11. Rekursion

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration, Lindenmayer Systeme

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich rekursiv definierbar.
- Das heisst, die Funktion erscheint in ihrer eigenen Definition.

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

Rekursion in Java: Genauso!

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n-1)!, & \text{andernfalls} \end{cases}$$

```
// POST: return value is n!
public static int fac (int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fac (n-1);
}
```


Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife. . .
- . . . nur noch schlechter (“verbrennt” Zeit und Speicher)

```
private static void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

„n wird mit jedem Aufruf kleiner."

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

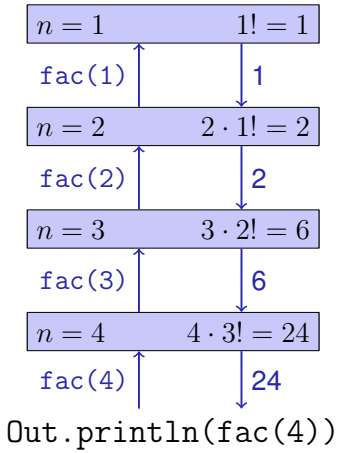
```
// POST: return value is n!
public static int fac (int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments: $n = 4$
Rekursiver Aufruf mit Argument $n - 1 == 3$

Der Aufrufstapel

Bei jedem Methodenaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion:

$$\gcd(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in Java

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
public static int gcd (int a, int b)
{
  if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$$

Fibonacci-Zahlen in Java

Laufzeit

`fib(50)` dauert „ewig”, denn es berechnet F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal, F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
public static int fib (int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib (n-1) + fib (n-2); // n > 1
}
```

Korrektheit und Terminierung sind klar.

Schnelle Fibonacci-Zahlen

Idee:

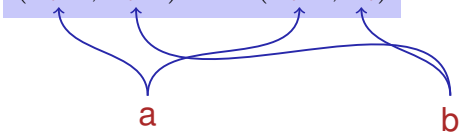
- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!
- Berechne die nächste Zahl als Summe von `a` und `b`!

Schnelle Fibonacci-Zahlen in Java

```
public static int fib (int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  int a = 1;  // F_1
  int b = 1;  // F_2
  for (int i = 3; i <= n; ++i){
    int a_old = a;  // F_i-2
    a = b;                    // F_i-1
    b += a_old;               // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

sehr schnell auch bei `fib(50)`

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$



Rekursion und Iteration

Rekursion kann *immer* simuliert werden durch

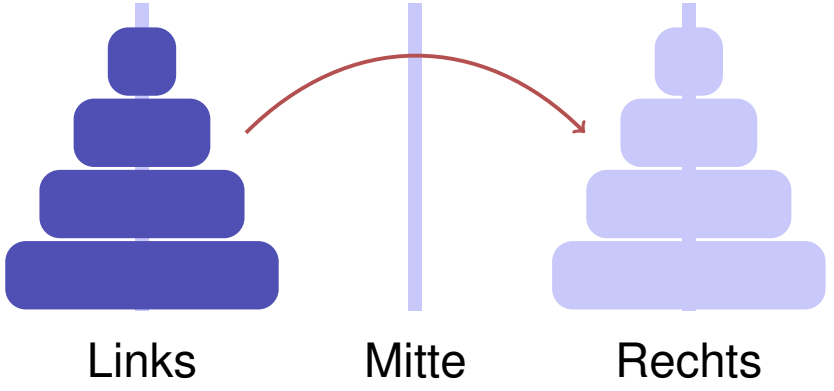
- Iteration (Schleifen)
- expliziten „Aufrufstapel” (z.B. Feld).

Oft sind rekursive Formulierungen einfacher, aber manchmal auch weniger effizient.

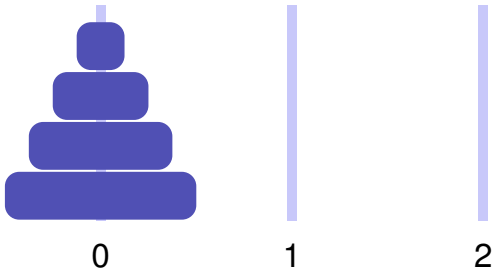
Die Macht der Rekursion

- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich einfacher lösbar.
- Beispiele: *Die Türme von Hanoi*, das n-Damen-Problem, Parsen von Ausdrücken, Sudoku-Löser, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren) $\rightarrow$ Informatik II

Experiment: Die Türme von Hanoi



Die Türme von Hanoi - Code



Bewege 4 Scheiben von 0 nach 2 mit Hilfsstapel 1:

```
Move(4, 0, 1, 2);
```


Die Türme von Hanoi - Code

```
Move(4, 0, 1, 2);
==
```

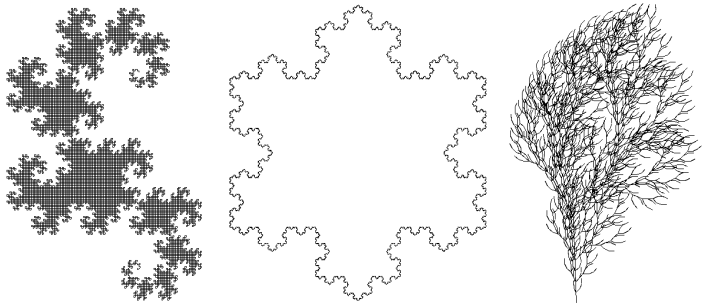
1. Bewege 3 Scheiben von 0 nach 1 mit Hilfsstapel 2:
   `Move(3, 0, 2, 1);`
2. Bewege 1 Scheibe von 0 nach 2
   `Move(1, 0, 1, 2);`
3. Bewege 3 Scheiben von 1 nach 2 mit Hilfsstapel 0
   `Move(3, 1, 0, 2);`

Die Türme von Hanoi - Code

```
public static void Move(int n, int source, int aux, int dest){
  if (n==1){
    Out.println("move " + source + "->" + dest);
  } else {
    Move(n-1, source, dest, aux);
    Move(1, source, aux, dest);
    Move(n-1, aux, source, dest);
  }
}
```


Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



L-Systeme wurden vom ungarischen Biologen Aristid Lindenmayer (1925–1989) zur Modellierung von Pflanzenwachstum erfunden.

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ
- Produktion $P : \Sigma \to \Sigma^*$
- Startwort $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
c	$P(c)$
F	F + F +
+	+
−	−
- F

Definition

Das Tripel $\mathcal{L} = (\Sigma, P, s_0)$ ist ein L-System.

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$ $\quad$ $w_0 := \mathrm{F}$

$w_1 := P(w_0)$ $\quad$ $w_1 := \mathrm{F} + \mathrm{F} +$

$w_2 := P(w_1)$ $\quad$ $w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$

$P(\mathrm{F})P(+)P(\mathrm{F})P(+)$

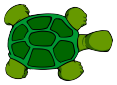
$\vdots$ $\quad$ $\vdots$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1)P(c_2)\ldots P(c_n)$

Turtle-Grafik

Schildkröte mit Position und Richtung

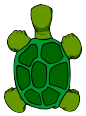


Schildkröte versteht 3 Befehle:

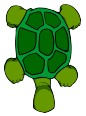
F: Gehe einen Schritt vorwärts ✓

Spur

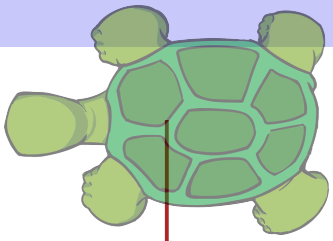
+: Drehe dich um 90 Grad ✓



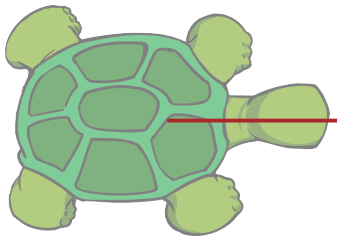
−: Drehe dich um −90 Grad ✓



Wörter zeichnen!



$w_1 = \mathrm{F} + \mathrm{F} +$ ✓



lindenmayer: Hauptprogramm

Wort $w_0 \in \Sigma^*$:

```
public static void main(String[] args){
    Out.print("Maximal Recursion Depth = ");
    int depth = In.readInt();
    Turtle t = new Turtle();
    produce(t,"F",depth);
    t.show();
}
```

$w = w_0 = \mathrm{F}$

lindenmayer: Produktion

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
static void produce (Turtle turtle, String word, int depth){
  if (depth > 0) {
    for (int k = 0; k < word.length(); ++k){
      produce(turtle, replace(word.charAt(k)), depth-1);
    }
  } else {
    draw(turtle,word);
  }
}
```

$w = w_i \rightarrow w = w_{i+1}$

Zeichne $w = w_n$!

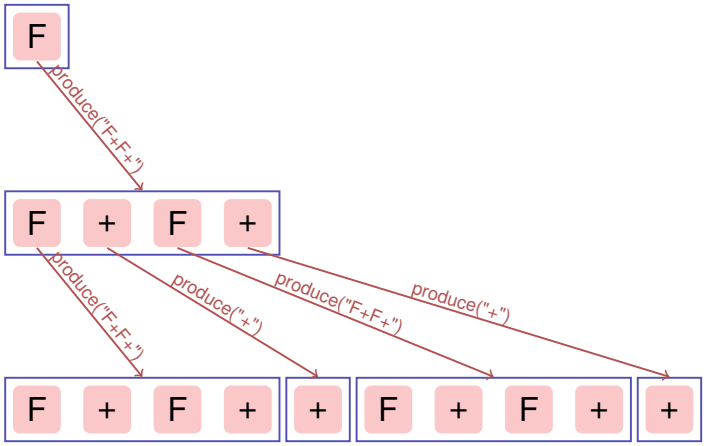
lindenmayer: replace

```
// POST: returns the production of c
static String replace (char c){
  switch (c) {
  case 'F':
    return "F+F+";
  default:
    return Character.toString(c); // trivial production c -> c
  }
}
```


lindenmayer: draw

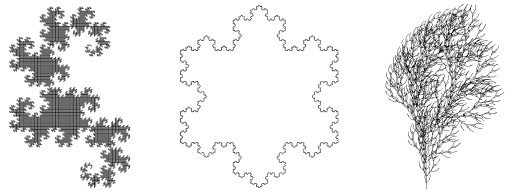
```
// POST: draws the turtle graphic interpretation of word
static void draw (Turtle turtle, String word) {
  for (int k = 0; k < word.length(); ++k){
    switch (word.charAt(k)) {
    case 'F':
      turtle.forward(1); // move one step forward
      break;
    case '+':
      turtle.left(90); // turn counterclockwise by 90 degrees
      break;
    }
  }
}
```

Die Rekursion



L-Systeme: Erweiterungen

- Beliebige Symbole ohne grafische Interpretation (dragon)
- Beliebige Drehwinkel (snowflake)
- Sichern und Wiederherstellen des Schildkröten-Zustandes → Pflanzen (bush)



Challenge: Wir freuen uns auf Ihre Beiträge