17. Rekursion 2

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

Motivation: Taschenrechner

Beispiel

Eingabe: `3 + 5`
Ausgabe: `8`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen

Motivation: Taschenrechner

Beispiel

Eingabe: `3 / 5`
Ausgabe: `0.6`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen
- Fliesskommaarithmetik

Motivation: Taschenrechner

Beispiel

Eingabe: `3 + 5 * 20`
Ausgabe: `103`

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

Motivation: Taschenrechner

Beispiel

Eingabe: `(3 + 5) * 20`
Ausgabe: `160`

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

Motivation: Taschenrechner

Beispiel

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";
```

Scheint zu klappen...

```
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 1 * 2 * 3 * 4 =
Ergebnis 24
```

Oops, Strich- vor Punktrechnung...

```
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:

$$13 + \ldots$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * \ldots$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * (15 - \ldots$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * (15 - 7 * \ldots$$

Analyse des Problems

Beispiel

Eingabe:

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

Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!

Analyse des Problems

Beispiel

Ergebnis:

$$13 + 4 * (15 - 21)$$

Analyse des Problems

Beispiel

Ergebnis:

$$13+4*(-6)$$

Analyse des Problems

Beispiel

Ergebnis:

$$13 + (-24)$$

Analyse des Problems

Beispiel

Ergebnis:

$$-11$$

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese Vorlesung

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese Vorlesung ist

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese Vorlesung ist insgesamt

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese Vorlesung ist insgesamt recht

Analyse des Problems

Beispiel

Ausdruck:

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

Beispiel

Diese Vorlesung ist insgesamt recht rekursiv.

Analyse des Problems

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von C++unabhängiges) Handwerkszeug.

Analyse des Problems

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von C++unabhängiges) Handwerkszeug.

Analyse des Problems

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von C++unabhängiges) Handwerkszeug.

Analyse des Problems

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von $\mathrm{C{+}{+}}$unabhängiges) Handwerkszeug.

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Zur Beschreibung der Grammatik verwenden wir:

Extended Backus Naur Form (EBNF)

What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, and
Xerox Palo Alto Research Center

Key Words and Phrases: syntactic description language, extended BNF
CR Categories: 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietory circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasymbols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as ⟨empty⟩ or ϵ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax     = {production}.
production = identifier "=" expression ".".
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | literal | "(" expression ")" |
             "[" expression "]" | "{" expression "}".
literal    = """" character {character} """".
```

Repetition is denoted by curly brackets, i.e. {a} stands for ϵ | a | aa | aaa | . . . . Optionality is expressed by square brackets, i.e. [a] stands for a | ϵ. Parentheses merely serve for grouping, e.g. (a|b)c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Received January 1977; revised February 1977

Ausdrücke

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

Was benötigen wir in einer Grammatik?

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- ? * ?,   ? / ?,   ...

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- ? * ?,   ? / ?,   ...
- ? - ?,   ? + ?,   ...

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- ? * ?,   ? / ?,   ...
- ? - ?,   ? + ?,   ...

Faktor

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- Faktor * Faktor,
  Faktor / Faktor , ...
- ? - ?,   ? + ?,   ...

Faktor

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- Faktor * Faktor,
  Faktor / Faktor , ...
- ? - ?, ? + ?, ...

Faktor

Term

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- Faktor ∗ Faktor, Faktor
  Faktor / Faktor , ...
- ? - ?, ? + ?, ...

Faktor

Term

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- Faktor ∗ Faktor, Faktor
  Faktor / Faktor , ...
- Term + Term,
  Term - Term, ...

Faktor

Term

Ausdrücke

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

Was benötigen wir in einer Grammatik?

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

Faktor

Term

Ausdruck

Ausdrücke

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

Was benötigen wir in einer Grammatik?

- Zahl , (?)
  -Zahl, -(?)
- Faktor ∗ Faktor, Faktor
  Faktor / Faktor , ...
- Term + Term, Term
  Term − Term, ...

Faktor

Term

Ausdruck

Ausdrücke

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

Was benötigen wir in einer Grammatik?

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

Faktor

Term

Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

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

$$\text{factor} = \text{unsigned_number} \mid \text{"("}\ \text{expression}\ \text{")"} \mid \text{"-"}\ \text{factor}\ .$$

Die EBNF für Ausdrücke

Ein Faktor ist

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

$$
\begin{aligned}
\text{factor} \quad &= \text{unsigned_number} \\
&\mid \text{"("}\ \text{expression}\ \text{")"} \\
&\mid \text{"}-\text{"}\ \text{factor}\ .
\end{aligned}
$$

Die EBNF für Ausdrücke

Ein Faktor ist

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

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

Die EBNF für Ausdrücke

Ein Faktor ist

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

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

Die EBNF für Ausdrücke

Ein Faktor ist

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

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

Nicht-terminales Symbol

Terminales Symbol

Alternative

Die EBNF für Ausdrücke

Ein Term ist

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

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

Die EBNF für Ausdrücke

Ein Term ist

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

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

Die EBNF für Ausdrücke

Ein Term ist

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

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

Die EBNF für Ausdrücke

Ein Term ist

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

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

Die EBNF für Ausdrücke

Ein Term ist

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

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

Optionale Repetition

Die EBNF für Ausdrücke

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

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

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

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der EBNF 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

Regeln

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

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

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

Funktionen (Parser)

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: returns true if and only if is = factor ...
//       and in this case extracts factor from is
bool factor (std::istream& is);

// POST: returns true if and only if is = term ...,
//       and in this case extracts all factors from is
bool term (std::istream& is);

// POST: returns true if and only if is = expression ...,
//       and in this case extracts all terms from is
bool expression (std::istream& is);
```

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

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 ;
}
```

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.
```

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 }.

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 }.

Rekursion!

Factor

Term

Expression

Rekursion!

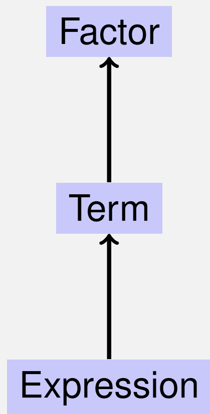
Factor

Term

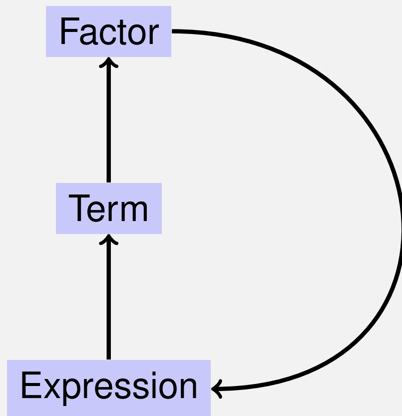
↑

Expression

Rekursion!



Rekursion!



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
```