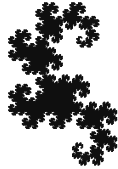


Lindenmayer-Systeme: Fraktale rekursiv zeichnen



Lindenmayer-Systeme: Definition

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

- $w_0 = F$
- $w_1 = F+F+$
- $w_2 = F+F++F+F++$
- $w_3 = F+F+++F+F++++F+F++++F+F++++$
- ...

w_2 entsteht aus w_1 durch Ersetzen aller Symbole mittels P .

$F \rightarrow F+F+$ $+ \rightarrow +$ $- \rightarrow -$

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° 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":

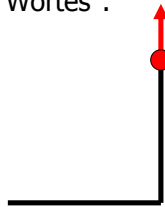
F+F+



Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

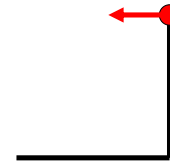
F+F+



Lindenmayer-Systeme: Turtle-Grafik

"Zeichnen eines Wortes":

F+F+



Lindenmayer-Systeme: Rekursives Zeichnen

Zeichnen von w_i :

Im Beispiel: $s = F$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von w_i :

- $w_0 = F$
- $w_1 = F+F+$
- $w_2 = F+F++F+F++$
- $w_3 = F+F+++F+F++++F+F++++F+F++++$
- ...

$$w_3 =: w_3(F) = w_2(F) w_2(+)$$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von w_i :

- $w_0 = F$
- $w_1 = F+F+$
- $w_2 = F+F++F+F++$
- $w_3 = \underbrace{F+F++F+F++++}_{F+F+} \underbrace{F+F++F+F++++}_{F+F+}$
- ...

$$w_i =: w_i(F) = w_{i-1}(F) \underbrace{w_{i-1}(+)}_+ w_{i-1}(F) \underbrace{w_{i-1}(+)}_+$$

Lindenmayer-Systeme: Die Rekursives Zeichnen

Zeichnen von w_i :

- $w_0 = F$
- $w_1 = F+F+$
- $w_2 = F+F++F+F++$
- $w_3 = \underbrace{F+F++F+F++++}_{F+F+} \underbrace{F+F++F+F++++}_{F+F+}$
- ...

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

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)

```
int main () {
    std::cout << "Number of iterations =? ";
    unsigned int n;
    std::cin >> n;

    // draw w_n = w_n(F)
    f(n);

    return 0;
}
```

Lindenmayer-Systeme: Erweiterungen

Neue Symbole (ohne Interpretation in Turtle-Grafik):

Beispiel *Drachen*:

- o $s = X$
- o $P(X) = X+YF+$, $P(Y) = -FX-Y$
- o $w_i = w_i(X) = w_{i-1}(X)+w_{i-1}(Y)F+$

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); // +
  }
}

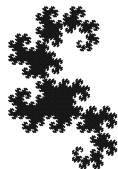
// POST: w_i^Y is drawn
void y (unsigned int i) {
  if (i > 0) {
    ifm::right(90); // -
    ifm::forward(); // F
    x(i-1); // w_{i-1}^X
    ifm::right(90); // -
    y(i-1); // w_{i-1}^Y
  }
}
```

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

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

Lindenmayer-Systeme: Drachen

Programm `dragon.C` :



Lindenmayer-Systeme: Erweiterungen

Drehwinkel α kann frei gewählt werden.

Beispiel *Schneeflocke*:

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

$$\underbrace{w_{i-1}(F)-w_{i-1}(F)++w_{i-1}(F)-w_{i-1}(F)}$$

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

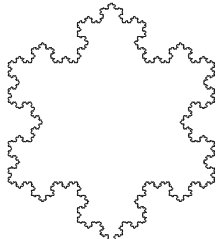
```
int main () {
    std::cout << "Number of iterations =? ";
    unsigned int n;
    std::cin >> n;

    // draw w_n = w_n^F ++ w_n^F ++ w_n^F
    f(n); // w_n^F
    ifm::left(120); // ++
    f(n); // w_n^F
    ifm::left(120); // ++
    f(n); // w_n^F

    return 0;
}
```

Lindenmayer-Systeme: Schneeflocke

Programm `snowflake.C` :



Structs und Referenztypen

Rechnen mit rationalen Zahlen

- o Rationale Zahlen (\mathbb{Q}) sind von der Form n/d , mit n und d aus \mathbb{Z}
- o C++ hat keinen "eingebauten" Typ für rationale Zahlen

Ziel: Wir bauen uns selbst einen C++ Typ für rationale Zahlen!

Rechnen mit rationalen Zahlen So könnte es aussehen

```
// Program: userational2.C
// Add two rational numbers.
#include <iostream>
#include "rational.h"

int main ()
{
    // input
    std::cout << "Rational number r:\n";
    rf::rational r;
    std::cin >> r;

    std::cout << "Rational number s:\n";
    rf::rational s;
    std::cin >> s;

    // computation and output
    std::cout << "Sum is " << r + s << ".\n";

    return 0;
}
```

Ein erstes Struct

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>
```

```
// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};
```

Invariante: Spezifiziert gültige Kombinationen von Werten (informell)

Ein `struct` definiert einen neuen Typ, dessen Wertebereich das *kartesische Produkt* der Wertebereiche existierender Typen ist (hier `int` \times `int`).

Ein erstes Struct

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>
```

```
// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};
```

Bedeutung: jedes *Objekt* des neuen Typs ist durch zwei Objekte vom Typ `int` repräsentiert, die die Namen `n` und `d` tragen.

Ein `struct` definiert einen *Typ*, keine *Variable*!

Ein erstes Struct : Funktionalität

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Mitglieds-Zugriff auf die
int- Objekte von a

Ein erstes Struct : Funktionalität

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

$$a + b = \frac{\text{Zähler}(a) \cdot \text{Nenner}(b) + \text{Nenner}(a) \cdot \text{Zähler}(b)}{\text{Nenner}(a) \cdot \text{Nenner}(b)}$$

Ein erstes Struct: Funktionalität

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Der neue Typ kann wie existierende
Typen benutzt werden, z.B. in

Variablendeklarationen...

Ein erstes Struct: Funktionalität

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Der neue Typ kann wie existierende
Typen benutzt werden, z.B. in

formalen Argumentlisten...

Ein erstes Struct: Funktionalität

```
// Program: userational.C
// Add two rational numbers.
#include <iostream>

// the new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

Der neue Typ kann wie existierende Typen benutzt werden, z.B. in Rückgabetypen...

Ein erstes Struct: Benutzung

```
int main ()
{
    // input
    std::cout << "Rational number r:\n";
    rational r;
    std::cout << " numerator =? "; std::cin >> r.n;
    std::cout << " denominator =? "; std::cin >> r.d;

    std::cout << "Rational number s:\n";
    rational s;
    std::cout << " numerator =? "; std::cin >> s.n;
    std::cout << " denominator =? "; std::cin >> s.d;

    // computation
    rational t = add (r, s);

    // output
    std::cout << "Sum is " << t.n << "/" << t.d << ".\n";

    return 0;
}
```

Struct-Definitionen

```
struct T {
    T1 name1;
    T2 name2;
    ...
    TN nameN;
};
```

Name des neuen Typs (Bezeichner)

Namen der Daten-Mitglieder (Bezeichner)

Wertebereich von T : $T1 \times T2 \times \dots \times TN$

Namen der zugrundeliegenden Typen

Struct-Definitionen: Beispiele

```
struct rational_vector_3 {
    rational x;
    rational y;
    rational z;
};
```

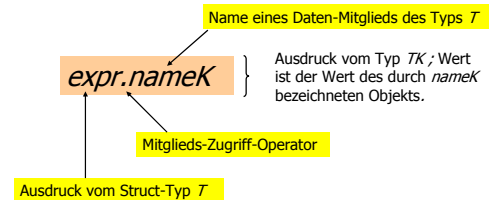
Die zugrundeliegenden Typen können fundamentale, aber auch benutzerdefinierte Typen sein.

Struct-Definitionen: Beispiele

```
struct extended_int {  
    // represents u if n==false and -u otherwise  
    unsigned int u; // absolute value  
    bool        n; // sign bit  
};
```

Die zugrundeliegenden Typen können natürlich auch **verschieden** sein.

Structs: Mitglieds-Zugriff



Structs: Initialisierung und Zuweisung

Initialisierung:

```
rational t = add (r, s);
```

```
"t.n = add (r, s).n;  
t.d = add (r, s).d;"
```

- o `t` wird mit dem Wert von `add (r, s)` initialisiert
- o Initialisierung erfolgt separat für jedes Daten-Mitglied

Structs: Initialisierung und Zuweisung

Default-Initialisierung:

```
rational t;
```

- o Daten-Mitglieder von `t` werden default-initialisiert
- o für Daten-Mitglieder fundamentaler Typen passiert nichts (Wert undefiniert)

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational t;  
t = add (r, s);
```

- o t wird default-initialisiert
- o der Wert von `add (r, s)` wird t zugewiesen (wieder separat für jedes Daten-Mitglied)

Structs und Felder

- o Felder können auch Mitglieder von Structs sein

```
struct rational_vector_3 {  
    rational v[3];  
};
```

- o Durch Verpacken in ein Struct kann das Kopieren von Feldern erreicht werden!

Structs und Felder

```
#include <iostream>  
  
struct point {  
    double coord[2];  
};  
  
int main()  
{  
    point p;  
    p.coord[0] = 1;  
    p.coord[1] = 2;  
  
    point q = p; // Hier wird ein Feld mit zwei Elementen kopiert  
    std::cout << q.coord[0] << " " // 1  
              << q.coord[1] << "\n"; // 2  
  
    return 0;  
}
```

Structs: Gleichheitstest?

Für jeden fundamentalen Typ gibt es die Vergleichsoperatoren `==` und `!=`, aber *nicht* für Structs! Warum?

- o Mitgliedsweiser Vergleich ergibt im allgemeinen keinen Sinn,...
- o ...denn dann wäre z.B. $2/3 \neq 4/6$

Benutzerdefinierte Operatoren

Anstatt

```
rational t = add (r, s);
```

würden wir lieber

```
rational t = r + s;
```

schreiben.

Dies geht mit *Operator-Überladung*.

Funktions- und Operator-Überladung

Verschiedene Funktionen können den *gleichen* Namen haben.

```
// POST: returns a * a
rational square (rational a);

// POST: returns a * a
extended_int square (extended_int a);
```

Der Compiler findet anhand der Aufruf-Argumente heraus, welche gemeint ist.

Operator-Überladung

- Operatoren sind spezielle Funktionen und können auch überladen werden
- Name des Operators *op* :
`operatorop`
- wir wissen schon, dass z.B. `operator+` für verschiedene Typen existiert

Additionsoperator für rationale Zahlen

Bisher:

```
// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
rational t = add (r, s);
```

Additionsoperator für rationale Zahlen

Neu:

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
rational t = r + s;
```

Infix-Notation

Additionsoperator für rationale Zahlen

Neu:

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
rational t = operator+ (r, s);
```

Äquivalent, aber unpraktisch: funktionale Notation

Andere binäre arithmetische Operatoren für rationale Zahlen

```
// POST: return value is the difference of a and b
rational operator- (rational a, rational b);
```

```
// POST: return value is the product of a and b
rational operator* (rational a, rational b);
```

```
// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator/ (rational a, rational b);
```

Unäres Minus

Hat gleiches Symbol wie binäres Minus, aber nur *ein* Argument:

```
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```

Relationale Operatoren

Sind für Structs nicht "eingebaut", können aber definiert werden:

```
// POST: return value is true if and only if a == b
bool operator== (rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

Arithmetische Zuweisungen

Wir wollen z.B. schreiben:

```
rational r;
r.n = 1; r.d = 2; // 1/2

rational s;
s.n = 1; s.d = 3; // 1/3
r += s;
std::cout << r.n << "/" << r.d << "\n"; // 5/6
```

Das Problem mit operator+=

Erster Versuch:

```
rational operator+= (rational a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Problem: Der Ausdruck `r += s` hat zwar den gewünschten Wert `r + s`, aber nicht den gewünschten Effekt der Veränderung von `r`, weil ein Funktionsaufruf die Werte der Aufrufargumente nicht ändert.

Das funktioniert nicht!

Das Problem mit operator+=

- Wir müssen Funktionen in die Lage versetzen, die Werte ihrer Aufrufparameter ändern zu können!
- Dazu brauchen wir kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen.

Referenztypen

Referenztypen: Definition

`T&` } gelesen als "T-Referenz"
zugrundeliegender Typ

- `T&` hat gleichen Wertebereich und gleiche Funktionalität wie `T`,...
- ... nur die Initialisierungs- und Zuweisungssemantik ist anders.

Referenztypen: Initialisierung

- Eine Variable mit Referenztyp (eine *Referenz*) kann nur mit einem L-Wert initialisiert werden
- die Variable wird dabei ein *Alias* des L-Werts (ein anderer Name für das Objekt hinter dem L-Wert)

Referenztypen: Initialisierung und Zuweisung

Beispiel:

```
int i = 5;
int& j = i;           // j becomes an alias of i

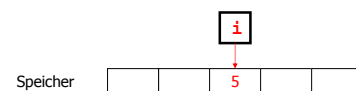
j = 6;               // changes the value of i
std::cout << i << "\n"; // outputs 6
```

Referenztypen: Initialisierung und Zuweisung

Beispiel:

```
int i = 5;
int& j = i;           // j becomes an alias of i

j = 6;               // changes the value of i
std::cout << i << "\n"; // outputs 6
```



Referenztypen: Initialisierung

Beispiel:

```
int i = 5;
int& j = i;           // j becomes an alias of i

j = 6;               // changes the value of i
std::cout << i << "\n"; // outputs 6
```



Referenztypen: Zuweisung

Beispiel:

```
int i = 5;
int& j = i;           // j becomes an alias of i

j = 6;               // changes the value of i
std::cout << i << "\n"; // outputs 6
```



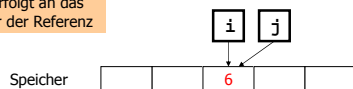
Referenztypen: Zuweisung

Beispiel:

```
int i = 5;
int& j = i;           // j becomes an alias of i

j = 6;               // changes the value of i
std::cout << i << "\n"; // outputs 6
```

Zuweisung erfolgt an das Objekt hinter der Referenz



Referenzen sind implizite Zeiger (aber einfacher und sicherer)

Mit Referenzen:

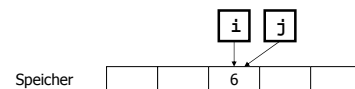
```
int i = 5;
int& j = i;

j = 6;
std::cout << i << "\n";
```

Mit Zeigern:

```
int i = 5;
int* j = &i;

*j = 6;
std::cout << i << "\n";
```



Referenztypen: Realisierung

- Intern wird ein Wert vom Typ $T\&$ durch die Adresse eines Objekts vom Typ T repräsentiert

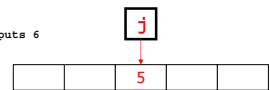
```
int& j; // error: j must be an alias of something
int& k = 5; // error: the literal 5 has no address
```

Call by Reference

- Referenztypen erlauben Funktionen, die Werte ihre Aufrufargumente zu ändern:

```
void increment (int& i)
{
  ++i;
}

int main ()
{
  int j = 5;
  increment (j);
  std::cout << j << "\n"; // outputs 6
  return 0;
}
```

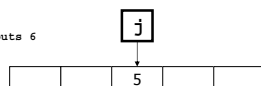


Call by Reference

- Referenztypen erlauben Funktionen, die Werte ihre Aufrufargumente zu ändern:

```
void increment (int& i)
{
  ++i;
}

int main ()
{
  int j = 5;
  increment (j);
  std::cout << j << "\n"; // outputs 6
  return 0;
}
```

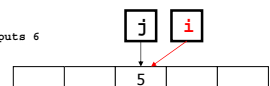


Call by Reference

- Referenztypen erlauben Funktionen, die Werte ihre Aufrufargumente zu ändern:

```
void increment (int& i)
{ // i becomes alias of call parameter ← Initialisierung der formalen Parameter
  ++i;
}
```

```
int main ()
{
  int j = 5;
  increment (j);
  std::cout << j << "\n"; // outputs 6
  return 0;
}
```

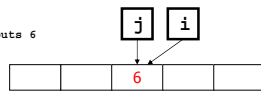


Call by Reference

- Referenztypen erlauben Funktionen, die Werte ihre Aufrufargumente zu ändern:

```
void increment (int& i)
{ // i becomes alias of call parameter
  ++i;
}

int main ()
{
  int j = 5;
  increment (j);
  std::cout << j << "\n"; // outputs 6
}
return 0;
```

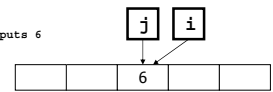


Call by Reference

- Referenztypen erlauben Funktionen, die Werte ihre Aufrufparameter zu ändern:

```
void increment (int& i)
{ // i becomes alias of call parameter
  ++i;
}

int main ()
{
  int j = 5;
  increment (j);
  std::cout << j << "\n"; // outputs 6
}
return 0;
```



Call by Value / Reference

- formales Argument hat Referenztyp:
call by reference

formales Argument wird (intern) mit der **Adresse** des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem **Alias**

- formales Argument hat keinen Referenztyp:

call by value

formales Argument wird mit dem **Wert** des Aufrufparameters (R-Wert) initialisiert und wird damit zu einer **Kopie**

Return by Value / Reference

- Auch der Rückgabetypp einer Funktion kann ein Referenztyp sein (*return by reference*)
- In diesem Fall ist der Funktionsaufruf selbst ein L-Wert

Return by Value / Reference

- o Auch der Rückgabetyt einer Funktion kann ein Referenztyp sein (*return by reference*)
- o In diesem Fall ist der Funktionsaufruf selbst ein L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```

exakt die Semantik des Prä-Inkremets:

Die Lösung für operator+=

Bisher (funktioniert nicht):

```
rational operator+= (rational a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Die Lösung für operator+=

Neu:

```
// POST: b has been added to a;
//      return value is the new value of a
rational& operator+= (rational& a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Der L-Wert a wird um den Wert von b inkrementiert und als L-Wert zurückgegeben

Ein-/Ausgabeoperatoren

- o können auch überladen werden!

Bisher:

```
std::cout << "Sum is " << t.n << "/" << t.d << "\n";
```

Neu (gewünscht):

```
std::cout << "Sum is " << t << "\n";
```



Ein-/Ausgabeoperatoren

- können auch überladen werden!

Das kann wie folgt erreicht werden:

```
// POST: a has been written to o
std::ostream& operator<< (std::ostream& o, rational r)
{
    return o << r.n << "/" << r.d;
}
```

schreibt `r` auf den Ausgabestrom `o` und gibt diesen als L-Wert zurück