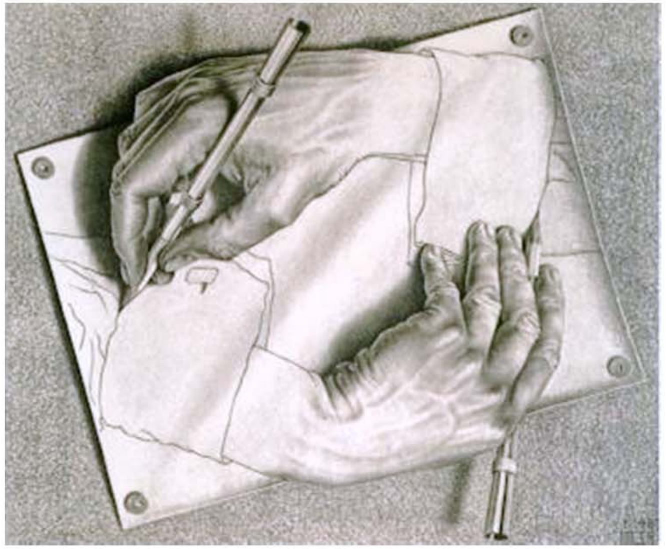
Rekursion



- Sie wissen wie man Programme rekursiv entwickelt
- Sie kennen typische Beispiele von rekursiven Algorithmen
- Sie kennen die Vor-/Nachteile von rekursiven Algorithmen

Rekursion

Definition:

> Eine Funktion ist rekursiv, wenn sie sich selbst direkt oder indirekt aufruft.
>
> Ein Datenstruktur heisst rekursiv, wenn sie sich selbst als Teil enthält oder mit Hilfe von sich selbst definiert ist.

- Vorteil der rekursiven Beschreibung ist die Möglichkeit, eine unendliche Menge durch eine endliche Aussage zu beschreiben
  - z.B. Objekt x enthält wieder Objekt x
- In Programmen wird Rekursion durch Funktionen implementiert, die sich selbst aufrufen
  - z.B. Funktion p ruft Funktion p auf

Beispiel einer rekursiv definierten Funktion: die Fakultät

- n! = 1*2*3…(n-1)*n   : 1, 2, 6, 24, 120, 720, 5040, 40320,

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

```
fak(3) = 3 * fak(2)
             2 * fak(1)
                 1 * fak(0)
                       1
                 1 * 1
             2 * 1
         3 * 2
           6
```

Fakultätsberechnung mit Rekursion

```
int fak(int n) {
    if (n == 0) return 1;
    else return n* fak(n-1);
}
```

Abbruch

rekursiver Aufruf

Uebung: Die Summenfunktion summiert die ersten n Summanden einer Reihe.
$s = a_1 + a_2 + ... + a_n$.
Definieren Sie diese Funktion rekursiv.

Beispiel einer rekursiv definierten Datenstruktur

```
p = first;
while (p != null) {
  doSomething(p);
  p = p.next;
}
```

Liste nicht rekursiv definiert

- Liste ::= (Element)*

Liste rekursiv definiert

- Liste ::= leer
- Liste ::= Element Liste

```
void traverse(List l) {
  if (l == null) // Abbruch
  else {
      doSomething(l);
      traverse(l.next);
  }
};
```

Notation

- ::= definiert
- ()* : beliebig oft, 0..∞
- ()? : optional, 0..1
- a b : a gefolgt von b

Übung

- Schreiben Sie eine rekursive Methode, welche die Elemente einer einfach verketteten Liste der Reihe nach ausgibt.

- Schreiben Sie eine rekursive Methode, welche die Elemente einer **einfach** verketteten Liste in **umgekehrter** Reihenfolge ausgibt.

Generelle Vorlage für rekursive Programme

- Das vorherige Beispiel (Fakultät) ist durch vollständige Induktion (Mathematik) beweisbar. Rekursive Programme sind das Programmäquivalent der vollständigen Induktion. Wesentlich ist somit, dass man zwischen zwei Fällen unterscheidet (wie bei den Beweisen):

- **1. Basis Fall** („Verankerung“):
  - Man weiss z.B., dass fak(0) = 1 ist.

- **2. Allgemeiner Fall** („Rekursionsschritt“):
  - Für alle anderen Fälle (z.B. n>0) weiss man, dass sich die Lösung des Problems X(n) zusammensetzt aus einigen Operationen und einem Problem X(n−1) was eine Dimension kleiner als X(n) ist, z.B. fak(n) = n * fak(n−1), n>0. Man zerlegt also das Problem für den allgemeinen Fall so lange, bis man auf den Basis Fall kommt.

Vorlage für rekursive Programme

Damit muss eine allgemeine Vorlage für rekursive Programme diese **beiden Fälle unterscheiden.** Der **Basis Fall** stellt sicher, dass die rekursiven Programme endlich sind und **terminieren** (d.h. die Anzahl der rekursiven Aufrufe ist begrenzt).

Vergisst man den Basis Fall, so werden im allgemeinen so viele rekursive Aufrufe durchgeführt, bis der Stack überläuft (Abbruch mit StackOverflow).

```
int p(int n) {
    int x;
    if (BaseCase==true) {
    // behandeln Basis Fall
    } else {
        p(n1); // behandeln allg. Fall
    }
    return x;
}
```

führt sicher zum Basis Fall, da jetzt ein kleineres Problem gelöst werden soll

Direkte/indirekte Rekursion

- Direkte Rekursion:
  - Bei der direkten Rekursion ruft eine Methode sich selber wieder auf (es fehlt der Rekursionsabbruch).

```
int p(int a) {
   int x = p(a1);
   return x;
}
```

- Indirekte Rekursion:
  - Bei der indirekten Rekursion rufen sich zwei oder mehrere Methoden gegenseitig auf.

```
int p(int a) {
   int x = q(a1);
   return x;
}
```

```
int q(int a) {
   int x = p(a-1);
   return x;
}
```

Einfache Rekursion → Schleife

Programm mit Rekursion

```
int fak(int n) {
    int res = 1;
    if (n > 0) res = n * fak(n-1);
    return res;
}
```

Programm mit Iteration

```
int fak(int n) {
    res = 1;
    while (n > 0) {
        res = n * res;
        n--;
    }
    return res;
}
```

Rekursion der Form:

k • f() oder f() • k

lassen sich leicht in eine nicht rekursive Form überführen

jedes rekursive Programm lässt sich prinzipiell in ein iteratives umschreiben

Übung

- Schleifen: Operationen werden endlich oft wiederholt

```
void p()  {
    int i = 0;
    while (i < 10)
       System.out.println(i++);
}
```

output

- Rekursion (Aufruf p(1))

```
void p(int i)  {
    if (i < 10) {
       System.out.println(i);
       p(i+1);
    }
}
```

```
void p(int i)  {
    if (i < 10) {
       p(i+1);
       System.out.println(i);
    }
}
```

Hamster Beispiel 1

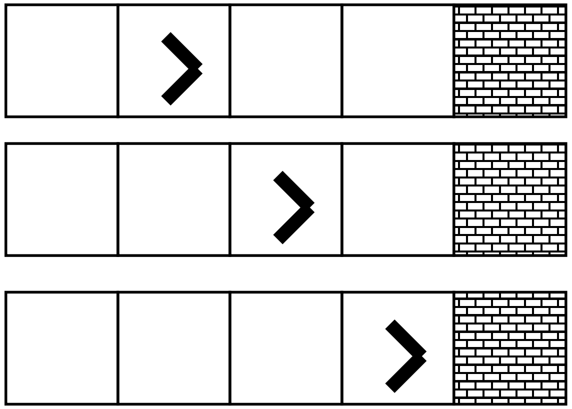
Der Hamster soll bis zur nächsten Wand laufen!

Iterative Lösung:

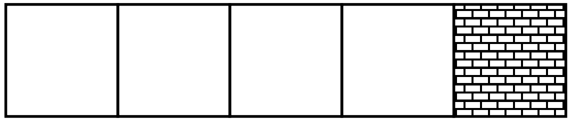
```
void zurMauer() {
  while (vorn_frei())
    vor();
}
```

Direkt rekursive Lösung:

```
void zurMauerR() {
  if (vorn_frei()) {
    vor();
    zurMauerR();
  }
}
```



Hamster Beispiel 2



Der Hamster soll bis zur nächsten Wand und dann zurück zur Ausgangsposition laufen!

Direkt rekursive Lösung:

```
void hinUndZurueckR() {
  if (!vorn_frei()) {
    kehrt();
  }
  else{
    vor();
    hinUndZurueckR();
    vor();
  }
}
```

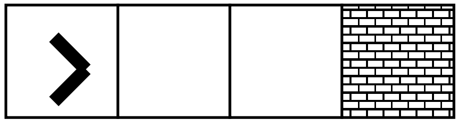
Wird erst aufgerufen, nachdem die Rekursion beendet ist

```
void hinUndZurueckR() {
  if (!vorn_frei()) {
    kehrt();
  }
  else{
    vor();
    hinUndZurueckR();
    vor();
  }
}
```

Hamster Beispiel 2

```
main:          hUZR (1.)          hUZR (2.)          hUZR (3.)

hUZR();        vorn_frei -> t
               vor();
               hUZR(); -----> vorn_frei -> t
                              vor();
                              hUZR(); -----> vorn_frei -> f
                                             kehrt();
                                     <-----
                              vor();
                      <-----
               vor();
       <-----
```



Befehlsfolge: vor(); vor(); kehrt(); vor(); vor();

Hamster Beispiel 3

Der Hamster soll die Anzahl Schritte bis zur nächsten Mauer zählen!

Iterative Lösung:

```
int anzahlSchritte() {
  int anzahl = 0;
  while (vorn_frei()) {
    vor();
    anzahl++;
  }
  return anzahl;
}
```

Rekursive Lösung:

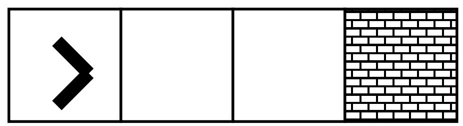
```
int anzahlSchritteR() {
  if (vorn_frei()) {
    vor();
    return 1 + anzahlSchritteR();
  } else
    return 0;
}
```

Hamster Beispiel 3

```
int anzahlSchritteR() {
  if (vorn_frei()) {
    vor();
    return 1 + anzahlSchritteR();
  } else
    return 0;
}
```

```
main:        aSR (1.)      aSR (2.)      aSR (3.)

i=aSR();     vorn_frei -> t
             vor();
             aSR()  -----> vorn_frei -> t
                           vor();
                           aSR()  -----> vorn_frei -> f
                                         return 0;
                                     0
                                  <-----
                           return 1 + 0;
                       1
                    <-----
             return 1 + 1;
        2
     <-----
i=2;
```

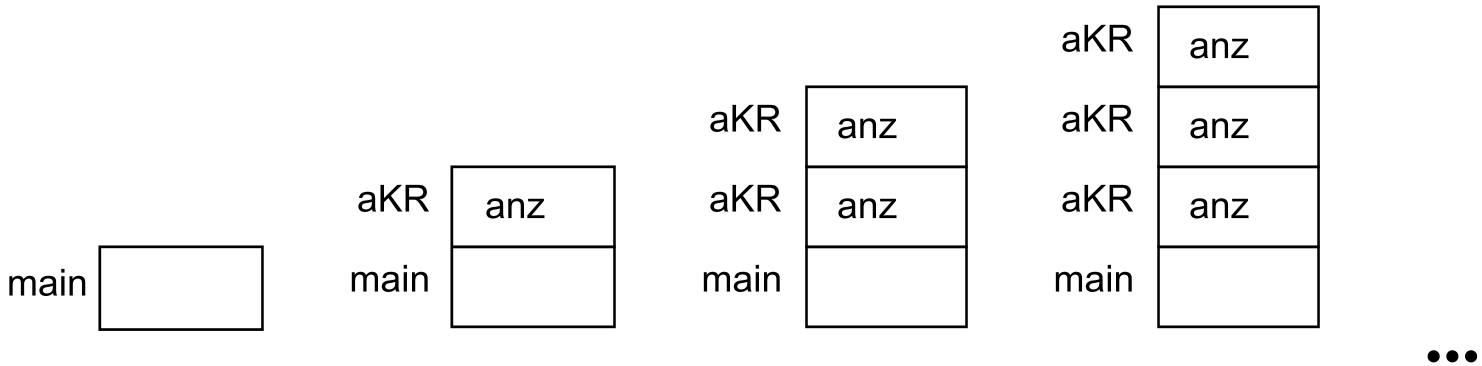


Hamster Beispiel 4: Lokale Variablen

Der Hamster soll die Anzahl Körner zählen

```
int anzahlKörner() {
  int anz;
  if (vorn_frei()) {
    anz = körner();
    return anz + anzahlKörnerR();
  } else
    return 0;
}
```

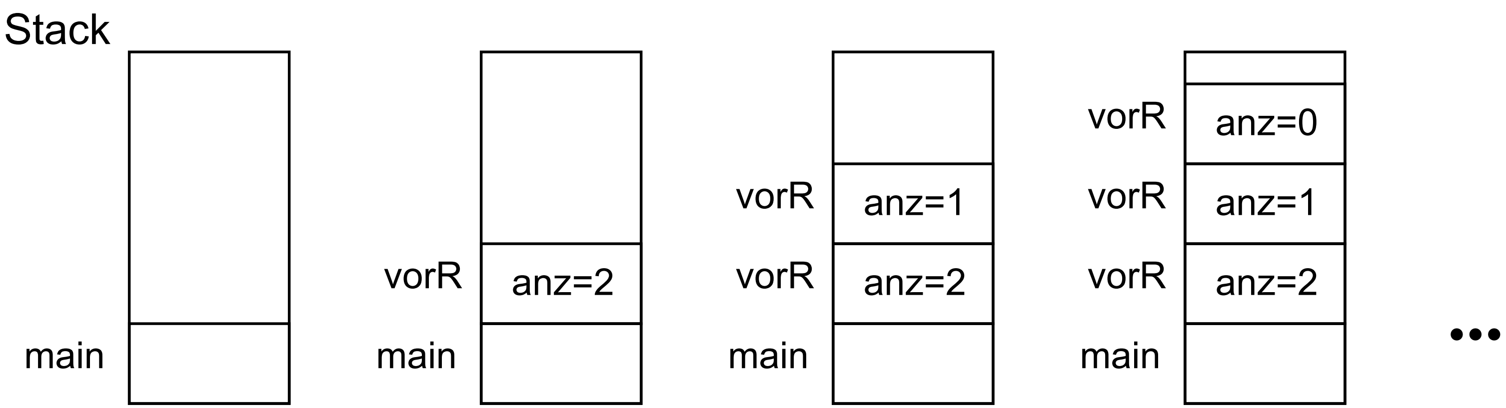
Stack



Hamster Beispiel 5: Parameter

Der Hamster soll „anz“-Schritte nach vorne gehen!

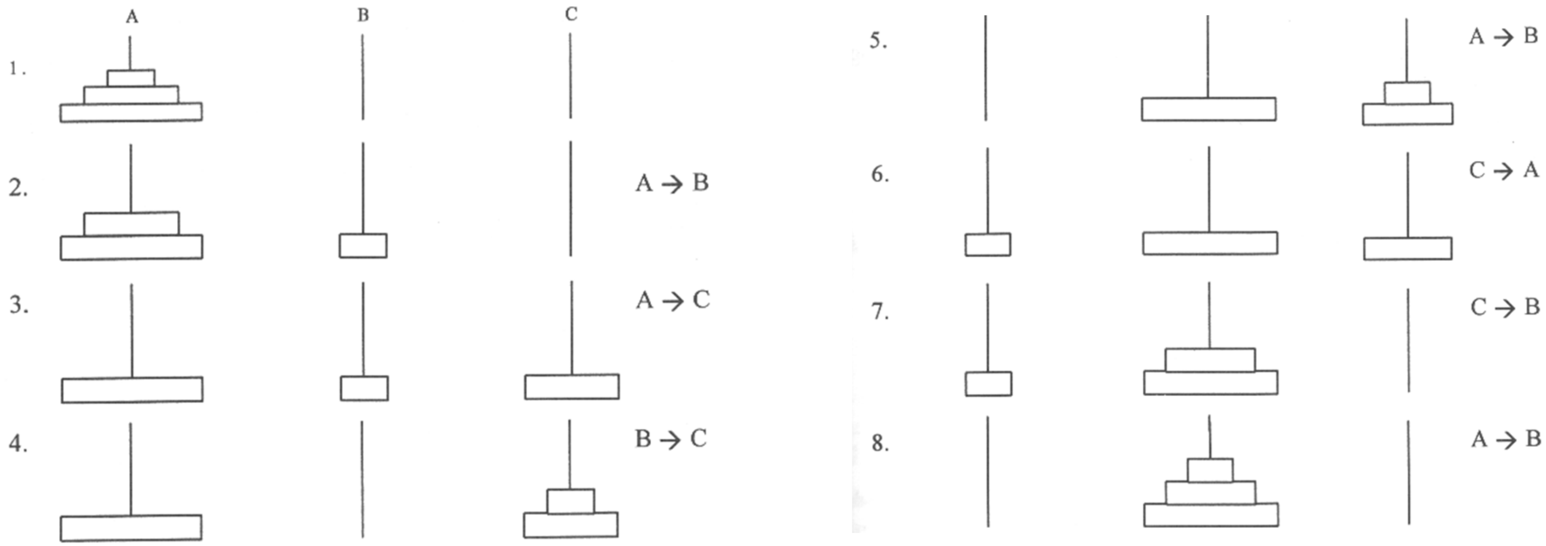
```
void vorR(int anz) {
  if ((anz > 0) && vorn_frei()) {
    vor();
    vorR(anz-1);
  }
}
```



Systematische Entwicklung von rekursiven Programmen

Beispiel 1: Türme von Hanoi

- In der Nähe von Hanoi sind Mönche damit beschäftigt insgesamt 64 unterschiedlich grossen Scheiben von einer Stange zur andern zu tragen. Dabei darf nie eine grössere Scheibe auf eine kleinere zu liegen kommen.
- Nach der Legende geht die Welt unter, wenn Sie mit Ihrer Arbeit fertig sind.



Vorgehen

- finde triviale Lösung (n = 1)
  - bewege Scheibe von A nach B
- Lösung für (n = 2)
  - bewege kleinere Scheibe von A nach C (Hilfsstange)
  - bewege grössere Scheibe von A nach B
  - bewege kleinere Scheibe von C (Hilfsstange) nach B
- Lösung für allgemeine n
  - bewege Stapel (n-1) von A nach C
  - bewege Scheibe von A nach B
  - bewege Stapel (n-1) von C nach B

Programm Beschreibung

```
void hanoi (int n, char from, char to, char help) {
    if (n == 1) {
        // bewege Scheibe von from nach to
    }
    else {
        // bewege Stapel n-1 von from auf help
        // bewege Scheibe von from nach to
        // bewege Stapel n-1 von help auf to
    }
}
```

weitere Vereinfachung

```
void hanoi (int n, char from, char to, char help) {
    if (n > 0) {
        // bewege Stapel n-1 von from auf help
        // bewege Scheibe von from nach to
        // bewege Stapel n-1 von help auf to
    }
}
```

Java Programm

```
void hanoi (int n, char from, char to, char help) {
   if (n > 0) {
      // bewege Stapel n-1 von from auf help
      hanoi(n-1,from,help,to);
      // bewege Scheibe von from nach to
      System.out.println("bewege "+from+" nach "+to);
      // bewege Stapel n-1 von help auf to
      hanoi(n-1,help,to,from);
   }
}

main {
  hanoi (3, 'A', 'B', 'C');
}
```

Rekursionstiefe, Zeitkomplexität, Speicherkomplexität

- **Rekursionstiefe**:
  - Maximale "Tiefe" der Aufrufe einer Methode minus 1
  - hanoi(3) -> hanoi(2) -> hanoi(1) -> hanoi(0) : Rekusionstiefe 3

- **Zeitkomplexität:** Rechenaufwand
  - Aufwand(n): 1 + 2 * Aufwand(n-1)
  - Aufwand(n-1): 1 + 2 * Aufwand(n-2)
  - Aufwand(n-2): 1 + 2 * Aufwand(n-3)
  - ……
  - Polynom: 2 * 2 * 2 * (n mal)… + ….
    - Wert des Polynoms für grosse Argumente durch Element mit dem grössten Exponenten bestimmt -> ~ 2^n

- Statt ~ schreibt man $O(2^n)$ -> **Aufwand ist exponentiell** (gross!!)
  - Anzahl Atome im Weltall (geschätzt): 10^{70}

- **Speicherkomplexität:** benötigter Speicher -> (auf Stack)

Übung

Uebung 1:

- Gegeben:
  - ein A4 Papier mit 0.1 mm Dicke
- Aufgabe:
  - Falten Sie dieses 32 mal und messen Sie die Dicke

Uebung 2:

- Ihr Programm löst das Problem der Türme von Hanoi mit 10 Scheiben in 0.1 sec.
- Wie lange dauert die Rechnung bei 30 Scheiben / bei 60 Scheiben?

Beispiel: Fibonacci-Zahlen

$$\text{fib(n)} = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ \text{fib(n-1) + fib(n-2)} & \text{sonst} \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12	..
fn	0	1	1	2	3	5	8	13	21	34	55	89	144	..

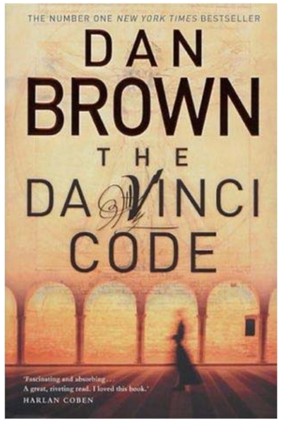
```
int fib(int n) {
    if (n==0) return 0;
    else if (n==1) return 1;
    else
      return fib(n-1) + fib(n-2);
}
```

Fibonacci-Zahlen

- Nach Leonardo Fibonacci (1170 – 1250)

- Erstmals erwähnt 450 oder 200 v. Chr.
- Kommen häufig in der Natur vor
- Zusammenhang mit dem
"Goldenen Schnitt"

- Interessanter Link:
http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html



Beispiel: Ackermannfunktion

$$\texttt{ack(n,m)} = \begin{cases} \texttt{m + 1} & \texttt{falls n = 0} \\ \texttt{ack(n-1,1)} & \texttt{falls m = 0} \\ \texttt{ack(n-1,ack(n,m-1))} & \texttt{sonst} \end{cases}$$

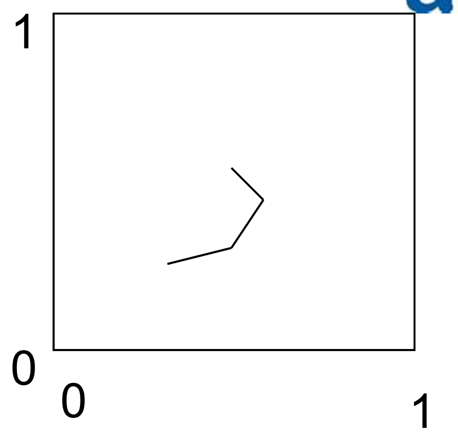
```
int ack(int n, int m) {
  if (n == 0) return m+1;
  else if (m == 0) return ack(n-1, 1);
  else return ack(n-1, ack(n,m-1));
}
```

Die Ackermannfunktion wächst sehr stark:

ack(4,2) besitzt 19729 Stellen

ack(4,4) ist größer als 10 hoch 10 hoch 10 hoch 19000

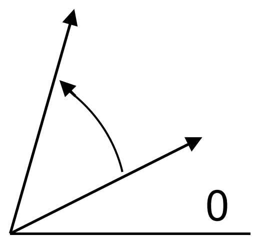
Beispiel:
Rekursiv definierte Kurven / Fraktale



Einschub: Schildkröten-Graphik

"Turtle" bewegt sich vorwärts

"Turtle" dreht sich um Winkel



in Grad

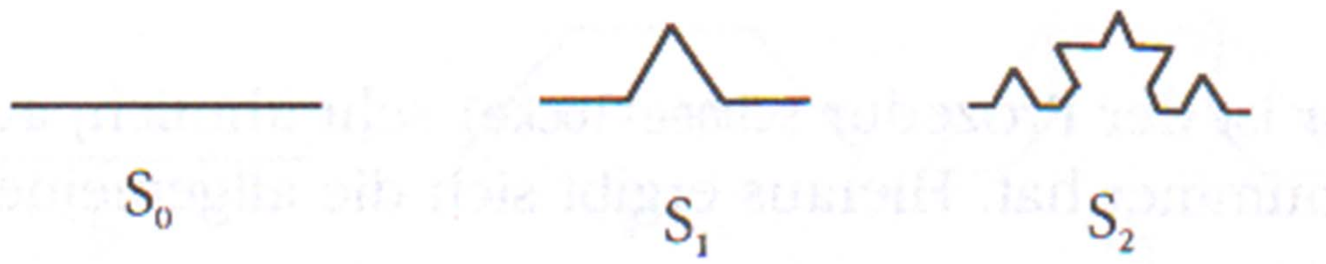
\+ Gegenuhrzeiger

0 bei 3 Uhr

```
class Turtle {
   double x, y;
   bewege(double distanz);
   drehe(double winkel):
}
```

Schneeflockenkurve

- die Strecke wird dreigeteilt; der mittlere Teil wird durch die zwei Seiten eines gleichseitigen Dreiecks ersetzt

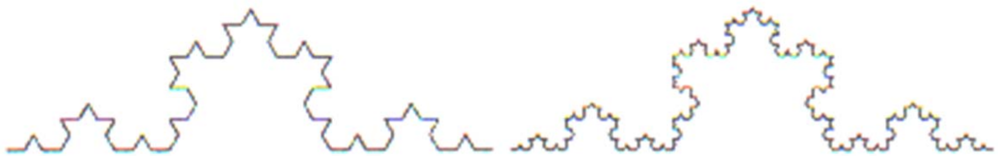



```
turte.bewege(dist);
```

```
dist = dist/3;
turtle.bewege(dist);
turtle.drehe(60);
turtle.bewege(dist);
turtle.drehe(-120);
turtle.bewege(dist);
turtle.drehe(60);
turtle.bewege(dist);
```

Java Programm für Schneeflocke

```
void schneeflocke(int stufe, double dist) {
if (stufe == 0) {
    turtle.bewege(dist)
else {
    stufe--;
    dist = dist/3;
    schneeflocke(stufe,dist);
    turtle.drehe(60);
    schneeflocke(stufe,dist);
    turtle.drehe(-120);
    schneeflocke(stufe,dist);
    turtle.drehe(60);
    schneeflocke(stufe,dist);
}
}
```



Beispiel: Parser für mathematische Ausdrücke

- Beispiele mathematischer Ausdrücke
  - 3 + 3 + 3
  - 3 + 3*2 + 2
  - 3 * (2 + 3)
- Grammatik dazu:
  - Expression ::= Term (("+" | "-") Term)*
  - Term ::= Factor (("*" | "/") Factor)*
  - Factor:: = Number | "(" Expression ")"

Parser für mathematische Ausdrücke: Expression

Expression ::= Term (("+" | "-") Term)*

```
static double expression() {
   double val = term();
    while (sym == '+' || sym == '-') {
     char s = sym;
     sym = getSym();
     double v = term();
     if (s=='+') val += v;
     else val -= v;
   }
   return val;
}
```

Parser für mathematische Ausdrücke: Term

Term ::= Factor (("*" | "/") Factor)*

```
static double term() {
   double val = factor();
   while (sym == '*' || sym == '/') {
      char s = sym;
      sym = getSym();
      double v = factor();
      if (s=='*') val *= v;
      else val /= v;
   }
   return val;
}
```

Parser für mathematische Ausdrücke: Factor

Factor:: = Number | "(" Expression ")"

```
static double factor() {
   double val = 0.0;
   if (sym == '(') {
      sym = getSym();
      val = expression();
      sym = getSym(); // ')'
   }
   else if (sym == NUMBER) {
      val = symValue(sym);
      sym = getSym();
   }
   return val;
}
```

Beispiel: Parser für mathematische Ausdrücke

- Beispiele mathematischer Ausdrücke
  - 3 + 3 + 3 oder 3 + 3*2 + 2 oder 3 * (2 + 3)
- Grammatik dazu:
  - Expression ::= Term (("+" | "-") Term)*
  - Term ::= Factor (("*" | "/") Factor)*
  - Factor:: = Number | "(" Expression ")"
- der Parser dazu

getSym() liefert das nächste Symbol
sym enthält das zuletzt eingelesene Symbol

```
static double expression() {
   double val = term();
   while (sym == '+' || sym == '-') {
      char s = sym;
      sym = getSym();
      double v = term();
      if (s=='+')val+=v; else val-=v;
   }
   return val;
}

static double term() {
   double val = factor();
   while (sym == '*' || sym == '/') {
      char s = sym;
      sym = getSym();
      double v = factor();
      if (s=='*') val*=v; else val/=v;
   }
   return val;
}
```

```
static double factor() {
   double val = 0.0;
   if (sym == '(') {
      sym = getSym();
      val = expression();
      sym = getSym(); // ')'
   }
   else if (sym == NUMBER) {
      val = symValue(sym);
      sym = getSym();
   }
   return val;
}

public static void main(String[] s) {
   sym = getSym();  // lesen des 1.
                    // Symbols
   double val = expression();
}
```

Rekursive Methoden Zusammenfassung

- Anmerkungen:
  - zu jedem rekursiv formulierten Algorithmus gibt es einen äquivalenten iterativen Algorithmus
- Vorteile rekursiver Algorithmen:
  - kürzere Formulierung
  - leichter verständliche Lösung
  - teilweise sehr effiziente Problemlösungen (z.B. Quicksort später)
- Nachteile rekursiver Algorithmen:
  - weniger effizientes Laufzeitverhalten (Overhead beim Funktionsaufruf)
  - Verständnisprobleme bei Programmieranfängern
  - Konstruktion rekursiver Algorithmen „gewöhnungsbedürftig“