

Float

Die grösste Zahl, die wir darstellen können ist

Alternativ: Bsp $F(2, 9, -2, 2)$: $x_{max} = 1.111_2 \cdot 2^2 = 11.1_2 = 7.5_{10}$

$$x_{max} := \max \{x \in F^*\} = b^{e_{max}} \cdot \sum_{n=0}^{p-4} (b-1) \cdot b^{-n} = b^{e_{max}} \cdot \frac{(1 - (\frac{1}{b})^p)(b-1)}{1 - \frac{1}{b}} = \frac{b^{e_{max}+1} \cdot (1 - b^{-p})}{b}$$

und $x_{min} := \min \{x \in F^*\} = -x_{max}$

Bsp $F(2, 4, -2, 2)$: $x_{max} = 2^3(1 - 2^{-4}) = 8 \cdot \frac{15}{16} = \frac{1}{2} \cdot 15 = 7.5$

Die kleinste positive Zahl ist dann $b^{e_{min}} \cdot x_{min} = 1 \cdot 2^{-2} = 0.01_2 = 0.25_{10}$

Anzahl Zahlen in $F^*(b, p, e_{min}, e_{max})$ für $b=2$:

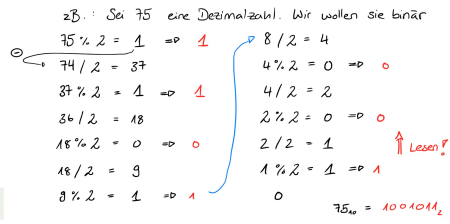
$$2 \cdot 2^{p-1} (e_{max} - e_{min} + 1)$$

negative hoch 0

Binärdarstellung von 1.1_{10}

x	b _i	x - b _i	2(x - b _i)
1.1	b ₀ = 1.	0.1	0.2
0.2	b ₁ = 0	0.2	0.4
0.4	b ₂ = 0	0.4	0.8
0.8	b ₃ = 0	0.8	1.6
1.6	b ₄ = 1	0.6	1.2
1.2	b ₅ = 1	0.2	0.4

Dezimal → Binär: Wir können hier wie folgt vorgehen



⇒ 1.00011, periodisch, nicht endlich

Pointer

```
int a[] = {0, 1, 2, 3};
int *ptr = a; // ptr zeigt auf a[0]
*ptr++ = 187; // <=> *ptr = 187; ptr++;
**ptr = 13; // <=> **ptr; *ptr = 13;
// => a = {187, 1, 13, 3}
// => ptr zeigt auf a[2]
for(int num: a) { // Ausgabe
    cout << num << " ";
}
```

Const-Faustregel:

Const vor dem * => gilt für Objekt
 Const nach dem * => gilt für Pointer
 Bem: const int *p <=> int const *p

Constness

```
int x = 3;
const int& r = x;
// r kann nicht benutzt werden, um
// x zu verändern, da const reference!
```

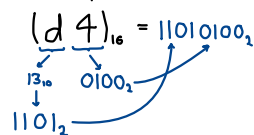
Number Conversion

```
#include<bitset>
```

```
cout << 0b1101; //Bin2Dec
cout << 0x7f; //Hex2Dec
cout << 0273; //Oct2Dec
```

```
cout << std::bitset<8>{187}; //Dec2Bin
cout << std::hex << 187; //Dec2Hex
cout << std::oct << 187; //Dec2Oct
```

Hex2Bin Bsp:



EBNF

```
// train = "[" ( open | compositions ) "]"
// open = loco cars.
// loco = "*" | "*" loco.
// cars = "-" | "(" cars ")".
// compositions = composition { composition }.
// composition = "<" open loco ">".
```

```
bool loco(std::istream& is){
    if(consume(is, '*')){
        loco(is);
        return true;
    }
    return false;
}
```

```
bool cars(std::istream& is){
    return consume(is, '-')
    || (consume(is, '(') && cars(is)
    && consume(is, ')'));
}
```

```
bool open(std::istream& is){
    return loco(is) && cars(is);
}
```

```
bool composition(std::istream& is){
    return consume(is, '<') && open(is)
    && loco(is) && consume(is, '>');
}
```

```
bool compositions(std::istream& is){
    if(composition(is)){
        while(lookahead(is) == '<'){
            if (!composition(is)){
                return false;
            }
        }
        return true;
    }
    return false;
}
```

```
bool train(std::istream& is) {
    return consume(is, '[') && (open(is)
    || compositions(is)) && consume(is, ']');
}
```

```
// Frogs = Frog Frog {Frog Frog}.
bool Frogs(std::istream& is){
    if (!Frog(is) || !Frog(is)){
        return false;
    }
    while(lookahead(is) == 'F'){
        if (!Frog(is) || !Frog(is)){
            return false;
        }
    }
    return true;
}
```

```
// Stack = Mattrass ( Frogs | Stack ) Mattrass.
bool Stack(std::istream& is){
    if (!Mattrass(is)){
        return false;
    }
    if (lookahead(is) == 'F') {
        if (!Frogs(is)) return false;
    } else {
        if (!Stack(is)) return false;
    }
    return Mattrass(is);
}
```

```
// Statement = "var" ((VarSingleName [VarValue]) | VarNames " ");
// VarValue = "=" Digit {Digit}
// VarSingleName = {"_"} Letter {Letter}
// VarNames = VarSingleName {"," VarSingleName}
// Letter = "a" | "b" | ... | "z"
// Digit = "0" | "1" | "2" | ... | "9"
```

```
bool VarSingleName(std::istream& is) {
    while (consume(is, "_")) {
        if (peek(is) == ' ') return false;
    }
    if (Letter(is)) {
        if (peek(is) == ' ') return true;
        while (Letter(is)) {
            if (peek(is) == ' ') return true;
        }
        return true;
    }
    return false;
}
```

```
bool Statement(std::istream& is) {
    if (consume(is, "var")) {
        unsigned int varNameCount = 0;
        do {
            if (!VarSingleName(is)) return false;
            varNameCount++;
        } while (consume(is, ","));
        if (lookahead(is) == '=') {
            if (varNameCount > 1 || !VarValue(is)) return false;
        }
        return consume(is, ";");
    }
    return false;
}
```

Bäume

```
void prune(Node* root){
    if(nonFullNodes(root) == 0){
        return;
    }
    Node* l = root->left;
    Node* r = root->right;
    if(l==nullptr && r!=nullptr){
        delete r;
        root->right = nullptr;
        return;
    } else if(l!=nullptr && r==nullptr){
        delete l;
        root->left = nullptr;
        return;
    }
    prune(l);
    prune(r);
}
```

```
void clear(Node* root){
    if(root != nullptr){
        clear(root->right);
        clear(root->left);
        delete root;
    }
}
```

Linked List

```
void deleteValue(unsigned int value) {
    if(first == sentinel){
        return;
    }

    if(first->value == value){
        ListNode* n = first;
        first = first->next;
        delete n; return;
    }

    ListNode* p = first;
    ListNode* n = first->next;
    while(n != sentinel && n->value != value){
        p = n;
        n = n->next;
    }

    if(n!=sentinel){
        p->next = n->next;
        delete n;
    }
}
```

```
void print_node(std::ostream& os, Node* n) {
    if (n != nullptr) {
        os << " " << n->value;
        print_node(os, n->next);
    }
}

void Queue::print(std::ostream& os) const {
    os << "[";
    if (first != nullptr) {
        os << first->value;
        print_node(os, first->next);
    } os << "]" ;
}
```

Overload

Rule of Three

```
avec::avec(const avec& vec) : count(vec.count) {
    elements = new tracked[count]();
    for (unsigned int i = 0; i < count; i++) {
        elements[i] = vec.elements[i];
    }
}

avec& avec::operator=(const avec& t) {
    if (elements != t.elements) {
        avec copy = avec(t);
        std::swap(copy.elements, elements);
        count = copy.count;
    }
    return(*this);
}

avec::~avec() {
    delete[] elements;
}
```

```
Vec3& Vec3::operator+=(const Vec3& other) {
    m_x += other.m_x;
    m_y += other.m_y;
    m_z += other.m_z;
    return (*this);
}
```

```
Vec3 Vec3::operator+(const Vec3& other) const {
    Vec3 v = *this;
    return v += other;
}
```

```
// input type: [-2,5]
bool read_input(std::istream& in, Complex& a){
    char first, mid, last;
    in >> first >> a.re >> mid >> a.im >> last;
    return (first == '[' && mid == ',' && last == ']');
}

std::ostream & operator<<(std::ostream& os, Complex& a){
    return os << '[' << a.re << ',' << a.im << ']';
}

Complex operator/(const Complex a, const Complex b){
    Complex x;
    double sqr;
    sqr = b.re*b.re + b.im*b.im;
    x.re = (a.re*b.re - a.im*b.im)/sqr;
    x.im = (a.im*b.re + a.re*b.im)/sqr;
    return x;
}

bool operator==(Complex a, Complex b){
    double d = 0.001;
    return (std::abs(a.re-b.re) < d
    && std::abs(a.im-b.im) < d);
}

bool operator!=(Complex a, Complex b){
    return !(a == b);
}
```

Algorithms

```
int* return_smax(std::vector<int> &v) {
    if(v.size() < 2){
        return nullptr;
    }
    int* max = &v.at(0);
    int* smax = &v.at(1);
    if(*max < *smax){
        std::swap(max, smax);
    }
    for(unsigned int i=2; i<v.size(); i++){
        if(v.at(i) > *smax){
            if(v.at(i) > *max){
                smax = max;
                max = &v.at(i);
            } else{
                smax = &v.at(i);
            }
        }
    }
    return smax;
}
```

```
void left_shift_vector(std::vector<int> &v){
    int first = v.at(0);
    for(unsigned int i=0; i<v.size()-1; i++){
        v.at(i) = v.at(i+1);
    }
    v.at(v.size()-1) = first;
}
```

```
void right_shift_vector(std::vector<int> &v){
    int last = v.at(v.size()-1);
    for(unsigned int i=v.size()-1; i > 0; --i) {
        v.at(i) = v.at(i-1);
    }
    v.at(0) = last;
}
```

```
// PRE: Input capital character between A and Z
// POST: Returns the input character rotated by key
char rot(char input, unsigned int key) {
    assert(input >= 65 && input <= 90);
    return (input - 65 + key) % 26 + 65;
}
```

```
int max_diff(iterator begin, iterator end){
    if (begin == end){
        return 0;
    }
    int min = *begin;
    int max = *begin;
    for (iterator it = begin+1; it != end; ++it){
        if (*it < min){
            min = *it;
        }
        if (*it > max){
            max = *it;
        }
    }
    return max - min;
}
```

Mengen

```
void transitively_close(Equalities& equalities) {
    Equalities newly_inferred;
    do {
        newly_inferred.clear();
        for (const Equality& eq1 : equalities) {
            for (const Equality& eq2 : equalities) {
                Symbol p, q;

                if (eq1.overlaps(eq2, p, q)) {
                    Equality eq3 = Equality(p, q);

                    if (equalities.find(eq3) == equalities.end()) {
                        newly_inferred.insert(eq3);
                    }
                }
            }
        }
        equalities.insert(newly_inferred.begin(), newly_inferred.end());
    } while (!newly_inferred.empty());
}
```

Rekursion

```
int knapsack(unsigned int max_weight,
            std::vector<item>::iterator begin,
            std::vector<item>::iterator end,
            unsigned int depth) {
    if(max_weight == 0 || begin == end){return 0;}
    int m1,m2;
    if(begin->weight > max_weight){
        return knapsack(max_weight, begin+1, end, depth);
    }else{
        m1 = begin->value +
        knapsack(max_weight-(begin->weight), begin, end, depth);
        m2 = knapsack(max_weight, begin+1, end, depth);
        return std::max(m1,m2);
    }
}
```

```
// PRE: value != 0
// POST: Delete a leaf node with given value from the (sub)tree starting at root.
// If a leaf node is deleted, remove it from its parent.
// If an inner node becomes empty, remove it from its parent and delete it.
// Return true if and only if the (sub)tree starting at root was deleted.
bool deleteLeaf(Node* root, unsigned int value) {
    if (root == nullptr) {
        return false;
    }
    if (root->isLeaf() && root->value == value) {
        delete root;
        return true;
    }
    for (unsigned int i = 0; i < root->children.size(); ++i) {
        if (deleteLeaf(root->children[i], value)) {
            root->removeFromChildren(root->children[i]);
            if (root->children.size() == 0) {
                delete root;
                return true;
            }
        }
    }
    return false;
}
```

```
using prices = std::vector<unsigned int>;
// pre: p.size() >= length + 1
// post: return the best value that can be attained by
// cutting a rod of the given length and given
// prices per length
unsigned int bestValue(const prices& p, unsigned int length) {
    if (length <= 0) return 0;
    unsigned int bv = 0;
    for (unsigned int i = 1; i < p.size(); ++i) {
        if (i <= length) {
            unsigned int nv = p.at(i) + bestValue(p, length - i);
            if (bv < nv) bv = nv;
        }
    }
    return bv;
}
```

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
operator+=	Append to string (public member function)
swap	Swap string values (public member function)
pop_back <small>C++11</small>	Delete last character (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

1 EXPRESSIONS & OPERATORS

Expressions: repräsentieren Berechnungen, haben Typ und Wert, entweder primär oder zusammengesetzt (mit Operatoren).

Literale: konstanter Wert, fester Typ. 42u, 3.14f, "bruh"

L-Wert	R-Wert
Identifiziert Speicherplatz (Adresse!)	Jeder L-Wert kann als R-Wert benutzt werden
Kann Wert ändern (z.B. Zuweisung)	Kann Wert nicht ändern
Bsp.: Variable	Bsp.: Literal

Operators: **R*R, !R, L=R, L+=R, ++L, L++, L>>L, L<<R**

Operator	Prec.	A
::	17	L
a++, f(), v[], o.m, p->n	16	L
--a, -a, !, ~, (cast), *p, &a, new, delete	15	R
*, /, %	13	L
a+b, a-b	12	L
<, <=, >, >=	9	L
==, !=	8	L
&, ^,	7, 6, 5	L
&&,	4, 3	L
=, +=, -=, *=, /=, %=, &=, ^=, =	2	R

Jede Expression kann eindeutig geklammert werden (→ Baum)

Short-circuit evaluation: Bei **&&** und **||** wird die rechte Seite nicht mehr ausgewertet, falls das Ergebnis schon nach der linken Auswertung feststeht. Bsp.: Division durch Null vermeiden:
`b != 0 && a/b < c`

Richtlinie: Vermeide das Verändern von Variablen, welche im selben Ausdruck noch einmal verändert werden! (**a*(a=2)**)

Modulo: Es gelten $(-a)/b = -(a/b)$ und $(a/b) \cdot b + a \% b = a$.

2 ZAHLENSYSTEME/-TYPEN

- » Conversion: `char/bool < int < uint < float < double`
- » Conversion: `bool → int` ok; `int → bool` möglich, aber bad practice (alles ausser 0 entspricht `true`).
- » Ein Overflow verursacht keine Warnung/Fehlermeldung!
 Overflow detection von $a + b$: `a < (int_max - b)`
- » Präfixe: `0b` binär; `0x` hexadec; `0` oktal(!)
`std::cout << 077; // 6310 (= 778)`

int:

- » signed = $[-2^{31}, 2^{31} - 1]$; max = 2'147'483'647
- » unsigned = $[0, 2^{32} - 1]$; max = 4'294'967'295
- » size = 4 Bytes = 32 Bits
- » Zweierkomplement: 1 Vorzeichen-Bit (1 = negativ)
 Umrechnung: Bits invertieren, dann 1 addieren
 Bsp.: $6 \rightarrow -6$: `0110 → 1001 + 1 = 1010`

char:

- » signed = $[-2^7, 2^7 - 1]$; max = 127
- » unsigned = $[0, 2^8 - 1]$; max = 255
- » size = 1 Bytes = 8 Bits
- » repräsentiert normalerweise Buchstaben (nach Ascii)

float/doubles:

- » $F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen $\pm d_0.d_1 \dots d_{p-1} \cdot \beta^e$, wobei $d_i \in \{0, \dots, \beta - 1\}$ und $e \in \{e_{\max}, \dots, e_{\min}\}$.
- » p = Stellenzahl inkl. Stelle vor Punkt! $p \geq 1$
- » Menge der normalisierten Zahlen ($d_0 \neq 0$)
 $|F^*(\beta, p, e_{\min}, e_{\max})| = 2 \cdot (\beta - 1) \cdot \beta^{p-1} \cdot (e_{\max} - e_{\min} + 1)$
- » Normalisierte Darstellung ist eindeutig!
- » 0 sowie $x < \beta^{e_{\min}}$ haben keine normalisierte Darstellung
- » Arithmetische Operatoren runden exaktes Ergebnis auf nächste darstellbare Zahl!
- » Float: $F^*(2, 24, -126, 127)$ (4 Bytes)
- » Double: $F^*(2, 53, -1022, 1023)$ (8 Bytes)
- » Konvertiere Dezimalzahl in die normalisierte Darstellung, runde auf die nächste darstellbare Zahl (p Bits), setze Exponenten (innerhalb des erlaubten Intervalls)
 Sei $F^*(2, 4, -3, 3)$ und $|F^*| = 112$
 $3.1416_{10} \Rightarrow 1.101_2 \cdot 2^1 \Rightarrow 3.25_{10}$ (Rundungsfehler: 0.1084)

hex	bin	dec	hex	bin	dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	a	1010	10
3	0011	3	b	1011	11
4	0100	4	c	1100	12
5	0101	5	d	1101	13
6	0110	6	e	1110	14
7	0111	7	f	1111	15

Primzahltest (n = Input):

```
unsigned int d;
for(d = 2; n%d != 0; ++d);
bool is_prime = (n == d);
```

Dec2bin (x ganze Zahl)

```
for (p = 1; p <= x / 2; p *= 2); // 2^p <= x
for (; p != 0; p /= 2) {
    if (x >= p) {
        std::cout << "1";
        x -= p;
    }
    else std::cout << "0"; }
```

Dec2bin (0 < x < 2; float/double)

```
for (int b_0; x != 0; x = 2 * (x - b_0)) {
    b_0 = (x >= 1);
    std::cout << b_0; } // d_0.d_1-d_n
```

1.25	-1	$d_0 = 1$
$0.25 \cdot 2 = 0.5$	-0	$d_1 = 0$
$0.5 \cdot 2 = 1$	-1	$d_2 = 1$
0		$\Rightarrow 1.25_{10} = 1.01_2$

Regeln:

- » Teste keine gerundeten Fließkommazahlen auf Gleichheit!
`if ((x - y) > 0) return ((x - y) < tol);`
`else return ((y - x) < tol);`
- » Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!
- » Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

3 VARIABLEN & ANDERE DATENTYPEN

Scopes {...}: Deklaration einer Variable nach aussen unsichtbar.

Shadowing: Variable in einem inneren Scope kann gleich heissen wie eine ausserhalb und wird nicht «verwechselt».

```
int i = 2;
for (int i = 0; i < 4; ++i) cout << i; // 0123
cout << i; // 2
```

Referenztypen T&

» Gleicher Wertebereich, gleiche Funktionalität, aber andere Initialisierung (&L = L) und Zuweisung (Synonym/Alias)

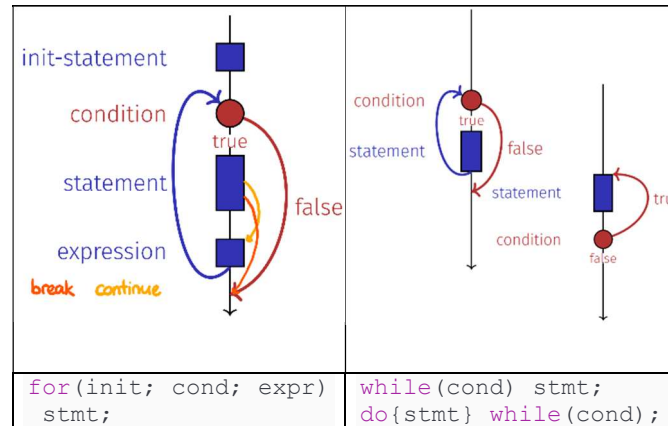
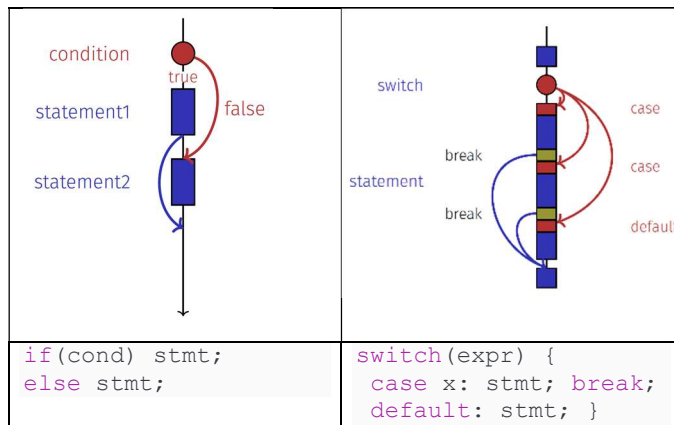
» const-Referenz: Read-only Reference auf ein Objekt

```
int n = 5; // original
int& rw = n; // read-write alias
const int& r = n; // read-only alias
```

Richtlinien:

- » Wo überall möglich **const** verwenden. Ein Programm, das diese Regel befolgt, heisst **const-korrekt**.
- » Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mind. so lange leben wie die Referenz selbst.
Besonders aufpassen bei return-by-reference-Funktionen.

4 STRUKTURIERUNG: LOOPS UND VERZWEIGUNGEN



Achtung: **switch** führt alle Statements bis zum nächsten **break** aus («Durchfallen»).

5 FUNKTIONEN

- » **Precondition** so offen wie möglich formuliert, "D"
- » **Postcondition** so stark wie möglich formuliert, "W"
- » Pre-/Postconditions können mit **assertions** überprüft werden. (`#include<cassert>`, dann `assert(cond);`)
- » Non-void Funktion müssen immer ein `return`-Stmt. erreichen!
- » Mit `return`; kann man eine void-Funktion abbrechen.
- » Funktionsaufruf ist ein R-Wert, ausser bei **T&-Funktionen**
- » **Pass by value**: Argument wird mit Wert initialisiert → Kopie
- » **Pass by reference**: Argument wird mit Adresse (als L-Wert) initialisiert → Alias
- » **Return by reference**: Aufpassen mit Lebensdauer (→ Richtlinie Kapitel 3)
- » **Const**: `const`-Objekte dürfen nur `const`-Funktionen aufrufen.
 - » `int getX() const { return x; }` Memberfunktion verändert das Objekt (implizites Argument 'this') nicht.
 - » `void f(const T& arg) { ... }` Funktion verändert Argument nicht

Overloading:

Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente.

```
int pow(int b, int e) {...}
int pow(int e) { return pow(2, e); }
```

Der Compiler wählt beim Aufruf die am besten passende Funktion.

Operator-Overloading:

```
rational operator+ (rational a, rational b)
{ return {a.n*b.d + a.d*b.n, a.d*b.d}; }
rational operator- (rational a)
{ a.n = -a.n; return a; }
bool operator== (rational a, rational b)
{ return a.n * b.d == a.d * b.n; }
rational& operator+= (rational& a, rational b)
{ a = a+b; return a; }
std::ostream& operator<< (std::ostream& out,
  rational r)
{ return out << r.n << "/" << r.d; }
std::istream& operator>> (std::istream& in,
  rational& r) // Input format: "n/d"
{ char c; return in >> r.n >> c >> r.d; }
rational& rational::operator++ () // pre-inc
{ n += d; return *this; }
rational rational::operator++ (int dummy)
{ rational tmp = *this; ++*this;
  return tmp; // return old value }
```

Achtung: Increment-Operatoren müssen Memberfunktionen der Klasse sein! Andere Operatoren können auch Member sein.

6 REKURSION & EBNF

Rekursion:

- » Voraussetzung für Terminierung
 - » Base case (Abbruchbedingung)
 - » Fortschritt der Variable in Richtung base case pro Iteration

Beispiel:

```
int gcd(int a, int b) {
  if(b == 0) return a; // base case
  return gcd(b, a%b); }
```


EBNF:

Definiert Gültigkeit einer formalen Grammatik

... ...	Alternative (OR)
{...}	Repetition (beliebig oft oder gar nie)
[...]	Repetition (max. 1x)
... = ...	Definition (endet mit .)
"..."	Enthält terminales Symbol

Beispiel:

```
digit = "0" | "1" | ... | "9".
number = digit { digit }.
factor = number | "(" expr ")" | "-" factor.
term = factor { "*" factor | "/" factor }.
expr = term { "+" term | "-" term }.
```

Parsing:

- » Regeln werden zu Funktionen
- » Alternativen/Optionen werden zu if-Statements
- » Nichtterminale Symbole auf der rechten Seite werden zu Funktionsaufrufen
- » Optionale Repetitionen werden zu while Anweisungen

EBNF Helpers:

- » `peek(is)`: returns next char of stream without consuming it.
- » `lookahead(is)`: returns next non-whitespace char of stream without consuming it.
- » `consume(is, c)`: consumes next char of stream and returns true if said char is equal to c.

Beispiel:

```
Hamburger = Bun { Onions } Patties [Salad] Bun
Salad = "S" "A".
Patties = "P" { "P" }.
Onions = "O" "O" "O" { Onions }.
Bun = "B".
```

```
bool Bun(std::istream& is) {
    return consume(is, 'B'); }
bool Patties(std::istream& is) {
    if (consume(is, 'P')) {
        while (lookahead(is) == 'P' &&
            consume(is, 'P'));
        return true; }
    return false; }
```

```
bool Onions(std::istream& is) {
    if (consume(is, 'O')) {
        unsigned int count = 1;
        while (lookahead(is) == 'O'
            && consume(is, 'O')) ++count;
        return count % 3 == 0; }
    return false; }
bool Salad(std::istream& is) {
    if(lookahead(is) == 'S' && consume(is, 'S'))
        return (lookahead(is) == 'A' &&
            consume(is, 'A'));
    return true; }
bool Hamburger(std::istream& is) {
    if (Bun(is)) {
        if (lookahead(is) == 'O' && !Onions(is))
            return false;
        if (!(Patties(is) && Bun(is)))
            return false;
        return !lookahead(is); }
    return false; }
```

7 POINTER

- » `T* ptr = &var;`
`int i = 5;`
`int* p = &i; // Adresse von i`
`int j = *p; // j = 5`
- » * zeigt an, dass es sich um einen Pointer handelt, und ist zugleich auch Dereferenzierungsoperator
- » `&var` gibt die Adresse von `var`.
- » `v[i]` Index-Operator, retourniert L-Wert
- » `p->n == (*p).n`

Const-Madness:

- » `const T var ⇔ T const var` (gilt auch für T&)
- » Deklaration von rechts nach links lesen:

<code>int const p1;</code>	p1 konstanter Int
<code>int const* p2;</code>	p2 Pointer auf konst. Int
<code>int* const p3;</code>	p3 konst. Pointer auf Int
<code>int const* const p4;</code>	p4 konst. Pointer auf konst. Int

» `const` ist nicht absolut:

```
int a = 5;
const int* p1 = &a; int* p2 = &a;
*p1 = 2; // Fehler
*p2 = 2; // ok, obwohl *p1 verändert wird
```

8 VEKTOREN, ARRAYS, LINKED LISTS ETC.

`std::vector`:

- » `#include <vector>`
- » `std::vector<T> name(length, init_val);`
- » `std::vector<T> name = {1, 2, 3};`
- » Wahlfreier Zugriff
 - » `vec[i]` gibt L-Wert zurück, deshalb ist eine Zuweisung `v[i] = 2;` möglich.
 - » `vec[i]` gibt keine Warnung bei out of border
 - » `vec.at(i)` gibt Warnung bei out of border
 - » Iteration: `for(int i = 0; i < 3; ++i) cout << v[i];`
Ineffizient: erfordert pro Zugriff Multiplikation + Addition
- » Sequenzieller Zugriff
 - » `for(int* it = p; it != p+3; ++it) cout << *it;`
Effizient: erfordert nur eine Addition pro Zugriff
- » Matrix/Multidimensionaler Vektor ist ein Vektor vom Typ Vektor: `std::vector<std::vector<T>>`
 - » Zugriff mit `v[i][j]` oder `v.at(i).at(j)`
 - » Jede Zeile ist ein separater Vektor ⇒ nicht jede Zeile hat gleich viele Spalten, Zeilen können auch 0 Spalten haben!
 - » Datentyp abkürzen:
`using imat = std::vector<std::vector<int>>;`
- » Memberfunktionen: `size`, `push_back`, `begin`, `end`, `pop_back`, `insert`, `reverse`, `swap`

`std::string`:

- » `#include <string>`
- » Statt `std::vector<char>` gibt es `std::string`
- » `std::string s(n, 'a')` s wird mit n a's gefüllt
- » Zeichen auslesen mit `s[i]` oder `s.at(i)`, `s[0] = 'a';`

» Operationen:

```
s+= "asdf";  
s = s1 + s2; // geht nur mit Variablen, nicht  
mit Literalen
```

» `.length()` statt `.size()` möglich

Dynamisches Array:

» `T* p = new T[n];`

» `int* p = new int[3]{1, 2, 3};`

» Zugriff auf Elemente mit Index-Operator

» Konvention: Übergabe eines Arrays durch zwei Pointer:

`begin` zeigt auf erstes Element, `end` zeigt hinter das letzte Element (past-the-end). Array ist leer, falls `begin == end`.

Linked List:

» Vorteil: Dynamisches Speichermanagement: Elemente können überall eingefügt/gelöscht werden, Grösse veränderbar

» Nachteil: Kein zusammenhängender Speicherbereich, kein wahlfreier Zugriff

» Einzelne Komponenten sind nodes mit value und Pointer zum nächsten Element (`struct llnode`) (muss mit `new` alloziert werden, damit ein node nicht direkt wieder gelöscht wird!)

» Gesamter Vektor (`class llvec`, s.u.) besteht aus einem Pointer zum ersten Element und versch. Memberfunktionen (Konstruktoren, `push_front/push_back`, `pop`, `insert`, `size`, `print`, versch. Operatoren etc.)

Container:

» `std::unordered_set<T>`: ungeordnete, duplikatfreie Menge

» `std::set<T>`: geordnete (z.B. alphabetisch, wenn `T = std::string`), duplikatfreie Menge (Rot-Schwarz-Baum)

Iterator:

» Sollten für jeden Container implementiert werden

» Container `c`:

» `it = c.begin()`: Iterator auf 1. Element

» `it = c.end()`: Past-the-end Iterator

» `*it`: Zugriff auf akuelles Element

» `++it`: Iterator um ein Element verschieben

» C++-Standardfunktionen (`find`, `fill`, `sort` etc.) funktionieren so auf beliebigen Containern, die Iterator implementiert haben.

» Eigener Iterator muss Operatoren `*`, `++`, `!=` implementiert und Funktionen `begin()` und `end()` implementiert haben.

» `iterator`: public subclass des Containers

Implementierung:

```
class llvec {  
public:  
    class iterator {  
        llnode* node;  
    public:  
        iterator(llnode* n) : node(n) {}  
        iterator& operator++() {  
            this->node = this->node->next;  
            return *this; }  
        int& operator*() const {  
            return this->node->value; }  
        bool operator!=(const iterator& it2) const {  
            return this->node != it2.node; }  
    };  
    iterator begin() {  
        return iterator(this->head); }  
    iterator end() {  
        return iterator(nullptr); } };
```

9 STRUCTS/CLASSES

» Class: standardmässig alles private

» Struct: standardmässig alles public

» Memberfunktionen: Deklaration innerhalb Klasse, Definition ausserhalb: `T className::funcName(...) {...}`

» Zugriff: `obj.memberfunc()`

» **Konstruktor T()**

» Spezielle Memberfunktion, wird bei Variablendeklaration aufgerufen, hat denselben Namen wie die Klasse

» **Muss public sein!**

» Default-Konstruktor: Verhindert undefiniertes Verhalten, indem er jeder Variable bei der Deklaration einen (neutralen) Wert zuweist (z.B. 0). Alternative: Default-Konstruktor löschen, so dass keine Variable uninitialized bleiben darf (`rational() = delete;`).

» **Destruktor ~T()**

» Eindeutige Memberfunktion, wird automatisch aufgerufen, wenn die Lebensdauer eines Klassenobjekts endet, z.B. bei `delete` oder am Ende eines Scopes.

» Falls kein Destruktor deklariert ist, wird er automatisch erzeugt und ruft die Destruktoren für Membervariablen auf. Im Fall von Pointern kann das zu memory leaks führen!

» **Copy-Konstruktor T(const T& x)**

» Eindeutiger Konstruktor, wird aufgerufen, wenn Werte vom Typ `T` mit Werten vom Typ `T` initialisiert werden.

» `T x = t;` (`t` vom Typ `T`)

» `T x (t);`

» Geht nicht: `T x;` `x = t;` (op= müsste überladen werden)

» Falls kein Copy-Konstruktor deklariert ist, wird er automatisch erzeugt und initialisiert memberweise, was bei Pointern zu Problemen führen kann!

» Überladung `operator=` als Memberfunktion *copy and swap idiom* (siehe unten)

» Mindestfunktionalität eines dynamischen Datentyps

» Konstruktor(en)

» Destruktor

» Copy-Konstruktor

» Zuweisungsoperator

» **Dreierregel**: Definiert eine Klasse eines davon, muss sie auch die anderen zwei definieren!

Initialisierung:

```
rational s; // Member-Variablen uninitialisiert  
rational t = {1, 5}; // Memberweise Init.  
rational u = t; // Memberweise Kopie  
t = u; // Memberweise Kopie  
rational v = u + t; // Memberweise Kopie
```

new/delete:

» Mit `new` erzeugte Objekte haben eine dynamische Lebensdauer: Sie leben, bis sie explizit mit `delete` gelöscht werden.

» `delete` ohne `new` verursacht einen Laufzeitfehler:

```
int* n = &var;  
delete n; // Laufzeitfehler
```

» `delete[] expr;` dealloziert ein mit `new` erzeugtes Array sein. `expr` muss ein `T*` sein, der auf das Array zeigt.

Richtlinie: Zu jedem `new` gibt es ein passendes `delete!`

Achtung: Dereferenzieren eines «dangling pointers»:

```
rational* t = new rational;
rational* s = t;
delete s;
int n = t->n; // t zeigt auf freigegebenen Speicher!
```

Mehrfaches Deallozieren eines Objekts mit `delete` ist ein ähnlich schwerer Fehler!

Implementierung `llvec`:

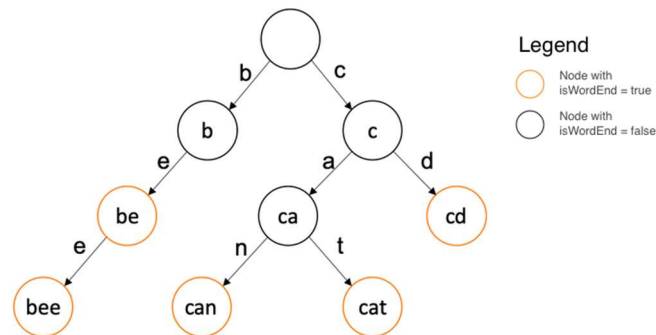
```
struct llnode {
    int value;
    llnode* next;
    llnode(int v, llnode* n) : value(v),
        next(n) {} };
class llvec {
    llnode* head;
public:
    // default constructor
    llvec() : head(nullptr) {}
    // constructor
    llvec(unsigned int i) {
        this->head = nullptr;
        for(; 0 < i; --i) this->push_front(0); }
    // copy constructor
    llvec(const llvec& vec) {
        this->head = nullptr;
        for(llnode* it = vec.head; it != nullptr;
            it = it->next) push_back(it->value); }
    // destructor
    ~llvec() {
        llnode* it = this->head;
        llnode* n = nullptr;
        while(it != nullptr) {
            n = it->next;
            delete it;
            it = n; } }
    // copy and swap idiom
    llvec& operator=(const llvec& vec) {
        // no self-assignment
        if(this->head != vec.head) {
            llvec copy = vec; // uses copy constructor
            std::swap(head, copy.head);
        } // swapped copy is destructed here
    }
}
```

```
// member functions
void push_front(int e) {
    this->head = new llnode(e, this->head); }
int& operator[](unsigned int i) {
    llnode* n = this->head;
    for(; 0 < i; --i) n = n->next;
    return n->value; }
void pop(unsigned int n) {
    unsigned int s = this->size();
    llnode* it = this->head;
    llnode* prev = nullptr;
    for(uint i = 0; i < n && i < s; ++i) {
        prev = it;
        it = it->next; }
    prev->next = it->next;
    delete it; } ;
```

Bäume:

Linked List, wo ein Node zu mehreren Nodes zeigen kann.

Beispiel Trie (Präfixbaum): Jeder Node hat 26 children.



```
struct TrieNode {
    TrieNode* children[26] = {};
    bool isWordEnd = false;
};
// add to tree
void insertWord(TrieNode* root,
    std::string word) {
    TrieNode *node = root;
    for (char c : word) {
        unsigned int index = charToIndex(c);
        if (node->children[index] == nullptr)
            node->children[index] = new TrieNode();
        node = node->children[index]; }
    node->isWordEnd = true; }
```

```
// search tree
bool containsWord(TrieNode* root,
    std::string word) {
    TrieNode *node = root;
    for (char c : word) {
        unsigned int index = charToIndex(c);
        if (node->children[index] == nullptr)
            return false;
        node = node->children[index]; }
    return node->isWordEnd; }
// print tree
void traverseAndPrint(TrieNode* node,
    std::string prefix) {
    if (node->isWordEnd)
        std::cout << prefix << "\n";
    for (unsigned int i = 0; i < 26; ++i) {
        if (node->children[i] != nullptr) {
            traverseAndPrint(node->children[i],
                prefix + indexToChar(i)); } } }
```

Anderer Baum (bel. viele children, in Vektor gespeichert):

```
struct Node {
    unsigned int value; // leaf: value != 0
    std::vector<Node*> children; //only inner nds
    // Default constructor
    Node() : value(0), children(0) {}
    // Construct a leaf node
    Node(unsigned int value) : value(value),
        children(0) {}
    // Construct an inner node
    Node(std::vector<Node*> children) : value(0),
        children(children) {}
    // Destructor
    ~Node() {
        for (Node* child : children)
            delete child; } }
unsigned int findMax(const Node* root) {
    unsigned int max = 0;
    // If the tree is empty, return 0
    if (root == nullptr) return 0;
    // If current node is a leaf, return value
    if (root->isLeaf()) return root->value;
    // Otherwise, search max recursively
    for (Node* child : root->children) {
        unsigned int childMax = findMax(child);
        if (childMax > max) max = childMax; }
    return max; }
```

```
bool deleteLeaf(Node* root,
unsigned int value) { //del node mit geg. val
if (root == nullptr) return false;
if (root->isLeaf() && root->value == value ){
delete root;
return true; }
for (unsigned int i = 0;
i < root->children.size(); ++i) {
if (deleteLeaf(root->children[i], value)) {
root->removeFromChildren(root->children[i]);
if (root->children.size() == 0) {
delete root;
return true; } } }
return false; }
```

10 REST: STREAMS AND OTHER STUFF

- » Komplexe Funktionen und Klassen sollten in andere Files ausgelagert werden.
- » Funktionsdefinitionen in .cpp-Files
- » Funktionsdeklarationen in .h-Files

Streams:

Immer als Referenz übergeben, da sie sich verändern!

Generic stream: `std::istream`, `std::ostream`

```
#include <iostream>
std::cin
std::cout
#include <fstream>
std::ifstream is_f("in.txt");
std::ofstream os_f("out.txt");
#include <sstream>
std::string in_str = "yeah boi";
std::istringstream is_s(in_str);
std::ostringstream os_s;
std::string out_str = os_s.str();
void f(std::istream& is, std::ostream& os){
is >> std::noskipws; // Leerzeichen beachten
char c;
while(is >> c) os << c; }
f(std::cin, is_f); // bel. Kombination möglich
```

Leerzeichen

```
is >> std::ws; // Leerzeichen überspringen
is >> std::noskipws; // Leerzeichen beachten
```

Einlesen verschiedener Typen:

```
// Input format: [a,b], e.g. [2,-5] = 2-5i
bool read_input(std::istream& in, Complex& a){
unsigned char c;
if(!(in >> c) || c != '['
|| !(in >> a.real)
|| !(in >> c) || c != ','
|| !(in >> a.imag)
|| !(in >> c) || c != ']')
return false;
else return true; }
```

Hex-Table:

00	0	40	64	80	128	c0	192
10	16	50	80	90	144	d0	208
20	32	60	96	a0	160	e0	224
30	48	70	112	b0	176	f0	240

ASCII:

dec	hex	bin	char	dec	hex	bin	char
0	0	0	NUL	64	40	1000000	@
1	1	1	SOH	65	41	1000001	A
2	2	10	STX	66	42	1000010	B
3	3	11	ETX	67	43	1000011	C
4	4	100	EOT	68	44	1000100	D
5	5	101	ENQ	69	45	1000101	E
6	6	110	ACK	70	46	1000110	F
7	7	111	BEL	71	47	1000111	G
8	8	1000	BS	72	48	1001000	H
9	9	1001	HT	73	49	1001001	I
10	0A	1010	LF	74	4A	1001010	J
11	0B	1011	VT	75	4B	1001011	K
12	0C	1100	FF	76	4C	1001100	L
13	0D	1101	CR	77	4D	1001101	M
14	0E	1110	SO	78	4E	1001110	N
15	0F	1111	SI	79	4F	1001111	O
16	10	10000	DLE	80	50	1010000	P
17	11	10001	DC1	81	51	1010001	Q
18	12	10010	DC2	82	52	1010010	R
19	13	10011	DC3	83	53	1010011	S
20	14	10100	DC4	84	54	1010100	T

21	15	10101	NAK	85	55	1010101	U
22	16	10110	SYN	86	56	1010110	V
23	17	10111	ETB	87	57	1010111	W
24	18	11000	CAN	88	58	1011000	X
25	19	11001	EM	89	59	1011001	Y
26	1A	11010	SUB	90	5A	1011010	Z
27	1B	11011	ESC	91	5B	1011011	[
28	1C	11100	FS	92	5C	1011100	\
29	1D	11101	GS	93	5D	1011101]
30	1E	11110	RS	94	5E	1011110	^
31	1F	11111	US	95	5F	1011111	_
32	20	100000	space	96	60	1100000	`
33	21	100001	!	97	61	1100001	a
34	22	100010	"	98	62	1100010	b
35	23	100011	#	99	63	1100011	c
36	24	100100	\$	100	64	1100100	d
37	25	100101	%	101	65	1100101	e
38	26	100110	&	102	66	1100110	f
39	27	100111	'	103	67	1100111	g
40	28	101000	(104	68	1101000	h
41	29	101001)	105	69	1101001	i
42	2A	101010	*	106	6A	1101010	j
43	2B	101011	+	107	6B	1101011	k
44	2C	101100	,	108	6C	1101100	l
45	2D	101101	-	109	6D	1101101	m
46	2E	101110	.	110	6E	1101110	n
47	2F	101111	/	111	6F	1101111	o
48	30	110000	0	112	70	1110000	p
49	31	110001	1	113	71	1110001	q
50	32	110010	2	114	72	1110010	r
51	33	110011	3	115	73	1110011	s
52	34	110100	4	116	74	1110100	t
53	35	110101	5	117	75	1110101	u
54	36	110110	6	118	76	1110110	v
55	37	110111	7	119	77	1110111	w
56	38	111000	8	120	78	1111000	x
57	39	111001	9	121	79	1111001	y
58	3A	111010	:	122	7A	1111010	z
59	3B	111011	;	123	7B	1111011	{
60	3C	111100	<	124	7C	1111100	
61	3D	111101	=	125	7D	1111101	}
62	3E	111110	>	126	7E	1111110	~
63	3F	111111	?	127	7F	1111111	DEL