Rekursion

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, d.h.
- die Funktion erscheint in ihrer eigenen Definition.

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar, d.h.
- die Funktion erscheint in ihrer eigenen Definition.

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

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel

Rekursion in C++

- modelliert oft direkt die mathematische Rekursionsformel.

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

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

Unendliche Rekursion

- ist so leicht zu erzeugen wie eine unendliche Schleife,
- sollte aber genauso vermieden werden.

```
void f()
{
  f(); // calls f(), calls f(), calls f(),...
}
```

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung in jedem rekursiven Aufruf.

`fac(n) :`

terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Parameter $< n$ aufgerufen.

Terminierung von rekursiven Funktionsaufrufen

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung in jedem rekursiven Aufruf.

`fac(n) :`

terminiert sofort für $n \leq 1$, andernfalls wird

die Funktion rekursiv mit Parameter $< n$

aufgerufen.

"n wird mit jedem Aufruf kleiner."

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

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

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Parameters mit dem Wert des Aufrufparameters

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Auswertung des Rückgabeausdrucks

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Ausführen des Funktionsrumpfs: Rekursiver Aufruf von fac mit Aufrufparameter **`n-1 == 2`**

Auswertung rekursiver Funktionsaufrufe (z.B. `fac(3)`)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Parameters mit dem Wert des Aufrufparameters

Auswertung rekursiver Funktionsaufrufe (z.B. **`fac(3)`**)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei! Den von **`fac (3)`**, und den von **`fac (2)`**

Initialisierung *des* formalen Parameters mit dem Wert des Aufrufparameters

Auswertung rekursiver Funktionsaufrufe (z.B. **`fac(3)`**)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Wir nehmen den Parameter des *aktuellen* Aufrufs, **`fac (2)`**

Initialisierung *des* formalen Parameters mit dem Wert des Aufrufparameters

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufparameters kommt auf einen Stapel (erst n = 3, dann n = 2,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende des Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

Der Aufrufstapel

Bei Auswertung jedes Funktionsaufrufs:

- Wert des Aufrufparameters kommt auf einen Stapel (erst n = 3, dann n = 2,...)
- es wird stets mit dem obersten Wert gearbeitet
- nach Ende eines Funktionsaufrufs wird der oberste Wert vom Stapel gelöscht

Bei mehreren Parametern analog (alle kommen auf den Stapel, wir arbeiten jeweils mit der obersten Version jedes Parameters)

Der Aufrufstapel - Beispiel

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

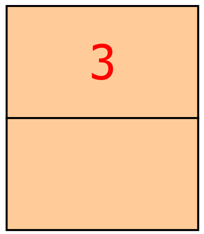
Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

fac(3)

Auswertung von **fac(3)** beginnt;

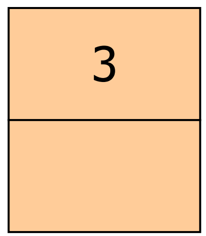
Aufrufparameter kommt auf Stapel



Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

fac(3)



Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

fac(2)

Auswertung von **`fac(2)`** beginnt;

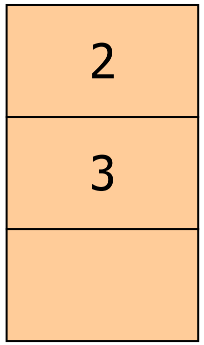
Aufrufparameter kommt auf Stapel

2
3

Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 2
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

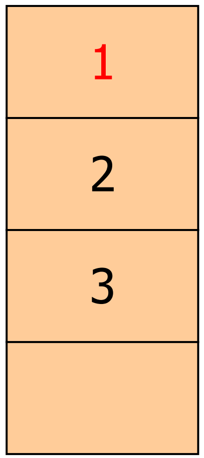
fac(2)



Der Aufrufstapel - Beispiel

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

fac(1)



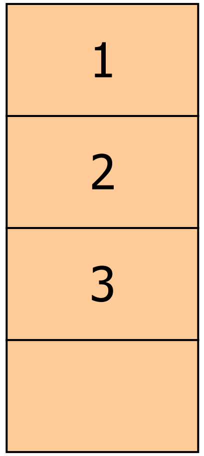
Auswertung von **`fac(1)`** beginnt;

Aufrufparameter kommt auf Stapel

Der Aufrufstapel - Beispiel

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

fac(1)



Der Aufrufstapel - Beispiel

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

fac(2)

fac(1) ausgewertet, Ergebnis 1;

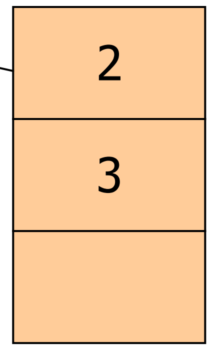
lösche obersten Wert vom Stapel

2
3

Der Aufrufstapel - Beispiel

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

fac(2)



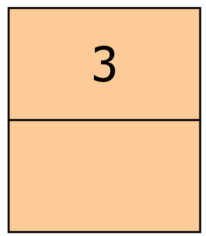
Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 1
  if (n <= 1) return 1;
  return n * 2       ; // n > 1
}
```

fac(3)

fac(2) ausgewertet, Ergebnis 2;

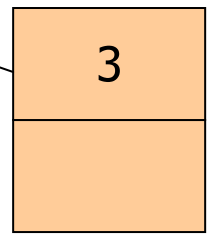
lösche obersten Wert vom Stapel



Der Aufrufstapel - Beispiel

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 1
  if (n <= 1) return 1;
  return n * 2       ; // n > 1
}
```

fac(3)



Der Aufrufstapel - Beispiel

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

`fac(3)` ausgewertet, Ergebnis 6;

lösche obersten Wert vom Stapel

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler *gcd (a, b)* zweier natürlicher Zahlen *a* und *b*

Grösster gemeinsamer Teiler

Euklidischer Algorithmus

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

$gcd\,(a, b) = gcd\,(b, a \bmod b)$ für $b > 0$.

Grösster gemeinsamer Teiler

gcd (a, b) = gcd (b, a mod b) für *b > 0* .

Beweis: Sei *k* Teiler von *b*. Aus

$$a = (a \text{ div } b)\, b + a \bmod b$$

folgt

$$a / k = (a \text{ div } b)\, \underbrace{b / k}_{\text{ganzzahlig!}} + (a \bmod b) / k$$

Grösster gemeinsamer Teiler

gcd (a, b) = gcd (b, a mod b) für $b > 0$.

Beweis: Sei *k* Teiler von *b*. Aus

$$a = (a \text{ div } b)\, b + a \bmod b$$

folgt

$$a / k = (a \text{ div } b)\, \underbrace{b / k}_{\text{ganzzahlig!}} + (a \bmod b) / k$$

ganzzahlig $\longleftrightarrow$ ganzzahlig

Grösster gemeinsamer Teiler

$gcd(a, b) = gcd(b, a \bmod b)$ für $b > 0$.

Beweis: Sei k Teiler von b. Aus

$$a = (a \operatorname{div} b)\, b + a \bmod b$$

folgt

k teilt a genau dann, wenn k *auch* $(a \bmod b)$ teilt.

Grösster gemeinsamer Teiler

$gcd\,(a, b) = gcd\,(b, a \bmod b)$ für $b > 0$.

Beweis: Sei *k* Teiler von *b*. Aus

$$a = (a \operatorname{div} b)\, b + a \bmod b$$

folgt

k teilt *a* genau dann, wenn *k auch (a mod b)* teilt.

k ist gemeinsamer Teiler von *a* und *b* genau dann, wenn k gemeinsamer Teiler von *a mod b* und *b* ist.

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit: $gcd(a, 0) = a$

$gcd(a, b) = gcd(b, a \bmod b), \quad b > 0$

Lemma!

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Terminierung: $a \text{ mod } b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Grösster gemeinsamer Teiler

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Korrektheit *und* Terminierung müssen bei rekursiven Funktionen stets separat bewiesen werden!

Fibonacci-Zahlen

- $F_0 := 0$
- $F_1 := 1$
- $F_n := F_{n-1} + F_{n-2}\,,\ n > 1$

Fibonacci-Zahlen

- $F_0 := 0$
- $F_1 := 1$
- $F_n := F_{n-1} + F_{n-2}\, ,\; n > 1$

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned 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.

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Laufzeit:

Fibonacci-Zahlen

```
// POST: return value is the n-th
//       Fibonacci number F_n
unsigned int fib (unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

Laufzeit:

`fib (50)` berechnet

F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal,

F_{45} 8-mal, F_{44} 13-mal, ...

Rekursion und Iteration

Rekursion kann im Prinzip ersetzt werden durch

- Iteration (Schleifen) und
- expliziten Aufrufstapel.

Oft sind direkte rekursive Formulierungen ein-facher, aber manchmal auch weniger effizient.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

Endrekursion

Eine Funktion ist endrekursiv, wenn sie genau einen rekursiven Aufruf ganz am Ende enthält.

```
// POST: return value is the greatest common
//       divisor of a and b
unsigned int gcd (unsigned int a, unsigned int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}
```

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"$(a,b) \;:=\; (b, a \bmod b)$"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b) := (b, a mod b)"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"(a,b)   := (b, a mod b)"

Endrekursion

- ist leicht iterativ zu schreiben.

```
// POST: return value is the greatest common divisor of a and b
unsigned int gcd2 (unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int a_prev = a;
    a = b;
    b = a_prev % b;
  }
  return a;
}
```

"*(a,b) := (b, a mod b)*"

...aber die rekursive Version is lesbarer und (fast) genauso effizient.

Fibonacci-Zahlen iterativ

Idee:

- berechne jede Zahl genau einmal, in der Reihenfolge $F_0, F_1, F_2, F_3, \ldots$
- speichere die jeweils letzten beiden berechneten Zahlen (Variablen **`a, b`**), dann kann die nächste Zahl durch eine Addition erhalten werden.

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

Fibonacci-Zahlen iterativ

```
// POST: return value is the n-th Fibonacci number F_n
unsigned int fib2 (unsigned int n)
{
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a;  // F_{i-2}
    a = b;                    // F_{i-1}
    b += a_prev;              // F_{i-1} += F_{i-2} -> F_i
  }
  return b;
}
```

“$(F_{i-2}, F_{i-1}) := (F_{i-1}, F_i)$”

a b

Die Ackermann-Funktion

$$A\,(m, n) = \begin{cases} n + 1\,, & \text{falls } m = 0 \\ A\,(m-1,\, 1)\,, & \text{falls } m > 0,\ n = 0 \\ A\,(m-1,\, A\,(m,\, n-1))\,, & \text{falls } m > 0,\ n > 0 \end{cases}$$

Die Ackermann-Funktion

$$A(m, n) = \begin{cases} n + 1, & \text{falls } m = 0 \\ A(m-1, 1), & \text{falls } m > 0, n = 0 \\ A(m-1, A(m, n-1)), & \text{falls } m > 0, n > 0 \end{cases}$$

- ist *berechenbar*, aber *nicht primitiv rekursiv* (man dachte Anfang des 20. Jahrhunderts, dass es diese Kombination gar nicht gibt)
- wächst *extrem* schnell

Die Ackermann-Funktion

```
// POST: return value is the Ackermann function
//       value A(m,n)
unsigned int A (unsigned int m, unsigned int n) {
  if (m == 0) return n+1;
  if (n == 0) return A(m-1,1);
  return A(m-1, A(m, n-1));
}
```

Die Ackermann-Funktion

	0	1	2	3	...	*n*
0	1	2	3	4	...	*n+1*
1						
2						
3						
4						

m

n

Die Ackermann-Funktion

m \ n	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2						
3						
4						

Die Ackermann-Funktion

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3						
4						

m

n

Die Ackermann-Funktion

	0	1	2	3	...	*n*
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4						

m

n

Die Ackermann-Funktion

	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{2}}}}-3$

m

n

Turm von n+3 Zweierpotenzen

Die Ackermann-Funktion

m \ n	0	1	2	3	...	n
0	1	2	3	4	...	$n+1$
1	2	3	4	5	...	$n+2$
2	3	5	7	9	...	$2n+3$
3	5	13	29	61	...	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	...	$2^{2^{\cdot^{\cdot^{2}}}}-3$

nicht mehr praktisch berechenbar

Die Ackermann-Funktion - Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

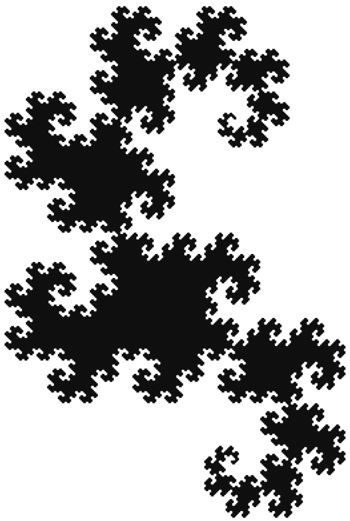
Die Ackermann-Funktion - Moral

- Rekursion ist sehr mächtig...
- ... aber auch gefährlich:

Es ist leicht, harmlos aussehende rekursive Funktionen hinzuschreiben, die theoretisch korrekt sind und terminieren, praktisch aber jeden Berechenbarkeitsrahmen sprengen.

Lindenmayer-Systeme: Zeichnen von Fraktalen

(Beispiel für Mächtigkeit der Rekursion)



Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*

Beispiel:

P(F) = F+F+

P(+) = +

P(-) = -

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*
- *s* aus Σ^* ein *Startwort* (Beispiel: F)

Lindenmayer-Systeme: Definition

- *Alphabet* Σ (Beispiel: {F, +, -})
- Σ^* = Menge aller endlichen Wörter über Σ (Beispiel: F+F+ ist in Σ^*)
- P: Σ -> Σ^* eine *Produktion*
- *s* aus Σ^* ein *Startwort* (Beispiel: F)

Def.: (Σ, P, s) ist *Lindenmayer-System*.

Lindenmayer-Systeme: Die beschriebenen Wörter

Die von (Σ, P, s) *beschriebenen* Wörter:

- s (F)
- P^1 (s) (F+F+)
- P^2 (s) (F+F++F+F++)
- P^3 (s) (F+F++F+F+++F+F++F+F+++)
- ...

Lindenmayer-Systeme: Die beschriebenen Wörter

Die von (Σ, P, s) *beschriebenen* Wörter:

- s (F)
- P^1 (s) (F+F+)
- P^2 (s) (F+F++F+F++)
- P^3 (s) (F+F++F+F+++F+F++F+F+++)
- ...

P^i (s) entsteht aus P^{i-1} (s) durch Ersetzen aller Symbole mittels P.

Lindenmayer-Systeme: Die beschriebenen Wörter

Die von (Σ, P, s) *beschriebenen* Wörter:

- s (F)
- $P^1(s)$ (F+F+)
- $P^2(s)$ (F+F++F+F++)
- $P^3(s)$ (F+F++F+F+++F+F++F+F+++)
- ...

$P^i(s)$ entsteht aus $P^{i-1}(s)$ durch Ersetzen aller Symbole mittels P.

F $\longrightarrow$ F+F+ + $\longrightarrow$ + - $\longrightarrow$ -

Lindenmayer-Systeme: Turtle-Grafik

Turtle-Grafik:

- Schildkröte mit *Position* und *Richtung*

Lindenmayer-Systeme: Turtle-Grafik

Turtle-Grafik:

- Schildkröte mit *Position* und *Richtung*

- versteht folgende Kommandos:
  - F: gehe einen Schritt in deine Richtung (und markiere ihn in Schwarz)
  - + / - : drehe dich um 90^{o} gegen / im UZS

Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

F+F+

Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

F+F+

Lindenmayer-Systeme: Turtle-Grafik

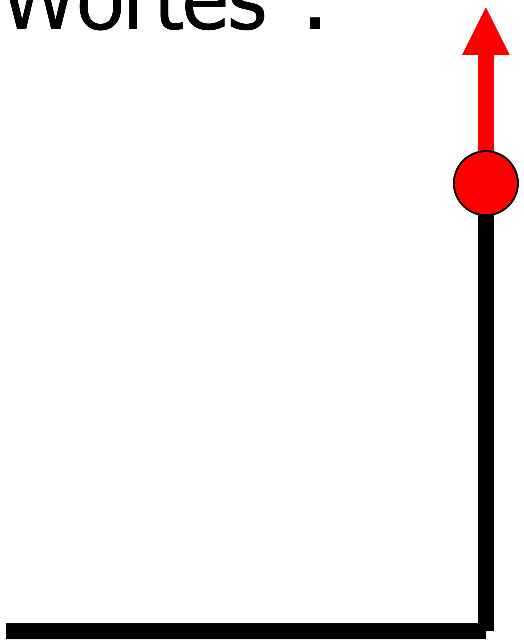
"Zeichnen eines Wortes":

F+F+

Lindenmayer-Systeme: Turtle-Grafik

“Zeichnen eines Wortes”:

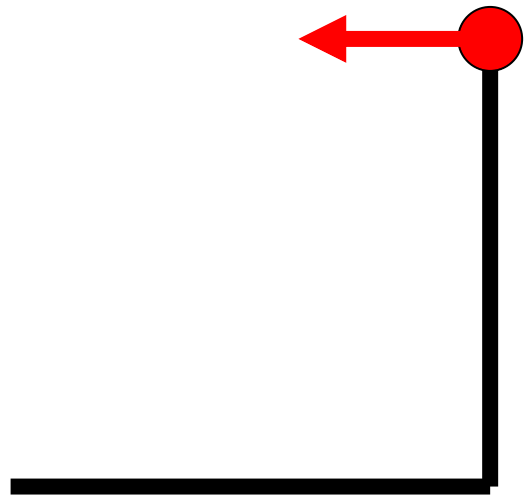
F+F+



Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

F+F+



Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

Im Beispiel: $s = F$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

- s (F)
- $P^1(s)$ (F+F+)
- $P^2(s)$ (F+F++F+F++)
- $P^3(s)$ ($\underbrace{F+F++F+F++}_{P^2(F)}\underbrace{+}_{P^2(+)}\underbrace{F+F++F+F++}_{P^2(F)}\underbrace{+}_{P^2(+)}$)

$P^3(F)$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

- s (F)
- $P^1(s)$ (F+F+)
- $P^2(s)$ (F+F++F+F++)
- $P^3(s)$ ($\underbrace{F+F++F+F++}_{P^2(F)}\underbrace{+}_{P^2(+)}\underbrace{F+F++F+F++}_{P^2(F)}\underbrace{+}_{P^2(+)}$)

$P^3(F)$

$P^2(F)$ $P^2(+)$ $P^2(F)$ $P^2(+)$

$\boxed{F + F + \; = P(F)}$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

- s (F)
- $P^1(s)$ (F+F+)
- $P^2(s)$ (F+F++F+F++)
- $P^3(s)$ ($\underbrace{\text{F+F++F+F++}}\,\text{+}\,\underbrace{\text{F+F++F+F++}}\,\text{+}$)

$P^3(F)$ $P^2(F)$ $P^2(+)$ $P^2(F)$ $P^2(+)$

Allgemein:

Für alle σ aus Σ gilt: $P^i(\sigma) = P^{i-1}(P(\sigma)) = P^{i-1}(\sigma_1 \ldots \sigma_k) = P^{i-1}(\sigma_1) \ldots P^{i-1}(\sigma_k)$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

- s (F)
- P^1 (s) (F+F+)
- P^2 (s) (F+F++F+F++)
- P^3 (s) ($\underbrace{\text{F+F++F+F++}}_{P^2(F)}\;\underset{P^2(+)}{+}\;\underbrace{\text{F+F++F+F++}}_{P^2(F)}\;\underset{P^2(+)}{+}$)

P^3 (F)

Allgemein:

Rekursion!

Für alle σ aus Σ gilt: $P^i(\sigma) = P^{i-1}(P(\sigma)) = P^{i-1}(\sigma_1 \dots \sigma_k) = P^{i-1}(\sigma_1) \dots P^{i-1}(\sigma_k)$

Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von $w_i := P^i(s)$:

- $s = s_1 \ldots s_k$.
- $w_i = P^i(s) = P^i(s_1 \ldots s_k) = P^i(s_1) \ldots P^i(s_k)$
  $=: w_i(s_1) \ldots w_i(s_k)$

- Zeichnungen der k Wörter $w_i(s_1) \ldots w_i(s_k)$ erfolgen rekursiv wie gerade hergeleitet.
- Die k Zeichnungen zusammen ergeben w_i.

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

Zeichnen von $w_i := P^i(s)$:

- $s = F$.
- $w_i = P^i(s) = P^i(s_1)$

  $=: w_i(s_1) = w_i(F)$

- Zeichnung des Wortes $w_i(F)$ erfolgt rekursiv wie gerade hergeleitet:

  $w_i(F) = w_{i-1}(F)\, w_{i-1}(+)\, w_{i-1}(F)\, w_{i-1}(+)$

  $= w_{i-1}(F)+w_{i-1}(F)+$

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

```
// POST: the word w_i^F is drawn
void f (unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
  }
}
```

$w_i(F) = w_{i-1}(F)+w_{i-1}(F)+$

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

```
// POST: the word w_i^F is drawn
void f (unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
    f(i-1);          // w_{i-1}^F
    ifm::left(90);   // +
  }
}
```

Befehle für Turtle-Grafik (aus der **`libwindow`**-Bibliothek)

$w_i(F) = w_{i-1}(F)+w_{i-1}(F)+$

Lindenmayer-Systeme: Rekursives Zeichnen (Beispiel)

Programm **`lindenmayer.C`** :

Lindenmayer-Systeme: Erweiterungen

Neue Symbole (ohne Interpretation in Turtle-Grafik):

Beispiel *Drachen*:

- s = X
- P(X) = X+YF+, P(Y) = -FX-Y

Lindenmayer-Systeme: Erweiterungen (Drachen)

```
// POST: w_i^X is drawn
void x (unsigned int i) {
  if (i > 0) {
    x(i-1);             // w_{i-1}^X
    ifm::left(90);      // +
    y(i-1);             // w_{i-1}^Y
    ifm::forward();     // F
    ifm::left(90);      // +
  }
}
```

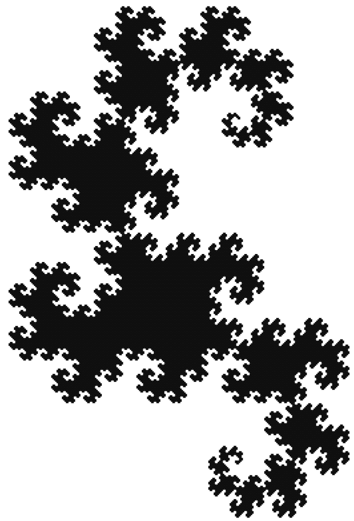
$w_i\,(X) = w_{i-1}\,(X)+w_{i-1}\,(Y)F+$

```
// POST: w_i^Y is drawn
void y (unsigned int i) {
  if (i > 0) {
    ifm::right(90);     // -
    ifm::forward();     // F
    x(i-1);             // w_{i-1}^X
    ifm::right(90);     // -
    y(i-1);             // w_{i-1}^Y
  }
}
```

$w_i\,(Y) = -w_{i-1}\,(F)X-w_{i-1}\,(Y)$

Lindenmayer-Systeme: Drachen

Programm **`dragon.C`** :



Lindenmayer-Systeme: Erweiterungen

Drehwinkel α kann frei gewählt werden.

Beispiel *Schneeflocke*:

- $\alpha = 60^o$
- s = F++F++F
- P(F) = F-F++F-F

Lindenmayer-Systeme: Schneeflocke

```
// POST: the word w_i^F is drawn
void f (unsigned int i) {
  if (i == 0)
    ifm::forward();  // F
  else {
    f(i-1);          // w_{i-1}^F
    ifm::right(60);  // -
    f(i-1);          // w_{i-1}^F
    ifm::left(120);  // ++
    f(i-1);          // w_{i-1}^F
    ifm::right(60);  // -
    f(i-1);          // w_{i-1}^F
  }
}
```

$w_i\,(F) = w_{i-1}\,(F)-w_{i-1}\,(F)++w_{i-1}\,(F)-w_{i-1}\,(F)$

Lindenmayer-Systeme: Schneeflocke

Programm **`snowflake.C`** :