

16. Rekursion 2

Bau eines Taschenrechners, Ströme, Formale Grammatiken, Extended Backus Naur Form (EBNF), Parsen von Ausdrücken

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

Motivation: 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 -

535

536

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * (15 - 7 * 3) =$$

Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!

Das "Verstehen" eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

537

538

Einschub als Vorbereitung: Ströme

Ein Programm verarbeitet Eingaben von einem konzeptuell unbegrenzten Eingabestrom.

Bisher: Eingabestrom der Kommandozeile `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

↑
Konsumiere `op` von `std::cin`,
Leseposition schreitet fort.

Wir wollen zukünftig aber auch von Dateien lesen können!

$$3 + 5 - 6 * 10 + 800 - 70$$

Beispiel: BSD 16-bit Checksum

```
#include <iostream>
```

```
int main () {
```

```
    char c;  
    int checksum = 0;  
    while (std::cin >> c) {  
        checksum = checksum / 2 + checksum % 2 * 0x8000 + c;  
        checksum %= 0x10000;  
    }  
    std::cout << "checksum = " << std::hex << checksum << "\n";  
}
```

Eingabe: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Erfordert in der Konsole manuelles Ende der Eingabe ⁷
Ausgabe: 0710

539

⁷Ctrl-D(Unix) / Ctrl-Z(Windows) am Anfang einer Zeile, welche durch ENTER abgeschlossen wird

540

Beispiel: BSD 16-bit Checksum mit Datei

```
#include <iostream>  
#include <fstream>
```

Ausgabe: 67fd

```
int main () {  
    std::ifstream fileStream ("loremispum.txt");  
    char c;  
    int checksum = 0;  
    while (fileStream >> c) {  
        checksum = checksum / 2 + checksum % 2 * 0x8000 + c;  
        checksum %= 0x10000;  
    }  
    std::cout << "checksum = " << std::hex << checksum << "\n";  
}
```

Gibt am Dateiende false zurück.

Beispiel: BSD 16-bit Checksum

Wiederverwendung gemeinsam genutzter Funktionalität?

Richtig: mit einer Funktion. Aber wie?

541

542

Beispiel: BSD 16-bit Checksum generisch!

```
#include <iostream>
#include <fstream>
```

Referenz nötig: wir verändern den Strom!

```
int checksum (std::istream& is)
{
    char c;
    int checksum = 0;
    while (is >> c) {
        checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
        checksum %= 0x10000;
    }
    return checksum;
}
```

543

Gleiches Recht für alle!

```
#include <iostream>
#include <fstream>
```

Eingabe: Lorem Yps mit Gimmick
Ausgabe: checksums differ

```
int checksum (std::istream& is) { ... }

int main () {
    std::ifstream fileStream("loremipsum.txt");

    if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
    else
        std::cout << "checksums differ.\n";
}
```

544

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.
- Unsere Variable `fileStream` ist vom Typ `std::ifstream`. Sie repräsentiert einen Eingabestrom auf einer Datei.
- Ein `std::ifstream` *ist auch ein* `std::istream`, kann nur etwas mehr.
- Somit kann `fileStream` überall dort verwendet werden, wo ein `std::istream` verlangt ist.

545

Nochmal gleiches Recht für alle!

```
#include <iostream>
#include <fstream>
#include <sstream>
```

Eingabe aus stringstream
Ausgabe: checksums differ

```
int checksum (std::istream& is) { ... }

int main () {
    std::ifstream fileStream ("loremipsum.txt");
    std::stringstream stringstream ("Lorem Yps mit Gimmick");

    if (checksum (fileStream) == checksum (stringstream))
        std::cout << "checksums match.\n";
    else
        std::cout << "checksums differ.\n";
}
```

546

Berge in Backus-Naur-Form (BNF)

`berg = "/" | "/" berg | berg berg.` **Regel**

Mögliche Berge

1 / \

2 / \ / \ ⇒ / \ / \ / \

3 / \ / \ / \ ⇒ / \ / \ / \ / \

Alternativen

Nichtterminal

Terminal

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

551

Ausdrücke

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

Was benötigen wir in einer BNF?

- Zahl, (Ausdruck)
- Faktor * Faktor, Faktor / Faktor, ...
- Term + Term, Term - Term, ...

Faktor

Term

Ausdruck

552

Die BNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

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

553

Die BNF für Ausdrücke

Ein Term ist

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

Wir brauchen eine Repetition!

554

EBNF

Extended Backus Naur Form: Erweiterung der BNF um

- Option [] und
- Optionale Repetition {}

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

NB: die EBNF ist nicht reichhaltiger als die BNF. Sie erlaubt nur eine kompaktere Schreibweise. Obiges Konstrukt lässt sich mit BNF z.B. so schreiben:

```
term = factor | factor T.  
T = "*" term | "+" term.
```

555

Die EBNF für Ausdrücke

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

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

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

556

Parse

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

557

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);
```

558

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)
{
    if (is.eof())
        return 0;
    is >> std::ws;      // skip whitespaces
    if (is.eof())
        return 0;      // end of stream
    return is.peek();  // next character in is
}
```

559

Rosinenpickerei

... um jeweils nur das gewünschte Zeichen zu extrahieren.

```
// POST: if ch matches the next lookahead then consume it
//       and return true. return false otherwise
bool consume (std::istream& is, char ch)
{
    if (lookahead(is) == ch){
        is >> ch;
        return true;
    }
    return false;
}
```

560

Faktoren auswerten

```
double factor (std::istream& is)
{
    double v;
    if (consume(is, '(')){
        v = expression (is);
        consume(is, ')');
    } else if (consume(is, '-'))
        v = -factor (is);
    else
        is >> v;
    return v;
}
```

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

561

Terme auswerten

```
double term (std::istream& is)
{
    double value = factor (is);
    while(true){
        if (consume(is, '*'))
            value *= factor (is);
        else if (consume(is, '/'))
            value /= factor(is)
        else
            return value;
    }
}
```

```
term = factor { "*" factor | "/" factor }
```

562

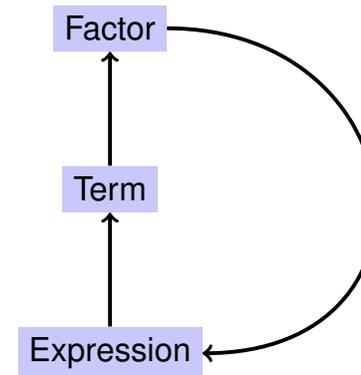
Ausdrücke auswerten

```
double expression (std::istream& is)
{
    double value = term(is);
    while(true){
        if (consume(is, '+'))
            value += term (is);
        else if (consume(is, '-'))
            value -= term(is)
        else
            return value;
    }
}
```

```
expression = term { "+" term | "-" term }
```

563

Rekursion!



564

EBNF — Und es funktioniert!

EBNF (calculator.cpp, Auswertung von links nach rechts):

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

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

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

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // -4
```

565

BNF — Und es funktioniert **nicht!**

BNF (calculator_r.cpp, Auswertung von rechts nach links):

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

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

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

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // 2
```

566

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

3, 3 - 5, 3 - 7 - 1

EBNF:

```
sum = value {"-" value | "+" value}.
```

BNF:

```
sum = value | value "-" sum | value "+" sum.
```

Die beiden Grammatiken erlauben dieselben Ausdrücke.

value

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

567

568

EBNF Variante

```
// sum = value {"-" value | "+" value}.
double sum(std::istream& is) {
    double v = value(is);
    while(true){
        if (consume(is, '-'))
            v -= value(is);
        else if (consume(is, '+'))
            v += value(is);
        else
            return v;
    }
}
```

Wir testen: EBNF Variante

- Eingabe: 1-2
Ausgabe: -1 ✓
- Eingabe: 1-2-3
Ausgabe: -4 ✓

569

570

BNF Variante

```
// sum = value | value "-" sum | value "+" sum.
double sum(std::istream& is){
    double v = value(is);
    if (consume(is, '-'))
        return v - sum(is);
    else if(consume(is, '+'))
        return v + sum(is);
    return v;
}
```

Wir testen: BNF Variante

- Eingabe: 1-2
Ausgabe: -1 ✓
- Eingabe: 1-2-3
Ausgabe: 2 😞

571

572

Wir testen

😞 Ist die BNF falsch ?

```
sum = value
    | value "-" sum
    | value "+" sum.
```

- 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

```
double sum (std::istream& is){
    double v = value (is);
    if (consume (is, '-'))
        v -= sum (is);
    else if (consume (is, '+'))
        v += sum(is);
    return v;
}
```

```
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

3	3
2 - "3"	-1
1 - "2 - 3"	2
"1 - 2 - 3"	2

573

574

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 = value | value "-" sum | value "+" sum.
```

führt sehr natürlich zu

```
1 - 2 - 3 = 1 - (2 - 3)
```

575

Eine Lösung: Linksrekursion

```
sum = value | sum "-" value | sum "+" value.
```

Implementationsmuster von vorher funktioniert nicht mehr.
Linksrekursion muss wieder zu Rechtsrekursion aufgelöst werden.

Das sähe dann so aus:

```
sum = value | value s.  
s = "-" sum | "+" sum.
```

Siehe `calculator_1.cpp`

576