

---

## 8 Grafik & Rekursion



Abb. 63: Ein Bild sagt mehr als tausend Worte – das gilt insbesondere beim Entdecken rekursiver Muster und deren funktionaler Modellierung

### 8.1 Übersicht

Grundsätzlich können zwei Aufgaben gestellt werden:

1. Es liegt eine rekursive Funktion vor, z.B. als Modell eines Sachproblems. Zur Analyse des quantitativen und qualitativen Verhaltens kann diese Funktion durch Diagramme visualisiert werden, d.h.

*Wie gelange ich zu einem geeigneten Diagramm?*

2. Es liegt ein rekursives (selbstbezügliches) graphisches Muster vor.

*Wie gelange ich zu einer passenden (rekursiven) Funktion bzw. Computerprogramm ?*

Bei der Beantwortung beider Fragen ist die Programmierung hilfreich (bei 2. sogar unverzichtbar); deshalb wird zunächst ein funktionales Graphikpaket als Erweiterung von DRACKET vorgestellt, s. Abschnitt 8.2.

## 8.2 Funktionale Graphik mit `image.ss`

Bei den meisten traditionellen Graphikpaketen einer Programmiersprache werden geometrische Figuren durch Prozeduren  $P$  erzeugt von der Form

$$P : p_1 p_2 \dots \rightarrow \langle \text{void} \rangle \quad (110)$$

d.h. sie liefern keinen Rückgabewert und sind somit keine Funktionen im mathematischen Sinn.


Dagegen liegt *funktionale Graphik* vor, wenn es sich bei den „Graphik-Befehlen“ um echte Funktionen handelt, die einen Wert liefern, also

$$F : p_1 p_2 \dots \rightarrow \text{Grafik-Typ} \quad (111)$$

Wir wollen die Umsetzung dieses Ansatzes anhand des Graphikpaketes `image.ss`<sup>10</sup> (sog. „Teachpack“) der HTDP-Sprachen von DRSCHEME kurz vorstellen:

- Grundlage ist der Datentyp **image**, der die von den unten aufgeführten Funktionen produzierten geometrischen Figuren und importierbare Bitmaps umfasst.
- Funktionen zur Erzeugung *elementarer geometrische Figuren*, wie z.B.


– `(circle 10 'solid 'gray)` → 

– `(triangle 30 'outline 'magenta)` → 

– usw.

Die Grafiken werden beim Funktionsaufruf – wie alle Funktionswerte – im Interaktionsfenster gezeigt, können aber – wie alle Funktionswerte – auch an einen Namen gebunden werden:


```
(define bild (ellipse 40 20 'outline 'black))
```

und durch `bild -->`  ebenfalls im Interaktionsfenster ausgegeben werden.

- Funktionen zur *relativen Überlagerung* von Graphiken, wie z.B.

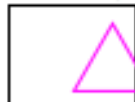
– `(overlay (triangle 30 'outline 'magenta) (circle 10 'solid 'gray))`

→ 

– `(beside bild (circle 10 'solid 'gray))` → 



– usw.

- Funktionen zur *absoluten Überlagerung* von Graphiken: z.B. kann mit `(empty-scene 50 40)` ein Bezugsrahmen erstellt werden, in dem andere Grafikobjekt absolut positioniert werden können mit

```
– (place-image
  → 
  (triangle 30 'outline 'magenta)
  40 20
  (empty-scene 50 40))
```

<sup>10</sup><http://www.eecs.northwestern.edu/~robby/pubs/papers/sfp2010-bff.pdf>

- usw.
- Funktionen zur *Manipulation* von Grafiken, wie z.B.

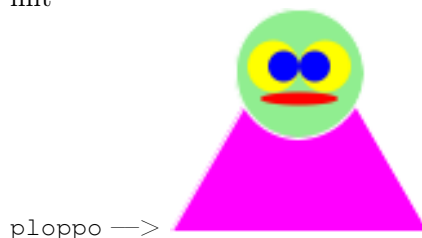
```
- (rotate 60 bild) —> 
- (scale 0.5 (circle 10 'solid 'green)) —> 
- usw.
```

- usw.

Somit können beliebige Figuren konstruiert werden, wie z.B.

```
(define ploppo
  (overlay/align 'middle 'top
    (overlay
      (above
        (overlay/align 'middle 'middle
          (beside
            (circle 6 'solid 'blue)
            (circle 6 'solid 'blue))
          (beside
            (circle 10 'solid 'yellow)
            (circle 10 'solid 'yellow)))
        (ellipse 30 10 'solid 'red))
      (circle 25 'outline 'white)
      (circle 25 'solid 'lightgreen))
    (triangle 100 'outline 'white)
    (triangle 100 'solid 'magenta)))
```

mit



ploppo —>

### 8.3 Visualisierung durch Diagramme

Wir unterscheiden zwischen

- *x-y-Diagramm*: die *Wertetabelle* (vgl. 3.2)  $((x_0, f(x_0)), (x_1, f(x_1)), \dots, (x_n, f(x_n)))$  wird in ein Diagramm umgesetzt, d.h. die Werte der Schrittfunction werden gegen die Werte der Rekursionsvariablen aufgetragen
- *Indexdiagramm* (*n-y-Diagramm*): die *Werteliste* (vgl. 3.2)  $(f(x_0), f(x_1), \dots, f(x_n))$  wird gegen den Index  $0, 1, 2, \dots$  aufgetragen

Bei einer Rekursion über  $\mathbb{N}$  (Typen ÜNQ und ÜNN, s. Abschnitt 6.1, d.h. primitives Rekursionsschema) stimmen *x-y-Diagramm* und *Indexdiagramm* überein. Bei allen anderen Rekursionen kann man zusätzlich oder alternativ zum *x-y-Diagramm* ein *Indexdiagramm* erstellen.

- *Streudiagramm* ( *$y_{n-1}$ - $y_n$ -Diagramm*, engl. *scatter plot*): die *Werteliste*  $(f(x_0), f(x_1), \dots, f(x_n)) = (y_0, y_1, \dots, y_n)$  wird so umgesetzt, daß  $y_n$  gegen  $y_{n-1}$  aufgetragen wird.

### Rechenblatt vs. Programmierung

```
(define (xy-diagramm tabelle BREITE HOEHE faktor farbe)
  (cond
    ((empty? tabelle) (empty-scene BREITE HOEHE))
    (else
     (place-image
      (circle 1 'solid farbe)
      (* faktor (first (first tabelle))) ; x-Koordinate
      (- HOEHE (* faktor (second (first tabelle)))) ; y-Koordinate
      (xy-diagramm (rest tabelle) BREITE HOEHE faktor farbe))))))

(define (index-diagramm liste n0 BREITE HOEHE faktor farbe)
  (cond
    ((empty? liste) (empty-scene BREITE HOEHE))
    (else
     (place-image
      (circle 1 'solid farbe)
      (+ n0 2) ; x-Koordinate
      (- HOEHE (* faktor (first liste))) ; y-Koordinate
      (index-diagramm (rest liste) (+ n0 1) BREITE HOEHE faktor farbe))))))

(define (streudiagramm liste BREITE HOEHE faktor farbe)
  (cond
    ((or (empty? liste) (empty? (rest liste)))
     (empty-scene BREITE HOEHE))
    (else
     (place-image
      (circle 1 'solid farbe)
      (* faktor (first liste)) ; x-Koordinate
      (- HOEHE (* faktor (first (rest liste)))) ; y-Koordinate
      (streudiagramm (rest liste) BREITE HOEHE faktor farbe))))))
```

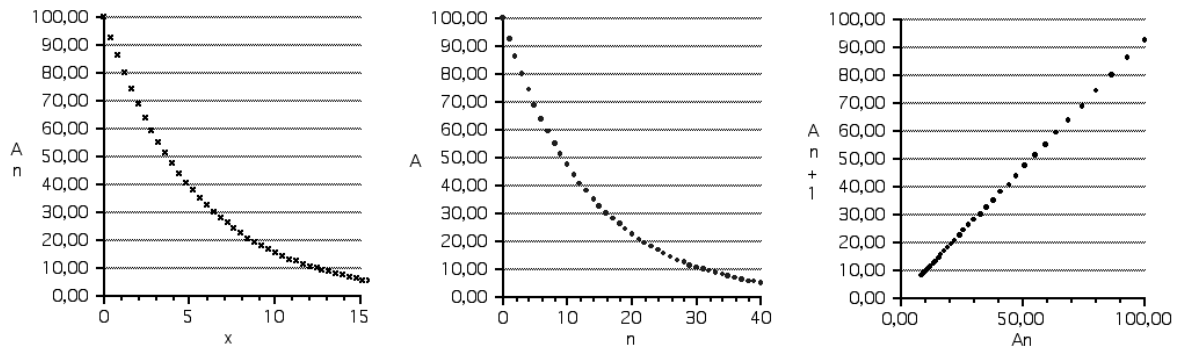
#### Beispiel 1

Wir greifen den Ansatz für exponentielles Wachstum (76) auf. Es handelt sich um eine Rekursion vom Typ ÄQ (vgl. 6.1, so daß im Rechenblatt aufgrund der Tabellierung von  $x$  und  $f(x)$  die Daten für ein  $x$ - $y$ -Diagramm bereitstehen:

	A	B	C	D	E	F	G		A	B	C	D	E	F	G	
1	Exponentielle Abnahme								1	Exponentielle Abnahme						
2	x0	0						2	x0	0						
3	A0	100						3	A0	100						
4	k	-0,18						4	k	-0,18						
5	$\Delta x$	0,4						5	$\Delta x$	0,4						
6				<i>n</i>	<i>x</i>	<i>A<sub>n</sub></i>	<i>A<sub>n+1</sub></i>	6				<i>n</i>	<i>x</i>	<i>A<sub>n</sub></i>	<i>A<sub>n+1</sub></i>	
7				0	=B\$2	=B\$3	=B	7				0	0	100,00	92,80	
8				=D7+1	=E7+B\$5	=F7+B\$4*B\$5*F7	=F9	8				1	0,4	92,80	86,12	
9				=D8+1	=E8+B\$5	=F8+B\$4*B\$5*F8	=F10	9				2	0,8	86,12	79,92	
10				=D9+1	=E9+B\$5	=F9+B\$4*B\$5*F9	=F11	10				3	1,2	79,92	74,16	
11				=D10+1	=E10+B\$5	=F10+B\$4*B\$5*F10	=F12	11				4	1,6	74,16	68,82	
12				=D11+1	=E11+B\$5	=F11+B\$4*B\$5*F11	=F12	12				5	2	68,82	63,87	

Abb. 64: Aufbereitung des Rechenblattes für Diagramme

Zusätzlich müssen der Index  $n$  (hier in Spalte D kursiv) für ein Indexdiagramm und  $y_{k+1}$  (hier in Spalte G kursiv) für ein Streudiagramm tabelliert werden:

Abb. 65: *x-y-Diagramm*, *Indexdiagramm* und *Streudiagramm* beim Rechenblatt

Die o.a. Programme zum Plotten der Diagramme benötigen als Eingabe die entsprechenden Wertetabellen bzw. -Listen.

Wir definieren uns zunächst die Hilfsfunktion

```
(define (runden zahl stellen)
  (/ (round (* zahl (expt 10 stellen))) (expt 10 stellen)))
```

für die gewünschte Stellenzahl.

```
(define (A-tabelle-hilfe x x0 k dx akku1 akku2 dezstellen)
  (cond
    ((= x x0) (list (list akku1 akku2)))
    (else
     (cons
      (list akku1 akku2)
      (A-tabelle-hilfe
       (- x dx) x0 k dx (+ dx akku1)
       (runden (* (+ 1 (* k dx)) akku2) dezstellen) dezstellen))))))
```

```
(define (A-tabelle x x0 A0 k dx dezstellen)
  (A-tabelle-hilfe x x0 k dx x0 A0 dezstellen))
```

mit z.B.

```
> (A-tabelle 2 0 100 -0.18 0.4 2)
```

```
-->
```

```
(list
 (list 0 100)
 (list 0.4 92.8)
 (list 0.8 86.12)
 (list 1.2 79.92)
 (list 1.6 74.17)
 (list 2 68.83))
```

und

```
(define (A-liste x x0 A0 k dx dezstellen)
  (cond
    ((= x x0) (list A0))
    (else
     (cons
      A0
      (A-liste
       (- x dx) x0
       (runden (* (+ 1 (* k dx)) A0) dezstellen) k dx dezstellen))))))
```

mit z.B.

```
> (A-liste 2 0 100 -0.18 0.4 2)
-->      (list 100 92.8 86.12 79.92 74.17 68.83)
```

und erhalten damit durch die Aufrufe der o.a. Plot-Programme für z.B.  $x = 16$

```
> (xy-diagramm (A-tabelle 16 0 100 -0.18 0.4 2) 300 300 4 'black)
> (index-diagramm (A-liste 16 0 100 -0.18 0.4 2) 300 300 0 4 'black)
> (streudiagramm (A-liste 16 0 100 -0.18 0.4 2) 300 300 4 'black)
```

die zugehörigen Diagramme:

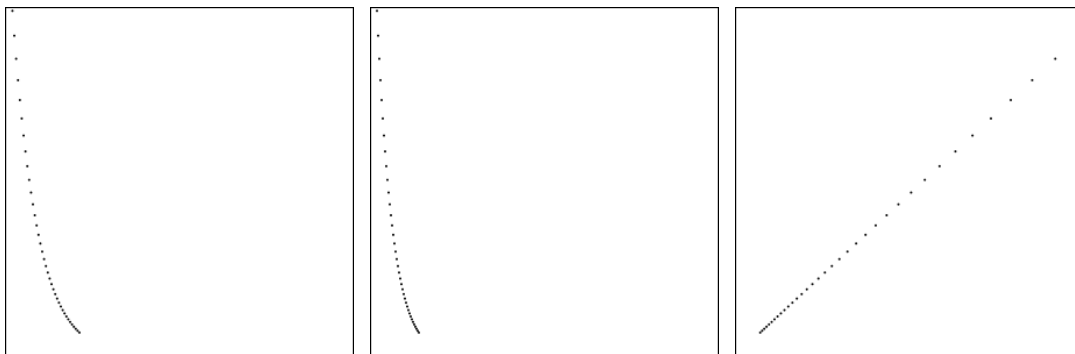


Abb. 66:  $x$ - $y$ -Diagramm, Indexdiagramm und Streudiagramm bei Programmierung

### Beispiel 2

Wir greifen die VERHULST-Folge (94) auf in der Form

1.

$$V_1(a, n) = \begin{cases} x_0 & \text{falls } n = 0 \\ a \cdot V_1(a, n-1) \cdot (1 - V_1(a, n-1)) & \text{falls } n > 0 \end{cases} \quad (112)$$

oder endrekursiv mit  $\varphi(a, x) = a \cdot x \cdot (1 - x)$

a)

$$V_2(a, x, n) = \begin{cases} x_0 & \text{falls } n = 0 \\ V_2(a, \varphi(a, x), n-1) & \text{falls } n > 0 \end{cases} \quad (113)$$

b)

$$V_3(a, x) = \begin{cases} x & \text{falls } x = \varphi(a, x) \\ V_3(a, \varphi(a, x)) & \text{sonst} \end{cases} \quad (114)$$

mit  $x_0 \in [0; 1]$  und untersuchen sie für  $a = 3,993$  (chaotischen Bereich) und  $x_0 = 0, 1$ .

Im Rechenblatt brauchen wir für das Streudiagramm zusätzlich die Tabellierung von  $y_{n+1}$ , d.h. die Werte von  $y_n$  werden um 1 nach oben verschoben (Spalte F):

	A	B	C	D	E	F		B	C	D	E	F	
1	VERHULST- Dynamik						1	RHULST- Dynamik					
2							2						
3	x0	0,1					3	0,1					
4	a	3,993					4	3,993					
5			n	Vn	Vn+1		5	n	Vn	Vn+1			
6			0	=B3	=E7		6	0	0,100000	0,359370			
7			=D6+1	=B\$4*E6*(1-E6)	=E8		7	1	0,359370	0,919281			
8			=D7+1	=B\$4*E7*(1-E7)	=E9		8	2	0,919281	0,296294			
9			=D8+1	=B\$4*E8*(1-E8)	=E10		9	3	0,296294	0,832555			
10			=D9+1	=B\$4*E9*(1-E9)	=E11		10	4	0,832555	0,556652			
11			=D10+1	=B\$4*E10*(1-E10)	=E12		11	5	0,556652	0,985434			
12			=D11+1	=B\$4*E11*(1-E11)	=E13		12	6	0,985434	0,057313			
13			=D12+1	=B\$4*E12*(1-E12)	=E14		13	7	0,057313	0,215735			
14			=D13+1	=B\$4*E13*(1-E13)	=E15		14	8	0,215735	0,675589			

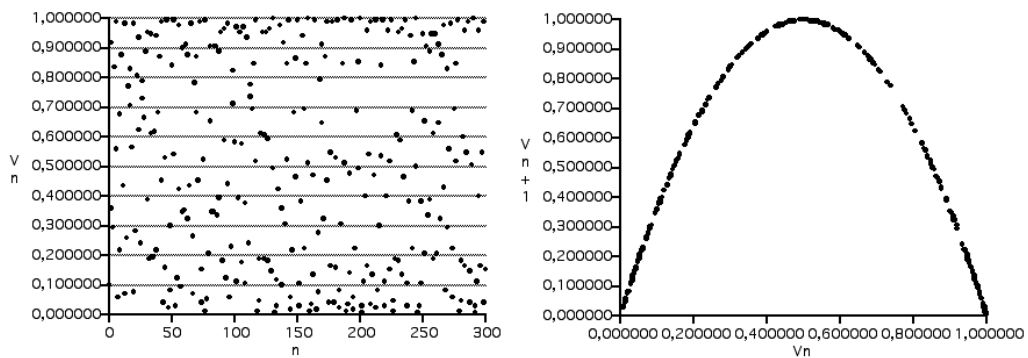


Abb. 67:  $x - y$ /Indexdiagramm und Streudiagramm bei Tabellenkalkulation

Während das Indexdiagramm das bekannte „Chaos“ darstellt, macht das Streudiagramm den Attraktor deutlich

Für die Programmierung wählen wir für (112)  $n = 300$  und erhalten durch die Aufrufe

```
> (index-diagramm (VERHULST-liste 0.1 3.993 300) 0 300 300 300 'black)
> (streudiagramm (VERHULST-liste 0.1 3.993 500) 300 300 300 'black)
```

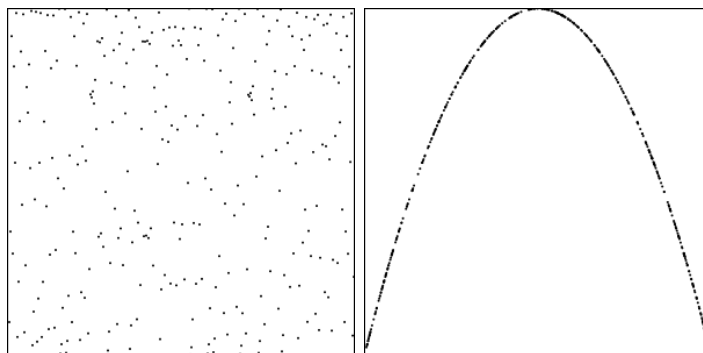


Abb. 68: Indexdiagramm und Streudiagramm bei Programmierung

Beispiel 3

Wir betrachten die sog. HÉNON-Abbildung

$$\begin{aligned}x_n &= 1 - ax_{n-1}^2 + y_{n-1} \\ y_n &= bx_{n-1}\end{aligned}\tag{115}$$

```
(define (henon-tabelle-hilfe a b akku n)
  (cond
    ((= n 0) akku)
    (else
     (henon-tabelle-hilfe
      a b
      (cons
       (list
        (runden
         (+ 1 (* a (sqr (first (first akku))) -1) (second (first akku))) 4)
        (runden (* b (first (first akku))) 4)
         akku)
       (- n 1))))))
```

```
(define (henon-tabelle a b x0 y0 n)
  (henon-tabelle-hilfe a b (list (list x0 y0)) n))
```

mit z.B.

```
> (henon-tabelle 1.4 0.3 0.5 0.5 4)
-->
(list
 (list 0.5 0.5)
 (list 1.15 0.15)
 (list -0.7015 0.345)
 (list 0.6561 -0.2104)
 (list 0.1869 0.1968))

> (streudiagramm2 (henon-liste 1.4 0.3 0.5 0.5 1000) 600 300 200 'black)
```

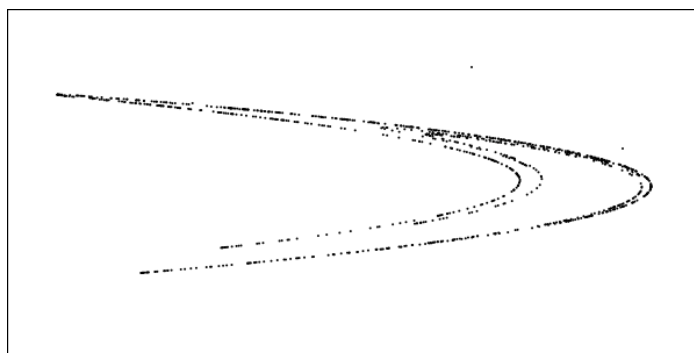


Abb. 69: HÉNON-Attraktor

## 8.4 Rekursive Muster

Fraktale können auf viele verschiedene Arten erzeugt werden, doch alle Verfahren beinhalten ein rekursives Vorgehen. Mögliche Verfahren sind:

1. *Iterierte Funktionen*, also Rekursionen vom IR (s. Abschnitt 6.1) ist die einfachste und bekannteste Art, Fraktale zu erzeugen; die Mandelbrot-Menge entsteht so. Eine besondere Form dieses Verfahrens sind IFS-Fraktale (Iterierte Funktionensysteme), bei denen mehrere Funktionen kombiniert werden. So lassen sich natürliche Gebilde erstellen.
2. *Nichtlineare dynamische Systeme* erzeugen fraktale Gebilde, z.B. „seltsame Attraktoren“.
3. *Lindenmayer-Systeme* beruhen auf wiederholter Symbolersetzung und eignen sich in besonderer Weise zur Modellierung natürlicher Gebilde wie Pflanzen und Zellstrukturen

### Muster aus implementierten Graphik-Funktionen („Zeichenbefehlen“)

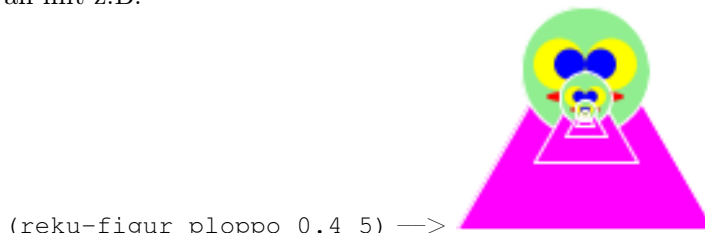
Der Typ `image` selbst ist kein induktiver Datentyp (nur intern rekursiv implementiert), d.h. eine Rekursionen über Variablen dieses Typs ist nicht möglich. Eine Alternative stellen strukturelle Rekursionen über vorhandene induktive Datentypen dar.

#### Beispiel 1

Auf die oben definierte Figur `ploppo` wenden wir die Funktion

```
(define (reku-figur bild faktor n)
  (cond
    ((zero? n) bild)
    (else
     (overlay (reku-figur (scale faktor bild) faktor (- n 1)) bild))))
```

an mit z.B.

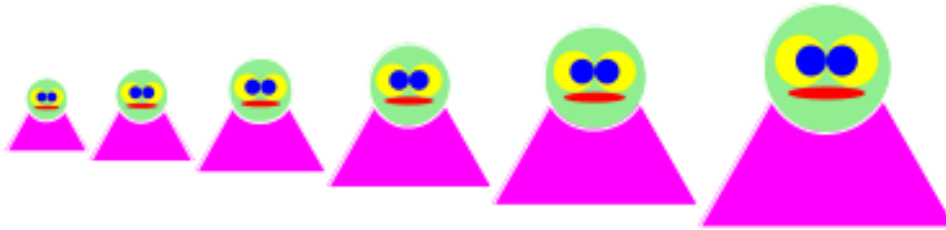


oder allgemeiner

```
(define (reku-figuren abbildung bild faktor n)
  (cond
    ((zero? n) bild)
    (else
     (abbildung
      (reku-figuren abbildung (scale faktor bild) faktor (- n 1) ) bild))))
```

mit

(reku-figuren beside ploppo 0.8 6) —>



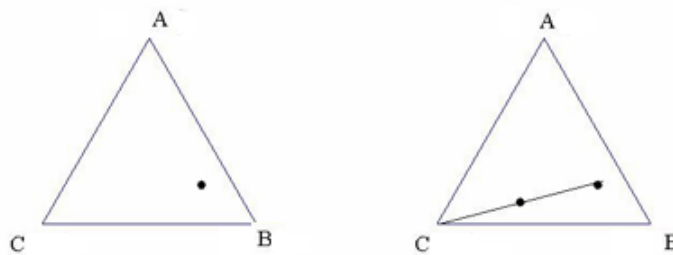
oder



(reku-figuren overlay ploppo 0.8 6) —>

### Beispiel 2

Zur Darstellung des bekannten SIERPINSKI-Dreiecks kann man auch sog. *Chaos-Spiel* einsetzen: Ein gleichseitiges Dreieck mit den Ecken A B C befindet sich bündig in einem Quadrat gleicher Kantenlänge. Jetzt wird ein zufälliger Punkt im Inneren des Quadrates gewählt. Dieser wird mit einer zufällig ausgewählten Ecke des Dreiecks verbunden. Die Mitte der so entstandenen Strecke wird markiert nun den Ausgangspunkt für die nächste Runde. Wiederholt man dies sehr oft bilden die Punkte eine Näherung des Sierpinski-Dreiecks.



Wir konstruieren dazu die Funktion

```
; (gleichseitiges) Dreieck:
(define KANTE 400)
; Ecken
(define A (make-posn 0 (/ (* KANTE (sqrt 3)) 2)))
(define B (make-posn (/ KANTE 2) 0))
(define C (make-posn KANTE (/ (* KANTE (sqrt 3)) 2)))
; Dreieck als Liste
(define dreieck (list A B C))
; Zufallspunkt (im Rechteck um das Dreieck)
(define P0 (make-posn
            (random (posn-x C))
```

```

      (random (floor (inexact->exact (/ (* KANTE (sqrt 3)) 2))))))
;sierpinski: posn number --> scene
(define (sierpinski-dreieck P0 P1 n)
  (local
    ((define ecke (list-ref dreieck (random 3))) ; Zufallsecke
     (define M
      (make-posn (/ (+ (posn-x P0) (posn-x ecke)) 2)
                 (/ (+ (posn-y P0) (posn-y ecke)) 2))))
    (cond
      ((zero? n)
       (empty-scene KANTE (/ (* KANTE (sqrt 3)) 2)))
      ((= n 1)
       (place-image
        (square 4 'outline 'black)
        (posn-x P1) (posn-y P1)
        (sierpinski-dreieck M P1 (- n 1))))
      (else
       (place-image
        (circle 1 'solid 'black)
        (posn-x M) (posn-y M)
        (sierpinski-dreieck M P1 (- n 1)))))))

```

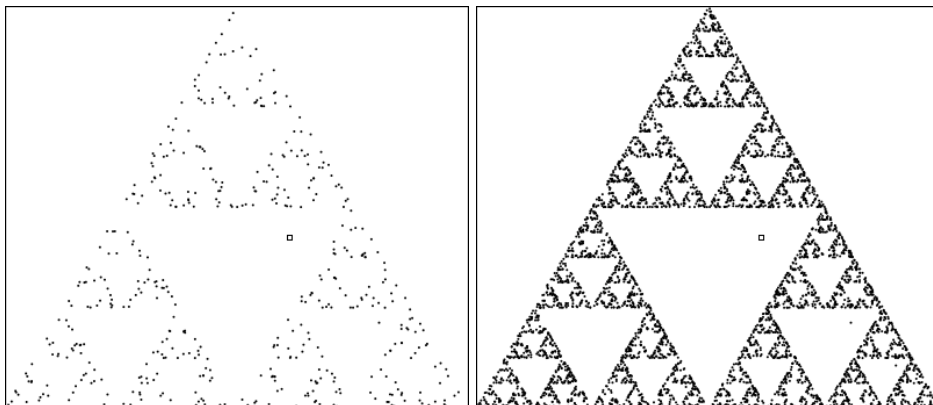
Die Aufrufe

```

> (sierpinski-dreieck P0 P0 500)
> (sierpinski-dreieck P0 P0 5000)

```

liefern z. B.



### Muster aus Bildobjekten

Für den Informatikunterricht kann es sicher interessant sein, einen induktiven Datentyp zu erstellen, über den die Rekursionen laufen können, statt den eingebauten Typ Liste zu benutzen.

Dazu definieren wir naheliegenderweise einen Typ RP („Russische Puppe“), der selbst kein Grafikobjekt enthält, dafür aber den Faktor, mit dem die Ausgangsgröße eines Grafikobjektes skaliert werden soll, und erhalten durch Paarbildung eine Listen-Struktur:

$$\text{Eine } \mathbf{RP} \text{ ist } \begin{cases} \text{eine Zahl} \\ \text{oder} \\ \text{ein Paar } (\mathbf{Zahl}, \mathbf{RP}) \end{cases}$$

Die Umsetzung erfolgt mit einem passenden Rekord

```
(define-struct schicht (faktor RP))
```

 (116)

Jetzt kann man z.B. ausgehend von einer äußeren Puppe der Größe 1 jeweils kleinere Exemplare ineinanderschachteln:

```
(define P0 1)
(define P1 (make-schicht 0.8 P0))
(define P2 (make-schicht 0.64 P1))
(define P3 (make-schicht 0.512 P2))
```

mit

```
> P0 --> 1
> P1 --> (make-schicht 0.8 1)
> P2 --> (make-schicht 0.64 (make-schicht 0.8 1))
> P3 --> (make-schicht 0.512 (make-schicht 0.64 (make-schicht 0.8 1)))
```

konstruieren.

Wenn sich zwei Schichten jeweils um einen bestimmten Faktor unterscheiden sollen, kann man auch eine passende Funktion konstruieren:

```
(define (make-puppe faktor f0 n)
  (cond
    ((= n 1) f0)
    (else
     (make-puppe faktor (make-puppe faktor (* faktor f0) (- n 1))))))
```

und erhalten z.B.

```
> (make-puppe 0.8 1 8)
-->
(make-schicht 1
  (make-schicht 0.8
    (make-schicht 0.64
      (make-schicht 0.512
        (make-schicht 0.4096
          (make-schicht 0.32768
            (make-schicht 0.262144 0.2097152))))))))
```

Genau genommen, stellt `make-schicht` nichts anderes als das `cons` der Listenerzeugung dar; die der Listenverarbeitung entsprechenden Operationen sind auch leicht zu konstruieren:

```
; äußere: RP --> number
; äußerste/hinterste/größte Puppe („first“)
(define (äußere-puppe eine-rp)
  (cond
    ((number? eine-rp) eine-rp)
    ((schicht? eine-rp) (äußere-puppe (schicht-RP eine-rp)))))

; innere-puppe: RP --> number
; innerste/vorderste/kleinste Puppe
```

```

(define (innere-puppe eine-rp)
  (cond
    ((number? eine-rp) eine-rp)
    ((schicht? eine-rp) (schicht-faktor eine-rp))))

; hülle RP --> RP
; RP ohne erste Puppe („rest“)
(define (hülle eine-rp)
  (cond
    ((number? (schicht-RP eine-rp)) (schicht-RP eine-rp))
    ((schicht? eine-rp) (schicht-RP eine-rp))))

; kern RP --> RP
; RP ohne letzte Puppe
(define (kern eine-rp)
  (cond
    ((number? (schicht-RP eine-rp)) (schicht-faktor eine-rp))
    ((schicht? eine-rp)
     (make-schicht (schicht-faktor eine-rp) (kern (schicht-RP eine-rp))))))

```

mit z.B.

```

> (hülle (make-puppe 0.8 1 8))
-->
(make-schicht 0.8
  (make-schicht 0.64
    (make-schicht 0.512
      (make-schicht 0.4096
        (make-schicht 0.32768
          (make-schicht 0.262144 0.2097152)))))))

```

oder

```

> (kern (make-puppe 0.8 1 8))
-->
(make-schicht 1
  (make-schicht 0.8
    (make-schicht 0.64
      (make-schicht 0.512
        (make-schicht 0.4096
          (make-schicht 0.32768 0.262144)))))))

```

usw.

Um ein Muster zu zeichnen brauchen wir ein Bildobjekt: Überraschenderweise definieren wir



```
(define matroschka )
```

Jetzt können wir z.B. unsere Puppe auf vielfältige Weise „auspacken“:

```
(define (auspacken abbildung winkel puppe bild)
  (cond
```

```

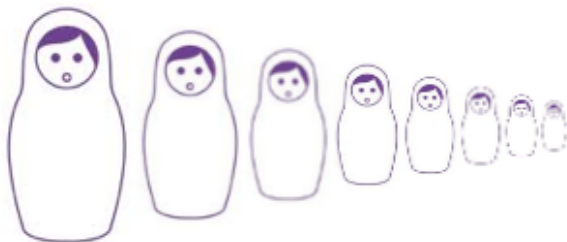
((number? puppe)
 (scale puppe bild))
((schicht? puppe)
 (abbildung
  (scale (schicht-faktor puppe) bild)
  (auspacken
   abbildung
   winkel
   (schicht-RP puppe)
   (rotate winkel bild))))))

```

> (auspacken underlay 0 (mache-puppe 0.5 1 8) matroschka) ->



> (auspacken underlay 40 (mache-puppe 0.6 1 8) matroschka) ->  
(> auspacken beside 0 (mache-puppe 0.8 1 8) matroschka) ->



(> auspacken beside 170 (mache-puppe 0.8 1 8) matroschka) ->



Wir schließen mit einem Beispiel, das natürlich nicht fehlen darf: der sich selbst enthaltende Spiegel, nämlich



(define spiegel

und benutzen die bereits oben definierte Funktion (reku-figures ...):



> (reku-figures overlay spiegel 0.4 6) ->

ÜBUNGEN:

1. Konstruiere zu (116) Funktionen, die den Listenoperation entsprechen
2. ...