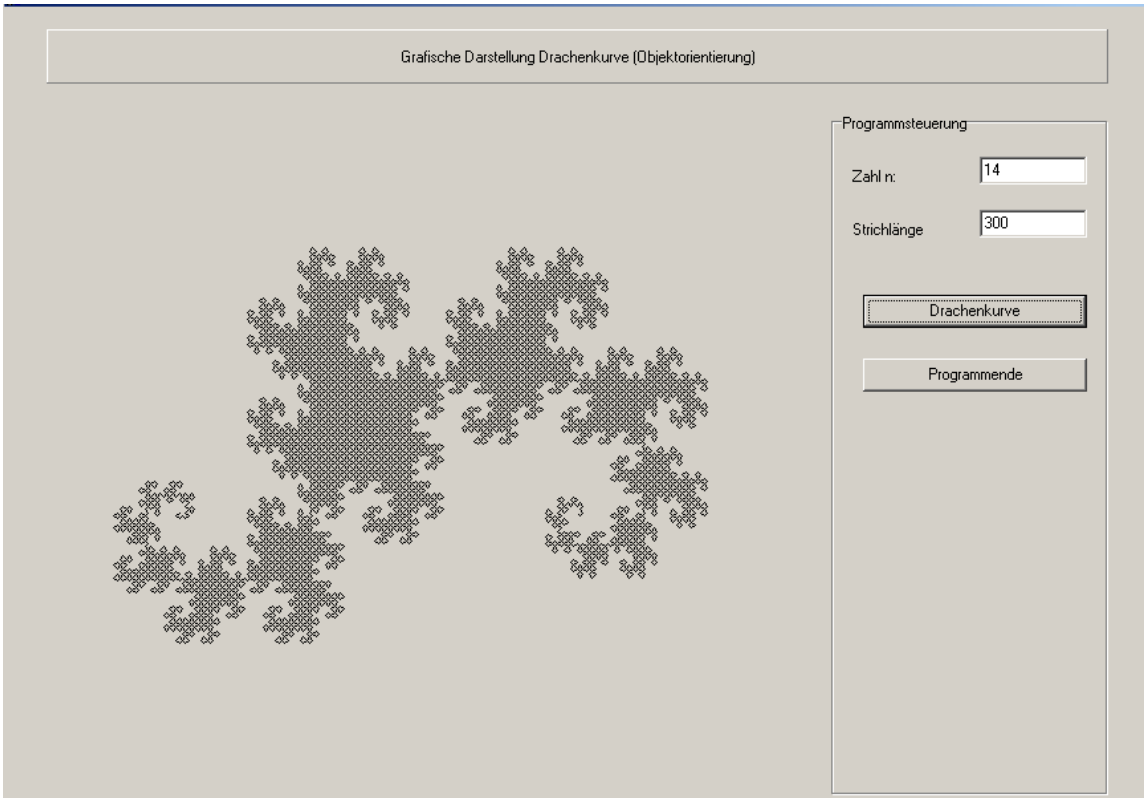
Nummer 3 besteht aus zwei rechten Winkeln



Nummer 4 besteht aus 4 rechten Winkeln



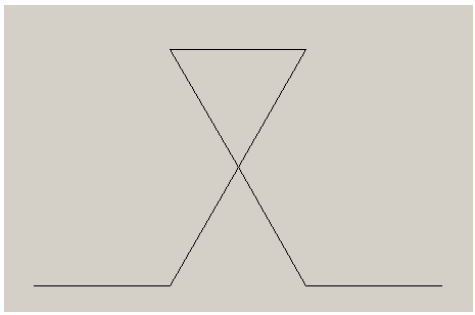
Drachenkurve n = 14 und Strichlänge 300



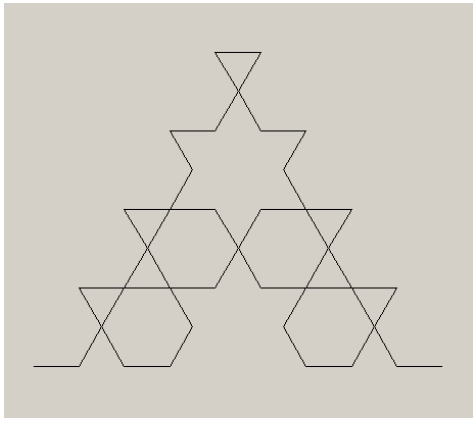
8.5.Schwammkurve

Die Schwammkurve hat als Initiator die Strecke der Länge 1
Nummer 1

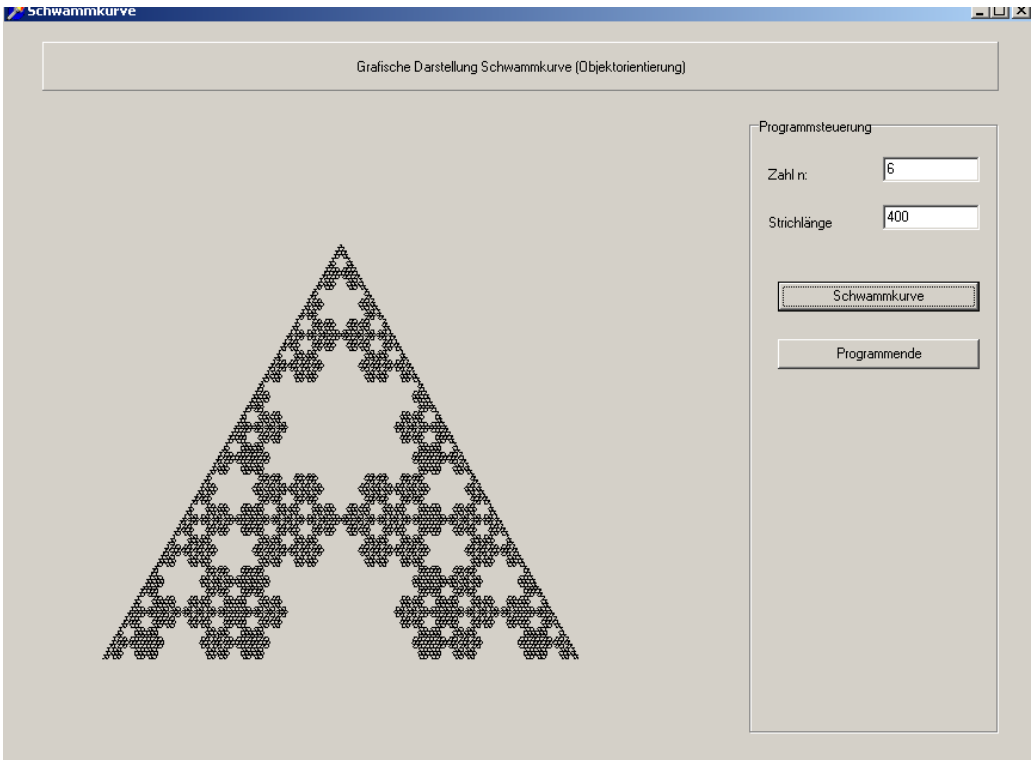
Generator Nummer 2



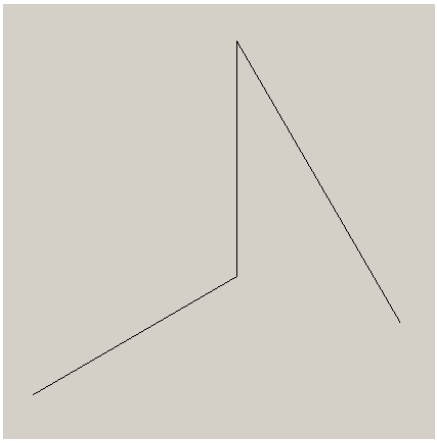
Nummer 3



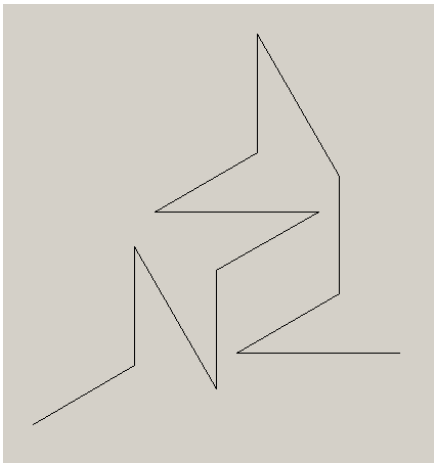
Schwammkurve n = 6 und Strichlänge 400



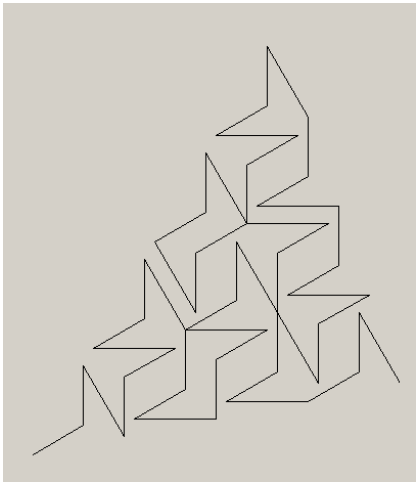
8.6. Sauzahnkurve



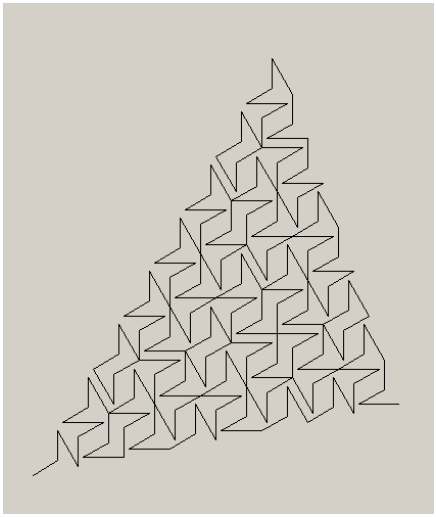
Initiator Nummer 1



Generator Nummer 2



Nummer 3



Nummer 4

9. Literatur

(1) Rüdeger Baumann
„Rekursion: Beispiele aus der Grafik“
LOGIN 3/85, Oldenbourg-Verlag

(2) Rüdeger Baumann
„Grafik im Informatikunterricht“
LOGIN 1/86 Oldenbourg-Verlag

(3) Mandelbrot
„Die fraktale Geometrie der Natur“
Berlin 1991, Birkhäuser-Verlag