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 Minimierung von Schaltfunktionen

3.5 Algorithmus von Quine-McCluskey

3.6 Programmbeweise

Zum Algorithmusbegriff:

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

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 Algorithmentexte 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:

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 Enderegel angewandt wurde, gekennzeichnet durch einen Endepunkt "▪", oder keine Regel mehr anwendbar ist.

Beispiel eines Markov-Algorithmus:
Dopplung einer Bitkette:

Regeln:

00β	$\rightarrow$	$0\beta0$	/* Vertauschung */
01β	$\rightarrow$	$1\beta0$	
10β	$\rightarrow$	$0\beta1$	
11β	$\rightarrow$	$1\beta1$	
$\alpha0$	$\rightarrow$	$0\beta0\alpha$	/* Dopplung */
$\alpha1$	$\rightarrow$	$1\beta1\alpha$	
β	$\rightarrow$	γ	/* Beenden der */
γ	$\rightarrow$		/* Dopplung */
α	$\rightarrow$	▪	
	$\rightarrow$	α	

Anwendung: Dopplung von 101:

101
$\alpha101$
$1\beta1\alpha01$
$1\beta10\beta0\alpha1$
$1\beta0\beta10\alpha1$
$1\beta0\beta101\beta1\alpha$
$1\beta0\beta11\beta01\alpha$
$1\beta0\beta1\beta101\alpha$
$1\gamma0\beta1\beta101\alpha$
$1\gamma0\gamma1\beta101\alpha$
$1\gamma0\gamma1\gamma101\alpha$
$10\gamma1\gamma101\alpha$
$101\gamma101\alpha$
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.

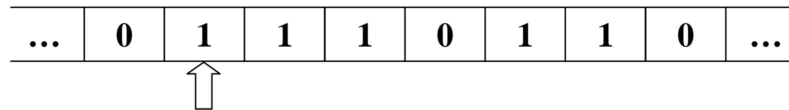
Bild einer Turing-Maschine:

Beispiel:

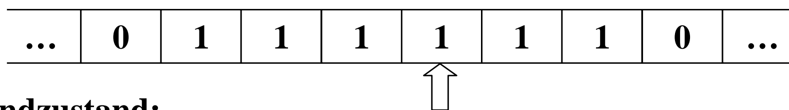
Programm für eine Turing-Maschine zur Addition zweier Zahlen in Einer-Darstellung:

	0	1
Z1	**1, Z2**	**R, Z1**
Z2	**R, Z3**	**L, Z2**
Z3	**R, Z4**	**0, Z3**
Z4	**R, Z4**	

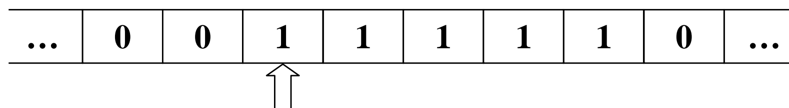
Anfangszustand:



Zwischenzustand:



Endzustand:



Eigenschaften eines Algorithmus:

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

Bemerkungen:

(i) Es existieren Probleme, zu deren Lösung sich kein Algorithmus finden läßt, ein Beispiel hierfür ist das Halteproblem.

(ii) Es existieren Probleme, zu deren Lösung man nur nicht effiziente Algorithmen kennt, ein Beispiel hierfür ist die Klasse der NP-vollständigen Probleme.

(iii) Man kann unsinnige Algorithmen formulieren, Beispiele hierfür sind Lösungsvorschläge für nicht wohlverstandene Probleme.

(iv) Zur Beschreibung von Algorithmen sucht man Formalismen, die es gestatten, Problemlösungen kompakt und eindeutig zu beschreiben, ein Beispiel hierfür sind die Formelsammlungen technischer Fächer.

Beispiele für Formeln:

8 * 4

true and false

"Wal" + "dlauf"

((7 * 3) < (4 * 5)) 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)

$$\equiv ((VAA + VAA))$$
$$\equiv (((VAA * VAA) + (VAA - VAA)))$$
$$\equiv (((VAA * (VAA - VAA)) + (VAA - VAA)))$$
$$\equiv (((3 \quad * (0 \quad - 7 \quad)) + (6 \quad - 2 \quad)))$$
$$\equiv (((3 * (0 - 7)) + (6 - 2)))$$

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:

Ausdruck: $3 + 2 * 4 * 5 + 2 + 6 + 4 * 3 * 7$

$$\Downarrow$$
$$3 + (2 * 4 * 5) + 2 + 6 + (4 * 3 * 7)$$
$$\Downarrow$$
$$3 + ((2 * 4) * 5) + 2 + 6 + ((4 * 3) * 7)$$
$$\Downarrow$$
$$((((3 + ((2 * 4) * 5)) + 2) + 6) + ((4 * 3) * 7))$$

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:	√, 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 │ – V │ V**
(R2)	**V ::= V + S │ V – S │ S**
(R3)	**S ::= S ∗ F │ S / F │ F**
(R4)	**F ::= Z │ (A) │ Y(B) │ Y()**
(R5)	**B ::= B , A │ 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.

::= – 400 – 3 ∗ (+2) ∗ 10

::= – 340

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

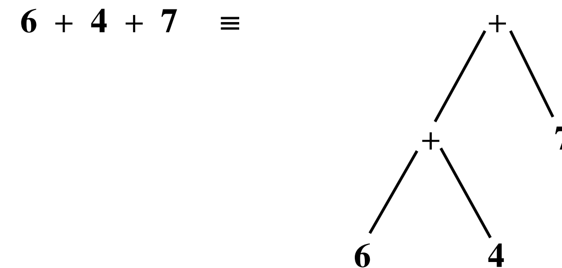
Bemerkung: **Erzeugung und Erkennung sind inverse Operationen.**

Darstellung arithmetischer Ausdrücke als Baum:

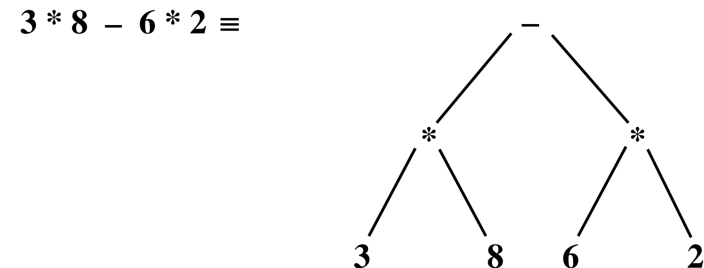
Beispiele:

3 + 5 ≡

6 + 4 + 7 ≡

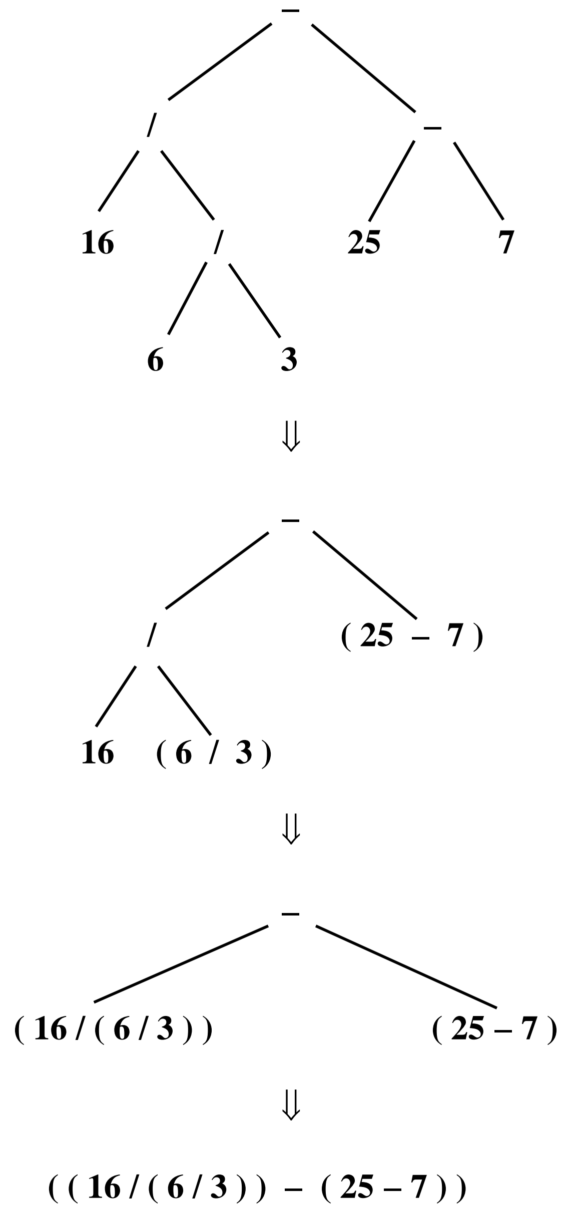


3 * 8 – 6 * 2 ≡



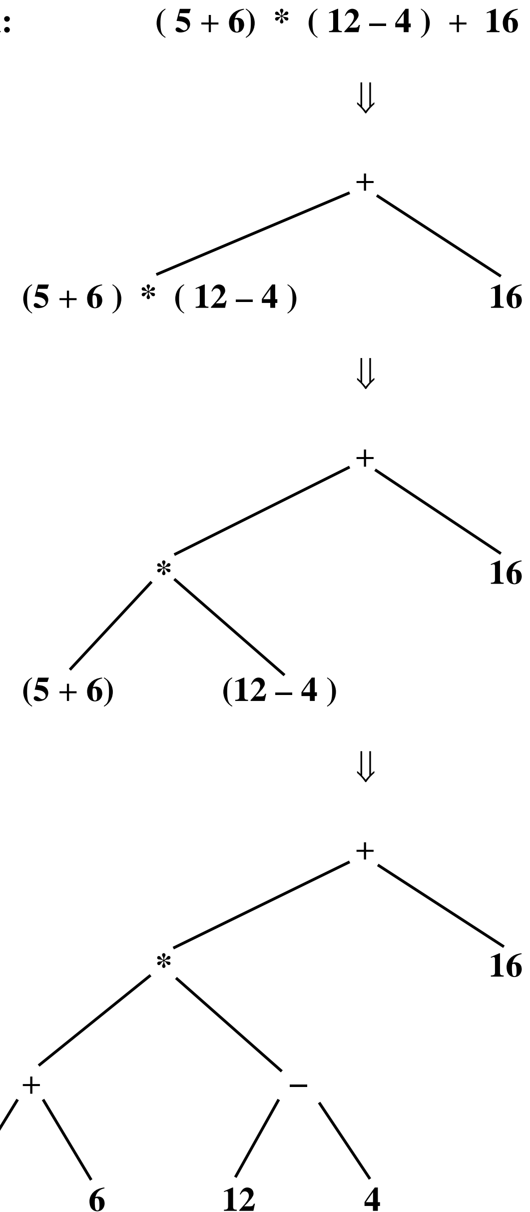
Umformung einer Baumdarstellung in eine lineare Darstellung:

Beispiel:



Umformung einer linearen Darstellung in eine Baumdarstellung:

Beispiel: **(5 + 6) * (12 – 4) + 16**



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:

P or Q ≡ Q or P
P and Q ≡ Q and P

Assoziativgesetze:

(P or Q) or R ≡ P or (Q or R)
(P and Q) and R ≡ P and (Q and R)

Distributivgesetze:

(P and Q) or R ≡ (P or R) and (Q or R)
(P or Q) and R ≡ (P and R) or (Q and R)

De Morgans Gesetze:

not (P or Q) ≡ not P and not Q
not (P and Q) ≡ not P or not Q

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 Elementarausdruck
- LK = logische Konstante
- VG = Vergleichsoperator
- AA = arithmetischer Ausdruck

Ein Regelsatz ("mehrdeutig"):

```
LA ::=  LA or LS     |   LS
LS ::=  LS and LF    |   LF
LF ::=  not LF       |   LE
LE ::=      LK       |   ( LA )
      |  LE = LE     |   LE ≠ LE
      | AA VG AA
VG ::=  <  |  ≤  |  >  |  ≥  |  ≠  |  =
```

Boolesche Algebra:

Sei A = { 0, 1 } eine zweielementige Wertemenge, 0 und 1 seien Synonyme für falsch und wahr, die Menge aller Abbildungen f: A $\longrightarrow$ 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 x A nach A:

Arg.	g_0	g_1	g_2	g_3	g_4	g_5	g_6	g_7	g_8	g_9	g_{10}	g_{11}	g_{12}	g_{13}	g_{14}	g_{15}
0 0	0	0	0	0	1	0	0	1	0	1	1	0	1	1	1	1
0 1	0	0	0	1	0	0	1	0	1	0	1	1	0	1	1	1
1 0	0	0	1	0	0	1	0	0	1	1	0	1	1	0	1	1
1 1	0	1	0	0	0	1	1	1	0	0	0	1	1	1	0	1

Fast alle Funktionen tragen einprägsame Namen:

- g_1 ≡ Konjunktion (AND-Funktion)
- g_{14} ≡ Sheffers Strich (NAND-Funktion)
- g_{11} ≡ Disjunktion (OR-Funktion)
- g_4 ≡ Peircescher Pfeil (NOR-Funktion)
- g_7 ≡ Äquivalenz
- g_8 ≡ Antivalenz (XOR-Funktion)
- g_{13} ≡ Implikation

Zwei Beispielrechnungen:

a and b	=	**not not (a and b)**
	=	**not (not a or not b)**
	=	**not a nor not b**
	=	**(a nor a) nor (b nor b)**
a or b	=	**not not (a or b)**
	=	**not (a nor b)**
	=	**(a nor b) nor (a nor b)**

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 g_{14} eine Basis ist.**
2. **Bestimmen Sie mindestens eine weitere Basis.)**

Ein Beispiel für die Vereinfachung von logischen Ausdrücken:

(P and Q) or (not P and Q)
≡ (P or not P) and Q
≡ true and Q
≡ Q

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) = (not x and y and z)
or (x and y and not z)
or (x and not y and not z)

Überprüfung für Argumente true, true, true:

f (true, true, true) = (not true and true and true)
or (true and true and not true)
or (true and not true and not true)
= false or false or false
= false

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

Wir betrachten nun n-stellige (n ≥ 2) logische Funktionen:

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

Es gilt das Lemma:

Für jede n-stellige logische Funktion f gilt:

$$
\begin{aligned}
& f(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n) \\
= \; & (a_i \text{ and } f(a_1, \ldots, a_{i-1}, 1, \ldots, a_n)) \\
\text{or } \; & (\text{not } a_i \text{ and } f(a_1, \ldots, a_{i-1}, 0, \ldots, a_n))
\end{aligned}
$$

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:

$$
\begin{aligned}
& f(a1, a2, \ldots, an) \\
& = (a1 \text{ and } a2 \text{ and } \ldots \text{ and } f(1, 1, \ldots, 1)) \\
& \text{or } (\text{not } a1 \text{ and } a2 \text{ and } a3 \ldots \text{ and } f(0, 1, \ldots, 1)) \\
& \ldots \\
& \text{or } (\text{not } a1 \text{ and not } a2 \text{ and } \ldots \text{ and } f(0, 0, \ldots, 0))
\end{aligned}
$$

Diese Darstellung nennt man die disjunktive Normalform.

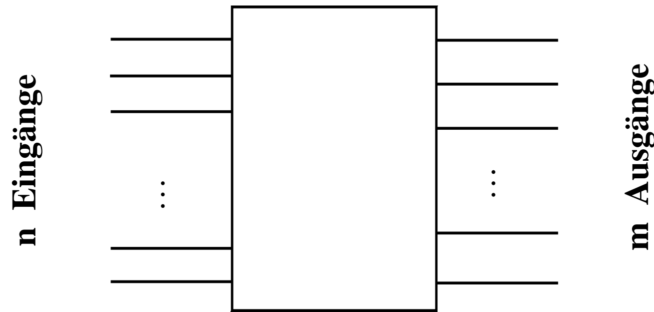
Boolesche Algebra:

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.

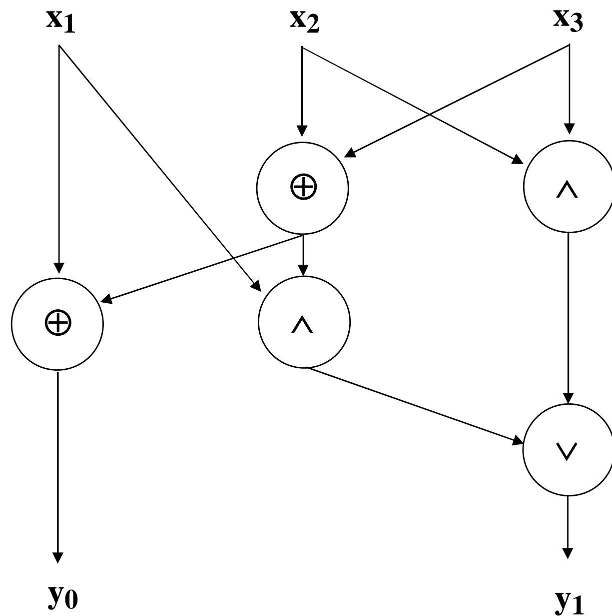
(B1)	Assoziativität:	$(x \wedge y) \wedge z = x \wedge (y \wedge z)$ $(x \wedge y) \wedge z = x \wedge (y \wedge z)$
(B2)	Kommutativität:	$x \wedge y = y \wedge x$ $x \vee y = y \vee x$
(B3)	Idempotenz:	$x \wedge x = x, \quad x \vee x = x$
(B4)	Verschmelzung:	$(x \wedge y) \vee x = x$ $(x \vee y) \wedge x = x$
(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:	$\neg(x \wedge y) = (\neg x) \vee (\neg y)$ $\neg(x \vee y) = (\neg x) \wedge (\neg y)$

Schaltglieder:

Darstellung eines Schaltglieds als schwarzen Kasten:



Beispiel eines Schaltnetzes:



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:

$$x \wedge y = x\,y,$$
$$x \vee y = x + y,$$
$$\text{not } x = x'$$

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 weitverbreitet:

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: Kosten $(a\,(b + c)'\,d + c)$ =
Kosten $(a\,(b + c)'\,d)$ + 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:

$f(a, b, c) = a'b'c + a'bc + ab'c + abc' + abc$

Kosten = 14

(ii) f in konjunktiver Normalform:

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

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

Kosten = 8

(iii) Minimierung von f mittels Karnaugh-Tafel:

a \ bc	00	01	11	10
0		1	1	
1		1	1	1

$f = c + ab$

Kosten = 2

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

$f(a, b, c) = a'b'c + a'bc + ab'c + abc' + abc$

$= a'b'c + a'bc + ab'c + abc' + abc + abc$

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

$= a'c + ac + ab$

$= c + ab$

Kosten = 2

Ein weiteres Beispiel:
Die Schaltfunktion f sei gegeben durch:

$f = a'b'c'd' + a'b'c'd + a'b'cd' + a'b'cd + a'bcd + ab'c'd' + ab'c'd + ab'cd' + abcd' + abcd$

Mögliche Vereinfachungen sind:

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

(ii) $f = acd' + bcd + a'b' + b'c$
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-Tafeln:

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

a' b

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

d

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

b d'

Lesen von Karnaugh-Tafeln:

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

= ?

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

= ?

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

= ?

Karnaugh-Tafel für c[2]:

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

		b[1]	0	0	1	1
a[1]	a[0]	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] / b[0]	0 / 0	0 / 1	1 / 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.

Das Verfahren von Quine-McCluskey:

Gegeben sei eine Schaltfunktion $f(x_1, x_2, \ldots, x_n)$ in den Variablen $x_1, x_2, \ldots, x_n$.

Bezeichnungen:

Eine Konjunktion $Y = y_1 y_2 \ldots y_n$, in der jede Variable von f genau einmal – nicht negiert oder negiert – auftritt, nennt man einen Minterm von f. Eine Konjunktion $Y = y_1 y_2 \ldots y_m$, in der jede Variable von f höchstens einmal – nicht negiert oder negiert – auftritt, nennt man einen Implikanten von f, falls aus Y = 1 f = 1 folgt. Ein Implikant I heißt Primimplikant, falls das Entfernen einer Variablen aus I dazu führt, daß I die Implikanteneigenschaft verliert.

Beispiele:

Sei f (a, b, c) = a' b c' + a' b c + a b' c' + a b c'.

Der Ausdruck a' b' c ist ein Minterm, aber kein Implikant.

Der Ausdruck a b' c' ist ein Implikant, aber kein Primimplikant.

Der Ausdruck a c' ist ein Primimplikant.

Das Minimierungsverfahren von Quine-McCluskey für Schaltfunktionen f besteht aus zwei Schritten:

1. **Bestimmung aller Primimplikanten von f.**
2. **Auswahl einer minimalen Überdeckung von f durch Primimplikanten.**

Beispiel:

Es wird c[0] des vorhergehenden Beispiels minimiert.

Schritt 1: Bildung der Minterme:

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

Schritt 2: Sortierung der Minterme nach Zahl der 1:

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

Schritt 3: Sukzessive Bildung der Implikanten:

Implikanten erster Stufe:

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

Fortführung von Schritt 3:

Implikanten zweiter Stufe:

a [1]	a[0]	b[1]	b[0]
–	0	–	1
–	1	–	0

Bemerkung: Die Implikanten zweiter Stufe sind in diesem Fall auch die Primimplikanten.

Schritt 4: Überdeckung der Minterme mittels Primimplikanten.

Dieser Schritt ist im Beispiel trivial. Man benötigt alle Primimplikanten und erhält:

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

Ein weiteres Beispiel zum Quine-McCluskey-Algorithmus:

Die Schaltfunktion $f(x_1, x_2, \ldots, x_n)$ sei in disjunktiver Normalform gegeben.

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'
+ a b c d'

Jede Konjunktion in der disjunktiven Normalform ist ein Implikant. Man schreibt vollständige Konjunktionen abgekürzt als Bitkette, wobei man die Reihenfolge der Variablen festlegt, die ungestrichenen Variablen durch eine 1 und die gestrichenen Variablen durch eine 0 darstellt.

Schritt 1a: **Darstellung der Funktion f als Tabelle kompakt codierter Minterme.**

f:		
	0:	0 0 0 0
	1:	0 0 0 1
	2:	0 0 1 0
	5:	0 1 0 1
	6:	0 1 1 0
	7:	0 1 1 1
	8:	1 0 0 0
	9:	1 0 0 1
	10:	1 0 1 0
	14:	1 1 1 0

Schritt 1b: **Sortierung der Minterme in Gruppen gemäß Zahl der unnegierten Variablen, gekenn- zeichnet durch eine 1.**

Gruppe 0:	0:	0 0 0 0	#
Gruppe 1:	1:	0 0 0 1	#
	2:	0 0 1 0	#
	8:	1 0 0 0	#
Gruppe 2:	5:	0 1 0 1	#
	6:	0 1 1 0	#
	9:	1 0 0 1	#
	10:	1 0 1 0	#
Gruppe 3:	7:	0 1 1 1	#
	14:	1 1 1 0	#

Schritt 1c: **Reduzierungen zwischen benachbarten Gruppen.**

Erste Reduzierung:

Gruppe 0:	0, 1:	0 0 0 -	#
	0, 2:	0 0 - 0	#
	0, 8:	- 0 0 0	#
Gruppe 1:	1, 5:	0 - 0 1	
	1, 9:	- 0 0 1	#
	2, 6:	0 - 1 0	#
	2, 10:	- 0 1 0	#
	8, 9:	1 0 0 -	#
	8, 10:	1 0 - 0	#
Gruppe 2:	5, 7:	0 1 - 1	
	6, 7:	0 1 1 -	
	6, 14:	- 1 1 0	#
	10, 14:	1 - 1 0	#

Zweite Reduzierung:

Gruppe 0:	0, 1, 8, 9:	- 0 0 -
	0, 2, 8, 10:	- 0 - 0
Gruppe 1:	2, 6, 10, 14:	- - 1 0

Es sind nun keine weiteren Reduzierungen möglich. Es wurden 6 Primimplikanten gefunden, nämlich:

(1, 5), (5, 7), (6, 7), (0, 1, 8 ,9), (0, 2, 8, 10) und (2, 6, 10, 14).

Schritt 2: **Bildung der Überdeckungstafel für Primimplikanten**

Primimplikant	Minterme 0	1	2	5	6	7	8	9	10	14
A = 1, 5		x		x						
B = 5, 7				x		x				
C = 6, 7					x	x				
D = 0, 1, 8, 9	x	x					x	x		
E = 0, 2, 8, 10	x		x				x		x	
F = 2, 6, 10 14			x		x				x	x

Eine vollständige Überdeckung wird beschrieben durch die Konjunktion:

$$\begin{aligned}\text{UEBER} &= (D \vee E) \wedge (A \vee D) \wedge (E \vee F) \wedge (A \vee B) \wedge (C \vee F) \\ &\quad \wedge (B \vee C) \wedge (D \vee E) \wedge D \wedge (E \vee F) \wedge F \\ &= D \wedge F \wedge (A \vee B) \wedge (B \vee C) \\ &= (D \wedge F \wedge A \wedge B) \vee (D \wedge F \wedge A \wedge C) \\ &\quad \vee (D \wedge F \wedge B) \vee (D \wedge F \wedge B \wedge C) \\ &= (D \wedge F \wedge B) \vee (D \wedge F \wedge A \wedge C)\end{aligned}$$

Damit ist eine mögliche Vereinfachung der Ausgangsfunktion:

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

Programmbeweise:

Zu zeigen ist, daß ein Programm P eine Spezifikation S erfüllt.

Dies wird oft reduziert auf folgenden Sachverhalt:

Sei P das zu verifizierende Programm.
Vor Ausführung von P möge eine Menge {V} von Vorbedingungen gelten.
Nach Ausführung von P erwartet man die Gültigkeit einer Menge {N} von Nachbedingungen.
V und N seien Ausdrücke einer vorgegebenen Logik.
Oft verwendet man die Prädikatenlogik.

Zu zeigen ist nun:
- **(i)** **Wenn P in einem Zustand, in dem {V} gilt, startet, dann terminiert P.**
- **(ii)** **Wenn P terminiert, dann überführt P jeden Zustand, in dem {V} gilt, in einen Zustand, in dem {N} gilt.**

Schreibweise nach Hoare: {V} P {N}.

Dies legt nahe, den Programmbeweis in einer Folge von Einzelschritten auszuführen.
Sei $P = a_1; a_2; a_3; \ldots; a_n$, **man sucht nun eine Folge von Bedingungen** B_i $(0 \leq i \leq n)$ **zu bestimmen mit**
$V = B_0$, $N = B_n$ **und** $\{B_0\}\ a_1\ \{B_1\}\ a_2\ \ldots\ \{B_{n-1}\}\ a_n\ \{B_n\}$.
Gilt $\{B_i\}\ a_i\ \{B_{i+1}\}$ **für** $0 \leq i < n$, **dann folgt** $\{V\}\ P\ \{N\}$.

Suche nach Vor- und Nachbedingungen:

Beispiel 1:

```
{ a und b seien vergleichbare Zahlen }
                b  :=  a + 1
                   { b > a }
```

Frage: Gibt es Fälle, in denen die Schlußaussage nicht gilt?

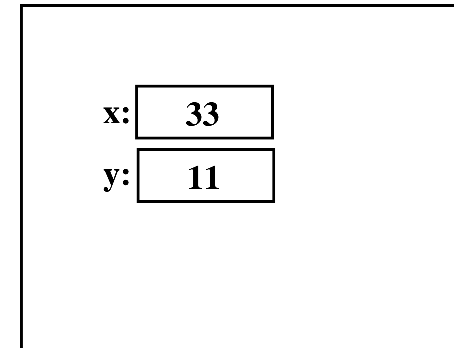
Beispiel 2:

```
{  Vorbedingungen  =  ? }
while  i ≤ n  do  begin
       p := p*s;
       i  :=  i + 1;
end
{   p  =  s^n }
```

Frage: Welche Forderungen an i, n, p und s sind zu stellen?

Zustandsaussagen im von Neumann Modell:

Speicher:



Programm:

```
x  :=  33
y  :=  11
```

Aussage: $x + y = 44$

Programm:

```
x  :=  x + 8
y  :=  y – 8
```

Aussage: $x + y = 44$

Bemerkung: Den einzelnen Punkten einer Programmausführung lassen sich logische Aussagen zuordnen. Diese lassen sich zu einem Beweis der Korrektheit eines Programms nutzen.

Was berechnet das folgende Programm B?

```
lies  (ein);

//  ein > 0  sei eine natürliche Zahl

x  :=  0;

y  :=  ein + 1;

while  y  >= 8
repeat

      x :=  x + y div 8;

      y := y div 8 + y mod 8

end

schreib ( x,  y – 1)
```

Testläufe von B:

Eingabe: 65
Ausgabe: 9, 2

Eingabe: 300
Ausgabe: 42, 6

Eingabe: 400
Ausgabe: 57, 1

Testläufe mit Zwischenausgaben:

ein = 65

x	y
8	10
9	3

ein = 300

x	y
37	42
42	7

ein = 400

x	y
50	51
56	9
57	2

Die Frage bleibt: Welche Bedeutung hat B?

Programm B mit Zusicherungen:

```
lies (ein);

{ ein > 0 and ein ganzzahlig }

x := 0;
y := ein + 1;

{ ein + 1 = 7 * x + y }
// obige Formel ist auch Schleifeninvariante

while y >= 8
repeat
    x := x + y div 8;
    y := y div 8 + y mod 8
end

{ ein = 7 * x + y – 1 and y < 8 and y > 0 }

{ x     = ein div 7 and
  y – 1 = ein mod 7 }

schreib (x, y-1)
```

Zunächst der Beweis, daß ein + 1 = 7 * x + y eine Schleifeninvariante ist.

Vor einem beliebigen Schleifendurchlauf gelte:

$$ein + 1 = 7 * x_{alt} + y_{alt} \text{ and } y_{alt} >= 8.$$

Zu zeigen ist: Nach einer Ausführung des Schleifenkörpers gilt:

$$ein + 1 = 7 * x_{neu} + y_{neu}.$$

Beweis:

$y_{alt} >= 8$ **garantiert den Eintritt in den Schleifenkörper; die Zuweisungen setzen:**

$x_{neu} = x_{alt} + y_{alt}$ div 8,

$y_{neu} = y_{alt}$ div 8 + y_{alt} mod 8.

Man rechnet aus:

$$\begin{aligned} & 7 * x_{neu} + y_{neu} \\ = \; & 7 * (x_{alt} + y_{alt} \text{ div } 8) + y_{alt} \text{ div } 8 + y_{alt} \text{ mod } 8 \\ = \; & 7 * x_{alt} + 8 * (y_{alt} \text{ div } 8) + y_{alt} \text{ mod } 8 \\ = \; & 7 * x_{alt} + y_{alt} \\ = \; & ein + 1 \qquad \text{q.e.d.} \end{aligned}$$

Nach der Initialisierung gilt:

$$\begin{aligned} & 7 * x + y \\ = \; & 7 * 0 + (ein + 1) \\ = \; & ein + 1 \end{aligned}$$

Nach Beendigung der Schleife gilt:

$$ein + 1 = 7 * x + y \text{ and } y < 8.$$

Da zusätzlich $y > 0$ eine Programminvariante ist, gilt die Endaussage.

Es bleibt zu zeigen, daß jeder Programmlauf terminiert.

Ein bequemer Weg, dies zu zeigen, besteht darin, für die Anzahl der Schleifendurchläufe eine (auch datenabhängige) obere Schranke anzugeben.

In diesem Fall ist der Anfangswert von y eine obere Schranke; dies folgt aus:

(i) $y > 0$ **Programminvariante**

(ii) $y_{neu} < y_{alt}$ **nach einem Schleifendurchlauf, denn**

$$y_{neu} = y_{alt} \text{ div } 8 + y_{alt} \text{ mod } 8$$
$$< 8 (y_{alt} \text{ div } 8) + y_{alt} \text{ mod } 8$$
$$= y_{alt},$$

da wegen Schleifenbedingung $y_{alt} \text{ div } 8 > 0$.

Ein Programm zur Berechnung einer ganzzahligen Wurzel:

Spezifikation: **Sei $n \geq 0$ eine natürliche Zahl; die natürliche Zahl a mit $a^2 \leq n$ und $n < (a+1)^2$ nennt man die ganzzahlige Wurzel von n.**

Programmfragment mit Zusicherungen:

```
{ n ≥ 0 }
a := 0;
b := n + 1;
{ a² ≤ n ∧ n < b² }
while b - a ≠ 1
repeat
      c := (a + b) div 2;
      if c² ≤ n then
            a := c
      else
            b := c
      fi
end
{ a² ≤ n ∧ n < (a + 1)² }
```

Aufgabe: **Führen Sie den Beweis der Korrektheit obigen Textes; als Schrankenfunktion zum Terminierungsnachweis kann man $b - a$ nutzen.**

Ein weiteres Programmfragment zur Berechnung einer ganzzahligen Wurzel:

$\{\ n \geq 0\ \}$

```
a := n;
```

$\{\ n < (a+1)^2 \ \wedge\ a \geq 0\ \}$

```
while a^2 > n
repeat
      a := (a + n div a) div 2;
end
```

$\{\ a^2 \leq n \ \wedge\ n < (a+1)^2\ \}$

Aufgabe: Führen Sie den Beweis der Korrektheit obigen Textes; als Schrankenfunktion zum Terminierungsnachweis kann man a nutzen.

Bemerkungen zu Beweisen:

(i) Programmbeweise sollten vollständig sein.

(ii) Programmbeweise finden oft in einer idealisierten Welt statt, z. B. werden die Beschränktheit der Zahlbereiche oder Rundungsfehler oft nicht berücksichtigt. Daher bieten Beweise nicht immer eine Gewähr für korrektes Arbeiten von Programmen.

(iii) Es ist schwierig, Beweise zu fertigen Programmen zu finden; daher ist es sinnvoll, ein Programm und seinen Beweis gleichzeitig zu entwickeln.

(iv) Es ist sinnvoll, einzelne Eigenschaften von Programmen wie Lebendigkeit und Vermeidung unsicherer Zustände zu beweisen.

(vi) Auch Beweise können fehlerhaft sein, daher sollte man niemals auf einen Programmtest verzichten. Viele Beispiele für fehlerhafte Beweise findet man in Gerhart und Yelowitz: Observations on Fallibility in Applications of Modern Programming Methodologies. IEEE Trans. Software Eng. 2 (1976).