15. Rekursion

Rekursive Funktionen, Korrektheit, Terminierung, Aufrufstapel, Bau eines Taschenrechners, BNF, Parsen

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 C++: Genauso!

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

```
// POST: return value is n!
unsigned int fac (const unsigned 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)

```
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)`

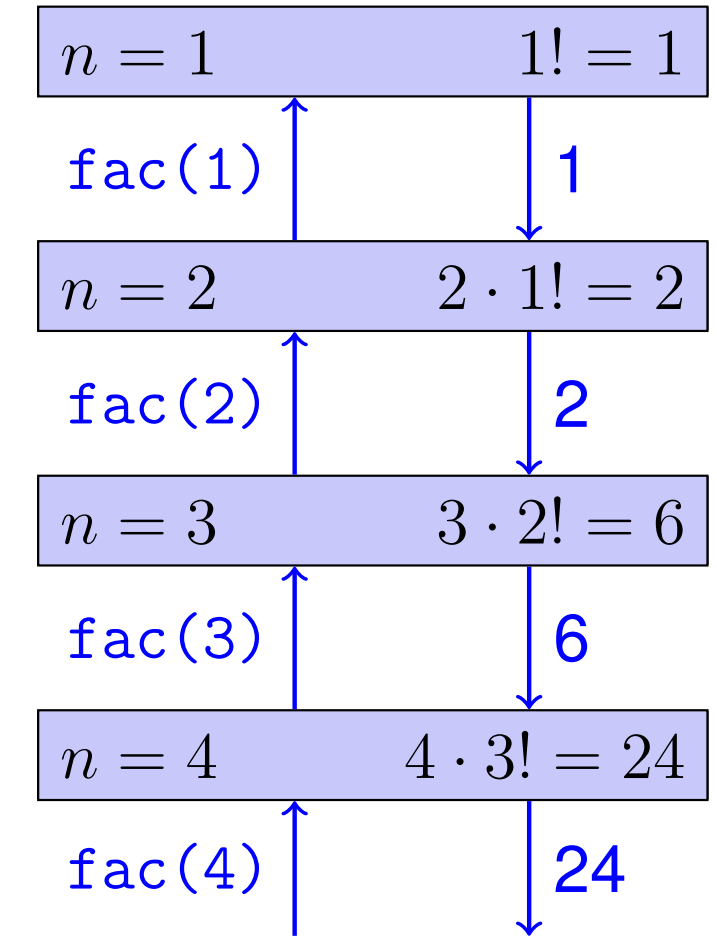
```
// POST: return value is n!
unsigned int fac (const unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

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

Der Aufrufstapel

Bei jedem Funktionsaufruf:

- 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 (Beweis im Skript):

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

Euklidischer Algorithmus in C++

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

```
unsigned int gcd
 (const unsigned int a, const unsigned 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 C++

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 (!)

```
unsigned int fib (const 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.

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 C++

```
unsigned int fib (const 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_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)`

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

a b

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.

Taschenrechner

Ziel: Bau eines Kommandozeilenrechners

Beispiel

Eingabe: `3 + 5`
Ausgabe: `8`
Eingabe: `3 / 5`
Ausgabe: `0.6`
Eingabe: `3 + 5 * 20`
Ausgabe: `103`
Eingabe: `(3 + 5) * 20`
Ausgabe: `160`
Eingabe: `-(3 + 5) + 20`
Ausgabe: `12`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen
- Fliesskommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung
- Unärer Operator `-`

Naiver Versuch (ohne Klammern)

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {

    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

```
Eingabe 2 + 3 * 3 =
Ergebnis 15
```

Analyse des Problems

Beispiel

Eingabe:

$$\underbrace{13 + 4 * (15 - 7*}\ 3) =$$

Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Formale Grammatiken

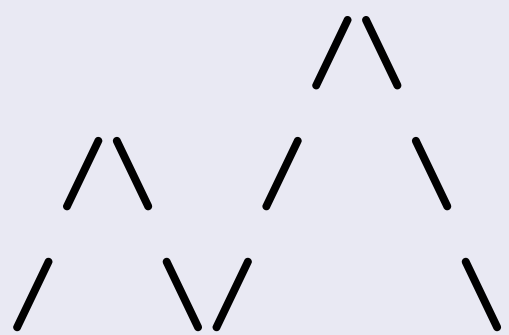
- Alphabet: endliche Menge von Symbolen Σ
- Sätze: endlichen Folgen von Symbolen Σ^*

Eine formale Grammatik definiert, welche Sätze gültig sind.

Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

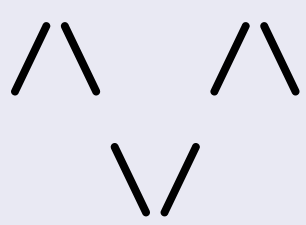
$m' =$ //\\//\\\



Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m''' = /\backslash\backslash//\backslash \notin \mathcal{M}$

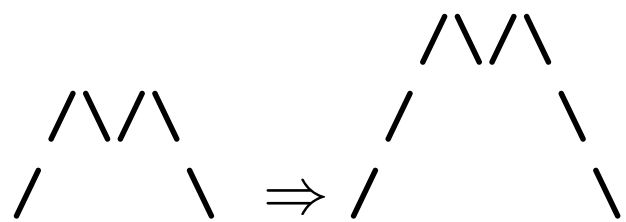


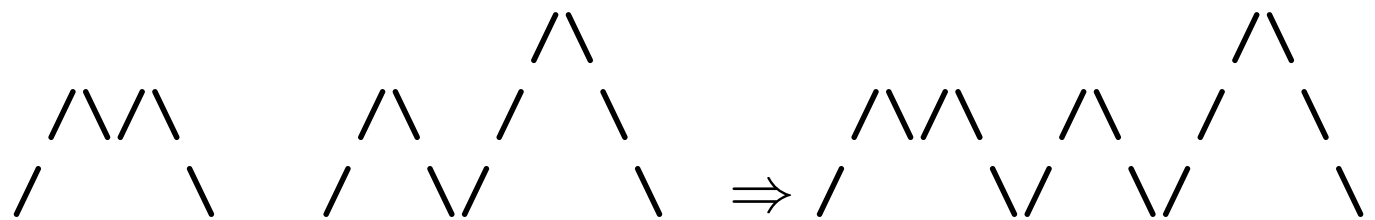
Beide Seiten müssen gleiche Starthöhe haben.
Ein Berg darf nicht unter seine Starthöhe fallen.

Berge in Backus-Naur-Form (BNF)

Mögliche Berge

1 /\

2 

3 

Man kann beweisen, dass diese BNF “unsere” Berge beschreibt, was a priori nicht ganz klar ist.

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (Ausdruck)
  -Zahl, -(Ausdruck)

  Faktor
- Faktor * Faktor, Faktor
  Faktor * Faktor / Faktor , ...

  Term
- Term + Term, Term
  Term - Term, ...

  Ausdruck

Die BNF für Ausdrücke

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.

term       = factor
           | factor "*" term
           | factor "/" term.

expression = term
           | term "+" expression
           | term "-" expression.
```

Parser und BNF

- **Parsen:** Feststellen, ob ein Satz nach der BNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der BNF kann (fast) automatisch ein Parser generiert werden:
  - Regeln werden zu Funktionen
  - Alternativen werden zu `if`–Anweisungen
  - Nichtterminale Symbole auf der rechten Seite werden zu Funktionsaufrufen

Funktionen (Parser mit Auswertung)

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: extracts a factor from is
//       and returns its value
double factor (std::istream& is);

// POST: extracts a term from is
//       and returns its value
double term (std::istream& is);

// POST: extracts an expression from is
//       and returns its value
double expression (std::istream& is);
```

Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: leading whitespace characters are extracted
//       from is, and the first non-whitespace character
//       is returned (0 if there is no such character)
char lookahead (std::istream& is)
{
  is >> std::ws;           // skip whitespaces
  if (is.eof())
    return 0;              // end of stream
  else
    return is.peek();      // next character in is
}
```

Faktoren auswerten

```
double factor (std::istream& is)
{
  double v;
  char c = lookahead (is);
  if (c == '(') {
    v = expression (is >> c);
    is >> c; // consume ")"
  } else if (c == '-') {
    v = -factor (is >> c);
  } else {
    is >> v;
  }
  return v;
}
```

```
factor = "(" expression ")"
       | "-" factor
       | unsigned_number.
```

Terme auswerten

```
double term (std::istream& is)
{
  double v = factor (is);
  char c = lookahead (is);
  if (c == '*')
      v *= term (is >> c);
  else if (c == '/')
      v /= term (is >> c);
  return v;
}
```

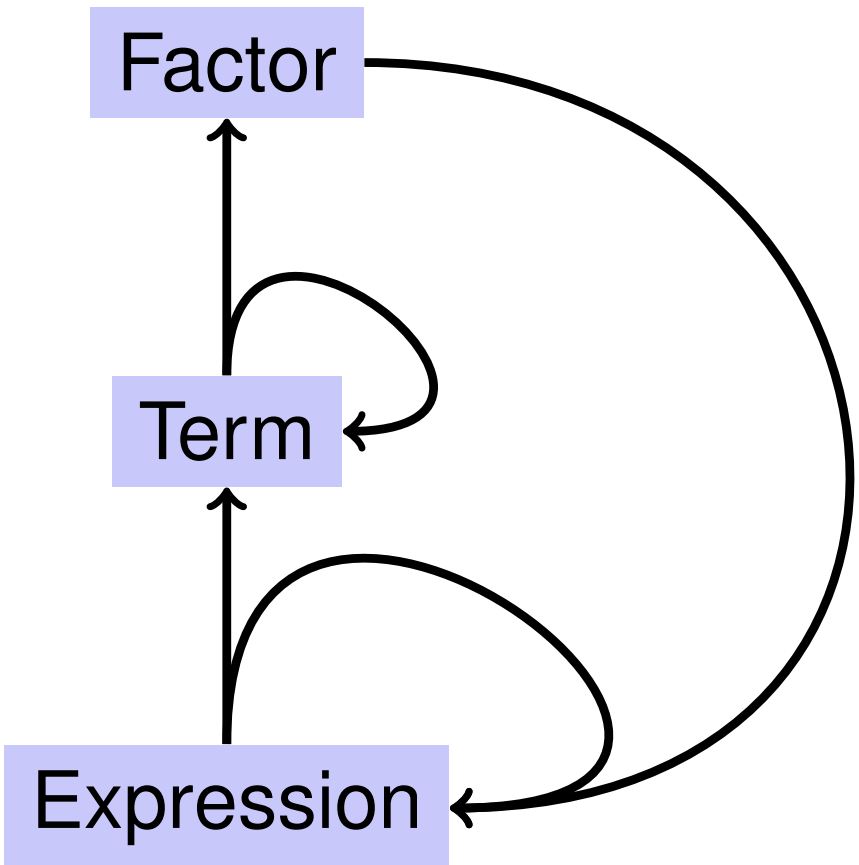
```
term = factor
     | factor "*" term
     | factor "/" term.
```

Ausdrücke auswerten

```
double expression (std::istream& is)
{
  double v = term(is);
  char c = lookahead (is);
  if (c == '+')
      v += expression (is >> c);
  else if (c == '-')
      v -= expression (is >> c);
  return v;
}
```

```
expression = term
           | term "+" expression
           | term "-" expression.
```

Rekursion!



Wir testen

- Eingabe: `-(3-(4-5))*(3+4*5)/6`
  Ausgabe: `-15.33333` ✓
- Eingabe: `3-4-5`
  Ausgabe: `4` 

Wir testen

 Ist die BNF falsch ?

```
expression = term
           | term "+" expression
           | term "−" expression.
```

- Nein, denn sie spricht nur über die Gültigkeit von Ausdrücken, nicht über deren Werte!
- Die Auswertung haben wir naiv “obendrauf” gesetzt.

Dem Problem auf den Grund gehen

Reduktion auf den kritischen Fall.

$$3 - 4 - 5$$

(Fast) minimale BNF:

```
sum  = val | val "−" sum | val "+" sum
```

```
double val (std::istream& is){
   double v;
   is >> v;
   return v;
}
```

Dem Problem auf den Grund gehen

```
// sum = val | val "-" sum | val "+" sum
double sum (std::istream& is){
   double v = val (is);
   char c = lookahead (is);
   if (c == '+')
      v += sum (is >> c);
   else if (c == '-')
      v -= sum (is » c);
   return v;
}
...
std::stringstream input ("3-4-5");
std::cout << sum (input) << "\n"; // 4
```

5	5
4 - "5"	-1
3 - "4 - 5"	4
"3 - 4 - 5"	4

Was ist denn falsch gelaufen?

Die BNF

- spricht offiziell zwar nicht über Werte,
- legt uns aber trotzdem die falsche Klammerung (von rechts nach links) nahe.

```
sum = val "-" sum
```

führt sehr natürlich zu

$3 - 4 - 5 = 3 - (4 - 5)$

Der “richtige” Wert einer Summe $v \pm s$

Bisher (Auswertung von rechts nach links):

$$\Sigma(v \pm s) = \begin{cases} v \pm v', & \text{falls } s = v' \\ v \pm \Sigma(s), & \text{sonst} \end{cases}$$

Neu (Auswertung von links nach rechts):

$$\Sigma(v \pm s) = \begin{cases} v \pm v', & \text{falls } s = v' \\ \Sigma((v \pm v') \pm s'), & \text{falls } s = v' + s' \\ \Sigma((v \pm v') \mp s'), & \text{falls } s = v' - s' \end{cases}$$

$\Sigma(v \pm s)$ ...in C++

```
// sum = val | val "+" sum | val "-" sum.
double sum (double v, char sign, std::istream& s){
   if (sign == '+')
       v += val (s);
   else  if (sign == '-')
       v -= val (s);
   char c = lookahead (s);
   if (c == '+' || c == '-')
       return sum (v, c, s >> c);
   else
       return v;
}
 ...
std::stringstream input ("3-4-5");
std::cout << sum (0, '+', input) << "\n"; // -6
```

-1, '-', "5"	-6
3, '-',"4 - 5",	-6
0,'+',"3 - 4 - 5"	-6

Zurück zur Vollversion

```
// expression = term | term "+" expression
//                   | term "-" expression.
double expression (double v, char sign, std::istream& is) {
    if (sign == '+')
        v += term (is);
    else if (sign == '-')
        v -= term (is);
    char c = lookahead (is);
    if (c == '+' || c == '-')
        return expression (v, c, is >> c);
    return v;
}

double expression (std::istream & is){
    return expression(0, '+', is );
}
```

Das gleiche für Terme – fertig

```
// term = factor | factor "*" term
//               | factor "/" term.
double term (double v, char sign, std::istream& is) {
    if (sign == '*')
        v *= factor (is);
    else if (sign == '/')
        v /= factor (is);
    char c = lookahead (is);
    if (c == '*' || c == '/')
        return term (v, c, is >> c);
    return v;
}

double term (std::istream & is){
    return term(1, '*', is);
}
```