

3	Ausdrücke
3.1	Zum Algorithmusbegriff
3.2	Arithmetische Ausdrücke
3.2.1	Eindeutigkeit der Auswertung
3.2.2	Vollgeklammerte Ausdrücke
3.2.3	Formbeschreibung
3.2.4	Baumdarstellung
3.3	Logische Ausdrücke
3.3.1	Formbeschreibung
3.3.2	Boolesche Algebra
3.4	C – Ausdrücke in C++
3.5	Minimierung von Schaltfunktionen
3.6	Begriffe aus der Syntaxtheorie
3.7	EBNF nach ISO/IEC 14977
3.8	EBNF-Notation für C++ – Ausdrücke

Zum Algorithmusbegriff:

Erläuterung: Ein Algorithmus ist ein allgemeines Verfahren zur Lösung einer Klasse gleichartiger Probleme.

Bemerkung: Das Wort Algorithmus geht zurück auf den Namen des persischen Gelehrten Abu Jafar Muhammad ibn Musa al-Khwarizmi, der etwa von 780 bis 850 n. Chr. in Bagdad lebte.

Beispiel: Algorithmus zur Bildung der Summe zweier ganzer Zahlen x und y .

1. Falls x kleiner als Null ist, fahre bei 7 fort.
2. Falls x größer als Null ist, fahre bei 5 fort.
3. Der Wert von y ist die gesuchte Summe.
4. Die Berechnung ist beendet.
5. Erniedrige x um 1.
6. Erhöhe y um 1, fahre bei 2 fort.
7. Erhöhe x um 1.
8. Erniedrige y um 1, fahre bei 1 fort.

Die obige Beschreibung hat einige wünschenswerte Eigenschaften. So ist die Anzahl der Verfahrensschritte klein, jeder Schritt ist leicht verständlich, jede Summation ist nach einer endlichen Zahl von Schritten beendet. Negativ ist zu vermerken, der Detaillierungsgrad ist zu hoch, falls ein Text wie der obige mehr als 1000 Verfahrensschritte enthält, dann läßt sich nur in seltenen Fällen nachweisen, daß er ein Lösungsschema für die ursprüngliche Problemklasse darstellt. Im vorhergehenden Kapitel wurden schon Strukturierungskonstrukte für Algorithmexte vorgestellt.

```
// Algorithmus zur Bestimmung des Ostersonntags nach
// Aloysius Lilius und Christopher Clavius für die Jahre
// nach 1582 A.D.
// Ostersonntag fällt auf den ersten Sonntag nach Voll-
// mond nach dem 20. März, damit ist das früheste Datum
// der 22. März, das späteste der 25. April.
```

```
int ostertag (int jahr) {
    int mj = jahr % 19 + 1;    // Jahr im
                               // Metonischen Zyklus
    int jh = jahr / 100 + 1;   // Jahrhundert

    int korr1 = 3*jh / 4 - 12; // Schaltjahrkorrektur
    int korr2 = (8*jh + 5) / 25 - 5; // Synchronisierung
                               // mit Mondzyklus

    int sonnt = 5*jahr/4 - korr1 - 10; // Sonntagsabgleich
    int epakte = (11*mj + 20 + korr2 - korr1) % 30;
    epakte = (epakte + 30) % 30; // Vorsichtsmaßnahme
    if (((epakte == 25) && (mj > 11)) || (epakte == 24))
        ++epakte;

    int vm = 44 - epakte;      // Vollmondbestimmung
    if (vm < 21)
        vm += 30;
    vm = vm + 7 - ((sonnt + vm) % 7);
        // Falls vm > 31, dann
        // Ostersonntag = (vm-31)-ter April, sonst
        // Ostersonntag = vm-ter März.

    return vm;
} //ostertag
```

Markov-Algorithmen (A. A. Markov, 1951):

```
Regeln:   a0  → 0a
          a1  → 1b
          b0  → 1a
          b1  → 0b
          a   → ▪
          b   → ▪
           → a
```

```
Ausführung: 1 0 0 1 1 1 0 1
             a 1 0 0 1 1 1 0 1
             1 b 0 0 1 1 1 0 1
             1 1 a 0 1 1 1 0 1
             1 1 0 a 1 1 1 0 1
             1 1 0 1 b 1 1 0 1
             1 1 0 1 0 b 1 0 1
             1 1 0 1 0 0 b 0 1
             1 1 0 1 0 0 1 a 1
             1 1 0 1 0 0 1 1 b
             1 1 0 1 0 0 1 1
```

Verfahren:

Das Wort auf der linken Seite einer Regel wird in der zu bearbeitenden Zeichenkette gesucht und durch das Wort auf der rechten Regelseite ersetzt. Dabei werden die Regeln von oben nach unten auf ihre Anwendbarkeit überprüft. Die erste passende Regel wird angewandt, indem das am weitesten links stehende Teilwort ersetzt wird. Das Verfahren endet, falls eine Endregel angewandt wurde, gekennzeichnet durch einen Endepunkt "▪", oder keine Regel mehr anwendbar ist.

**Beispiel eines Markov-Algorithmus:
Dopplung einer Bitkette:**

Regeln: 00β → 0β0 /* Vertauschung */
 01β → 1β0
 10β → 0β1
 11β → 1β1
 α0 → 0β0α /* Dopplung */
 α1 → 1β1α
 β → γ /* Beenden der */
 γ → /* Dopplung */
 α → ▪
 → α

Anwendung: Dopplung von 101:

101
 α101
 1β1α01
 1β10β0α1
 1β0β10α1
 1β0β101β1α
 1β0β11β01α
 1β0β1β101α
 1γ0β1β101α
 1γ0γ1β101α
 1γ0γ1γ101α
 10γ1γ101α
 101γ101α
 101101α
 101101

Berechenbarkeit:

Es gibt viele Präzisierungen des Begriffs der "Berechenbarkeit einer Funktion".

Beispiele sind:

Markov Algorithmen,

Turing-Maschinen,

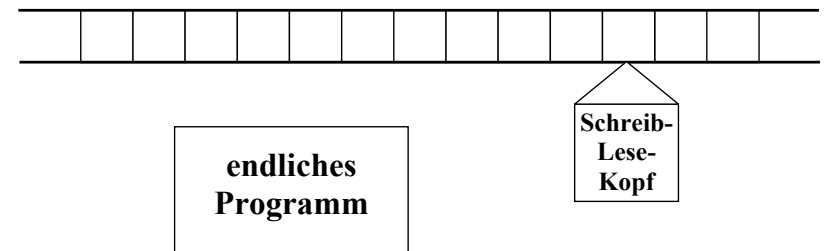
Lambda Kalkül,

Rekursive Funktionen.

Churchsche Vermutung:

Alle "sinnvollen" Definitionen des Begriffs der Berechenbarkeit sind gleichwertig.

Bild einer Turing-Maschine:



Erläuterung von Kleene:

"An algorithm is a finitely described procedure, sufficient to guide us to the answer to any one of infinitely many questions, by finitely many steps in the case of each question."

Eigenschaften eines Algorithmus:

- Finitheit der Beschreibung
- Effektivität
- Determiniertheit
- Terminierung
- Effizienz

Bemerkungen:

- Es existieren Probleme, zu deren Lösung sich kein Algorithmus finden läßt, ein Beispiel hierfür ist das Halteproblem.
- Es existieren Probleme, zu deren Lösung man nur nicht effiziente Algorithmen kennt, ein Beispiel hierfür ist die Klasse der NP-vollständigen Probleme.
- Man kann unsinnige Algorithmen formulieren, Beispiele hierfür sind Lösungsvorschläge für nicht wohlverstandene Probleme.
- Zur Beschreibung von Algorithmen sucht man Formalismen, die es gestatten, Problemlösungen kompakt und eindeutig zu beschreiben, ein Beispiel hierfür sind die Formelsammlungen technischer Fächer.
- Die Eigenschaft der Determiniertheit wird häufig abgeschwächt. Man kennt probabilistische und auch nichtdeterministische Algorithmen.

Beispiele für Formeln:

$$8 * 4$$

true and false

"Wal" + "dlauf"

$$((7 * 3) < (4 * 5)) \text{ und } (2 * 2 = 7)$$

$$x! \approx (\sqrt{2 \pi x}) x^x e^{-x}$$

$$\sin^2 x + \cos^2 x = 1$$

$$\sin(\varphi \pm 2n\pi) = \sin \varphi$$

Auswertung arithmetischer Ausdrücke:

Beispiel: $3 + 6 * 4 - 5 + 8 * 2$

(i) Auswertung von links nach rechts:
 $(((((3 + 6) * 4) - 5) + 8) * 2) = 78$

(ii) Auswertung von rechts nach links:
 $(3 + (6 * (4 - (5 + (8 * 2)))))) = -99$

(iii) übliche Interpretation:
 $((3 + (6 * 4)) - 5) + (8 * 2) = 38$

Bemerkungen:

- (i) Nur im Fall (iii) muß eine Analyse des Ausdrucks erfolgen.
- (ii) Vollgeklammerte arithmetische Ausdrücke vermeiden jedweden Interpretationsspielraum.

Beschreibung vollgeklammerter arithmetischer Ausdrücke mittels Muster:

Ein arithmetischer Ausdruck (= VAA) ist von der Form:

- (1) eine Zahl
- oder (2) (VAA)
- oder (3) (VAA + VAA)
- oder (4) (VAA - VAA)
- oder (5) (VAA * VAA)
- oder (6) (VAA / VAA)

Beispiel: (VAA)

$$\begin{aligned} &\equiv ((VAA + VAA)) \\ &\equiv (((VAA * VAA) + (VAA - VAA))) \\ &\equiv (((VAA * (VAA - VAA)) + (VAA - VAA))) \\ &\equiv (((3 * (0 - 7)) + (6 - 2))) \\ &\equiv (((3 * (0 - 7)) + (6 - 2))) \end{aligned}$$

Bemerkung:

Will man die Zahl der Klammern gering halten, z. B. aus Gründen der Lesbarkeit, dann muß man zusätzliche Auswertungsregeln einführen. Beispiele hierfür sind:

- (i) Punktrechnung geht vor Strichrechnung.
- (ii) Gleichgeordnete Operationen sind links-assoziativ.

Nun ist ein aufwendiger Analyseschritt notwendig, als Beispiel möge Rutishausers Klammergebirge dienen:

$$\begin{aligned} \text{Ausdruck: } & 3 + 2 * 4 * 5 + 2 + 6 + 4 * 3 * 7 \\ & \quad \quad \quad \downarrow \\ & 3 + (2 * 4 * 5) + 2 + 6 + (4 * 3 * 7) \\ & \quad \quad \quad \downarrow \\ & 3 + ((2 * 4) * 5) + 2 + 6 + ((4 * 3) * 7) \\ & \quad \quad \quad \downarrow \\ & (((3 + ((2 * 4) * 5)) + 2) + 6) + ((4 * 3) * 7) \end{aligned}$$

Bemerkung: Das Hauptproblem bei der Erstellung der ersten Compiler war die Entdeckung zugkräftiger Analyse-Algorithmen für arithmetische Ausdrücke.

Formale Beschreibung arithmetischer Ausdrücke:

Beispiele: $3 + 4 * 5$
 $(8,43 + 7,921) * \sin(37,765)$
 $-(-38,472 / 173 + 345,88)$
 $\tan(34 * 33,3366)$

Elemente arithmetischer Ausdrücke:

Zahlen:	"Form bekannt"
Operatoren:	+, -, *, /
Funktionssymbole:	$\sqrt{\quad}$, exp, log, sin, cos, ...
Klammern:	(,)
Sonderzeichen:	","

Bemerkung: Die unterschiedliche Verwendung des Kommas, einmal als Trennsymbol bei Aufzählungen und einmal als Bestandteil arithmetischer Ausdrücke wird durch die Benutzung des Zeichens " angedeutet.

Regelsatz zur Formbeschreibung:

Abkürzungen:

A = Arithmetischer Ausdruck
B = Folge arithmetischer Ausdrücke
V = Vorzeichenloser arithmetischer Ausdruck
S = Summand
F = Faktor
Z = Zahl
Y = Funktionssymbol

Formmuster:

(R1) $A ::= + V \mid - V \mid V$
(R2) $V ::= V + S \mid V - S \mid S$
(R3) $S ::= S * F \mid S / F \mid F$
(R4) $F ::= Z \mid (A) \mid Y(B) \mid Y()$
(R5) $B ::= B, A \mid A$

Bemerkungen:

- (i) Das Symbol "|" dient als Metasymbol zur Trennung der einzelnen Anwendungsfälle einer Regel, es hat die Bedeutung eines umgangssprachlichen oder.
- (ii) Die einzelne Regel ist etwa so zu lesen: Der Begriff auf der linken Seite von ::= muß der Form xxx oder der Form yyy oder ... oder der Form zzz genügen. Als Beispiel möge die Regel R3 dienen: Ein Summand hat die Form "Summand gefolgt von '*'-Zeichen gefolgt von Faktor" oder die Form "Summand gefolgt von '/'-Zeichen gefolgt von Faktor" oder die Form eines Faktors.
- (iii) Die Formregeln berücksichtigen die normalen Bindungsregeln für arithmetische Operationen.
- (iv) Den Regelsatz kann man zur Erzeugung formgerechter arithmetischer Ausdrücke nutzen.
- (v) Den Regelsatz kann man zur Analyse arithmetischer Ausdrücke nutzen.
- (vi) Die Form einer Zahl oder eines Funktionssymbols wird hier als bekannt vorausgesetzt.
- (vii) Der obige Beschreibungsmechanismus entspricht in etwa dem für kontextfreie Grammatiken.

Nutzung des Regelsatzes zur Erzeugung wohlgeformter arithmetischer Ausdrücke:

Beispiel:

- (R1) $A ::= - V$
- (R2) $::= - V - S$
- (R2) $::= - S - S$
- (R3) $::= - F - S$
- (R4) $::= - Z - S$
- (R3) $::= - Z - S * F$
- (R3) $::= - Z - S * F * F$
- (R4) $::= - Z - S * (A) * F$
- (R1) $::= - Z - S * (+V) * F$
- (R3) $::= - Z - F * (+V) * F$
- (R4) $::= - Z - Z * (+V) * F$
- (R4) $::= - Z - Z * (+V) * Z$
- (R2) $::= - Z - Z * (+S) * Z$
- (R3) $::= - Z - Z * (+F) * Z$
- (R4) $::= - Z - Z * (+Z) * Z$

Die Z-Symbole werden durch Zahlen ersetzt.

$$\begin{aligned} & ::= - 400 - 3 * (+2) * 10 \\ & ::= - 340 \end{aligned}$$

Bemerkungen:

- (i) Der Anwendungsort einer Regel eines Regelsatzes wird nicht vorgeschrieben.
- (ii) Die Entstehungsgeschichte eines Ausdrucks bestimmt seinen Wert.

- (iii) Man wünscht, daß die Entstehungsgeschichte eines Ausdrucks, soweit es seine Wertbestimmung betrifft, eindeutig ist. Trifft dies für obiges Beispiel zu?
- (iv) Der vorgestellte Regelsatz entspricht nicht dem Alltagsgebrauch, so hat der Ausdruck $0 - 3 - 4$ einen anderen Wert als der Ausdruck $- 3 - 4$. Wie läßt sich dies korrigieren?

Benutzung des Regelsatzes zur Erkennung arithmetischer Ausdrücke:

Beispiel:

- $3 * 4 + \log(32)$
- $::= Z * Z + Y(Z)$
- (R4) $::= Z * Z + Y(F)$
- (R3) $::= Z * Z + Y(S)$
- (R2) $::= Z * Z + Y(V)$
- (R1) $::= Z * Z + Y(A)$
- (R5) $::= Z * Z + Y(B)$
- (R4) $::= Z * Z + F$
- (R4) $::= Z * F + F$
- (R4) $::= F * F + F$
- (R3) $::= S * F + F$
- (R3) $::= S + F$
- (R3) $::= S + S$
- (R2) $::= V + S$
- (R2) $::= V$
- (R1) $::= A$

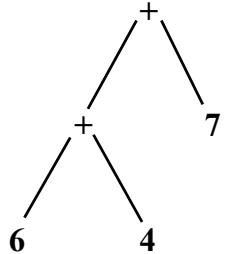
Bemerkung: Erzeugung und Erkennung sind inverse Operationen.

Darstellung arithmetischer Ausdrücke als Baum:

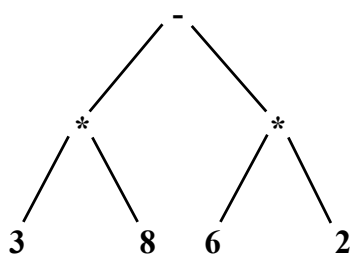
Beispiele:

$$3 + 5 \equiv$$


A tree diagram representing the expression 3 + 5. The root node is a plus sign (+). It has two children: the number 3 on the left and the number 5 on the right.

$$6 + 4 + 7 \equiv$$


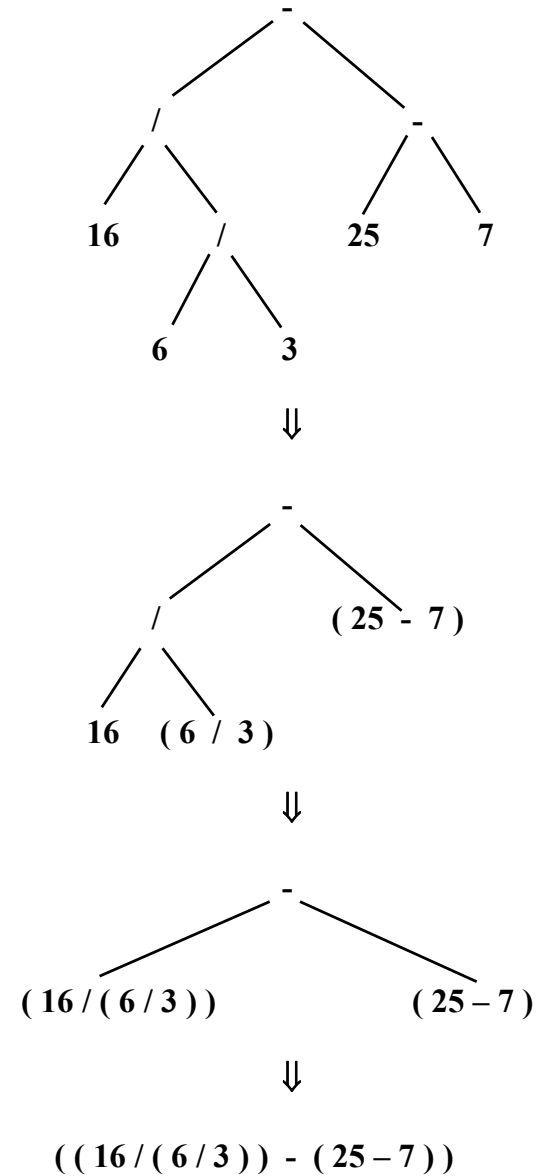
A tree diagram representing the expression 6 + 4 + 7. The root node is a plus sign (+). Its left child is another plus sign (+), and its right child is the number 7. The second plus sign has two children: the number 6 on the left and the number 4 on the right.

$$3 * 8 - 6 * 2 \equiv$$


A tree diagram representing the expression 3 * 8 - 6 * 2. The root node is a minus sign (-). It has two children: a multiplication sign (*) on the left and another multiplication sign (*) on the right. The left multiplication sign has children 3 and 8. The right multiplication sign has children 6 and 2.

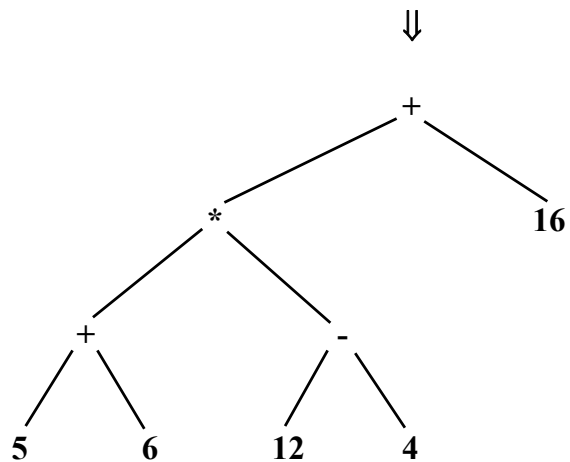
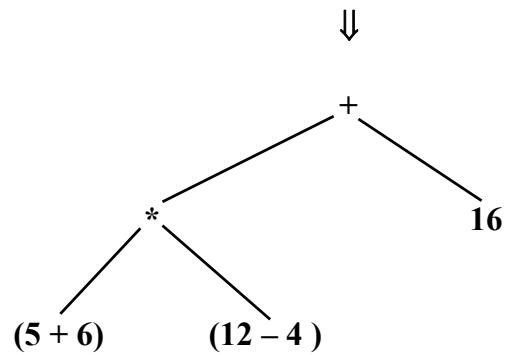
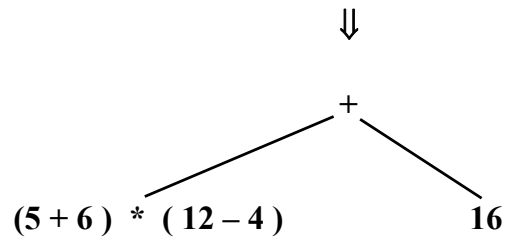
Umformung einer Baumdarstellung in eine lineare Darstellung:

Beispiel:



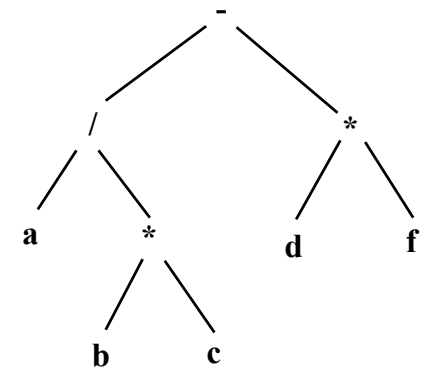
Umformung einer linearen Darstellung in eine Baumdarstellung:

Beispiel: $(5 + 6) * (12 - 4) + 16$

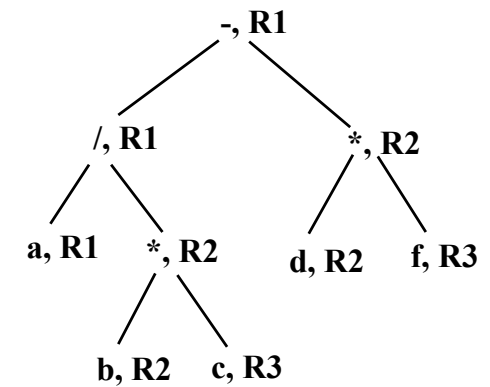


Ein Beispiel zur Übersetzung arithmetischer Ausdrücke:

Beispielausdruck:

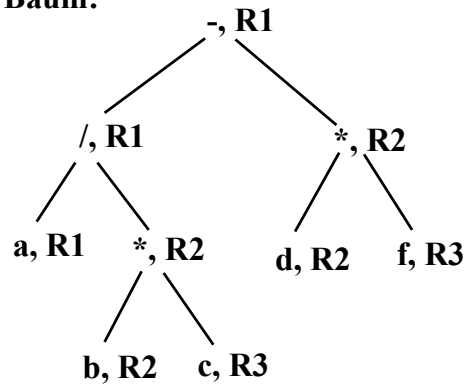


Registervergabe:



Bemerkung: Die Registervergabe erfolgt in einem Durchlauf durch den Baum von oben nach unten und von links nach rechts.

Geschmückter Baum:



Bemerkung: Den Maschinencode erzeugt in einem Durchlauf von oben nach unten und von links nach rechts.

Erzeugtes Maschinenprogramm für einen einfachen Befehlssatz:

```

mov R1, a
mov R2, b
mov R3, c
mul R2, R3
div R1, R2
mov R2, d
mov R3, f
mul R2, R3
sub R1, R2
    
```

Bemerkung: Das Resultat der Ausdrucksauswertung findet man in Register 1.

Logische Ausdrücke:

Konstante: true, false

Operationen: not, and, or

Wahrheitstafeln:

Negation:

not	
false	true
true	false

Konjunktion:

and	false	true
false	false	false
true	false	true

Disjunktion:

or	false	true
false	false	true
true	true	true

Beispiele:
 not not not true
 not (false or true)
 (true and false) and true
 not (27 < 12)
 3 < 6 = 7 > 5
 false or (false = true) or 5 > 7

Einige Rechengesetze für logische Ausdrücke:

Seien P, Q, R logische Variable, dann gelten die folgenden Identitäten:

Kommutativgesetze:

$$\begin{aligned} P \text{ or } Q &\equiv Q \text{ or } P \\ P \text{ and } Q &\equiv Q \text{ and } P \end{aligned}$$

Assoziativgesetze:

$$\begin{aligned} (P \text{ or } Q) \text{ or } R &\equiv P \text{ or } (Q \text{ or } R) \\ (P \text{ and } Q) \text{ and } R &\equiv P \text{ and } (Q \text{ and } R) \end{aligned}$$

Distributivgesetze:

$$\begin{aligned} (P \text{ and } Q) \text{ or } R &\equiv (P \text{ or } R) \text{ and } (Q \text{ or } R) \\ (P \text{ or } Q) \text{ and } R &\equiv (P \text{ and } R) \text{ or } (Q \text{ and } R) \end{aligned}$$

De Morgans Gesetze:

$$\begin{aligned} \text{not } (P \text{ or } Q) &\equiv \text{not } P \text{ and } \text{not } Q \\ \text{not } (P \text{ and } Q) &\equiv \text{not } P \text{ or } \text{not } Q \end{aligned}$$

Bemerkung: Es wurde angenommen, daß der Operator "not" stärker bindet als die Operatoren "and" und "or".

Beschreibung logischer Ausdrücke:

Abkürzungen:

LA = logischer Ausdruck
LS = logischer Summand
LF = logischer Faktor
LE = logischer Elementar Ausdruck
LK = logische Konstante
VG = Vergleichsoperator
AA = arithmetischer Ausdruck

Ein Regelsatz ("mehrdeutig"):

(S1) LA ::= LA or LS | LS
(S2) LS ::= LS and LF | LF
(S3) LF ::= not LF | LE
(S4) LE ::= LK | (LA)
 | LE = LE | LE ≠ LE
 | AA VG AA
(S5) VG ::= < | ≤ | > | ≥ | ≠ | =

Alle einstelligen und zweistelligen Booleschen Funktionen:

Sei $A = \{ 0, 1 \}$ eine zweielementige Wertemenge, 0 und 1 seien Synonyme für falsch und wahr, die Menge aller Abbildungen $f: A \rightarrow A$ läßt sich leicht aufzählen:

Argumente:	Wertebereiche:			
	f0	f1	f2	f3
0	0	0	1	1
1	0	1	0	1

mit f0 = Konstant 0, f1 = Identität, f2 = Negation, f3 = Konstant 1.

Es existieren genau 16 Funktionen von $A \times A$ nach A :

Arg.	g0	g1	g2	g3	g4	g5	g6	g7	g8	g9	g10	g11	g12	g13	g14	g15
00	0	0	0	0	1	0	0	1	0	1	1	0	1	1	1	1
01	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1
10	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1
11	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1

Viele dieser Funktionen tragen einprägsame Namen:

- g1 ≡ Konjunktion (AND-Funktion)
- g14 ≡ Sheffers Strich (NAND-Funktion)
- g11 ≡ Disjunktion (OR-Funktion)
- g4 ≡ Peircescher Pfeil (NOR-Funktion)
- g7 ≡ Äquivalenz
- g8 ≡ Antivalenz (XOR-Funktion)
- g13 ≡ Implikation

Zwei Beispielrechnungen:

$$\begin{aligned}
 a \text{ and } b &= \text{not not } (a \text{ and } b) \\
 &= \text{not } (\text{not } a \text{ or not } b) \\
 &= \text{not } a \text{ nor not } b \\
 &= (a \text{ nor } a) \text{ nor } (b \text{ nor } b)
 \end{aligned}$$

$$\begin{aligned}
 a \text{ or } b &= \text{not not } (a \text{ or } b) \\
 &= \text{not } (a \text{ nor } b) \\
 &= (a \text{ nor } b) \text{ nor } (a \text{ nor } b)
 \end{aligned}$$

Es läßt sich zeigen, daß sich alle 16 Funktionen allein aus der Peirce-Funktion herleiten lassen. Man nennt daher die Peirce-Funktion eine Basis der zweistelligen logischen Funktionen; es gibt weitere Basen.

(Aufgaben:

1. Zeigen Sie, daß g4 eine Basis ist.
2. Bestimmen Sie mindestens eine weitere Basis.)

Ein Beispiel für die Vereinfachung von logischen Ausdrücken:

$$\begin{aligned}
 &(P \text{ and } Q) \text{ or } (\text{not } P \text{ and } Q) \\
 &\equiv (P \text{ or not } P) \text{ and } Q \\
 &\equiv \text{true and } Q \\
 &\equiv Q
 \end{aligned}$$

Beschreibung einer dreistelligen logischen Funktion durch eine Wahrheitstafel:

x	true false true false	true false true false
y	true true false false	true true false false
z	true true true true	false false false false
f(x, y, z)	false true false false	true false true false

Darstellung von f mittels eines Ausdrucks:

$$f(x, y, z) = (\text{not } x \text{ and } y \text{ and } z) \\ \text{or } (x \text{ and } y \text{ and not } z) \\ \text{or } (x \text{ and not } y \text{ and not } z)$$

Überprüfung für Argumente true, true, true:

$$f(\text{true}, \text{true}, \text{true}) = (\text{not true and true and true}) \\ \text{or } (\text{true and true and not true}) \\ \text{or } (\text{true and not true and not true}) \\ = \text{false or false or false} \\ = \text{false}$$

Bemerkung: Für die folgenden Ausführungen werden die Synonyme 0 und 1 für false und true benutzt.

Wir betrachten nun n-stellige ($n \geq 2$) logische Funktionen:

$$\underbrace{A \times A \times \dots \times A}_{n\text{-mal}} \longrightarrow A$$

Es gilt das Lemma:

Für jede n-stellige logische Funktion f gilt:

$$f(a_1, \dots, a_{i-1}, a_i, \dots, a_n) \\ = (a_i \text{ and } f(a_1, \dots, a_{i-1}, 1, \dots, a_n)) \\ \text{or } (\text{not } a_i \text{ and } f(a_1, \dots, a_{i-1}, 0, \dots, a_n))$$

Beweis:

Man untersucht die beiden Fälle $a_i = 0$ und $a_i = 1$ getrennt.

Fall $a_i = 0$: trivial

Fall $a_i = 1$: trivial

Durch iterative Anwendung des Lemmas erhält man das **Boolesche Normalform-Theorem**:

Jede n-stellige logische Funktion läßt sich eindeutig als Disjunktion von Konjunktionen darstellen:

$$f(a_1, a_2, \dots, a_n) \\ = (a_1 \text{ and } a_2 \text{ and } \dots \text{ and } f(1, 1, \dots, 1)) \\ \text{or } (\text{not } a_1 \text{ and } a_2 \text{ and } a_3 \dots \text{ and } f(0, 1, \dots, 1)) \\ \dots \\ \text{or } (\text{not } a_1 \text{ and not } a_2 \text{ and } \dots \text{ and } f(0, 0, \dots, 0))$$

Diese Darstellung nennt man die **disjunktive Normalform**.

Boolesche Algebra:

Einen vollständigen, komplementären, distributiven Verband $V = (A, \wedge, \vee, \neg)$ nennt man eine Boolesche Algebra. Hierbei ist A die Trägermenge, \wedge und \vee sind zweistellige Operationen auf A , \neg ist eine einstellige Operation auf A . Seien K und G das kleinste bzw. das größte Element in A . Eine Boolesche Algebra erfüllt die folgenden zehn Gesetze.

- (B1) Assoziativität: $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
 $(x \vee y) \vee z = x \vee (y \vee z)$
- (B2) Kommutativität: $x \wedge y = y \wedge x$
 $x \vee y = y \vee x$
- (B3) Idempotenz: $x \wedge x = x, \quad x \vee x = x$
- (B4) Verschmelzung: $(x \wedge y) \vee x = x$
 $(x \vee y) \wedge x = x$
- (B5) Distributivität: $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$
 $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$
- (B6) Modularität: falls $x \leq z$, dann
 $x \wedge (y \vee z) = (x \wedge y) \vee z$
- (B7) neutrale Elemente: $x \wedge K = K, \quad x \vee K = x$
 $x \wedge G = x, \quad x \vee G = G$
- (B8) Komplement: $x \wedge (\neg x) = K, \quad x \vee (\neg x) = G$
- (B9) Involution: $\neg(\neg x) = x$
- (B10) De Morgan: $\neg(x \wedge y) = (\neg x) \vee (\neg y)$
 $\neg(x \vee y) = (\neg x) \wedge (\neg y)$

3.5 C-Operatoren in C++:

Bemerkung: Die Bindungsstärken von Operatoren werden nur indirekt durch die Grammatiken von C und C++ festgelegt, daher differieren sie in den Sprachen C und C++ leicht. Hier werden nur die Bindungsstärken innerhalb von C++ angegeben.

Präzedenz	Name	Assoziativität
1	Postfix-Operatoren: [] () . -> ++ --	links nach rechts
2	Unäre Operatoren: ++ -- (Präfix-Operatoren) ! ~ + - * & sizeof cast: (type name)	rechts nach links
3	* / %	links nach rechts
4	+ -	links nach rechts
5	<< >>	links nach rechts
6	< <= > >=	links nach rechts
7	== !=	links nach rechts
8	&	links nach rechts
9	^	links nach rechts
10		links nach rechts
11	&&	links nach rechts
12		links nach rechts
13	? :	rechts nach links
14	= += -= *= /= %= &= ^= = <<= >>=	rechts nach links
15	Komma-Operator: ,	links nach rechts

Beispiele zu Operatoren:

```
int a = 5;
int b = a++;
int c = 1;
```

Wert der Ausdrücke:

b	:	5
a	:	6
b = ~a	:	-7
b = !a	:	0
a << 3	:	48
a >> 2	:	1
a == b	:	0
a & b	:	4
a ^ b	:	3
a b	:	7
a && b	:	1
a b	:	1
a-- != b ? a : b	:	5
(a = 6, b = a, b++)	:	6
(b = 5, b = a += b, b == a)	:	1
(a = 6, b = 5, b = (a += b, b == a))	:	0
a = 1, b = 1		
a += b++ + ++c	:	4
a += b+++ ++c	:	9
a += (b++)+(++c)	:	16
a += b+++++c		!! Syntaxfehler !!
int m = 17, n = 3		
m % n	:	2
m % -n	:	2
-m % n	:	-2
-m % -n	:	-2

```
// Beispiel zum Zugriffsoperator ".".
// Eine Struktur mit zwei Funktionen
```

```
#include <iostream>
#include <string>

struct Rechteck {
    std::string name;
    double length;
    double width;

    double area () {
        return length*width;
    }
    double circum () {
        return 2*(length+width);
    }
};//Rechteck

int main () {
    Rechteck eins = {"Handtuch", 80.55, 35.33};
    Rechteck zwei;
    zwei.name = "test";
    zwei.length = 15;
    zwei.width = 40;
    std::cout << "Fläche von " << eins.name
        << " = " << eins.area ()
        << std::endl;
};//main

/*
Fläche von Handtuch = 2845.83
*/
```

```
// Beispiel zum Funktionsoperator "()".
```

```
#include <iostream>
```

```
struct Neuerwert {  
    int x;  
    int operator () () {      // Funktionsoperator  
        x += 3;  
        return x;  
    }  
};
```

```
int main () {  
    Neuerwert neu = {12};  
  
    std::cout << "Wert = " << neu () << std::endl;  
    std::cout << "Wert = " << neu () << std::endl;  
    if (neu () < neu ())  
        std::cout << "Wert wächst!" << std::endl;  
} //main
```

```
/*  
Wert = 15  
Wert = 18  
Wert wächst!  
*/
```

```
#include <iostream>  
#include <cmath>  
using namespace std;
```

```
typedef double (*f1p) (double);
```

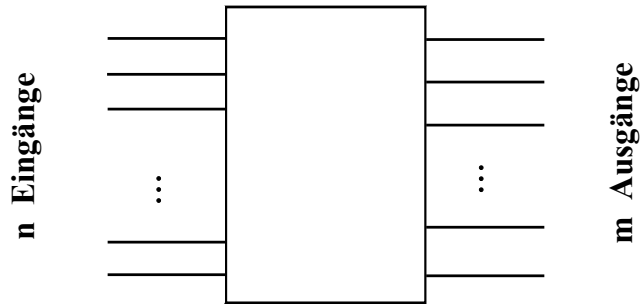
```
f1p fa (int x) {  
    switch (x) {  
        case 10: return sqrt;  
        break;  
        case 25: return exp;  
        break;  
        default: return log;  
    }  
} //fa
```

```
int main () {  
    cout << "sqrt: " << fa (10) (345.67) << endl;  
    cout << "log:  " << fa (105) (345.67) << endl;  
    cout << "exp:  " << fa (25) (345.67) << endl;  
} //main
```

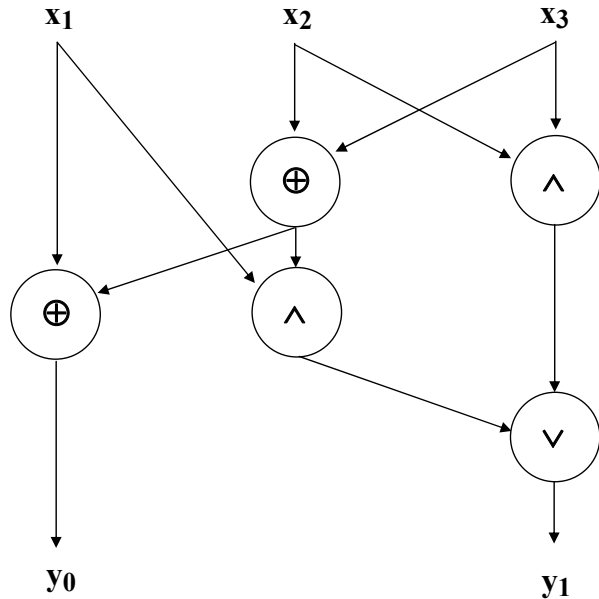
```
/*  
sqrt: 18.5922  
log:  5.84548  
exp:  1.32609e+150  
*/
```

Schaltglieder:

Darstellung eines Schaltglieds als schwarzer Kasten:



Beispiel eines Schaltnetzes:



Legende: \wedge = Und, \vee = Oder, \oplus = Antivalenz

Minimierung von Schaltfunktionen:

Im folgenden betrachten wir nur logische Funktionen, deren beschreibender Ausdruck ausschließlich die Operatoren Konjunktion, Disjunktion und Negation enthält.

Bemerkung: Im Bereich der Schaltfunktionen benutzt man folgende Notation:

$$\begin{aligned}x \wedge y &= x y, \\x \vee y &= x + y, \\ \text{not } x &= x'\end{aligned}$$

Man kann nun einem logischen Ausdruck L Kosten zuordnen und nach einem zu L gleichwertigen logischen Ausdruck K geringster Kosten suchen. Als Kostenmaß für einen logischen Ausdrucks bietet sich die Zahl der Operatoren an.

Beispiel: $F(a, b, c, d) = a(b + c)'d + c$
Kosten (F) = 5

In der Theorie der Minimierung logischer Funktionen ist das folgende Kostenmaß weit verbreitet:

Kosten (Variable) = Kosten (0) = Kosten (1) = 0,
da Leitungen kostenlos sind,
Kosten (a + b) =
Kosten (a b) = Kosten (a) + Kosten (b) + 1,
Kosten (a') = Kosten (a),
da die Negation kostenlos ist.

Beispiel: $\text{Kosten } (a(b + c)'d + c) =$
 $\text{Kosten } (a(b + c)'d) + \text{Kosten } (c) + 1 = 4$

Sei eine Schaltfunktion f gegeben durch:

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(i) f in disjunktiver Normalform:

$$f(a, b, c) = a' b' c + a' b c + a b' c + a b c' + a b c$$

Kosten = 14

(ii) f in konjunktiver Normalform:

$$f(a, b, c) = (a' b' c' + a' b c' + a b' c')'$$

$$= (a + b + c) (a + b' + c) (a' + b + c)$$

Kosten = 8

(iii) Minimierung von f mittels Karnaugh-Tafel:

		b c			
		00	01	11	10
a	0		1	1	
	1		1	1	1

$$f = c + a b$$

Kosten = 2

(iv) Minimierung von f unter Nutzung der Idempotenz-, Distributiv- und Komplementgesetze:

$$f(a, b, c) = a' b' c + a' b c + a b' c + a b c' + a b c$$

$$= a' b' c + a' b c + a b' c + a b c' + a b c$$

$$+ a b c$$

$$= a' c (b' + b) + a c (b' + b) + a b (c' + c)$$

$$= a' c + a c + a b$$

$$= c + a b$$

Kosten = 2

Ein weiteres Beispiel:

Die Schaltfunktion f sei gegeben durch:

$$f = a' b' c' d' + a' b' c' d + a' b' c d' + a' b' c d$$

$$+ a' b c d + a b' c' d' + a b' c' d + a b' c d'$$

$$+ a b c d' + a b c d$$

Mögliche Vereinfachungen sind:

(i) $f = a' c d + a b c + b' c + b' d$
Kosten = 9

(ii) $f = a c d' + b c d + a' b' + b' c$
Kosten = 9

Bemerkung: Die Systematisierung des unter (iv) demonstrierten Verfahrens zur Minimierung von Schaltfunktionen nennt man den Quine-McCluskey Algorithmus.

Beispiel zum Verfahren von Karnaugh :

Das nach Karnaugh benannte Minimierungsverfahren wurde vorgestellt in:

Maurice Karnaugh: "The Map Method for Synthesis of Combinational Logic Circuits", Transactions of the American Institute of Electrical Engineers, Vol. 72, Pt. I, pp. 593-599, 1953.

Funktionstafel zur Addition zweier zweistelliger Dualzahlen: $c = a + b$:

a [1]	a[0]	b[1]	b[0]	c[2]	c[1]	c[0]
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	0	1
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	1	0	0
1	0	0	0	0	1	0
1	0	0	1	0	1	1
1	0	1	0	1	0	0
1	0	1	1	1	0	1
1	1	0	0	0	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

Lesen von Karnaugh-Tabellen:

		c d				
		0 0	0 1	1 1	1 0	
a b	0 0					a' b
	0 1	1	1	1	1	
	1 1					
	1 1					

		c d				
		0 0	0 1	1 1	1 0	
a b	0 0		1	1		d
	0 1		1	1		
	1 1		1	1		
	1 0		1	1		

		c d				
		0 0	0 1	1 1	1 0	
a b	0 0					b d'
	0 1	1			1	
	1 1	1			1	
	1 0					

Karnaugh-Tafel für c[2]:

a[1]	a[0]	b[1]	0	0	1	1
		b[0]	0	1	1	0
0	0					
0	1				1	
1	1			1	1	1
1	0				1	1

Man liest ab:

$$c[2] = a[1] b[1] + a[1] a[0] b[0] + a[0] b[0] b[1]$$

Karnaugh-Tafel für c[1]:

a[1]	a[0]	b[1]	0	0	1	1
		b[0]	0	1	1	0
0	0				1	1
0	1			1		1
1	1		1		1	
1	0		1	1		

Man liest ab:

$$c[1] = a[1]' a[0]' b[1] + b[1] b[0]' a[1]' + b[0]' b[1]' a[1] + a[1] a[0]' b[1]' + a[0] a[1]' b[0] b[1]' + a[0] a[1] b[0] b[1]$$

Karnaugh-Tafel für c[0]:

a[1]	a[0]	b[1]	0	0	1	1
		b[0]	0	1	1	0
0	0				1	1
0	1			1		1
1	1			1		1
1	0				1	1

Man liest ab:

$$c[0] = a[0] b[0]' + a[0]' b[0]$$

Bemerkungen:

- (i) Es erfordert einige Übung, um den Ausdruck für c[0] aus der Karnaugh-Tafel abzulesen.
- (ii) Bei hohen Variablenzahlen versagt die optische Inspektion.
- (iii) Das Minimierungsproblem für Schaltfunktionen ist NP-vollständig.
- (iv) Das Verfahren von Quine-McCluskey erfordert keine visuelle Geschicklichkeit.

3.6 Begriffe aus der Syntaxtheorie

Ein Beispiel aus Haider: Deutsche Syntax – generativ

Vorgestern sagte man ihr, habe jemand zu Protokoll gegeben, sei Herr Lothringer abgereist.

Vorgestern sagte man ihr, habe jemand zu Protokoll gegeben, Herr Lothringer sei abgereist.

Vorgestern sagte man ihr, jemand habe zu Protokoll gegeben, Herr Lothringer sei abgereist.

Bemerkung: In den drei vorhergehenden Sätzen wird das Adverb vorgestern an jeweils andere Satzteile gebunden. Die Bindungen eines Adverbs an den entsprechenden Satzteil versucht man grammatikalisch zu erfassen.

Wenden wir uns nun einem einfacheren Beispiel zu.

Satz: Heute scheint die Sonne.

Dieser Satz besteht aus den Satzteilen:

Umstandsbestimmung der Zeit, Prädikat, Subjekt; das Subjekt seinerseits besteht aus Artikel und Substantiv.

Bemerkung: Die Partikel Subjekt, Objekt, Prädikat, Nebensatz, Korrelat, Infinitivkomplement, u. s. w. dienen der Analyse natürlich sprachlicher Sätze und werden daher Metasymbole oder Nichtterminale genannt.

Bezeichnungen und Definitionen:

Definition: Eine endliche, nichtleere Menge von Zeichen nennt man ein Alphabet.

Beispiel: Zur Beschreibung der Sprache C++ verwendet man u. a. die Zeichen: if, for, while, do, +, -, *, /, ++, +=, throw.

Definition: Eine Folge von Zeichen aus einem Alphabet A nennt man ein Wort über dem Alphabet A. Die Zahl der Zeichen in einem Wort w nennt man die Wortlänge von w (kurz: Länge (w)).

Bezeichnungen: Zu jedem Alphabet fordert man die eindeutige Existenz eines Wortes ϵ mit folgenden Eigenschaften:

1. Für jedes Wort w über A gilt:

$$\epsilon w = w \epsilon = w;$$

2. Länge (ϵ) = 0.

Das Wort ϵ nennt man das leere Wort.

Bezeichnung: Die Menge aller endlichen Wörter über einem Alphabet A einschließlich des leeren Wortes bezeichnet man mit A^* .

Definition: Eine Sprache S über dem Alphabet A ist eine Teilmenge von A^* .

Bemerkung: Auf dieser Betrachtungsebene sind Sprachen nur Sammlungen von Zeichenketten; die Zeichenketten einer Sprache verkörpern nur selten eine Bedeutung.

Definition: Ein Quadrupel $G = (T, N, S, P)$ nennt man eine Grammatik, falls gilt:

1. T und N sind Alphabete,
2. $T \cap N = \emptyset$,
3. $S \in N$,
4. P ist eine nichtleere, endliche Menge von Regeln der Form $u \rightarrow w$ mit $u \in (N \cup T)^+$ und $w \in (N \cup T)^*$.

Bezeichnungen: Man nennt

- T die Menge der Terminalzeichen (kurz: Terminale),
- N die Menge der Nichtterminalzeichen (auch: Nichtterminale, Nonterminale Variable, Metasymbole),
- S das Startsymbol (auch: Satzsymbol),
- P die Menge der Produktionen (auch: Regelmenge),
- $V = N \cup T$ das Vokabular der Grammatik.

Bezeichnungen: In einer Regel $u \rightarrow w$ nennt man

- w die rechte Seite (auch: Argument),
- u die linke Seite (auch: Ziel).

Beispiel: $G1 = (T1, N1, Z, P1)$
mit $T1 = \{+, -, 0, 1\}$,
 $N1 = \{Z, S, K, E\}$,
 $P1 = \{Z \rightarrow SK, S \rightarrow +, S \rightarrow -, K \rightarrow E, K \rightarrow KE, E \rightarrow 0, E \rightarrow 1\}$

Beispiel: $G2 = (T2, N2, A, P2)$
mit $T2 = \{a, b, c\}$,
 $N2 = \{A, B, C\}$,
 $P2 = \{A \rightarrow abc, A \rightarrow aBbc, Bb \rightarrow bB, Bc \rightarrow Cbcc, bC \rightarrow Cb, aC \rightarrow aaB, aC \rightarrow aa\}$.

Der Anfang einer Ersetzungskette ist:

$A \rightarrow aBbc$
 $\rightarrow abBc$
 $\rightarrow abCbcc$
 $\rightarrow aCbbcc$
 $\rightarrow aaBbbcc$
 $\rightarrow aabBbcc$
 $\rightarrow aabbBcc$
 $\rightarrow aabbCbcc$
 $\rightarrow aabCbbccc$
 $\rightarrow aaCbbccc$
 $\rightarrow aaaBbbbccc$
 $\rightarrow aaabBbbccc$
 $\rightarrow aaabbBbcc$
 $\rightarrow aaabbbBccc$
 \vdots

Definition: Seien $G = (T, N, S, P)$ eine Grammatik, $\alpha \in (T \cup N)^*$, $\beta \in (T \cup N)^*$; die Zeichenkette β nennt man direkt ableitbar aus der Zeichenkette α gemäß der Grammatik G , wenn gilt:

1. Es existieren Zeichenketten γ, δ, u, w mit $\alpha = \gamma u \delta$ und $\beta = \gamma w \delta$;
2. $u \rightarrow w$ ist eine Produktion aus P .

Bezeichnung: Bei gegebener Grammatik G schreibt man für β ist direkt ableitbar aus α : $\alpha \xrightarrow{G} \beta$, ist G aus dem Zusammenhang eindeutig bestimmt, schreibt man: $\alpha \Rightarrow \beta$.

Beispiel: $aaBaaCa \xrightarrow{G_2} aaBaaaa$

Definition: Seien $G = (T, N, S, P)$ eine Grammatik, $\alpha \in (T \cup N)^*$, $\beta \in (T \cup N)^*$; man nennt β ableitbar von α gemäß G , wenn gilt:
 $\alpha = \beta$
 oder

es existieren $\gamma_0, \gamma_1, \dots, \gamma_n \in (T \cup N)^*$ mit $n \geq 1$, $\gamma_0 = \alpha$, $\gamma_n = \beta$, $\gamma_i \xrightarrow{G} \gamma_{i+1}$ für $i = 0, 1, \dots, n-1$.

Bezeichnung: Für β ist aus α gemäß G ableitbar benutzt man die Notation $\alpha \xrightarrow{G}^* \beta$ (kurz: $\alpha \Rightarrow^* \beta$).

Beispiel: Eine Grammatik liest man als Erzeugendensystem für Terminalketten. Für die Grammatik G_1 läßt sich folgende Ersetzungskette bilden:

$Z \Rightarrow SK$
 $\Rightarrow SKE$
 $\Rightarrow SKEE$
 $\Rightarrow SKEEE$
 $\Rightarrow SEEEE$
 $\Rightarrow -EEEE$
 $\Rightarrow -1EEE$
 $\Rightarrow -10EE$
 $\Rightarrow -100E$
 $\Rightarrow -1001$

Zusammenfassend: $Z \xrightarrow{*} -1001$

Eine andere Ersetzungskette ist:

$Z \Rightarrow SK$
 $\Rightarrow +K$
 $\Rightarrow +KE$
 $\Rightarrow +K0$
 $\Rightarrow +KE0$
 $\Rightarrow +EE0$
 $\Rightarrow +0E0$
 $\Rightarrow +010$

Bemerkung: Eine Grammatik kann man auch als Erkennungssystem für Terminalketten lesen.

Definition: Sei $G = (T, N, S, P)$, $\alpha \in (T \cup N)^*$; α heißt eine Satzform von G , falls $S \xRightarrow{*} \alpha$ gilt; α heißt ein Satz von G , falls $S \xRightarrow{*} \alpha$ und $\alpha \in T^*$ gilt.

Definition: Sei $G = (T, N, S, P)$;
mit $L(G) = \{\alpha : \alpha \text{ Satz von } G\}$
bezeichnet man die von G erzeugte Sprache.

Beispiel: SEE10E0E ist eine Satzform der Grammatik G_1 ; -0010 ist ein Satz von G_1 ;
 $L(G_1) = \{+, -\} \{0, 1\}^+$;
 $L(G_2) = \{a^n b^n c^n : n = 1, 2, 3, \dots\}$.

Bemerkung: Jede wohldefinierbare Menge von Zeichenketten betrachten wir als eine Sprache. Wir sind interessiert an denjenigen Sprachen, die wir mittels eines endlichen Regelsystems beschreiben können. Das Regelsystem soll später so erweitert werden, daß den einzelnen Sätzen einer Sprache auf übersichtliche Weise eine Bedeutung zuordbar ist. Im Compilerbau nutzt man zur Beschreibung der syntaktischen Struktur von Sprachen aus praktischen Gründen fast nur kontextfreie Grammatiken.

Definition: Ein Grammatik $G = (T, N, S, P)$ heißt kontextfrei, falls für ihre Regeln $u \rightarrow w \in P$ gilt: $u \in N$ und $w \in (N \cup T)^*$.

3.7 EBNF nach ISO/IEC 14977

Bemerkung: ENBF ist eine Abkürzung und bedeutet Extended Backus-Naur Form.

Bemerkung: Die Wörter einer Sprache werden aus einzelnen Zeichen zusammengestellt; die Sätze wiederum werden aus Wörtern und Satzzeichen gebildet. Sätze einer Sprache werden nur aneinandergereiht. Bei Programmiersprachen kann man klar unterscheiden zwischen den Wörtern in den Sprachsätzen und den beschreibenden Begriffen. Die beschreibenden Begriffe nennt man Nichtterminale oder Metasymbole und die Grundelemente der Sprache Terminale.

Bemerkung: Die möglichen Sätze einer Programmiersprache beschreibt man durch Regeln der Form Nichtterminal wird definiert durch Folge von Terminalen und Nichtterminalen, unter den Nichtterminalen ist eines ausgezeichnet, das Satzsymbol; es sollte nur auf der linken Seite von Regeln auftreten.

Ziel von ISO-EBNF:

Definition einer einfachen, leicht verständlichen, einheitlichen und universellen Notation, in der sich die Grammatiken jeder Programmiersprache beschreiben lassen.

Charakteristika von ISO-EBNF:

- (i) Terminale werden in einfache oder doppelte Anführungszeichen eingeschlossen, Nichtterminale werden normal notiert.
- (ii) Nichtterminale können aus mehreren Wörtern bestehen.
- (iii) In ISO-EBNF werden nur weitverbreitete Zeichen aus dem ASCII-Zeichensatz verwendet.
- (iv) Der Formalismus von ISO-EBNF kann genutzt werden, um den Formalismus von ISO-EBNF vollständig zu beschreiben.
- (v) ISO-EBNF beschreibt Grammatiken mittels einer linearen Zeichenkette.

In ISO-EBNF eingesetzte Zeichen mit Sonderbedeutung. In der Tabelle sind die Zeichen entsprechend ihrer Bindungsstärke aufgeführt, von stark nach schwach.

Zeichen	Bedeutung
*	Wiederholungszeichen
-	Ausnahmezeichen
,	Konkatenationszeichen
	Trennzeichen für Alternativen
=	Definitionszeichen
;	Endezeichen einer Regel

Eine Regel hat die Form:

Nichtterminal = Folge von Terminalen und Nichtterminalen ;

Auf der rechten Seite einer Regel können die folgenden Klammerpaare auftreten, sie überschreiben die Präzedenzen der Sonderzeichen.

[...]	Optionales Auftreten
{ ... }	Beliebige Wiederholung
(...)	Gruppierung
' ... '	Anführungszeichen 1
" ... "	Anführungszeichen 2
(* ... *)	Kommentar
? ... ?	Spezielle Sequenz

Die spezielle Sequenz dient der Erweiterung der EBNF.

Eine Sonderrolle nimmt die leere Symbolfolge ein, sie wird nicht durch ein Sondersymbol dargestellt, so bedeutet

$ttt = \{ "A" \} -, "B";$

die Gesamtheit der Folgen

AB AAB AAAB AAAAB u. s. w.,

denn B ist wegen des Minuszeichens nicht zugelassen

Einige Beispiele aus ISO/IEC 14977:

aa = "A";
bb = 3 * aa, "B";
cc = 3 * [aa], "C";
dd = {aa}, "D";
ee = aa, {aa}, "E";
ff = 3 * aa, 3 * [aa], "F";

Durch obige Regeln definierte Terminalketten:

aa: A
bb: AAAB
cc: C AC AAC AAAC
dd: D AD AAD AAAD AAAAD u. s. w.
ee AE AAE AAAE AAAAE u. s. w.
ff AAAF AAAAF AAAAAF AAAAAAF

Beispiel einer Erweiterung:

form feed = ? ISO 6429 character Form Feed ?

Alternativzeichen:

/	statt	
!	statt	
/)	statt]
:)	statt	}
(/	statt	[
(:	statt	{
.	statt	;

Unerlaubte Zeichenfolgen:

(*)
(:)
(/)

3.8 Ausdrücke in C++

Die Teilgrammatik für die Ausdrücke wurde dem Working Draft, Standard Programming Language C++ vom 09. 11. 2009 entnommen.

primary-expression:

literal
this
(expression)
id-expression
lambda-expression

In EBNF:

primary expression = literal

| "this"
| "(" , expression , ")"
| id expression
| lambda expression ;

(* lambda expression ist eine Erweiterung des C++ - Standards von 1998.

*)

id-expression:

unqualified-id
qualified-id

In EBNF:

id expression = unqualified id | qualified id ;

unqualified-id:

identifier
operator-function-id
conversion-function-id
literal-operator-id
~ class-name
template-id

In EBNF:

unqualified id = identifier | operator function id
| **conversion function id**
| **literal operator id**
| **"~" , class name | template id ;**

qualified-id:

::_{opt} nested-name-specifier template_{opt} unqualified-id
:: identifier
:: operator function id
:: literal-operator-id
:: template-id

In EBNF:

qualified id = ["::"] , nested name specifier ,
[template] , unqualified id
| **"::" , identifier**
| **"::" , operator function id**
| **"::" , literal operator id**
| **"::" , template id ;**

nested-name-specifier:

type-name ::
namespace-name ::
nested-name-specifier identifier ::
nested-name-specifier template_{opt} simple-template-id ::

In EBNF:

nested name specifier = type name , "::"
| **namespace name , "::"**
| **nested name specifier , identifier , "::"**
| **nested name specifier , [template] ,**
simple template id , "::" ;

lambda-expression:

lambda-introducer lambda-declarator_{opt} compound-
statement

In EBNF:

lambda expression = lambda introducer ,
[lambda declarator] , compound statement ;

lambda-introducer:

[lambda-capture_{opt}]

In EBNF:

lambda introducer = "[" , [lambda capture] , "]" ;

lambda-capture:
capture-default
capture-list
capture-default , capture-list

In EBNF:
lambda capture = capture default
 | capture list
 | capture default , "," , capture list ;

capture-default:
&
=

In EBNF:
capture default = "&" | "=" ;

capture-list:
capture
capture-list , capture

In EBNF:
capture list = capture
 | capture list , "," , capture ;

capture:
identifier
& identifier
this

In EBNF:
capture = ["&"] identifier | "this" ;

lambda-declarator:
(parameter-declaration-clause) attribute-specifier_{opt}
mutable_{opt} exception-specification_{opt}
trailing-return-type_{opt}

In EBNF:
lambda declarator =
 ("parameter declaration clause")
 [attribute specifier] [mutable]
 [exception specification] [trailing return type] ;

postfix-expression:
primary-expression
postfix-expression [expression]
postfix-expression [braced-init-list]
postfix-expression (expression-list_{opt})
simple-type-specifier (expression-list_{opt})
typename-specifier (expression-list_{opt})
simple-type-specifier braced-init-list
typename-specifier braced-init-list
postfix-expression . template_{opt} id-expression
postfix-expression -> template_{opt} id-expression
postfix-expression . pseudo-destructor-name
postfix-expression -> pseudo-destructor-name
postfix-expression ++
postfix-expression --
dynamic_cast <type-id> (expression)
static_cast <type-id> (expression)
reinterpret_cast <type-id> (expression)
const_cast < type-id > (expression)
typeid (expression)
typeid (type-id)

In EBNF:

postfix expression =

| primary expression
| postfix expression , "[" , expression , "]"
| postfix expression , "[" , braced init list , "]"
| postfix expression , "(" , [expression list] , ")"
| simple type specifier , "(" [expression list] , ")"
| typename specifier "(" , [expression list] , ")"
| simple type specifier , braced init list
| typename specifier , braced init list
| postfix expression , "." , [template] , id expression
| postfix expression , "->" , [template] , id expression
| postfix expression , "." , pseudo destructor name
| postfix expression , "->" , pseudo-destructor-name
| postfix expression , ("++" | "--")
| "dynamic_cast" , "<" , type id , ">" , "(" , expression , ")"
| "static_cast" , "<" , type id , ">" , "(" , expression , ")"
| "reinterpret_cast" , "<" , type id , ">" ,
 , "(" , expression , ")"
| "const_cast" , "<" , "type id" , ">" , "(" , expression , ")"
| "typeid" , "(" , (expression | type id) , ")" ;

expression-list:

initializer-list

In EBNF:

expression list = initializer list ;

pseudo-destructor-name:

::opt nested-name-specifier_{opt} type-name ::
 ~ type-name
::opt nested-name-specifier template
 simple-template-id :: ~ type-name
::opt nested-name-specifier_{opt} ~ type-name

In EBNF:

pseudo destructor name =

["::"] , [nested name specifier] , type name , "::" ,
 "~" , "type name"
| ["::"] , nested name specifier , template
 simple template id , "::" , "~" , type name
| ["::"] , [nested name specifier] , "~" , type name ;

unary-expression:

postfix-expression
++ cast-expression
— cast-expression
unary-operator cast-expression
sizeof unary-expression
sizeof (type-id)
sizeof . . . (identifier)
alignof (type-id)
new-expression
delete-expression

In EBNF:

unary expression =
 postfix expression
 | ("++" | "--") , **cast expression**
 | **unary operator** , **cast expression**
 | "sizeof" , **unary expression**
 | "sizeof" , "(" , **type id** , ")"
 | "sizeof" , "... " , "(" , **identifier** , ")"
 | "alignof" , "(" , **type id** , ")"
 | **new expression**
 | **delete expression** ;

unary-operator: one of
 * & + - ! ~

In EBNF:

unary-operator = "*" | "&" | "+" | "-" | "!" | "~" ;

new-expression:

 ::opt **new** **new-placement**_{opt} **new-type-id**
 new-initializer_{opt}
 ::opt **new** **new-placement**_{opt} (**type-id**)
 new-initializer_{opt}

In EBNF:

new expression =
 ["::"] , "new" , [new placement] , new type id
 [new initializer]
 | ["::"] , "new" , [new placement] , "(" , type id , ")" ,
 [new initializer] ;

new-placement:
 (expression-list)

In EBNF:

new placement = "(" , expression list , ")" ;

new-type-id:

type-specifier-seq **new-declarator**_{opt}

In EBNF:

new type id = **type specifier seq** , [new declarator] ;

new-declarator:

ptr-operator **new-declarator**_{opt}
 noPtr-new-declarator

In EBNF:

new declarator =
 ptr operator , [new declarator]
 | **noPtr new declarator** ;

noPtr-new-declarator:

 [expression]
 noPtr-new-declarator [constant-expression]

In EBNF:

noPtr new declarator:
 "[" , expression , "]"
 noPtr new declarator , "[" , constant expression , "]" ;

new-initializer:

**(expression-list_{opt})
braced-init-list**

In EBNF:

**new-initializer = "(" , expression list , ")"
| braced init list ;**

delete-expression:

**::_{opt} delete cast-expression
::_{opt} delete [] cast-expression**

In EBNF:

**delete expression =
["::_{opt}] , "delete" , cast expression
| ["::_{opt}] , "delete" , "[]" , cast expression ;**

cast-expression:

**unary-expression
(type-id) cast-expression**

In EBNF:

**cast expression =
unary expression
| "(" , type id , ")" , cast expression ;**

pm-expression:

**cast-expression
pm-expression .* cast-expression
pm-expression ->* cast-expression**

In EBNF:

**pm expression = cast expression
| pm expression , ".*" , cast expression
| pm expression , "->*" , cast-expression ;**

multiplicative-expression:

**pm-expression
multiplicative-expression * pm-expression
multiplicative-expression / pm-expression
multiplicative-expression % pm-expression**

In EBNF:

**multiplicative expression = pm expression
| multiplicative expression , "*" , pm expression
| multiplicative-expression , "/" , pm expression
| multiplicative-expression , "%" , pm expression ;**

additive-expression:

**multiplicative-expression
additive-expression + multiplicative-expression
additive-expression - multiplicative-expression**

In EBNF:

**additive expression = multiplicative expression
| additive expression , "+" , multiplicative expression
| additive-expression , "-" , multiplicative expression ;**

shift-expression:

additive-expression

shift-expression << additive-expression

shift-expression >> additive-expression

In EBNF:

shift expression = additive-expression

| **shift expression , "<<" , additive expression**

| **shift expression , ">>" , additive expression ;**

relational-expression:

shift-expression

relational-expression < shift-expression

relational-expression > shift-expression

relational-expression <= shift-expression

relational-expression >= shift-expression

In EBNF:

relational expression = shift expression

| **relational expression , "<" , shift expression**

| **relational expression , ">" , shift expression**

| **relational expression , "<=" , shift expression**

| **relational expression , ">=" , shift expression ;**

equality-expression:

relational-expression

equality-expression == relational-expression

equality-expression != relational-expression

In EBNF:

equality expression = relational expression

| **equality-expression , "==" , relational expression**

| **equality-expression , "!=" , relational expression ;**

and-expression:

equality-expression

and-expression & equality-expression

In EBNF:

and expression =

[and expression , "&"] , equality expression ;

exclusive-or-expression:

and-expression

exclusive-or-expression ^ and-expression

In EBNF:

exclusive or expression =

[exclusive or expression , "^"] , and expression ;

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | exclusive-or-expression

In EBNF:
inclusive or expression =
[inclusive or expression "|"], exclusive or expression ;

logical-and-expression:
inclusive-or-expression
logical-and-expression && inclusive-or-expression

In EBNF:
logical and expression = inclusive or expression
| logical and expression , "&&" ,
inclusive or expression ;

logical-or-expression:
logical-and-expression
logical-or-expression || logical-and-expression

In EBNF:
logical or expression =
[logical or expression , "||"] , logical and expression ;

conditional-expression:
logical-or-expression
logical-or-expression ? expression :
assignment-expression

In EBNF:
conditional-expression =
logical-or-expression
| logical or expression , "?" , expression , ":" ,
assignment expression ;

assignment-expression:
conditional-expression
logical-or-expression assignment-operator
initializer-clause
throw-expression

In EBNF:
assignment expression =
conditional expression
| logical or expression , assignment operator ,
initializer clause
| throw expression ;

assignment-operator: one of
= *= /= %= += -= >>= <<= &= ^= |=

In EBNF:
assignment operator = "=" | "*=" | "/=" | "%=" |
| "+=" | "-=" | ">>=" | "<<=" |
| "&=" | "^=" | "|=" ;

expression:

assignment-expression

expression , assignment-expression

In EBNF:

expression = [expression , ","] , assignment expression ;

constant-expression:

conditional-expression

In EBNF:

constant expression = conditional expression