

15. Rekursion 2

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

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
- Präcedenzen und Assoziativitäten wie in C++
- Klammerung
- Unärer Operator -

497

498

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

499

Analyse des Problems

Beispiel

Eingabe:

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

Muss gespeichert bleiben,
damit jetzt ausgewertet werden kann!

500

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.

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)

Short Communications
Programming Languages

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 proprietary 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, 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.

Communications
of
the ACM

November 1977
Volume 20
Number 11

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasympols 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 ε | a. 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?

- 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. *Nicht-terminales Symbol*

factor = number
| "(" expression ")"
| "-" factor.
Alternative *Terminales Symbol*

505

Die EBNF für Ausdrücke

Ein Term ist

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

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

506

Die EBNF für Ausdrücke

factor = number
| "(" expression ")"
| "-" factor.

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

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

507

Zahlen

Eine *Ganzzahl* hat mindestens eine Ziffer, gefolgt von beliebig vielen Ziffern.

number = digit { digit }.
digit = '0' | '1' | '2' | ... | '9'.

508

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

Parser bauen

- 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

509

510

Regeln (ohne number)

```
factor      = 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);
```

511

512

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);
```

513

Vorausschau von einem Zeichen...

... um jeweils die richtige Alternative zu finden.

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

514

Rosinenpickerei

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

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

515

Faktoren auswerten

```
double factor (std::istream& input)
{
    double value;
    if (consume (input, '(')) {
        value = expression (input);    // "(" expression
        consume (input, ')');          // ")"
    } else if (consume (input, '-'))
        value = -factor (input);      // - factor
    else
        value = number(input);
    return value;
}
```

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

516

Terme auswerten

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

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

517

Ausdrücke auswerten

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

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

518

Ziffern ...

```
// POST: returns the digit that could be consumed from a stream
//       (0 if no digit available)
// digit = '0' | '1' | ... | '9'.
char digit(std::istream& input){
    char ch = input.peek(); // one symbol lookahead
    if (input.eof()) return 0; // nothing available on the stream
    if (ch >= '0' && ch <= '9'){
        input >> ch; // consume
        return ch;
    }
    return 0;
}
```

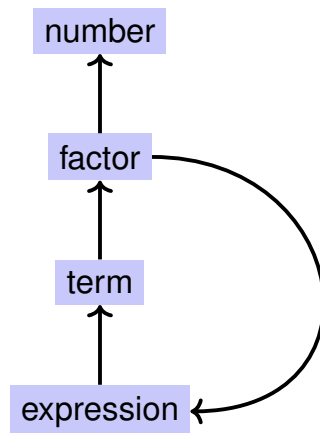
519

... und Zahlen

```
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int number (std::istream& input){
    input >> std::skipws; // skip whitespaces before the first digit
    char ch = digit(input);
    input >> std::noskipws; // no whitespaces allowed within a number
    unsigned int num = 0;
    while(ch > 0){ // skip remaining digits
        num = num * 10 + ch - '0';
        ch = digit(input);
    }
    return num;
}
```

520

Rekursion!



EBNF — Und es funktioniert!

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

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

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

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

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

521

522

16. Structs

Rationale Zahlen, Struct-Definition, Funktions- und Operatorüberladung

Rechnen mit rationalen Zahlen

- Rationale Zahlen ($\mathbb{Q}$) sind von der Form $\frac{n}{d}$ mit n und d in $\mathbb{Z}$
- C++ hat keinen „eingebauten“ Typ für rationale Zahlen

Ziel

Wir bauen uns selbst einen C++-Typ für rationale Zahlen! 😊

523

524

Vision

So könnte (wird) es aussehen

```
// compute
std::cout << "Rational number r=? ";
rational r;
std::cin >> r;
std::cout << "Rational number s=? ";
rational s;
std::cin >> s;

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

Ein erstes Struct

```
struct rational {
    int n; ← Member-Variable (numerator)
    int d; ← INV: d != 0
};
```

Member-Variable (**d**enominator)

Invariante: spezifiziert gültige Wertkombinationen (informell).

- struct definiert einen neuen *Typ*
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen
- Echter Wertebereich: `rational` $\subsetneq$ `int` $\times$ `int`.

525

526

Zugriff auf Member-Variablen

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

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;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

Ein erstes Struct: Funktionalität

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

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

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

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

Member-Zugriff auf die `int`-Objekte von `a`.

527

528

Eingabe

```
// Input r
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? ";
std::cin >> r.n;
std::cout << " denominator =? ";
std::cin >> r.d;

// Input s the same way
rational s;
...
```

Vision in Reichweite ...

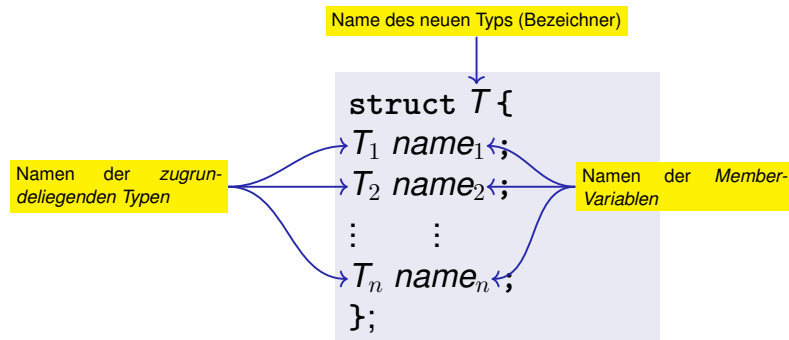
```
// computation
const rational t = add (r, s);

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

529

530

Struct-Definitionen



Wertebereich von T : $T_1 \times T_2 \times \dots \times T_n$

Struct-Definitionen: Beispiele

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

Zugrundeliegende Typen können fundamentale aber auch **benutzerdefinierte** Typen sein.

531

532

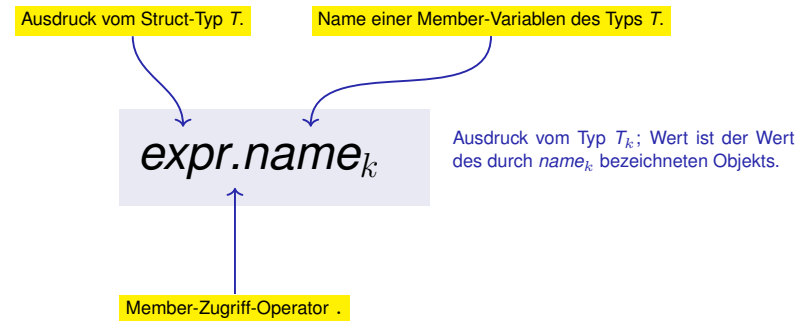
Struct-Definitionen: Beispiele

```
struct extended_int {  
    // represents value if is_positive==true  
    // and -value otherwise  
    unsigned int value;  
    bool is_positive;  
};
```

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

533

Structs: Member-Zugriff



534

Structs: Initialisierung und Zuweisung

Default-Initialisierung:

```
rational t;
```

- Member-Variablen von t werden default-initialisiert
- für Member-Variablen fundamentaler Typen passiert dabei nichts (Wert undefiniert)

535

Structs: Initialisierung und Zuweisung

Initialisierung:

```
rational t = {5, 1};
```

- Member-Variablen von t werden mit den Werten der Liste, entsprechend der Deklarationsreihenfolge, initialisiert.

536

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational s;  
...  
rational t = s;
```

- Den Member-Variablen von `t` werden die Werte der Member-Variablen von `s` zugewiesen.

537

Structs: Initialisierung und Zuweisung

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

Initialisierung:

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

- `t` wird mit dem Wert von `add(r, s)` initialisiert

538

Structs: Initialisierung und Zuweisung

Zuweisung:

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

- `t` wird default-initialisiert
- Der Wert von `add(r, s)` wird `t` zugewiesen

539

Structs: Initialisierung und Zuweisung

```
rational s; ← Member-Variablen uninitialized (wird  
sich bald ändern)
```

```
rational t = {1,5}; ← Memberweise Initialisierung:  
t.n = 1, t.d = 5
```

```
rational u = t; ← Memberweise Kopie
```

```
t = u; ← Memberweise Kopie
```

```
rational v = add(u,t); ← Memberweise Kopie
```

540

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`, ...) gibt es die Vergleichsoperatoren `==` und `!=`, aber nicht für Structs! Warum?

- Memberweiser Vergleich ergibt im allgemeinen keinen Sinn,...
- ...denn dann wäre z.B. $\frac{2}{3} \neq \frac{4}{6}$

Structs als Funktionsargumente

```
void increment(rational dest, const rational src)
{
    dest = add (dest, src); // veraendert nur lokale Kopie
}
```

Call by Value !

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a); // kein Effekt!
std::cout << b.n << "/" << b.d; // 1 / 2
```

541

542

Structs als Funktionsargumente

```
void increment(rational &dest, const rational src)
{
    dest = add (dest, src);
}
```

Call by Reference

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a);
std::cout << b.n << "/" << b.d; // 2 / 2
```

543

Benutzerdefinierte Operatoren

Statt

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

würden wir lieber

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

schreiben.

Das geht mit *Operator-Überladung*.

544

Überladen von Funktionen

- Funktionen sind durch Ihren Namen im Gültigkeitsbereich ansprechbar
- Es ist sogar möglich, mehrere Funktionen des gleichen Namens zu definieren und zu deklarieren
- Die „richtige“ Version wird aufgrund der *Signatur* der Funktion ausgewählt

545

Funktionsüberladung

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

```
double sq (double x) { ... }      // f1
int sq (int x) { ... }           // f2
int pow (int b, int e) { ... }   // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“ (wir vertiefen das nicht)

```
std::cout << sq (3);           // Compiler wählt f2
std::cout << sq (1.414);       // Compiler wählt f1
std::cout << pow (2);          // Compiler wählt f4
std::cout << pow (3,3);        // Compiler wählt f3
```

546

Operator-Überladung (Operator Overloading)

- 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

547

rational addieren, 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;
}
...
const rational t = add (r, s);
```

548

rational addieren, 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;
}
...
const rational t = r + s;
```

↑
Infix-Notation

549

Andere binäre Operatoren für rationale Zahlen

```
// POST: return value is 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);
```

550

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;
}
```

551

Vergleichsoperatoren

Sind für Structs nicht eingebaut, können aber definiert werden:

```
// POST: returns true iff a == b
bool operator== (rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

$$\frac{2}{3} = \frac{4}{6} \quad \checkmark$$

552

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;    // 5/6
```

Operator +=

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

Das funktioniert!

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

`r += s;` hat nun den gewünschten Effekt.

553

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;  
}
```

Das funktioniert nicht! Warum?

- Der Ausdruck `r += s` hat zwar den gewünschten Wert, weil die Aufrufargumente R-Werte sind (call by value!) jedoch **nicht den gewünschten Effekt** der Veränderung von `r`.
- Das Resultat von `r += s` stellt zudem entgegen der Konvention von C++ keinen L-Wert dar.

554

Ein-/Ausgabeoperatoren

können auch überladen werden.

- Bisher:

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

- Neu (gewünscht):

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

555

556

Ein-/Ausgabeoperatoren

können auch überladen werden wie folgt:

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

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

557

Eingabe

```
// PRE: in starts with a rational number
// of the form "n/d"
// POST: r has been read from in
std::istream& operator>> (std::istream& in,
                        rational& r){
    char c; // separating character '/'
    return in >> r.n >> c >> r.d;
}
```

liest `r` aus dem Eingabestrom
und gibt diesen als L-Wert zurück.

558

Ziel erreicht!

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

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

operator >>

operator +

operator<<

559