INFORMATIK I

Technische und sprachliche Grundlagen

Fachhochschule Fulda
Fachbereich Elektrotechnik
Prof. Dr. Timm Grams

Datei: INFORM_1.DOC
20. Januar 2009 (Erstausgabe: 11.8.94)

Beschreibung der Lehrveranstaltung

Die Methoden der Informatik nehmen heute in der Elektrotechnik großen Raum ein. Anwendungsgebiete sind (unter vielen anderen):

- Entwicklung und Konstruktion höchstintegrierter Schaltungen
- Anforderungserfassung, Planung, Bau, Diagnose, Dokumentation und Pflege von Anlagen (Computer Aided Engineering, CAE)
- Simulation von Schaltungen, Nachrichtennetzen und Automatisierungssystemen
- Anwendungssoftware in informationstechnischen Systemen und in Automatisierungssystemen.

Ziel: Die Lehrveranstaltung vermittelt grundlegende Konzepte des Aufbaus und der Programmierung von Rechnern. Der Teilnehmer ist schließlich in der Lage, kleinere Aufgaben und Probleme mit Hilfe des Rechners selbst zu lösen.

Konzept der Lehrveranstaltung: Die Sichtweise dieser Lehrveranstaltung ist grundsätzlich abstrahierend und generalisierend. Elementare Techniken der Informatik werden auf konstruktivem Wege eingeführt und nicht einfach vorgestellt. Das dient dem besseren Verständnis der bekannten Lösungen und erleichtert die Übertragung auf neue Probleme - also das Erfinden.

Aufbau: Die Lehrveranstaltung besteht aus einer Vorlesung mit 24 Lektionen. Einige zusätzliche Lektionen sind in der Gliederung mit einem Sternchen gekennzeichnet. Parallel dazu läuft das Praktikum. Die 12 Lektionen des ersten Semesters behandeln die technischen und sprachlichen Grundlagen der Informatik. Am Computer finden im ersten Semester Übungen mit dem Betriebssystem, mit einem Editor und mit fertigen Programmen statt: `MS-DOS, SPICE, LogScope, LogTrans.` Im zweiten Semester soll der Einstieg in höhere Programmiersprachen `(Pascal, C)` durch die Programmierung grundlegender Algorithmen vollzogen werden. Für die Grafikausgabe wird ein Tabellenkalkulationsprogramm genutzt (`Excel`).

Begleitmaterial: Das Skriptum besteht aus zwei Teilen, in denen der Stoff der zweisemestrigen Lehrveranstaltung knapp zusammengefasst ist, und aus einem Begleitheft zum Praktikum. Das Skriptum ist gedacht als Leitfaden, nicht etwa als Lehrbuch. Zum Gebrauch von guten Büchern wird dringend geraten! Finden Sie selbst heraus, was Ihnen weiterhilft. Die Literaturhinweise des Skriptums sollen bei der Auswahl helfen. Will man die Wirkung von Programmen und Algorithmen verstehen, muss man sich die von ihnen gesteuerten Abläufe klar machen. Für die Darstellung solcher Abläufe eignet sich ein statischer Text nicht sehr gut. Auch an der Wandtafel entsteht eher Verwirrendes. Deshalb werden in Ergänzung zum Skriptum Animationen angeboten. Sie sind auf der Web-Page `http://www.fh-fulda.de/~grams/informat.htm` zu finden.

Tips

1. Beschäftigen Sie sich *von Anfang an* intensiv mit dem angebotenen Stoff und nehmen Sie die Übungsangebote wahr: die Klausur ist schneller da als man glaubt.
2. Sollten Sie die Bedeutung eines in der Lehrveranstaltung verwendeten Begriffes einmal nicht parat haben, sehen Sie im *Sachverzeichnis* nach. Sie finden an der ersten Verweisstelle den Begriff in Kursivschrift und meist auch eine Begriffsbestimmung.
3. Nicht alle *Übungen* und *Aufgaben* des Skriptums werden in der Lehrveranstaltung behandelt. Versuchen sie jedenfalls immer, diese Aufgaben selbst zu lösen. Wenn Sie Fragen dazu haben, dann können diese, je nach Bedeutung für die Allgemeinheit, in der Vorlesung, im Praktikum oder in der Sprechstunde behandelt werden.

Gliederung

Teil I

Literatur

Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Massachusetts 1983

Blieberger, J.; Schildt, G.-H.; Schmid, U.; Stöckler, S.: Informatik. Springer, Wien 1990

Fricke, K.: Digitaltechnik. Vieweg, Braunschweig 1999

Grams, T.: Codierungsverfahren. BI-Taschenbuch, Band 625, BI-Mannheim, 1986

Gumm, H.-P.; Sommer, M.: Einführung in die Informatik. Oldenbourg, München 1998. Umfassende Darstellung der Grundlagen der heutigen Informatik.

Knuth, D.: The Art of Computer Programming. Vol. 1: Fundamental Algorithms. Addison-Wesley 1973

Knuth, D.: The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. Addison-Wesley 1981

MS-DOS 5.0: Benutzerhandbuch. Microsoft 1991

Schiffmann, W.; Schmitz, R.: Technische Informatik. Band 1: Grundlagen der digitalen Elektronik. Springer, Berlin, Heidelberg, New York 1993

Schiffmann, W.; Schmitz, R.: Technische Informatik. Band 2: Grundlagen der Computertechnik. Springer, Berlin, Heidelberg, New York 1992

Wendt, S.: Nachrichtenverarbeitung (Nachrichtentechnik Band 3 von K. Steinbuch und W. Rupprecht). Springer-Verlag, Berlin, Heidelberg, New York 1982

Nachschlagewerke

Bronstein, I. N.; Semendjajew, K. A.; Musiol, G.; Mühlig, H.: Taschenbuch der Mathematik. Verlag Harri Deutsch, Thun, Frankfurt am Main, 1993

Hütte: Die Grundlagen der Ingenieurwissenschaften. 25. Auflage. Springer, Berlin, Heidelberg 1991

Informatik-Duden: Ein Sachlexikon für Studium und Praxis. Bibliographisches Institut & F.A. Brockhaus AG, Mannheim 1988

Löffler, H.,; Meinhardt, J.; Werner, D.: Taschenbuch der Informatik. Fachbuchverlag Leipzig 1992

1 Technische Grundlagen

„Meines Erachtens gibt es einen Weltmarkt für vielleicht fünf Computer“
IBM Präsident Thomas Watson, 1943

zitiert nach Hermann Maurer
Informatik-Spektrum 23 (2000) 1, S. 51

1.1 Die Mechanisierung des Rechnens

Stellenwertsystem und Algorithmus. Anmerkungen zur historischen Entwicklung und zum heutigen Stand der Informatik.

Eriks Trick: Sack für Sack tragen die Packleute in das Lager. Erik zählt mit: Für jeden Sack legt er - aus einem Vorrat von Steinen - ein Steinch∈n in ein rechteckiges Feld, das er sich im Sand aufgemalt hat. Gegen Mittag wird der Steinchenvorrat allmählich knapp. 202 Säcke sind gezählt. Glücklicherweise ist jetzt Mittagspause und Erik überlegt, wie er mit weniger Steinen beim Zählen auskommen kann. Er erinnert sich an einen alten Trick: Zunächst einmal ergänzt er sein Rechteck um einige weitere Felder, **Bild 1.1-1 a**.

(a)				202
(b)			40	2
(c)		8	0	2
(d)	1	3	0	2

Bild 1.1-1 Eriks Trick

Dann teilt er seine „Zählsteine“ gleichmäßig auf ebensoviele Haufen auf, wie er Finger an einer Hand hat. Dabei bleiben 2 Steine übrig, die lässt er liegen. Von den fünf Steinhaufen legt er alle bis auf einen wieder zum Vorrat zurück. Einen der Haufen aus 40 Steinen hingegen legt er in das nächste Feld weiter links, **Bild 1.1-1 b**.

Nun wiederholt er diesen Schritt: Er teilt die 40 Steine auf fünf Haufen auf. Diesmal bleibt nichts übrig. Er lässt das Kästchen leer (wir schreiben eine 0 hinein), übernimmt einen der fünf Haufen zu je acht Steinen in das nächste Feld und wirft die anderen zum Vorrat, **Bild 1.1-1 c**. Die acht Steine teilt er erneut auf, wirft vier der fünf „Haufen“ (je ein Stein) zum Vorrat, überträgt einen in das nächste Feld und lässt den Rest liegen, **Bild 1.1-1 d**.

Jetzt gibt es nichts mehr aufzuteilen. In jedem der vier Felder liegen weniger als 5 Steine. Erik hat sein Ziel erreicht: Anstelle der ursprünglich 202 Steine benötigt er nur noch sechs.

Wenn Erik wieder für jeden Sack einen Stein haben will, geht er genau umgekehrt vor: Er muss die Steine nur von links nach rechts transportieren und immer, wenn ein Stein von einem Feld in das rechts daneben liegende Nachbarfeld wechselt, vier weitere Steine aus dem Vorrat dazulegen.

Was Erik da macht, ist nichts anderes als die Zahlendarstellung im *Stellenwertsystem* zur Basis fünf. Als Basis kann man jede andere natürliche Zahl größer 1 wählen. Falls eine andere Basis als 10 gewählt wird und Verwechslung droht, schreiben wir die Basis als tiefgestellte Dezimalzahl an die Ziffernfolge. Offenbar gilt $202 = (1302)_5$.

Die Bezeichnung „Stellenwertsystem“ ergibt sich aus der Tatsache, dass den Steinen je nach Position verschiedene Werte zukommen: Jeder Stein im ganz rechten Kästchen steht für sich

selbst. Jeder Stein im zweiten Feld von rechts steht für fünf Steine. Jeder Stein im dritten Feld von rechts steht für fünf mal fünf, also fünfundzwanzig Steine usw.

Eriks Vorgehen lässt sich verallgemeinern: Um die Darstellung einer positiven Zahl z in einem Stellenwertsystem zur Basis b zu erhalten, verfährt man so[1]:

(1) Setze $x_0 = z$

(2) Berechne x_1 und r_0, so dass $x_0 = bx_1 + r_0$ und $r_0 < b$
Berechne x_2 und r_1, so dass $x_1 = bx_2 + r_1$ und $r_1 < b$
Berechne x_3 und r_2, so dass $x_2 = bx_3 + r_2$ und $r_2 < b$

...

(3) Beende die Rechnung, sobald für einen Index n gilt $x_n = 0$

Es wird also mehrmals hintereinander die *ganzzahlige Division* mit Rest angewendet. Durch sukzessives Einsetzen ergibt sich

$$
\begin{aligned}
& z \\
&= x_0 \\
&= b \cdot x_1 + r_0 \\
&= b \cdot (b \cdot x_2 + r_1) + r_0 = b^2 \cdot x_2 + b \cdot r_1 + r_0 \\
&= b^2 \cdot (b \cdot x_3 + r_2) + b \cdot r_1 + r_0 = b^3 \cdot x_3 + b^2 \cdot r_2 + b \cdot r_1 + r_0 \\
&\ldots \\
&= b^{n-1} \cdot r_{n-1} + \ldots + b^2 \cdot r_2 + b \cdot r_1 + r_0 \\
&= b^{n-1} \cdot r_{n-1} + \ldots + b^2 \cdot r_2 + b^1 \cdot r_1 + b^0 \cdot r_0 \\
&= (r_{n-1} \ldots r_2 r_1 r_0)_b
\end{aligned}
$$

Dabei ist die letzte Zeile nur als abgekürzte Schreibweise der vorhergehenden zu verstehen.

Dass in der kleinen Geschichte von Erik das uns geläufige Stellenwertsystem zur Basis 10 - das *Dezimalsystem* - bereits auftritt, tut nichts zur Sache: Es wird nur zur Vereinfachung der Darstellung benötigt.

Übung 1: Erläutern Sie, wie Erik wohl am Nachmittag beim Zählen verfahren wird.

Übung 2: Was wird er tun, wenn Säcke aus dem Lager heraus getragen werden.

Übung 3: Wie sieht die Zahl 88 im Stellenwertsystem zur Basis 3 aus?

Erik kommt zu seiner Zahlendarstellung durch einen *Algorithmus*. Ein Algorithmus ist ein Satz von Regeln bzw. Handlungsanweisungen, deren wiederholte und mechanische Anwendung auf ein Problem dieses von einem Anfangszustand in einen erwünschten Zielzustand überführt.

Die Grundeigenschaften eines Algorithmus sind Bestimmtheit, Allgemeingültigkeit und Zielorientiertheit. Was heißt das?

1. *Bestimmtheit* bedeutet, dass auf jeder Bearbeitungsstufe durch den erreichten Bearbeitungszustand und die Handlungsanweisungen der Folgezustand eindeutig festgelegt ist:

[1] Alle auftretenden Zahlen werden als nichtnegativ vorausgesetzt. Außer den natürlichen Zahlen kommt höchstens noch die Null vor.

$z_{k+1} = f(z_k)$. Hierin ist k eine Stufennummer; z_k steht für den Bearbeitungszustand auf dieser Stufe; die Funktion f ist durch die Handlungsanweisungen gegeben. Erik befolgt die Handlungsanweisung „Teile einen Haufen mit mehr als vier Steinen auf fünf Haufen gleichmäßig auf, lasse den Rest liegen, übernehme einen der Haufen in das linke Nachbarfeld, werfe die anderen vier Haufen zum Vorrat". Auf diese Weise kommt Erik vom Anfangszustand z_0 (Bild 1.1-1 a) zum Zustand z_1 (Bild 1.1-1 b), danach zu z_2 (Bild 1.1-1 c) und schließlich zu z_3 (Bild 1.1-1 d).

2. Unter *Allgemeingültigkeit* ist zu verstehen, dass der Algorithmus auf eine ganze Klasse von Problemen anwendbar ist, und nicht bloß auf einen speziellen Fall. Das heißt, dass der Anfangszustand zu einer Klasse möglicher Anfangszustände gehört: $z_0 \in Z_{\text{Anfang}}$. Klar ist, dass Eriks Trick auch dann noch funktioniert, wenn eine andere Zahl von Steinen als 202 vorgegeben ist. Zugelassen ist jede natürliche Zahl.
3. *Zielorientiertheit* meint, dass nach einer endlichen Anzahl n von Schritten ein Zielzustand erreicht wird: $z_n \in Z_{\text{Ziel}}$. Die Menge möglicher Zielzustände Z_{Ziel} wird durch eine *Zielbedingung* - auch *Endebedingung* genannt - definiert. Für Eriks Algorithmus lautet die Zielbedingung: „In jedem Feld befinden sich weniger als fünf Steine". Im konkreten Fall ist das Ziel nach drei Schritten erreicht; also ist $n = 3$.

Die Bedeutung der Algorithmen liegt darin, dass sich mit ihrer Hilfe die Lösung eines Problems in zwei Teilschritte zerlegen lässt:

1. Möglichst präzise und unmissverständliche Formulierung von Handlungsanweisungen bzw. Rechenvorschriften.
2. Wiederholte und mechanische Anwendung der Rechenvorschriften auf das konkrete Problem.

Der erste Teilschritt verlangt Kreativität. Das ist die Domäne des Menschen. Den zweiten Teilschritt überlässt man besser einer Maschine. Sie kann ohne Ermüdungserscheinungen die notwendigen Rechnungen stur, schnell und genau ausführen.

Das Stellenwertsystem erlaubt es, wirksame Algorithmen für die Zahlenrechnung (Addition, Subtraktion, Multiplikation und Division) zu formulieren. Jeder kennt sie aus der Schule. Zusammenfassend lässt sich sagen: Stellenwertsystem und Algorithmen sind die zentralen Voraussetzungen für eine *Mechanisierung des Rechnens*.

Anmerkungen zur historischen Entwicklung

Die historische Einordnung von Erfindungen und Entdeckungen kann die Orientierung in einem Fachgebiet erleichtern. In der Zeittafel **(Tabelle 1.1-1)**, sind einige Daten aus der Wissenschafts- und Technikgeschichte zusammengestellt.

Die Basis einer Mechanisierung des Rechnens - und damit der heutigen Informatik - wurde bereits im Mittelalter gelegt. (So finster war das Mittelalter wohl doch nicht.)

Die Mechanisierung des Rechnens ist eng mit den Namen zweier Personen verbunden: *al-Khwarizmi* (im Lexikon auch unter *Hwarizmi, Charismi, Chwarismi, Khuwarizmi* oder *Chorezmi* zu finden) und *Fibonacci* (eigentlich: *Leonardo von Pisa*).

Al-Khwarizmi erläuterte in einer seiner Arbeiten zur Arithmetik das dezimale Stellenwertsystem (Dezimalsystem) und die darauf beruhenden Rechenverfahren - beides aus Indien übernommen. Auf seinen Namen geht das Wort Algorithmus zurück. Sein Werk wurde ins Lateinische übersetzt und hatte großen Einfluss auf Fibonacci.

Fibonacci verwendete konsequent die indisch-arabischen Ziffern und zeigte damit die Vorteile des Dezimalsystems. Er sorgte mit seinem Buch der Rechenkunst (Liber Abaci) für die Verbreitung dieser Gedanken.

Den mathematischen Fachausdruck Funktion haben wir von *Gottfried Wilhelm Leibniz*. Er baute eine arbeitsfähige Rechenmaschine und erkannte die Bedeutung des binären Zahlensystems, also des Stellenwertsystems zur Basis zwei.

Der Bau von Rechenanlagen profitiert vor allem von der algebraischen Begründung der Aussagenlogik durch *George Boole*.

Die formalsprachlichen Aspekte der Logik und das von *Gottlob Frege* erstmals vollständig beschriebene Prädikatenkalkül bilden die Grundlage der Programmierung von Rechenanlagen.

Konrad Zuse ist der „Schöpfer der ersten vollautomatischen, programmgesteuerten und frei programmierbaren, in binärer Gleitpunktrechnung arbeitenden Rechenanlage. Sie war 1941 betriebsfähig." Das schreibt F. L. Bauer im Geleitwort zu Zuses Autobiographie (1984).

John von Neumann wird die Idee zugeschrieben, Programme genauso zu speichern wie Daten (Bauer, 1998). Das ist der Anfang der modernen Rechnerarchitekturen.

Friedrich L. Bauer erkannte den Kellerspeicher (Stack) als ein Schlüsselelement der universellen Programmiersprachen. Er ermöglicht unter anderem die direkte Auswertung algebraischer Formeln, wenn diese in der (klammerfreien) umgekehrten polnischen Notation gegeben sind. Diese Notation wurde von dem polnischen Logiker *Jan Łukasiewicz* eingeführt.

Erläuterung: Der Ausdruck „3*(*x*+1)-*a*/*x*" lautet in umgekehrter polnischer Notation „3 *x* 1 + * *a x* / -".

Gliederung der Informatik

Heute umfasst die Wissenschaft vom Computer, die *Informatik* (englisch: *Computer Science*), nicht nur das Rechnen!

Die Informatik errang Ende der sechziger Jahre den Status einer neuen Grundlagenwissenschaft und wird seither in eigenen Studiengängen an den Hochschulen gelehrt. Die Teilgebiete der Informatik sind in Anlehnung an die Empfehlungen der Gesellschaft für Informatik[1] in der **Tabelle 1.1-2** erfasst.

[1] Empfehlungen der Gesellschaft für Informatik e. V. zur Stärkung der Anwendungsorientierung in Diplom-Studiengängen der Informatik an Universitäten. Informatik-Spektrum 22 (1999) 6, S. 444-448

Tabelle 1.1-1 Zeittafel zur Mathematik, Naturwissenschaft und Technik (kursiv: unmittelbare Bedeutung für die Informatik)

Jahr	*Erfindungen und Entdeckungen*
3000 v.C.	Sumerer: Rad
300 v.C.	Euklid: Elemente der Geometrie
820	*al-Khwarizmi: Algorithmus, Algebra*
1202	*Fibonacci: Dezimalzahlen*
1445	Gutenberg: Buchdruck
1543	Kopernikus: heliozentrisches Weltsystem
1616	Galilei beobachtet Venusphasen und widerlegt das geozentrische Weltsystem
1674	*Leibniz: funktionsfähige Rechenmaschine*
1765	Watt: Dampfmaschine
1824	Carnot: Kreisprozess
1831	Faraday: elektrische Induktion
1833-37	Gauß, Weber, Morse: Telegraph
1842	Darwin: Abstammungslehre Mayer: Energieerhaltungssatz
um 1850	Anwendung wissenschaftlicher Methoden in Medizin (Asepsis) und Landwirtschaft (Mineraldünger)
1854	*Boole: Algebra der Logik*
1861-76	Reis, Bell, Gray: Telefon
1865-77	Clausius, Boltzmann: Entropie
1879	*Frege: Prädikatenkalkül*
1865-88	Maxwell, Hertz: elektromagnetische Wellen
1896	Marconi: drahtlose Telegrafie
1923	Rundfunk in Deutschland
1941	*Zuse: Computer*
1944	*von Neumann: Konzept allgemeiner Programmverarbeitung*
1948	*Shockley u.a.: Transistor*
1950	*Bauer: Kellerspeicher*

Tabelle 1.1-2 Teilgebiete und Fächer der Informatik

Technische Informatik	Praktische Informatik	Theoretische Informatik	Anwendungen der Informatik (Beispiele)
Technologische Grundlagen	Programmiersprachen, Programmiermethodik	Automaten, Formale Sprachen, Berechenbarkeit, Komplexität	Automatisierungstechnik
Rechnerarchitektur, Rechnerorganisation	Datenstrukturen, Algorithmen	Logik, Semantik, Wissensrepräsentation	Computergraphik
Maschinennahe Programmierung, Systemprogrammierung	Software-Engineering	Formale Spezifikation, Verifikation	Simulation
Betriebssysteme, Rechnerkommunikation, Netze	Datenbanken und Informationssysteme	Entwurf und Analyse von Algorithmen	Wirtschaftsinformatik
Echtzeitsysteme, eingebettet Systeme, Robotik	Mensch-Maschine-Interaktion		Rechtsinformatik
Verteilte Systeme	Wissensbasierte Systeme		Medizinische Informatik

Literaturhinweise

Für diese Lektion wurden neben den Büchern von Knuth (1973, Abschnitt 1.1 „Algorithms“; 1981, Abschnitt 4.1 „Positional Number Systems“) folgende zusätzliche Quellen benutzt:

Bauer, F. L.: Wer erfand den von-Neumann-Rechner? Informatik-Spektrum 21 (1998) 2, 84-88. *Ein Computer-Pionier schreibt über die Urheberschaft der modernen Rechnerarchitektur.*

Lexikon bedeutender Mathematiker. Bibliographisches Institut Leipzig 1990

Lüneburg, H.: Leonardi Pisani Liber Abacci oder Lesevergnügen eines Mathematikers. BI Mannheim 1992

Petzold, H.: Moderne Rechenkünstler. Die Industrialisierung der Rechentechnik in Deutschland. C. H. Beck, München 1992

Vorndran, E. P.: Entwicklungsgeschichte des Computers. VDE-Verlag, Berlin, Offenbach 1986. Eine knappe und allgemeinverständliche Darstellung der Entwicklung vom Rechenbrett zum heutigen Computer mit ausgewählten und gut erklärten technischen Details.

Zemanek, H.: Das geistige Umfeld der Informationstechnik. Springer-Verlag, Berlin, Heidelberg 1992

Zuse, K.: Der Computer - Mein Lebenswerk. Springer-Verlag, Berlin, Heidelberg 1984

1.2 Binäre Codierung

Computer verarbeiten Nachrichten in standardisierter Form. Es ist zunächst zu klären, was unter einer Nachricht zu verstehen ist, und wie sie computergerecht dargestellt werden kann: Codierungsverfahren, Codetabellen für Texte, Zahlendarstellungen.

Zeitungsartikel, Startschuss, geschlossene Schranke, SOS, Achselzucken, Kavalierstart, Raumtemperatur, Zahlenkombination eines Schlosses, Rundfunkkommentar: Diese unsortierte Sammlung von Beispielen für Nachrichten zeigt, wie aussichtslos es ist, den Nachrichtenbegriff präzise und zugleich allgemeingültig zu definieren.

Nach dem Duden ist eine *Nachricht* „das, wonach man sich zu richten hat". Wir wollen den alltäglichen Begriff der Nachricht nicht definieren, sondern - für den technischen Bereich - lediglich feststellen, was alles zu einer Nachricht gehört, nämlich

1. ein Ort, eine Variable bzw. deren Namen, Index usw.: v
2. ein Zeitpunkt oder Zeitraum t, für den die Nachricht gültig ist
3. der Wert bzw. das Zeichen w aus dem Wertebereich bzw. dem *Zeichenvorrat* W der Variablen

Um die Zeit- und Ortsgebundenheit der Zeichen deutlich zu machen, wird hin und wieder die Funktionsschreibweise verwendet:

$$w = w(v, t) \in W$$

Durch die Verkettung von Zeichen zu *Zeichenfolgen* lassen sich komplexe Nachrichten erzeugen. Die Verkettung geschieht über die Variablen (also *parallel*) oder über die Zeit (also *seriell*). Bei paralleler Darstellung hat man eine Folge von Variablen (z.B. Speicherplätze in einem Rechner, Druckpositionen auf einer Buchseite, Eriks Felder usw.): $v_1, v_2, v_3, \ldots v_N$.

Die einzelnen Zeichen der Folge sind: $w_k = w(v_k, t)$ für ein bestimmtes t und für $k = 1, 2, \ldots, N$. Bei serieller Darstellung nimmt eine Variable v zu bestimmten Zeiten t_k, $k = 1, 2, \ldots, N$, die Werte $w_k = w(v, t_k)$ an.

Die Zeichenfolgen der Länge N werden einfach durch Nebeneinanderschreiben dargestellt:

$$w_1 w_2 w_3 \ldots w_N$$

Diese Zeichenfolgen heißen *Wörter*. Wörter sind selbst wieder Zeichen (Hyperzeichen), die sich erneut verketten lassen. Die neu entstehenden Zeichenvorräte werden von Mal zu Mal umfangreicher. Es entstehen Zeichenhierarchien, wie sie in natürlichen Texten durch Buchstaben oder Laute, Wörter, Sätze, Abschnitte usw. gegeben sind. Die Menge aller Zeichenfolgen der Länge N, deren Zeichen der Menge W entnommen sind, wird mit W^N bezeichnet.

Die Mächtigkeit einer Menge M - also die Anzahl ihrer Elemente - bezeichnen wir mit $|M|$. Offenbar gilt

$$|W^N| = |W|^N$$

Beispiel: Mit A sei ein vierelementiger Zeichenvorrat bezeichnet

$$A = \{a, b, c, d\}.$$

Zeichenfolgen der Länge 2 sind aa, ab, ac, ..._Zeichenfolgen der Länge 3 sind aaa, aab, aac, ... Sie bilden die Zeichenvorräte A^2 und A^3:

$A^2 = \{aa, ab, ac, ad, ba, bb, bc, bd, ca, cb, cc, cd, da, db, dc, dd\}$

A^3 = {aaa, aab, aac, aad, aba, abb, ..., ddc, ddd}

Diese Zeichenvorräte besitzen $4^2 = 16$ bzw. $4^3 = 64$ Elemente.

Den kleinsten sinnvollen Zeichenvorrat bezeichnen wir mit *B*. Er enthält zwei Elemente. Das sind die *Elementarzeichen,* auch *Binärzeichen* oder *Bits* (Binary Digits) genannt. Wir führen für die Binärzeichen die Bezeichnungen 0 und 1 ein:

$B = \{0, 1\}$

Die Binärzeichen spielen in der Informatik eine herausragende Rolle. Die Speicherung und Verarbeitung von Binärzeichen kommt mit Schaltern aus, die sich verhältnismäßig einfach realisieren lassen.

Eine Nachricht im hier beschriebenen Sinne hat zunächst keine Bedeutung. Erst durch ihre Verarbeitung erhält die Nachricht einen Sinn. Eine Nachricht, die einen Sachverhalt ausdrückt, einem Zweck dient oder eine Aktion auslöst, nennen wir *Information.*

Codierung

Ein gegebener Zeichenvorrat kann für die Übermittlung, die Verarbeitung oder die Speicherung von Nachrichten bei Benutzung bestimmter Medien und Apparate ungeeignet sein. Die Lösung heißt dann: Wahl eines neuen, dem Zweck besser angepassten Zeichenvorrats.

Dazu ein paar Beispiele: Die Laute der natürlichen Sprache werden zur Speicherung von Mitteilungen auf Papier durch die Buchstaben der Schrift ersetzt. Zur maschinengerechten Darstellung von Nachrichten werden die Binärzeichen gewählt. Zahlen und Texte sind folglich in Binärzeichenfolgen umzuwandeln.

Ein weiteres Problem liegt in der Unzuverlässigkeit von Speichermedien und Übertragungskanälen: Die Information ist in besonders gesicherter Form darzustellen. Andererseits kann aus Aufwandsgründen eine besonders knappe Darstellung erforderlich werden.

In allen diesen Fällen ist also die ursprüngliche Information durch *Codierung* in die geeignetere Darstellung zu überführen.

Einen *Code* definieren wir durch seine *Decodierungsfunktion g*. Das ist eine Abbildung einer Codezeichenmenge *C* in die ursprüngliche Zeichenmenge *U*:

$g: C \rightarrow U$

Jedem Zeichen der Codezeichenmenge $c \in C$ wird also ein Element u des ursprünglichen Zeichenvorrats *U* zugeordnet. In Zeichen:

$u = g(c)$

oder

$c \rightarrow u$

Die Decodierungsfunktion garantiert die eindeutige Decodierbarkeit von Codezeichen. Eineindeutigkeit, also Umkehrbarkeit, wird nicht gefordert, da es sinnvoll sein kann, mehreren Codezeichen ein und dasselbe ursprüngliche Zeichen zuzuordnen. Die Abbildung muss surjektiv (also eine Abbildung von *C auf U*) sein, da man für jedes ursprüngliche Zeichen auch ein Codezeichen braucht:

$g(C) = U$

Wegen der Eindeutigkeit von Abbildungen zieht das die Forderung nach sich, dass die Codezeichenmenge wenigstens so groß wie die ursprüngliche Zeichenmenge sein muss, kurz:

$|U| \leq |C|$

Beispiel: Für die Zeichen der Menge *A* ist wegen $|A| = |B^2|$ folgende Decodierungsfunktion möglich:

$00 \rightarrow a$
$01 \rightarrow b$
$10 \rightarrow c$
$11 \rightarrow d$

Die Informationsdarstellung in Rechnern geschieht vorzugsweise in der Form von Binärzeichenfolgen gleicher Länge. Eine solche Binärzeichenfolge nennt man auch *Block*. Dementsprechend spricht man bei solchen Codierungen auch von *Blockcodes*. Die *Wortlänge* in Rechnern beträgt z.B. 8, 16 oder 32 Bits. Ein 8-Bit-Wort heißt *Byte*.

Zahlendarstellung

Eine Möglichkeit der Zahlendarstellung in Rechnern bietet das *binäre Zahlensystem*, also das Stellenwertsystem zur Basis zwei. Die *n*-stellige Bitfolge *c* sei Codezeichen:

$$c = c_{n-1}c_{n-2} \ldots c_1c_0$$

Wird diese Bitfolge als Darstellung einer Zahl im binären Zahlensystem interpretiert, dann ergibt sich ihr Wert folgendermaßen:

$$g(c) = c_{n-1}\cdot 2^{n-1} + c_{n-2}\cdot 2^{n-2} + \ldots + c_1\cdot 2^1 + c_0\cdot 2^0 = u$$

Wobei wir davon ausgehen, dass die ursprünglichen Zeichen *u* im Dezimalsystem dargestellt wird. Das binäre Zahlensystem wird auch *Dualsystem* genannt.

Mit 16-Bit-Wörtern lassen sich die ganzen Zahlen von 0 bis 65 535 darstellen.

Übung: Stellen Sie die Zahlen 1994 und 1875 im binären Zahlensystem dar und subtrahieren Sie in diesem System beide Zahlen voneinander. Stellen Sie das Ergebnis im Dezimalsystem dar.

Weitere Zahlendarstellungen in Rechnern: Für negative Zahlen gibt es mehrere Darstellungmöglichkeiten; eine davon ist die Einführung eines zusätzlichen Bits für das Vorzeichen. Festpunktzahlen unterscheiden sich rechnerintern von den ganzen Zahlen nur dadurch, dass ein konstanter Faktor vereinbart ist, mit dem die ganzen Zahlen zu multiplizieren sind, damit sich der Festpunktwert ergibt. Gleitpunktzahlen setzen sich aus zwei Zahlen zusammen. Eine (die Mantisse) ist eine Festpunktzahl, die andere (der Exponent) ist eine ganze Zahl.

Mit diesem Überblick verlassen wir die rechnerinterne Zahlenwelt und wenden uns den Schnittstellen zwischen dem Rechner und seiner Umwelt zu.

Bei Zahleneingabe über die Tastatur ist es offenbar sinnvoll, in einem Zwischenschritt die Dezimalzahlen zunächst einmal ziffernweise binär zu codieren. Dafür braucht man vierstellige Codewörter. Wählt man das bereits oben erwähnte Stellenwertsystem, so erhält man den viel verwendeten *BCD-Code* (BCD steht für „Binary Coded Decimal“). Diese Codierung taucht auch in den - anschließend zu besprechenden - *Codetabellen* auf. Die Zahl 253 (dezimal) sieht in BCD-Schreibweise so aus: 001001010011.

Das Stellenwertsystem eignet sich nicht für die Codierung stetig veränderlicher Größen. Am Beispiel der Längenmessung wird das deutlich: Die Position eines Schiebers werde durch Schleifkontakte erfasst, die ein Band abtasten, das mit elektrisch leitendem Material beschichtet ist. Das Muster der Beschichtung in Querrichtung entspricht dem binär codierten Zahlenwert der jeweiligen Position des Schiebers (**Bild 1.2-1**).

Wird das Muster durch das Stellenwertsystem festgelegt, entsteht folgendes Problem: Steht der Schieber auf Position 7, greift er bei vierstelliger Codierung folgendes Muster ab: 0111; auf Position 8 ist es das Muster 1000. Bei kontinuierlichem Übergang von Position 7 nach Postition 8 kann es passieren, dass vorübergehend die Kontakte zur 7. Position noch bestehen und die zur nächsten Position schon hergestellt sind. Das kann dazu führen, dass irrtümlich das Binärmuster 1111 erfasst wird. Es ergibt sich also der völlig falsche Wert 15.

An einen Code für die digitale Erfassung kontinuierlicher Größen wird deshalb die Anforderung gestellt, dass sich die binären Muster zwischen benachbarten Positionen nur in einer Stelle unterscheiden. Das leisten die *Gray-Codes*. **Bild 1.2-2** zeigt einen vierstelligen. Er wurde bereits in Bild 1.2-1 verwendet.

Übung: Konstruieren Sie einen fünfstelligen Gray-Code. Beschreiben Sie das allgemeine Schema zu Konstruktion mehrstelliger Gray-Codes. (Das hier nahe gelegte Verfahren ist nicht das einzige. Es gibt viele verschiedene Gray-Codes mit vorgegebener Stellenzahl.)

Für die abgekürzte Schreibweise von Dualzahlen oder auch beliebige anders zu interpretierende Bitfolgen sind im Programmieralltag noch das Oktalsystem und das Hexadezimalsystem wichtig. Beim Oktalsystem werden Abschnitte aus je 3 Bits in die Ziffern 0, 1, ..., 7 umgewandelt. Beim Hexadezimalsystem gehen Abschnitte aus je vier Bits in die Zeichen 0, 1, 2,..., 9, A, B,..., F über.

Die Codierungsvorschriften sind der **Tafel 1.2-1** zu entnehmen.

Codetabellen

Für die Umwandlung von Text in maschinengerechte Binärzeichenfolgen gibt es national und international genormte Codetabellen. Für den Computeranwender besonders wichtig ist der *ASCII-Code* (ASCII steht für American Standard Code for Information Interchange). Er umfasst 128 Zeichen und erfordert demzufolge 7-stellige Bitfolgen für die Codierung.

Es ist allgemein üblich, diese Bitfolgen als (im Stellenwertsystem) binär codierte Zahlen zu interpretieren und die Zahlen in Dezimalschreibweise anzugeben. In dieser Form ist der Code in der **Tafel 1.2-2** definiert. Die ASCII-Code-Tabelle entspricht im wesentlichen dem international vereinbarten 7-Bit-Code (Internationales Alphabet Nr. 5). Die deutsche Norm DIN 66 003 enthält außerdem die nationale Version.

Die ersten 32 Zeichen (Nr. 0 bis Nr. 31) sind *Steuerzeichen*. Zu den Formatsteuerzeichen gehören BS (Backspace, Rückwärtsschritt), LF (Line Feed, Zeilenvorschub) und CR (Carriage Return, Wagenrücklauf). Außerdem gibt es Datenübertragungssteuerzeichen, Gerätesteuerzeichen, Informationstrennzeichen und Zeichen für Codeerweiterungen. Außer diesen Steuerzeichen und den 94 darstellbaren Zeichen (Nr. 33 bis 126) erscheinen in der Tabelle noch der Zwischenraum SP (Space, Nr. 32) sowie das Löschzeichen DEL (Delete, Nr. 127).

Beispiel: Gegeben sei eine Folge von Codewörtern des 7-Bit-Codes: „0110101 0110000 0100000 1001010 1100001 1101000 1110010 1100101“. In Dezimalschreibweise lautet die Folge „53 48 32 74 97 104 114 101“. Die Codetabelle liefert den Klartext: „50 Jahre“.

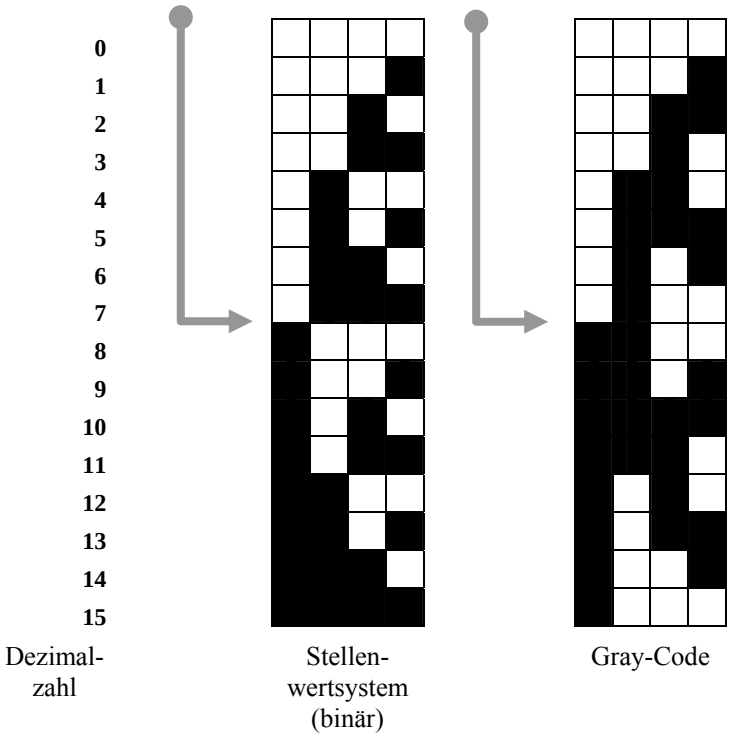


Bild 1.2-1 Längenmessung

```
0000 →  0
0001 →  1
0011 →  2
0010 →  3
0110 →  4
0111 →  5
0101 →  6
0100 →  7
1100 →  8
1101 →  9
1111 → 10
1110 → 11
1010 → 12
1011 → 13
1001 → 14
1000 → 15
```

Bild 1.2-2 Ein vierstelliger Gray-Code

Tafel 1.2-1: Zahlendarstellungen

Dual	Oktal	Dezimal	Hexadezimal
00000	0	0	0
00001	1	1	1
00010	2	2	2
00011	3	3	3
00100	4	4	4
00101	5	5	5
00110	6	6	6
00111	7	7	7
01000	10	8	8
01001	11	9	9
01010	12	10	A
01011	13	11	B
01100	14	12	C
01101	15	13	D
01110	16	14	E
01111	17	15	F
10000	20	16	10
10001	21	17	11
10010	22	18	12
10011	23	19	13
10100	24	20	14

Literaturhinweise

Diese Lektion folgt der Linie meines Buches Codierungsverfahren (1986). Dort ist auch die Zweierkomplementdarstellung der negativen Zahlen dargestellt. Wer mehr wissen will, ist mit dem Standardwerk von Knuth (1981, Kapitel 4 „Arithmetic“) bestens bedient. Eine deutschsprachige Fassung der wesentlichen Dinge bieten Blieberger, Schildt, Schmid und Stöckler (1990, Abschnitt 5 „Zahlendarstellungen“). Im Taschenbuch der Mathematik (Bronstein u. a., 1993) findet man eine kurze Zusammenfassung der Zahlendarstellungen mit Hinweisen auf die Normung.

Tafel 1.2-2 ASCII-Codetabelle (7-Bit-Code)

(Die acht in der deutschen Version des 7-Bit-Codes abweichenden Zeichen sind durch Komma getrennt an zweiter Stelle aufgeführt)

Dezimalzahl	Zeichen	Dezimalzahl	Zeichen	Dezimalzahl	Zeichen	Dezimalzahl	Zeichen
000	NUL	032	SP	064	@, §	096	`
001	SOH	033	!	065	A	097	a
002	STX	034	"	066	B	098	b
003	ETX	035	#	067	C	099	c
004	EOT	036	$	068	D	100	d
005	ENQ	037	%	069	E	101	e
006	ACK	038	&	070	F	102	f
007	BEL	039	'	071	G	103	g
008	BS	040	(	072	H	104	h
009	HT	041	)	073	I	105	i
010	LF	042	*	074	J	106	j
011	VT	043	+	075	K	107	k
012	FF	044	,	076	L	108	l
013	CR	045	-	077	M	109	m
014	SO	046	.	078	N	110	n
015	SI	047	/	079	O	111	o
016	DLE	048	0	080	P	112	p
017	DC1	049	1	081	Q	113	q
018	DC2	050	2	082	R	114	r
019	DC3	051	3	083	S	115	s
020	DC4	052	4	084	T	116	t
021	NAK	053	5	085	U	117	u
022	SYN	054	6	086	V	118	v
023	ETB	055	7	087	W	119	w
024	CAN	056	8	088	X	120	x
025	EM	057	9	089	Y	121	y
026	SUB	058	:	090	Z	122	z
027	ESC	059	;	091	[, Ä	123	{. ä
028	FS	060	<	092	\, Ö	124	|, ö
029	GS	061	=	093	], Ü	125	}, ü
030	RS	062	>	094	^	126	~, ß
031	US	063	?	095	_	127	DEL

1.3 Computerarithmetik

Probleme der Computerarithmetik. Exzessdarstellung ganzer Zahlen. Fest- und Gleitpunktdarstellung reeller Zahlen. Gleitpunktrechnung. Genauigkeitsabschätzungen.

Rechnerintern steht für die Darstellung von Zahlen nur eine begrenzte Anzahl von Stellen (Bitpositionen) zur Verfügung. Es kann zu überraschend großen *Rundungsfehlern* kommen. Die Rechenregeln der Arithmetik dürfen für die *Computerarithmetik* nicht mehr als allgemein gültig vorausgesetzt werden.

Beispiel 1: Der Rechner - auch der Taschenrechner - wird für den Ausdruck $a+b-a$ kaum das korrekte Ergebnis liefern, wenn a gegenüber b sehr groß ist. Setzen wir beispielsweise $a = 10^{34}$ und $b = -2$, dann erhält man im Normalfall den falschen Wert 0 und nicht -2, wie es richtig wäre.

Beispiel 2: Wenn man versucht, mit dem PC oder auch mit dem Taschenrechner, den Wert des Ausdrucks $9x^4 - y^4 + 2y^2$ für $x = 10\,864$ und $y = 18\,817$ zu berechnen, wird man im allgemeinen nicht den korrekten Wert 1 erhalten. Mein Taschenrechner beispielsweise liegt mit seinem Ergebnis von 1 158 978 ziemlich daneben.

Schlechte Erfahrungen mit der Computerarithmetik führen dazu, dass aus dem computergläubigen Anfänger ein Skeptiker wird: Der Anfänger glaubt dem Computer alles; der Skeptiker traut ihm grundsätzlich nicht.

Um nicht einem der beiden extremen Standpunkte zu verfallen - der erste ist gefährlich und der zweite hinderlich - bleibt uns nichts anderes übrig, als die Computerarithmetik genauer zu studieren.

Letztlich ist nur durch das Kennenlernen der fehlerträchtigen Situationen und gegebenenfalls das Abschätzen der Rechen- und Rundungsfehler den drohenden Fallen zu entkommen. Tatsächlich sollte unsere Einstellung gegenüber dem Computer stets ein gesundes Misstrauen sein. Genau darum geht es in dieser Lektion.

Exzessdarstellung ganzer Zahlen

Die Dualzahlen sind die Basis für sämtliche Zahlendarstellungen im Computer. Dualzahlen stehen für natürliche Zahlen.

Will man negative Zahlen darstellen, muss man die Zahlendarstellung erweitern. Dafür gibt es mehrere Möglichkeiten. Eine davon ist die *Exzessdarstellung*.

Mit der Bitfolge $c_{n-1}c_{n-2} \ldots c_1c_0$ lassen sich alle Zahlen von 0 bis 2^n-1 darstellen - genausogut aber auch die Zahlen von -2^{n-1} bis $2^{n-1}-1$. Im letzten Fall haben wir einfach von jeder Dualzahl den Wert 2^{n-1} subtrahiert. Wir setzen $q = 2^{n-1}$ und bezeichen q als *Exzess*. Die tatsächliche gemeinte Zahl ist also gleich dem Wert, um den die Dualzahl den *Exzess* q überschreitet.

Allgemein lautet die Decodierungsvorschrift für die *Exzessdarstellung* ganzer Zahlen mit der *Stellenzahl* n, der *Basis* b und dem *Exzess* q:

$$u = g(c) = c_{n-1}\cdot b^{n-1} + c_{n-2}\cdot b^{n-2} + \ldots + c_1\cdot b^1 + c_0\cdot b^0 - q$$

Wählt man speziell $b = 2$ und $q = 2^{n-1}$, dann erhält man

$$u = g(c) = (c_{n-1}-1)\cdot 2^{n-1} + c_{n-2}\cdot 2^{n-2} + \ldots + c_1\cdot 2^1 + c_0\cdot 2^0$$

Bei den nichtnegativen Zahlen ist das höchstwertige Bit, nämlich c_{n-1}, gleich 1 und bei den negativen Zahlen gleich 0.

Übung 1: Gesucht ist die Zahl -17 in der Exzessdarstellung mit der Basis 2 und dem Exzess 32. Führen Sie die Codierung durch und zur Kontrolle auch die Decodierung.

Die ganzen Zahlen bilden einen *kommutativen Ring mit Einselement:* Addition und Multiplikation sind einschränkungslos durchführbar (Abgeschlossenheit). Die 0 ist neutral hinsichtlich der Addition. Die 1 ist neutral hinsichtlich der Multiplikation. Es gelten die Assoziativ- und Kommutativgesetze für Addition und Multiplikation, ferner die Distributivgesetze. Und zu jedem Element a gibt es ein Inverses bezüglich der Addition: $-a$.

Übung 2: Für die Exzessdarstellung von Zahlen gelten einige der Ringgesetze nicht mehr. Welche sind das? Wie sieht es mit der Abgeschlossenheit aus? Gibt es zu jedem Element ein Inverses bezüglich der Addition?

Festpunktdarstellung reeller Zahlen

Für die *Festpunktdarstellung* von Zahlen im Stellenwertsystem zur Basis b wird eine feste Zahl t von Stellen hinter dem Punkt fest vereinbart (Mantissenlänge). Im Dezimalsystem mit zwei Stellen nach dem Dezimalpunkt hat man beispielsweise $3.14 = 314/100 = 314 \cdot 10^{-2}$.

Eine im Computer gespeicherte Bitfolge wird als Dualzahl (eventuell mit vorangestelltem Vorzeichen) interpretiert und zur Gewinnung des wahren Wertes noch mit 2^{-t} multipliziert. Auf diese Weise lassen sich die Dualzahlen mit t Nachpunktstellen darstellen.

Beispiel 3: Wir führen die Zahlenwandlung der Zahl $(0.1)_{10}$ in die Darstellung zur Basis zwei durch: $0.1 = 0.1 \cdot 2^0 = 0.2 \cdot 2^{-1} = 0.4 \cdot 2^{-2} = 0.8 \cdot 2^{-3} = 1.6 \cdot 2^{-4} = 3.2 \cdot 2^{-5} = ((11)_2 + (0.2)_{10}) \cdot 2^{-5}$. Die Dezimalzahl 0.2 lässt sich nach demselben Schema entwickeln: $0.2 = ((11)_2 + 0.2) \cdot 2^{-4}$. Und das wiederholt sich nun. Damit ist die periodische Darstellung gefunden: $(0.1)_{10} = (0.00011\,\overline{0011})_2$. Die Zahl wird auf 20 Nachpunktstellen gerundet. Das Ergebnis: $(0.1)_{10} \approx (0.00011001100110011010)_2 = (00011001100110011010)_2 \cdot 2^{-20}$.

Gleitpunktdarstellung reeller Zahlen

Die *Gleitpunktdarstellung* reeller Zahlen ist von zentraler Bedeutung für die Computeranwendungen in Wissenschaft und Technik - und das vor allem wegen des praktisch unbegrenzten Wertebereichs. Ein Beispiel ist die Gleitpunktdarstellung der Lichtgeschwindigkeit: $2.99792458 \cdot 10^8$ m/s.

Die Gleitpunktdarstellung einer Zahlen ist ein Paar von Zahlen (e, m). Dabei ist e eine ganze Zahl, und m ist eine (vorzeichenbehaftete) Festpunktzahl mit t Stellen nach dem Punkt. Beiden Zahlendarstellungen wird das Stellenwertsystem zur Basis b zu Grunde gelegt. Und e möge in Exzessdarstellung mit dem Exzess q vorliegen. Rechnerintern wird eine solche Zahl folgendermaßen gespeichert:

±	$e+q$	$\lvert m\rvert \cdot b^t$

Im Fall der Basis 2 enthält das erste Feld ein Bit für das Vorzeichen (0 für positive und 1 für negative Zahlen) und die übrigen Felder beinhalten Dualzahlen.

Der Wert der Gleitpunktzahl (e, m) ist gegeben durch

$$z = m \cdot b^e$$

Durch geeignete Wahl des Exponenten lässt sich bei Zahlen ungleich null erreichen, dass

$$b^{-1} = 1/b \le |m| < 1 \qquad (*)$$

gilt. In diesem Fall besitzt $|m| \cdot b^t$ die Darstellung $(c_{t-1}c_{t-2} \ldots c_1c_0)_b$ mit c_{t-1} ungleich null. Gleitpunktdarstellungen, die die Bedingung (*) erfüllen, heißen *normalisiert*. Die Zahl m wird *Mantisse* genannt; e ist der *Exponent*.

Beispiel 4: Gesucht ist für die Zahl $(5.75)_{10}$ die normalisierte Gleitpunktdarstellung zur Basis 2, mit dem Exzess 16 und mit 6 Stellen nach dem Punkt ($b = 2$, $q = 16$ und $t = 6$). Die einzelnen Umformungsschritte:

$5.75 = 11.5 \cdot 2^{-1}$ //Verdoppeln der Dezimalzahl

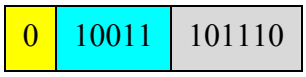
$= 23 \cdot 2^{-2}$ //bis kein Dezimalbruch mehr erscheint

$= (10111)_2 \cdot 2^{-2}$ //Dezimalzahl -> Dualzahl

$= (0.10111)_2 \cdot 2^3$ //Normalisierung

$= (0.101110)_2 \cdot 2^{19-16}$ //Exzessdarstellung

Da $19 = (10011)_2$ haben wir folgende computerinterne Zahlendarstellung.



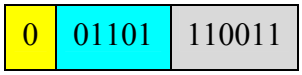
Die Zahl des letzten Beispiels erlaubt eine exakte (gebrochene) Darstellung im Binärsystem. Deshalb ging die Zahlenwandlung so problemlos. Aber schon die Zahl 0.1 lässt sich nicht mehr exakt als Festpunktzahl mit endlicher Stellenzahl darstellen.

Beispiel 5: Gesucht ist für die Zahl $(0.1)_{10}$ die normalisierte Gleitpunktdarstellung mit $b = 2$, $q = 16$ und $t = 6$. Durch das wiederholte Multiplizieren der Mantisse mit 2 und entsprechender Korrektur des Exponenten werden wir die Nachpunktstellen im Dezimalsystem nicht los: $0.1 = 0.2 \cdot 2^{-1} = 0.4 \cdot 2^{-2} = 0.8 \cdot 2^{-3} = 1.6 \cdot 2^{-4} = 3.2 \cdot 2^{-5} = \ldots$ Ab jetzt wiederholt sich die Sache: Hinter dem Dezimalpunkt erscheinen nacheinander die Ziffern 2, 4, 8, 6, 2, 4, 8, 6, 2, 4, 8, 6, ...

Man kann auf einfache Weise zu einer möglichst genauen Zahlendarstellung im Binärsystem kommen:

1. Man macht sich klar, in welche Zahlenbereich man die ganze Zahl vor dem Dezimalpunkt bringen muss. Zu guter Letzt hat die gewandelte Zahl - mit den Platzhaltern x und y für die Binärzeichen 0 und 1 - die Gestalt $\pm (0.1\text{xxxxx})_2 \cdot 2^{(\text{yyyyy})_2 - (16)_{10}}$.
2. Die Mantisse wird als ganze Zahl dargestellt: $(1\text{xxxxx})_2$. Sie muss also zwischen $(100000)_2 = 32$ und $(111111)_2 = 63$ liegen. Wir müssen die oben begonnene Folge fortsetzen: $0.1 = 3.2 \cdot 2^{-5} = 6.4 \cdot 2^{-6} = 12.8 \cdot 2^{-7} = 25.6 \cdot 2^{-8} = 51.2 \cdot 2^{-9} \approx 51 \cdot 2^{-9}$. Die gerundete Mantisse liegt im gewünschten Zahlenbereich.
3. Wandlung aus der Dezimaldarstellung in die Binärdarstellung: $51 \cdot 2^{-9} = (110011)_2 \cdot 2^{-9}$ $= (0.110011)_2 \cdot 2^{-3} = (0.110011)_2 \cdot 2^{13-16} = (0.110011)_2 \cdot 2^{(01101)_2 - (16)_{10}}$.

4. Speicherung:



Für Gleitpunktzahlen gelten einige der Rechenregeln nicht, die wir von den reellen Zahlen gewohnt sind (Körpergesetze). Weiterhin ist zu beachten:

1. Gleitpunktzahlen liegen nicht dicht wie die reellen Zahlen. Bei der näherungsweisen Darstellung reeller Zahlen durch Gleitpunktzahlen muss mit einem *Rundungsfehler* gerechnet werden.
2. *Bereichsüberschreitungen:* Für sehr große Zahlen z existiert keine Gleitpunktnäherung. Dieser Fall wird als *Überlauf* (englisch: *Overflow*) bezeichnet. Auch sehr kleine Zahlen ungleich null lassen sich nicht als Gleitpunktzahl repräsentieren. Hier spricht man von *Unterlauf* (englisch: *Underflow*).

Diese beiden Effekte führen dazu, dass für die Gleitpunktarithmetik die Rechenregeln der üblichen Arithmetik nicht alle gelten. Insbesondere das Assoziativgesetz wird immer wieder verletzt. Gefahr droht besonders dann, wenn nahezu gleich große Zahlen voneinander abzuziehen sind. Als Gegenmaßnahme sollte man Ausdrücke unter Ausnutzung der Rechenregeln der Arithmetik geeignet umformen, bevor man Sie dem Rechner überantwortet.

Übung 3: Überlegen Sie, was in Situationen wie in Beispiel 1 und 2 zu tun wäre.

Dass beim zweiten Beispiel der Wert 1 herauskommen muss, habe ich mir ohne allzuviel Rechnerei so klar gemacht: Gefahr droht, wenn $9x^4$ und y^4 - bzw. $3x^2$ und y^2 - halbwegs gleich groß sind. Tatsächlich gilt $y^2 = 3x^2 + 1$. Diesen Nachweis führt man sicherheitshalber von Hand und überprüft ihn mittels Taschenrechner oder PC. Der Rest ergibt sich durch Einsetzen und einfache Umformungen.

Übung 4: Stellen Sie die Zahl $\pi = 3.1415927...$ im Gleitpunktformat wie in Beispiel 5 dar ($b = 2$, $q = 16$ und $t = 6$). Achten Sie darauf, wann und wie zu runden ist, so dass eine möglichst genaue Zahlendarstellung gewährleistet ist.

Die IEEE-Formate zur Darstellung von Gleitpunktzahlen

Die IEEE-Datenformate für Gleitpunktzahlen unterscheiden sich von der oben besprochenen computerinternen Darstellung dadurch, dass die normalisierte Form der Mantisse die Darstellung $(1.xx...x)_2$ besitzt und dass die Eins vor dem Dezimalpunkt nicht mit abgespeichert wird. Die Gleitpunktzahlen vom Typ `float` bzw. `double` haben eine Gesamtlänge von 32 bzw. 64 Bits. Für den Exponenten werden im ersten Fall 8 und im zweiten Fall 11 Bits verwendet. Der Exzess q hat den Wert 127 bzw. 1023. Da noch ein Bit für das Vorzeichen abgeht, bleiben für die Mantisse 23 bzw. 52 Bits übrig. Die jeweils kleinsten Exponenten (-127 bzw. -1023) sind für die Sonderdarstellung besonders kleiner Zahlen reserviert (in der Exzessdarstellung werden diese Exponenten durch Nullen repräsentiert). Die Zahl Null selbst wird durch die Mantisse $(1.00...0)_2$ und das Vorzeichenbit 0 repräsentiert. In diesem Fall bestehen sämtliche Bits der Zahlendarstellung aus Nullen. Der größtmögliche Exponent (128 bzw. 1024) ist für die Zahlen $+\infty$ und $-\infty$ sowie für Fehlermeldungen reserviert. Im Ausdruck liest man dann manchmal NaN (Not a Number). Weitere Festlegungen sind im ANSI/IEEE Standard 745-1985 zu finden (s. a. Capper, 1994, S. 399 ff.).

Abschätzung des Rundungsfehlers

Für die Praxis ist die Abschätzung des *Rundungsfehlers* bei Gleitpunktdarstellungen bedeutsam. Wir schätzen zunächst den Abstand benachbarter normalisierter Gleitpunktzahlen ab. Sei z eine beliebige normalisierte Gleitpunktzahl mit der Stellenzahl t:

$$z = m \cdot b^e$$

Der Abstand δ der nächstliegenden Gleitpunktzahlen von diesem Wert ist (höchstens) gleich $b^{-t} \cdot b^e$. Der relative Abstand $\delta/|z|$ lässt sich folgendermaßen abschätzen:

$$\delta/|z| \le b^{-t} \cdot b^e / (|m| \cdot b^e) = b^{-t}/|m| \le b^{-t}/b^{-1} = b^{1-t}$$

Die ober Grenze für den relativen Abstand von Gleitpunktzahlen ist nicht von deren Größe abhängig, sondern nur von der Basis und der Anzahl von Stellen nach dem Punkt.

Beim üblichen Rundungsverfahren ist $\frac{1}{2} \cdot b^{1-t}$ eine obere *Schranke für den relativen Rundungsfehler*. Sei rd(z) die Gleitpunktzahl, die durch die Rundung aus der reellen Zahl z entsteht, so gilt

$$|\mathrm{rd}(z) - z| / |\mathrm{rd}(z)| \le \tfrac{1}{2} \cdot b^{1-t}$$

Übung 5: Wie groß ist der relative Rundungsfehler bei den IEEE-Gleitpunktformaten? *Hinweis:* Da die führende 1 der Mantisse nicht mit abgespeichert wird, ist die Mantissenlänge t um eins größer die Anzahl der Nachpunktstellen.

Iterationsverfahren zur Lösung von Gleichungen

In der numerischen Mathematik findet man häufig *Iterationsverfahren* folgender Art: Die Lösung x der Gleichung $x = f(x)$ heißt *Fixpunkt* von f. Unter bestimmten Bedingungen und bei geeignet gewähltem Anfangswert x_0 strebt die Folge der Werte x_0, x_1, x_2, ..., die nach der *Rekursionsformel*

$$x_{k+1} = f(x_k)$$

berechnet werden, gegen den Fixpunkt.

Die bisherigen Überlegungen legen nahe, als *Abbruchbedingung* für diese Iteration eine *relative Genauigkeit* vorzugeben:

$$|x_{k+1} - x_k| \le \varepsilon \cdot |x_k|$$

Da bei der Berechnung von f und zusätzlich in der Abbruchbedingung Rundungen auftreten, muss die relative Genauigkeit ε so gewählt sein, dass sie ein Vielfaches der Schranke für den relativen Rundungsfehler ist. Ansonsten kann es passieren, dass das Iterationsverfahren nie endet, obwohl bei exakter Rechnung der Fixpunkt schnell erreicht wäre.

Literaturhinweise

Der Abschnitt 5.10 des Buches von Blieberger, Schildt, Schmid und Stöckler (1990) behandelt die Darstellung reeller Zahlen. Eine gründliche Darstellung der Computerarithmetik ist im zweiten Band des Werkes von Knuth (Band II, 1981, S. 198 ff.) zu finden. In diesen Büchern findet man auch Hinweise auf fehlerträchtige Situationen. Der Informatik-Duden behandelt den Gegenstand unter den Stichwörtern „Gleitpunktdarstellung“ und „Rundung“. Für diese Lektion wurden außerdem die folgenden Bücher und Aufsätze benutzt:

Alex, J.; Flessner, H.; Mons, W.; Pauli, K.; Zuse, H.: Konrad Zuse. Der Vater des Computers. Parzeller, Fulda 2000

Capper, D. M.: Introducing C++ for scientists, engineers and mathematicians. Springer, London 1994, S. 399 ff.

Grams, T.: Denkfallen und Programmierfehler. Springer, Berlin, Heidelberg 1990

Rojas, R. (Hrsg.): Die Rechenmaschinen von Konrad Zuse. Springer, Berlin, Heidelberg 1998

Rump, S. M.: Wie zuverlässig sind unsere Rechenanlagen? Jahrbuch Überblicke Mathematik 1983. Bibliographisches Institut Mannheim 1983

Schwetlick, H.; Kretzschmer, H.: Numerische Verfahren für Naturwissenschaftler und Ingenieure - Eine computerorientierte Einführung. Fachbuchverlag Leipzig, 1991

Das Buch von Schwetlick und Kretzschmer enthält eine weitere ausführliche Darstellung der Computerarithmetik. Der Aufsatz vom Rump bringt vor allem eine Zusammenstellung von Beispielen fehlerhafter Computerarithmetik. Mein Buch behandelt die Denkmechanismen, die zu Programmierfehlern führen und geht auch auf die Computerarithmetik ein.

Das Problem der Umwandlung von Dezimalbrüchen in die binäre Festpunktdarstellung wird von Gries und Knuth in deren Beiträgen zum Buch „Beauty is Our Business“ (Herausgeber: W. H. J. Feijen u. a.; Springer-Verlag, New York 1990) in einiger Tiefe behandelt.

In der Würdigung Zuses als Erfinder des Computers erwähnt F. L. Bauer ganz besonders die Zahlendarstellung: Zuse ist der „Schöpfer der ersten vollautomatischen, programmgesteuerten und frei programmierten, in binärer Gleitpunktrechnung arbeitenden Rechenanlage. Sie war 1941 betriebsfähig“. Weitere Informationen darüber enthalten die Bücher von Alex/Flessner/Mons/Pauli und Rojas. Insbesondere das Buch von Rojas enthält eine ausführliche Darstellung der internen binären Zahlendarstellung, die Ein- und Ausgabe als Dezimalzahlen sowie die Architektur zur Realisierung der Rechenoperationen.

1.4 Boolesche Algebra und Verknüpfungsglieder

Grundlagen der Nachrichtenverarbeitung. Boolesche Algebra. Schaltfunktionen und Gatter. Schalter. Blockdiagramme.

Die elektronische Datenverarbeitung basiert darauf, dass man Nachrichten in Form von endlichen Bitfolgen codieren kann, dass man Nachrichten zu neuen Nachrichten verknüpfen, und dass man Nachrichten festhalten - also speichern - kann.

Die Codierung war Thema des letzten Abschnitts. In dieser und den folgenden Lektionen geht es um die Verknüpfung und die Speicherung von Nachrichten. Mathematische Grundlage ist die so genannte *Boolesche Algebra.*

Unter einer *Booleschen Funktion* oder *Schaltfunktion f* mit *n* unabhängigen Variablen verstehen wir eine Abbildung der Art

$f: B^n \rightarrow B$

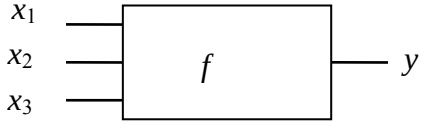
Eine solche Schaltfunktion nennen wir kurz *n-stellig*. Sie lässt sich durch eine *Funktionstabelle* definieren. Jede Zeile enthält genau eine der 2^n möglichen Wertebelegungen der unabhängigen Variablen sowie den zugehörigen Funktionswert. Die Zahl *m* der Zeilen der Funktionstabelle ist gegeben durch $m = 2^n$. Die Zahl aller möglichen *n*-stelligen Funktionen ist 2^m.

Der Definitionsbereich der Schaltfunktion ist gleich B^n, also *n*-dimensional. Bei allgemeinen Verknüpfungsoperationen ist auch der Bildbereich mehrdimensional. Aber das macht die Angelegenheit nicht komplizierter, weil jede Komponente der mehrdimensionalen Funktion für sich betrachtet eine Schaltfunktion im obigen Sinne - also eindimensional - ist. Wir können uns also auf das Studium der (eindimensionalen) Schaltfunktionen beschränken.

Die Funktionstabelle und das allgemeine Schaltzeichen einer Schaltfunktion mit drei unabhängigen Variablen $y = f(x_1, x_2, x_3)$ zeigt **Tafel 1.3-1**. Das Schaltzeichen wird als Blockdiagramm dargestellt.

Ein- und zweistellige Schaltfunktionen bezeichnet man auch als *Verknüpfungen (Junktoren). Elementare Verknüpfungsglieder* sind Realisierungen für Schaltfunktionen mit einer bzw. zwei unabhängigen Variablen. Die elementaren Verknüpfungsglieder nennt man auch *Gatter*. Es gibt 4 verschieden einstellige und 16 zweistellige Schaltfunktionen. Die wichtigsten elementaren Verknüpfungsglieder werden mit ihren Schaltzeichen in der **Tafel 1.3-2** definiert.

Tafel 1.3-1 Funktionstabelle und Blockdiagramm einer Schaltfunktion



$y = f(x_1, x_2, x_3)$

x_1	x_2	x_3	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Für die *Negation* NOT(x) schreiben wir ¬x. Bei den zweistelligen Funktionen gehen wir zur so genannten *Infixnotation* über: Anstelle von AND(x_1, x_2), das ist die *Konjunktion*, steht

x_1 AND x_2 bzw. $x_1 \wedge x_2$, und anstelle von OR(x_1, x_2), das ist die *Disjunktion*, heißt es x_1 OR x_2 bzw. $x_1 \vee x_2$.

Mit den Grundoperationen NOT, AND und OR lassen sich alle ein- und zweistelligen Verknüpfungen darstellen. (Dieser Satz wird - in verallgemeinerter Form - in der folgenden Lektion bewiesen.) Für NAND und NOR beispielsweise sieht das so aus:

$$\text{NAND}(x_1, x_2) = \neg(x_1 \wedge x_2)$$

$$\text{NOR}(x_1, x_2) = \neg(x_1 \vee x_2)$$

Weitere wichtige zweistellige Verknüpfungen sind die *Äquivalenz* (EQUIV), die *Antivalenz* (exklusives Oder, XOR) und die *Implikation* (IMP). Sie werden hier über die Grundopertionen definiert:

$$\text{EQUIV}(x_1, x_2) = (x_1 \wedge x_2) \vee(\neg x_1 \wedge \neg x_2)$$

$$\text{XOR}(x_1, x_2) = (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$$

$$\text{IMP}(x_1, x_2) = \neg x_1 \vee x_2$$

Übung 1: Zeigen Sie, dass sich jede der Grundoperationen NOT, AND und OR allein aus NAND-Gliedern zusammensetzen lässt. Zeigen Sie ferner, dass auch das NOR-Glied ausreicht (Wendt, 1982, S. 20). Zeichnen Sie die zugehörigen Schaltbilder (Blockdiagramme).

Die NAND-Verknüpfung heißt auch *Shefferscher Strich* und man schreibt die Verknüpfung in der Form $y = x_1 \mid x_2$. Die NOR-Verknüpfung wird auch *Peirce-Operator* genannt.

Die elementaren Schaltfunktionen und daraus abgeleitete häufig benötigte Schaltkreise stehen in verschiedenen Halbleitertechnologien als integrierte Schaltkreise zur Verfügung: Die wichtigsten sind TTL (Transistor Transistor Logic), ECL (Emitter Coupled Logic) und CMOS (Complementary Metal Oxide Semiconductor).

Für das Rechnen mit Verknüpfungen gelten die Rechenregeln (Gesetze) der *Booleschen Algebra*, **Tafel 1.3-3**. Am besten macht man sich die Gesetze anhand von Funktionstabellen klar: Die Funktionen auf der linken und auf der rechten Seite der Gleichungen müssen jeweils identisch sein.

Die Assoziativgesetze erlauben es, Klammern wegzulassen: Statt $a \wedge (b \wedge c)$ darf man auch $a \wedge b \wedge c$ schreiben, ohne dass Missverständnisse auftreten können; denn egal wie man die Klammern setzt, immer kommt dasselbe Ergebnis zustande. Dasselbe gilt für $a \vee b \vee c$.

Aus den Rechengesetzen der Booleschen Algebra lässt sich ein *Satz zur Auflösung von Gleichungen* herleiten: Aus $a \wedge x = 0$ und $a \vee x = 1$ folgt $x = \neg a$.

Beweis: Dass $x = \neg a$ eine mögliche Lösung der Gleichungen ist, entnimmt man dem 6. Rechengesetz über komplementäre Elemente. Die Eindeutigkeit der Lösung zeigt man so: Sei neben x eine weitere Lösung x' gegeben: $a \wedge x' = 0$ und $a \vee x' = 1$. Dann muss sein[1]:

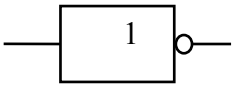
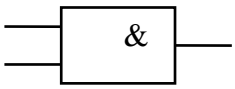
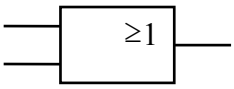
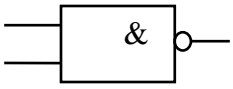
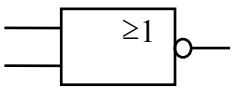
x'	
$= 1 \wedge x'$	//Neutrales Element (5.)
$= x' \wedge 1$	//Kommutativgesetz (1.)
$= x' \wedge (a \vee x)$	//Prämisse
$= (x' \wedge a) \vee (x' \wedge x)$	//Distributivgesetz (3.)
$= (a \wedge x') \vee (x' \wedge x)$	//Kommutativgesetz (1.)

[1] Kommentare stehen hinter dem Doppelstrich

$= 0 \vee (x' \wedge x)$	//Prämisse
$= (a \wedge x) \vee (x' \wedge x)$	//Prämisse
$= (x \wedge a) \vee (x \wedge x')$	//Kommutativgesetz (1.)
$= x \wedge (a \vee x')$	//Distributivgesetz (3.)
$= x \wedge 1$	//Prämisse
$= 1 \wedge x$	//Kommutativgesetz (1.)
$= x$	//Neutrales Element (5.)

Das war zu beweisen.

Tafel 1.3-2 Verknüpfungsglieder: Funktionen und Schaltzeichen

Name	Schaltzeichen nach DIN 40700/14	Funktionstabelle
NOT		x | y 0 | 1 1 | 0
AND		x_1 x_2 | y 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		x_1 x_2 | y 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NAND		x_1 x_2 | y 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		x_1 x_2 | y 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0

Übung 2: Zeigen Sie durch Anwendung der Gesetze der Booleschen Algebra, dass $a \wedge a = a$ und $a \vee a = a$ gilt. *Hinweis:* Wenden Sie die Rechengesetze 5, 1 und 4 an.

Übung 3: Zeigen Sie, dass $0 \wedge a = 0$ und dass $1 \vee a = 1$ ist.

Übung 4: Beweisen Sie das *Gesetz der Verneinung* $\neg(\neg a) = a$ allein mit den Rechengesetzen der Booleschen Algebra. *Hinweis:* Zeigen Sie zunächst, dass $\neg a \wedge a = 0$ und $\neg a \vee a = 1$, und wenden Sie dann den Satz zur Auflösung von Gleichungen an.

Übung 5: Beweisen Sie die *de Morganschen Gesetze*

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\neg(a \vee b) = \neg a \wedge \neg b$$

allein mit den Rechengesetzen der Booleschen Algebra. *Hinweis:* Zeigen Sie mit Hilfe des Distributivgesetzes zunächst, dass $(a \wedge b) \wedge (\neg a \vee \neg b) = 0$ und $(a \wedge b) \vee (\neg a \vee \neg b) = 1$ gilt und wenden Sie dann den Satz zur Auflösung von Gleichungen an.

Übung 6: Zeigen Sie durch Anwendung der Definitionsgleichungen und durch alleinigen Gebrauch der Rechengesetze der Booleschen Algebra, dass XOR(a, b) = NOT(EQUIV(a, b)).

Diese Lektion hat eine sehr technische Sicht der so genannten *Aussagenlogik* geboten. Im zweiten Teil der Veranstaltung werden wir den Blickwinkel ändern und die Sache aus sprachlicher Sicht erneut behandeln. Erstere ist für den Bau der Computer, letztere für deren Programmierung bedeutend.

Literaturhinweise

Der Abschnitt hält sich an die Bücher von Blieberger u. a. (1990, Abschnitt 7), Schiffmann und Schmitz (1993, Abschnitt 4), Wendt (1982, Abschnitt 10).

Hinsichtlich der Bezeichnungen wurde auf das Buch von Hermes zurückgegriffen:

Hermes, H.: Einführung in die mathematische Logik. Teubner, Stuttgart 1991

Eine umfassende Behandlung der Logik aus ingenieurwissenschaftlicher Sicht bietet das Buch von Schneeweiss:

Schneeweiss, W. G.: Boolean Functions - with Engineering Applications and Computer Programs. Springer-Verlag, Berlin, Heidelberg 1989

Tafel 1.3-3 Die Gesetze der Booleschen Algrebra

1. Kommutativgesetze

$a \wedge b = b \wedge a$
$a \vee b = b \vee a$

2. Assoziativgesetze

$(a \wedge b) \wedge c = a \wedge (b \wedge c)$
$(a \vee b) \vee c = a \vee (b \vee c)$

3. Distributivgesetze

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

4. Absorptionsgesetze

$a \wedge (a \vee b) = a$
$a \vee (a \wedge b) = a$

5. Existenz neutraler Elemente

$0 \vee a = a$
$1 \wedge a = a$

6. Existenz komplementärer Elemente

$a \vee \neg a = 1$
$a \wedge \neg a = 0$

1.5 Schaltnetze und deren Minimierung

Normalformen (DNF, KNF), Minimierung, Produktlebensphasen, Entwurf eines Schaltnetzes.

Schaltnetze sind schaltungtechnische Realisierungen von Schaltfunktionen.

Mit den Verknüpfungen AND, OR und NOT kann jede beliebige Schaltfunktion aufgebaut werden. Da sich diese Verknüpfungen wiederum allein mit NAND-Gliedern (oder NOR-Gliedern) realisieren lassen, reicht sogar ein Gatter-Typ zur Realisierung beliebiger Schaltfunktionen aus. Diesen Sachverhalt drückt man kurz so aus: AND, OR und NOT bilden eine *funktional vollständige Verknüpfungsmenge.* Auch das NAND allein ist funktional vollständig. Dasselbe gilt für das NOR allein.

Wie findet man zu einer vorgegebene Schaltfunktion eine Realisierung, die nur AND-, OR- und NOT-Gatter enthält? Das grundsätzliche Vorgehen sei an der Schaltfunktion *f* der Tafel 1.3-1 des letzten Abschnitts erläutert.

Für jede Zeile, die für die Funktion den Ergebniswert 1 aufweist, schreiben wir eine *Vollkonjunktion* hin; das ist eine konjunktive Verknüpfung sämtlicher unabhängigen Variablen. Falls der Wert einer Variablen in der Zeile gleich 1 ist, wird die Variable nicht negiert hingeschrieben, andernfalls negiert. Alle diese Vollkonjunktionen werden disjunktiv verknüpft.

Jede Vollkonjunktion nimmt für genau eine Variablenbelegung den Wert 1 an, nämlich für die Variablenbelegung derjenigen Zeile, für die sie aufgestellt worden ist. Auf diese Weise entsteht folgende Funktionsdarstellung für die Schaltfunktion aus Tafel 1.3-1:

$$f(x_1, x_2, x_3)$$
$$= (\neg x_1 \wedge \neg x_2 \wedge x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge \neg x_3)$$

Eine solche Funktionsdarstellung heißt *kanonische disjunktive Normalform (KDNF).*

Zur *kanonischen konjunktiven Normalform (KKNF)*, kommt man, indem man zunächst die KDNF der negierten Funktion bestimmt:

$$\neg f(x_1, x_2, x_3)$$
$$= (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3)$$
$$\vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge x_3)$$

Dann negiert man diese und wendet eine der de Morganschen Regeln an:

$$f(x_1, x_2, x_3)$$
$$= \neg(\neg f(x_1, x_2, x_3))$$
$$= (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$$
$$\wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$$

Man kann auch - analog zur Erstellung der KDNF - von vornherein *Volldisjunktionen* zur Erzeugung von Nullen hinschreiben und diese konjunktiv verknüpfen.

KDNF und KKNF lösen die Aufgabe der Darstellung von Schaltfunktionen mittels Grundoperationen. Allerdings nicht immer auf die einfachste Weise. Meist sind mehr Verknüpfungen beteiligt als unbedingt nötig.

Je weniger verschiedene Verknüpfungsarten auftreten und je geringer die Gesamtzahl der Verknüpfungen ist, desto einfacher ist die technische Realisierung. Durch geschickt Anwendung der Regeln und Sätze der Booleschen Algebra kann man sich diesem Ziel nähern.

Es gibt systematische Methoden der *Minimierung*. Da sie Gegenstand anderer Lehrveranstaltungen sind, seien die bekanntesten hier nur erwähnt: Das Verfahren von *Quine* und *McCluskey* liefert einen Algorithmus, dessen Abarbeitung man dem Computer überlassen kann. Das Verfahren von *Karnaugh* und *Veitch* ist besonders anschaulich und eignet sich für den „Entwurf von Hand“. Beide Verfahren sind in der angegebenen Literatur zu finden.

Durch Reduzierung der an jedem konjunktiven (bzw. distributiven) Ausdruck beteiligten Variablen bleibt die Grundstruktur der Ausdrücke erhalten. Man spricht dann von einer *Disjunktiven Normalform* (*DNF*) bzw. von einer *konjunktiven Normalform* (*KNF*). Beispielsweise lässt sich die KDNF $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c)$ mit Hilfe der Gesetze der Booleschen Algebra vereinfachen zur DNF $(a \wedge c)$.

Übung: Zeigen Sie unter alleiniger Anwendung der Rechenregeln der Booleschen Algebra, dass tatsächlich gilt: $(a \wedge b \wedge c) \vee (a \wedge \neg b \wedge c) = (a \wedge c)$.

In dieser Lektion soll das Prinzipielle des Entwurfs von Schaltnetzen an einem Beispiel, dem Entwurf eines *Volladdierers*, vorgeführt werden. Bei dieser Gelegenheit ist es angebracht, auch den Gesamtzusammenhang zu beleuchten, in dem der reine Schaltungsentwurf steht. Dazu führe man sich einmal die *Produktlebensphasen* vor Augen:

1. *Anforderungserfassung:* Ergebnis ist die verbale Beschreibung des Produkts, die nichtformale Spezifikation.
2. *Spezifizierung:* Ergebnis ist die *formale Spezifikation*, beispielsweise eine Funktionstabelle.
3. *Entwurf:* Ergebnis ist die *Architektur* des Produkts, beispielsweise ein Blockschaltbild mit den Schaltzeichen der Tafel 1.3-2.
4. *Implementierung:* Ergebnis sind die Konstruktionsunterlagen und Schaltbilder in der gewählten Technologie, beispielsweise CMOS.
5. *Realisierung:* Ergebnis sind die Fertigungsunterlagen (Stücklisten und dergleichen)
6. Fertigung/Montage/Prüfung
7. Vertrieb/Beratung/Verkauf
8. Gebrauch/Verbrauch/Wartung
9. *Recycling/Entsorgung*

Für den Bau eines Addierers sollen hier die ersten drei Phasen durchlaufen werden.

Anforderungserfassung: Es geht darum, die Addition von Zahlen im binären Zahlensystem zu realisieren. Da das Stellenwertsystem es erlaubt, die Addition stellenweise zu betrachten, geht es also zunächst nur darum, zwei Bits zu addieren: 0+0=0, 0+1=1, 1+0=1, 1+1=0 plus Übertrag von einem Bit zur nächsten Stelle. Das führt zu Schluss, dass ein *Volladdierer* eigentlich drei Eingangsgrößen hat, denn wir müssen auf jeder Stufe auch den Übertrag vom letzten Bit berücksichtigen. Im höchsten Fall sind also drei Einsen zu addieren: 1+1+1 = 1 plus ein Übertrag von 1 zur nächsten Stelle. Als Ausgangsgrößen hat jeder Volladdierer das Summenbit und den Übertrag zur nächsten Stelle (Carry-Bit).

Spezifizierung: Aus der verbalen Beschreibung lässt sich leicht die Funktionstabelle herleiten. Wir bezeichnen die Eingangsgrößen mit *A*, *B* (die Bits auf den entsprechenden Stellen der Summanden) und $C_{\ddot{u}}$ (Carry-Bit von der niedrigerwertigen Stelle). Die Ausgangsgrößen sind das Summenbit *S* und der Übertrag zur nächsten Stelle *C*. Die Spezifikation mittels Funktionstabelle:

A	B	$C_{ü}$	S	C
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Entwurf: Aus der Funktionstabelle liest man unmittelbar die KDNF für die Ausgangsgrößen ab. Zur Verkürzung der Darstellung lassen wir die Klammern und das Verknüpfungszeichen $\wedge$ weg.

$$S = \neg A\neg BC_{ü} \vee \neg AB\neg C_{ü} \vee A\neg B\neg C_{ü} \vee ABC_{ü}$$

$$C = \neg ABC_{ü} \vee A\neg BC_{ü} \vee AB\neg C_{ü} \vee ABC_{ü}$$

Der Ausdruck für das Carry-Bit lässt sich vereinfachen:

$$C = \neg ABC_{ü} \vee A\neg BC_{ü} \vee AB\neg C_{ü} \vee ABC_{ü}$$

$$= \neg ABC_{ü} \vee A\neg BC_{ü} \vee AB\neg C_{ü} \vee ABC_{ü} \vee ABC_{ü} \vee ABC_{ü}$$

$$= (\neg ABC_{ü} \vee ABC_{ü}) \vee (A\neg BC_{ü} \vee ABC_{ü}) \vee (AB\neg C_{ü} \vee ABC_{ü})$$

$$= BC_{ü} \vee AC_{ü} \vee AB$$

Ferner könnte man beispielsweise noch $AC_{ü} \vee AB$ vereinfachen zu $A \wedge (C_{ü} \vee B)$. Aber damit ginge die Grundstruktur der DNF verloren. Aber gerade für die Realisierung von Schaltfunktionen in dieser Form gibt es *programmierbare Logikbausteine (PLD, Programmable Logic Devices)*. Diese gehören zur Klasse der *ASICs (Application Specified Integrated Circuits)*. Wir begnügen uns also mit dem erreichten Stand der Minimierung. **Bild 1.4-1** zeigt das zugehörige Blockschaltbild.

Konvention für Blockschaltbilder: *Eingangsgrößen* gehen von links oder von oben in Blöcke hinein. *Ausgangsgrößen* verlassen die Blöcke nach rechts oder nach unten.

Übung: Zeichnen Sie das Blockschaltbild für einen Addierer, der 4-Bit-Wörter addieren kann. Das Additionsergebnis sei ebenfalls ein 4-Bit-Wort. Was passiert, wenn die Summe gleich 16 oder gar größer wird? *Hinweis:* Mit k mod n bezeichnet man den ganzzahligen Rest der Division von k durch n. Die Zahlen werden als ganzzahlig vorausgesetzt, außerdem sei $k \geq 0$ und und $n > 0$.

Literaturhinweise

Entwurf und Minimierung von Schaltnetzen werden im Buch von Schiffmann und Schmitz (1993, Abschnitt „4 Schaltnetze“, Abschnitt „7.1.4 ASICs“) dargestellt. Zum allgemeinen

Vorgehen bei Entwurf und Konstruktion wurde auf die Hütte (1991, Teil „K Entwicklung und Konstruktion“) und folgende weitere Literatur zurückgegriffen:

Zemanek, H.: Das geistige Umfeld der Informationstechnik. Springer, Berlin, Heidelberg 1992

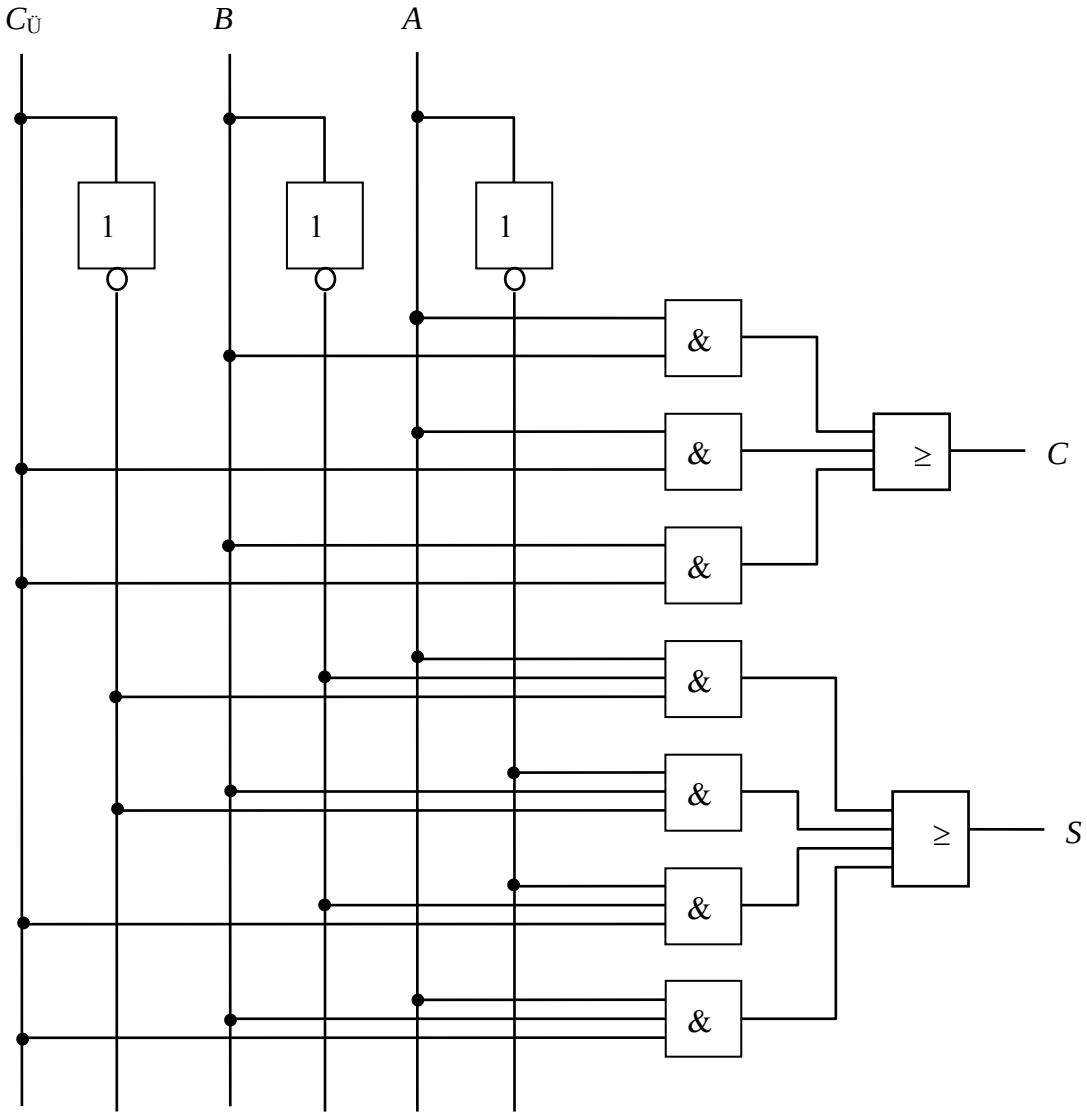


Bild 1.4-1 Volladdierer

1.6 Schaltwerke und Speicher

Schaltungen mit Rückkopplungen. Entwurf von Ein-Bit-Speichern.

Mit Schaltnetzen lassen sich boolesche Funktionen realisieren. Abläufe, wie sie bei Algorithmen auftreten, sind damit nicht möglich. Für die Lösung allgemeiner algorithmischer Aufgaben benötigen wir Einrichtungen mit Gedächtnis, also Geräte, die einen Zustand speichern können.

Grundsätzlich für diesen Zweck geeignet sind *Schaltwerke*. Diese unterscheiden sich von den Schaltnetzen dadurch, dass sie Rückkopplungen enthalten: Beim Schaltnetz kommt man, wenn man den Signalfluss verfolgt, immer zwangsläufig zum Ausgang; kein Pfad wird mehrfach durchlaufen. Demgegenüber kommen in Schaltwerken Schleifen vor.

Ein Schaltwerk beschreiben wir mit den Variablen x, y und z - das sind die *Eingangsgröße*, die *Ausgangsgröße* und die *Zustandsvariable* des Schaltwerks. Die Werte der Größen sind Binärzeichenfolgen: $x \in B^m$, $y \in B^n$, $z \in B^q$.

Außerdem sei eine *Übergangsfunktion f* gegeben, die in Abhängigkeit von dem momentan gültigen Zustand und der momentan anliegenden Eingangsgröße den Folgezustand z' festlegt:

$$z' = f(z, x)$$

Dabei wollen wir voraussetzen, dass der Folgezustand stabil ist, dass also $z' = f(z', x)$ gilt. (Genaugenommen wären noch Einschränkungen in Bezug auf Laufzeiteffekte nötig. Aber diese Betrachtungen führen hier zu weit. Sie gehören in die Lehrveranstaltungen der Digitaltechnik.)

Die Ausganggrößen sind durch die *Ausgabefunktion g* gegeben:

$$y = g(z, x)$$

Für $m = 0$ hat das Schaltwerke keine Eingangsgröße, und für $q = 0$ liegt der Grenzfall des Schaltnetzes vor.

Die abstrakten Definitionen sollen nun mit Gehalt gefüllt werden. Wir nehmen uns den Bau eines einfachen Speichers vor. Zunächst wollen wir die Anforderungen formulieren.

Anforderungserfassung: Die Schaltung hat zwei Eingänge: den S-Eingang zum Setzen des Bits (Set) und den R-Eingang zum Rücksetzen (Reset). Setzen und Rücksetzen sollen jeweils durch den Wert 1 der jeweiligen Eingangsgröße bewirkt werden. Ansonsten haben die Eingangsgrößen den Wert 0. Die Ausgangsgröße soll nach dem Setzen des Bits den Wert 1 annehmen und nach dem Rücksetzen den Wert 0.

Spezifizierung: Die Ausgangsgröße ist zugleich Zustandsvariable. Wir nennen sie Q. Die Eingangsgrößen sind S und R. Mit den obigen Bezeichnungen ist also $x = (S, R)$, $y = z = Q$. Die Übergangsfunktion ist gegeben durch folgende Tabelle:

S	*R*	*Q*	*Q'*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

In dieser Tabelle bedeutet „-“ soviel wie „don't care“. In diesen Fällen ist es also egal, welchen Wert der Ausgang annimmt. Bereits die Eingangswertkombination wird als unsinnig angesehen und sollte eigentlich nie vorkommen. Wir können hier willkürlich 1 oder 0 einsetzen. Dabei bietet sich die Möglichkeit, den Entwurf zu optimieren. Bei Ansatz einer KDNF wird man diese Werte auf null setzen und bei Ansatz einer KKNF wählt man diese Werte zu eins.

Übung: Stellen Sie fest, ob die Folgezustände alle stabil sind.

Entwurf: Nach der Methode des letzen Abschnitts ergibt sich die KDNF zu

$Q' = \neg S\neg RQ \vee S\neg R\neg Q \vee S\neg RQ.$

Diese Funktion ist zu vereinfachen, denn es sind - gemessen an der einfachen Aufgabe - zu viele verschiedene Gattertypen und diese in zu großer Anzahl beteiligt. Nun gibt es viele Möglichkeiten, die Regeln der Booleschen Algebra auf diesen Ausdruck anzuwenden. Von vornherein ist allerdings nie klar, ob der eingeschlagene Weg tatsächlich zu Vereinfachungen führt. Am besten man probiert viele Wege aus. In dieser Lektion werden natürlich nur die Wege beschritten, die schließlich von Erfolg gekrönt sind. Zunächst stellen wir fest, dass der letzte Term sich von den beiden vorhergehenden jeweils in nur einer Variablen unterscheidet. Das führt unmittelbar zu folgenden Vereinfachungsschritten:

$Q' = \neg S\neg RQ \vee S\neg R\neg Q \vee (S\neg RQ \vee S\neg RQ)$ //a = a ∨a

$= (\neg S\neg RQ \vee S\neg RQ) \vee (S\neg R\neg Q \vee S\neg RQ)$ //Assoziativgesetz

$= \neg RQ(\neg S \vee S) \vee S\neg R(\neg Q \vee Q)$ //Kommutativ- und Distibutivgesetze

$= \neg RQ \vee S\neg R$ //Exitenz komplementärer Elemente

$= \neg R(Q \vee S)$ // Kommutativ- und Distibutivgesetze

Nun stört nur noch, dass die Verknüpfungen drei verschiedene Gatter erfordern. Eine einfache Umformung beseitigt diesen Missstand. Schließlich sind nur zwei NOR-Bausteine nötig.

$Q' = \neg R\ (Q \vee S)$

$= \neg(R \vee \neg\ (Q \vee S))$ //De Morgansche Gesetze

$= \mathrm{NOR}(R, \mathrm{NOR}(Q, S))$ //Funktionsdefinition

Zur Realisierung mit NAND-Gattern kommt man, indem man die KDNF für $\neg Q'$ aufstellt und dabei die „Don't Care“-Terme auf 1 anstatt auf 0 setzt:

$\neg Q' = \neg S\neg R\neg Q \vee \neg SR\neg Q \vee \neg SRQ$

$= \neg S\neg Q \vee \neg SR = \neg S(\neg Q \vee R) = \neg S\neg(Q\neg R)$

Oder so

$Q' = \neg(\neg Q') = \neg(\neg S \neg(Q \neg R)) = \text{NAND}(\neg S, \text{NAND}(Q, \neg R))$

Das Schaltbild ist in **Bild 1.5-1** zu sehen.

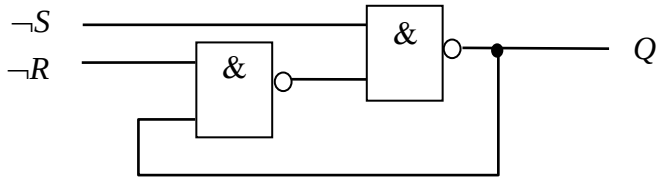


Bild 1.5-1 Ein-Bit-Speicher

Der Speicher aus Bild 1.5-1 wird *RS-Flip-Flop* genannt. Den negierten Q-Ausgang erhält man, indem man die Größe zwischen den NAND-Gattern als Ausgangsgröße herauszieht.

Übung: Setzen Sie ein Flip-Flop aus einem Vorschaltnetzwerk und nachfolgendem RS-Flip-Flop gemäß **Bild 1.5-2** zusammen. Folgende Anforderungen sollen erfüllt sein: Die Schaltung soll den Wert der Eingangsgröße *D* an den Ausgang *Q* übernehmen, sobald das Taktsignal *C* (Clock) den Wert 1 annimmt. Die unzulässigen Eingangskombinationen für das RS-Flip-Flop sollen vermieden werden. Entwerfen Sie das Vorschaltnetzwerk aus NAND-Gattern. Der so entstehende Ein-Bit-Speicher heißt *taktpegelgesteuertes D-Flip-Flop* (Wendt, 1982, S. 64). Eine Schaltungsvariante kommt mit nur zwei NAND-Gattern aus.

Anmerkung: Das D-Flip-Flop liefert den Werte der Eingangsvariablen *D* um einen Takt verzögert am Ausgang *Q* ab. Daher der Name: D steht für *Delay*.

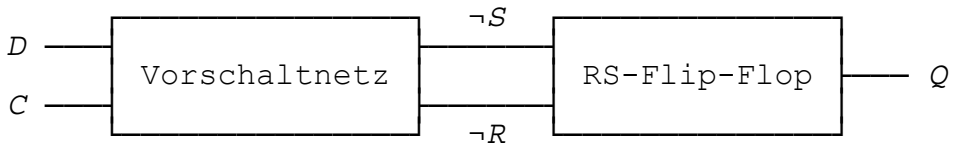


Bild 1.5-2 Flip-Flop mit Vorschaltnetzwerk

Literaturhinweise

Diese Lektion orientiert sich hinsichtlich des Aufbaus am Buch von Wendt (1982, Abschnitt „10.3 Schaltwerke“, S. 47 ff.).

1.7 Automaten

Automaten und synchrone (getaktete) Schaltwerke.

Mit den Möglichkeiten des Codierens, Speicherns und Verarbeitens von Nachrichten liegen die Grundlagen bereit. Alle notwendigen Voraussetzungen für den Computerbau sind damit gegeben. Jetzt geht es nur noch um den effizienten Einsatz der Ressourcen - um die Rechnerarchitektur.

Die wesentlichen Fähigkeiten zur Verarbeitung von Information und zur Abarbeitung von Algorithmen lassen sich mit *Automaten* darstellen. Und Automaten wiederum lassen sich durch getaktete Schaltwerke realisieren.

Einen Automaten beschreiben wir in derselben Weise wie allgemeine (also auch ungetaktete) Schaltwerke: Gegeben ist also eine Übergangsfunktion *f* und eine Ausgabefunktion *g*.

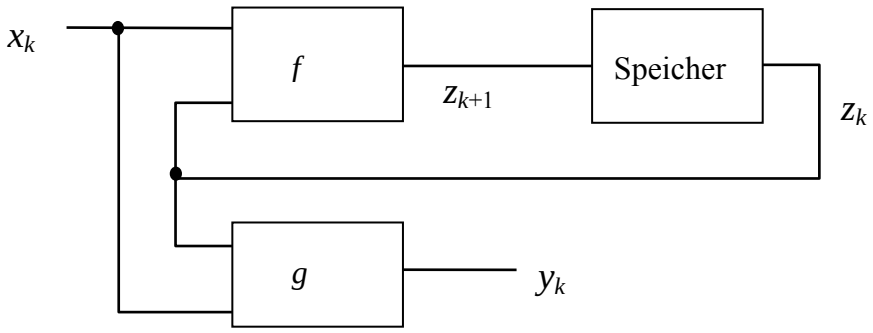


Bild 1.6-1 Blockschaltbild eines Automaten

Im Unterschied zur letzten Lektion wollen wir nun aber voraussetzen, dass für das Verhalten des Systems die Werte der Variablen *x*, *y* und *z* nur zu festen Zeitpunkten t_0, t_1, t_2, t_3, ... relevant sind. Wir haben also eine Folge x_0, x_1, x_2, x_3, ... von Werten der Eingangsgröße *x*, eine Folge y_0, y_1, y_2, y_3, ... von Werten der Ausgangsgröße *y* und eine Folge z_0, z_1, z_2, z_3, ... von Werten der Zustandsgröße *z* zu betrachten. Der Zusammenhang zwischen den Größen wird durch folgende Gleichungen hergestellt:

$z_{k+1} = f(z_k, x_k)$

$y_k = g(z_k, x_k)$

Das allgemeine Blockschaltbild zeigt das **Bild 1.6-1**. Die Aufgabe des Speichers ist es, innerhalb der auf den Zeitpunkt t_k folgenden *k*-ten Periode den für die Periode gültigen Zustand festzuhalten und jeweils zum nächsten markierten Zeitpunkt t_{k+1} den neuen Zustand zu übernehmen.

Grundsätzliche Lösungen für die drei Blöcke in Bild 1.6-1 sind uns bekannt: Für *f* und *g* sind das die Schaltnetze und die wesentlichen Komponenten des Speichers sind D-Flip-Flops.

Der Speicher erfordert eine eingehende Betrachtung. Folgendes Problem nämlich tritt auf, wenn man die Speicherung der Bits mit einem taktpegelgesteuerten D-Flip-Flop versucht: Bei einem Übergang des Taktes von 0 nach 1 wird der Wert des Flip-Flop-Eingangs an den Ausgang übernommen. Das Schaltnetz *f* berechnet den nächsten Zustandswert. Dadurch ändern sich die Werte am Eingang des Speichers. Aber zu diesem Zeitpunkt kann das Taktsignal

noch den Wert 1 haben und das kann zu unerwünschten Veränderungen der Ausgangsgrößen des Speichers innerhalb des alten Taktschrittes führen.

Eine grundsätzliche Möglichkeit, dieses Problem zu lösen und den Ausgang des Speichers vom Eingang zu trennen, bietet die Hintereinanderschaltung von zwei taktpegelgesteuerten Flip-Flops. Es entsteht das *Master-Slave-Flip-Flops*, dessen Funkionsprinzip in **Bild 1.6-2** dargestellt ist. Der Wert des Eingangs wird erst dann an den Ausgang übergeben, wenn das Taktsignal zum Wert 0 zurückkehrt (retardierter Ausgang).

Eine weitere Möglichkeit zur Entkopplung der Ein- und Ausgänge von Speichern bieten *taktflankengesteuerte* Flip-Flops. Weitere Informationen zur Entkopplung und zu den Details der Realisierung sind in den angegebenen Quellen zu finden.

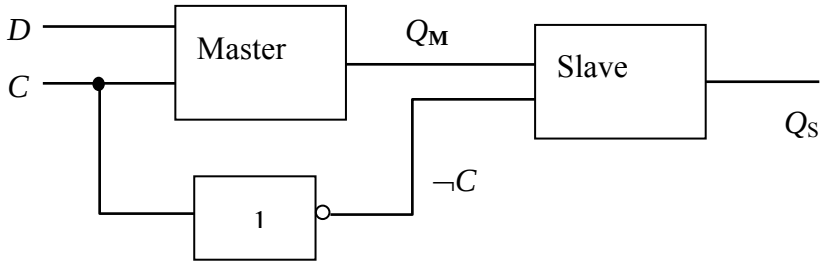


Bild 1.6-2 Prinzip des Master-Slave-Flip-Flops

Als Beispiel für den Entwurf eines getakteten Schaltwerks nehmen wir einen umschaltbaren *2-Bit-Gray-Code-Zähler* (Schiffmann, Schmitz, 1992, S. 227). Wir durchlaufen wieder die ersten drei Produktlebensphasen.

Anforderungserfassung: Ausgangsgröße des Zählers ist ein 2-Bit-Wort. Die Eingangsgröße besteht aus einem Bit. Falls der Wert der Eingangsgröße x gleich 0 ist, soll die Ausgangsgöße y zyklisch - Takt für Takt - folgende Werte liefern: 00, 01, 11, 10 ... Für $x = 1$ sollen die Werte in umgekehrter Reihenfolge durchlaufen werden 00, 10, 11, 01, ...

Spezifizierung: Der Zählerstand wird in der Zustandsvariablen z gespeichert. Sie ist zugleich Ausgangsgröße: $y = z = (z_1, z_0) = z_1z_0$. Da wir die Indexschreibweise schon für die Komponenten des Zustands verwenden, wollen wir hier auf den Index für die Zeit verzichten. Die für den Index k gültigen Werte werden mit x, y und z, z_1 und z_0 bezeichnet. Der Zustand des folgenden Zeitschritts ist z^+; er hat die Komponenten z_1^+ und z_0^+. Die Übergangsfunktion f ist gegeben durch die nebenstehende Funktionstabelle

z_1	z_0	x	z_1^+	z_0^+
0	0	0	0	1
0	0	1	1	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	0
1	1	1	0	1

Entwurf: Aus den Schaltfunktionen lesen wir die KDNF der Übergangsfunktionen für die Zustandskomponenten ab:

$$z_1^+ = \neg z_1 \neg z_0 x \vee \neg z_1 z_0 \neg x \vee z_1 \neg z_0 x \vee z_1 z_0 \neg x$$

$$z_0^+ = \neg z_1 \neg z_0 \neg x \vee \neg z_1 z_0 \neg x \vee z_1 \neg z_0 x \vee z_1 z_0 x$$

Diese Funktionen lassen sich durch Anwendung von Regeln der Form $abc \vee ab\neg c = ab$ und mit Hilfe der Kommutativ- und Assoziativgesetze vereinfachen zu:

$$z_1^+ = \neg z_0 x \vee z_0 \neg x$$

$$z_0^+ = \neg z_1 \neg x \vee z_1 x$$

Die Architektur in Form des Blockschaltbildes zeigt **Bild 1.6-3**. Für die Speicherung wurden in diesem Fall taktflankengesteuerte D-Flip-Flops genommen. Da die Flip-Flops üblicherweise nicht nur den Q-Ausgang, sondern auch noch den negierten Ausgang anbieten, können die Gatter für die Negation der Zustandsgrößen entfallen.

Übung 1: Entwerfen Sie ein Schaltwerk, das die Funktion eines Tastschalters (einer Nachttischlampe zum Beispiel) realisiert. Ein- und Ausschalten geschieht über das Taktsignal des Schaltwerks. Führen Sie die ersten drei Produktlebensphasen aus.

Übung 2: Ergebnis der ersten Übung ist eine Schaltung, die man auch als Frequenzteiler bezeichnen kann. Zeigen Sie, dass man mit solchen Schaltungen einen zyklisch arbeitenden Dualzähler aufbauen kann.

Literaturhinweise

Diese Lektion folgt dem Abschnitt 6 des Buches von Schiffmann und Schmitz (1993, S. 217 ff.). Anregungen wurden dem Abschnitt „J Technische Informatik“ (1991, S. J 16 ff.) der Hütte und dem Abschnitt „10.3 Schaltwerke“ des Buches von Wendt (1982, S. 47 ff.) entnommen.

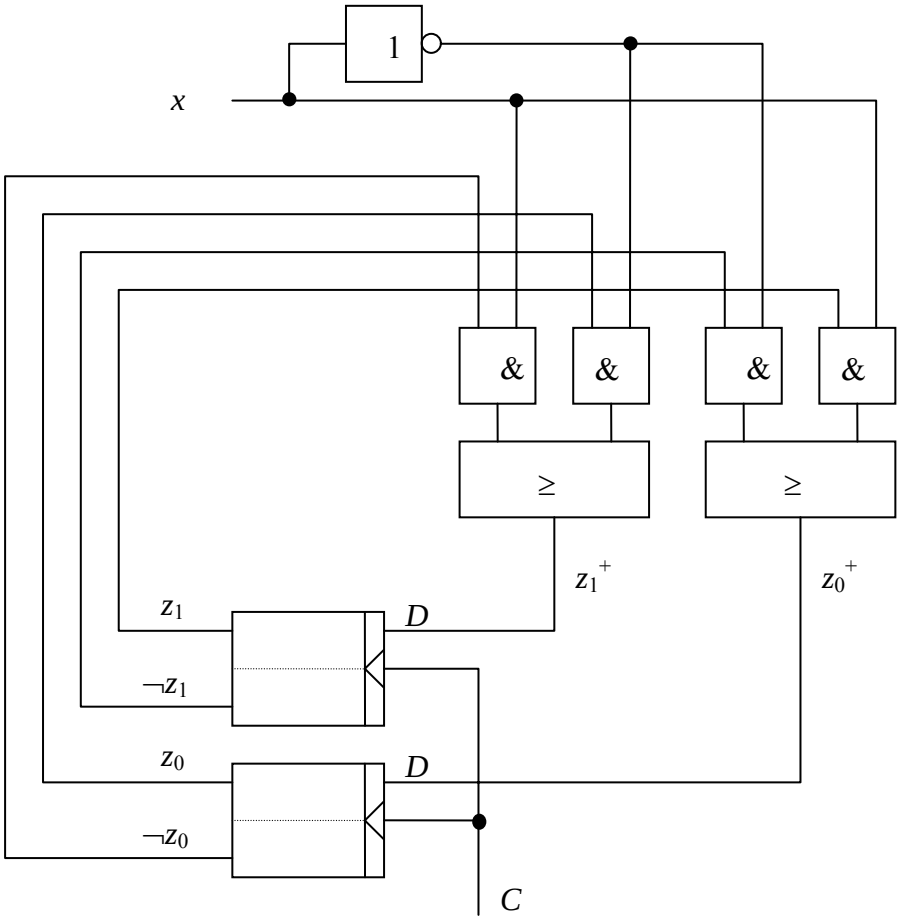


Bild 1.6-3 Umschaltbarer Gray-Code-Zähler

1.8 Der Von-Neumann-Rechner

Eine einfache Rechnerarchitektur.

Mit Schaltwerken kann man alle denkbaren Informationsverarbeitungsaufgaben lösen. Nur ein Nachteil scheint ihnen noch anzuhängen: Ändert sich die Aufgabe, muss ein neues Schaltwerk her. Die Hardware muss neu gebaut werden.

Die Sache bessert sich, wenn man ein steuerbares Schaltnetz verwendet, dessen Funktion durch Steuergrößen umgeschaltet werden kann. Nun werden auch die Steuerungsbefehle als Binärwörter codiert und dem Schaltnetz als zusätzliche Eingangsgrößen angeboten. Das Berechnungsergebnis wird dann nicht nur durch die Eingangsgrößen für die gewählte Verarbeitungsfunktion festgelegt (*Eingabedaten*), sondern auch vom gerade anstehenden Wert der *Steuergrößen*.

Ein steuerbares Schaltnetz, die so genannte *ALU* (Arithmetic Logic Unit), ist zentraler Bestandteil des *Rechenwerks* eines *Von-Neumann-Rechners*, **Bild 1.7-4**. Weitere Bestandteile dieser *Rechnerarchitektur* sind das *Leitwerk*, der *Speicher* und die *Ein-/Ausgabeeinheit*.

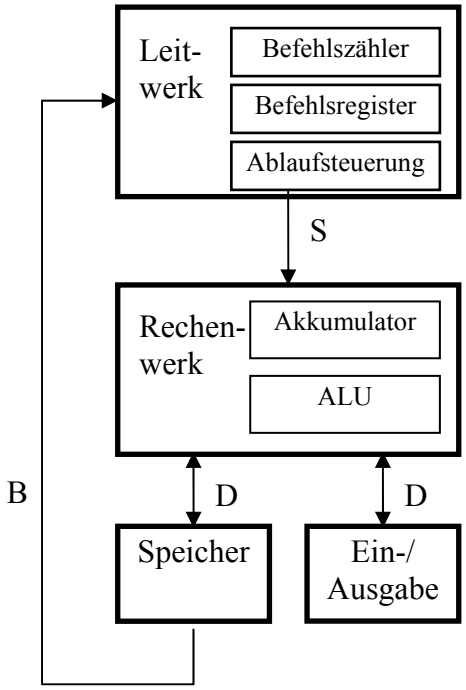


Bild 1.7-4 Architektur des Von-Neumann-Rechners

Zeichenerklärung:
S: Steuerverbindung (nur die wichtigste)
D: Datenverbindungen
B: Befehlsverbindungen
Adressverbindungen fehlen ganz

Neben der ALU enthält das Rechenwerk mehrere *Register*. Sie dienen der kurzzeitigen Speicherung von Daten. Das zentrale Register im Rechenwerk ist der *Akkumulator*. Er sammelt (akkumuliert) Ergebnisse fortlaufender Operationen, zum Beispiel beim Addieren. Über ihn liefert das Rechenwerk das Berechnungsergebnis ab.

Das *Leitwerk* steuert den Ablauf des Programms, indem es *Maschinenbefehle* aus dem Speicher holt, im *Befehlsregister* zwischenspeichert und die einzelnen Operationscodes mittels einer *Ablaufsteuerung* in Steueralgorithmen umsetzt. Holphase und Ausführungsphase bilden zusammen den *Befehlszyklus*. Er wird vom Leitwerk ständig wiederholt. Mit dem *Befehlszähler* wird der Speicher adressiert. Während der Holphase zeigt er auf den nächsten Maschinenbefehl. Nachdem der Befehl geholt ist, wird die Adresse fortgeschaltet. Das geschieht im Normalfall durch eine Erhöhung des Wertes um eins.

Das Leitwerk hat auch die Übertragung der Daten und Befehle zwischen den einzelnen Komponenten des Rechners mit Hilfe von *Datenbussen* zu bewerkstelligen. Ein Datenbus ist im wesentlichen ein Bündel von Adress-, Daten- und Steuerleitungen, die von mehreren Komponenten des Rechners gemeinsam genutzt werden. Datenbusse sind in Bild 1.7-4 nicht eingezeichnet, sondern nur die wesentlichen der durch die Datenbusse hergestellten Verbindungen (Kommunikationsbeziehungen).

Im *Speicher* eines Von-Neumann-Rechners werden sowohl *Daten*, also die Eingangs- und Ausgangsgrößen der Berechnungen, als auch die *Befehle* aufbewahrt.

Die *Ein-/Ausgabeeinheit* dient als Schnittstelle zur Peripherie des Rechners. Zur Peripherie gehören die Tastatur, die Maus, Monitore, Drucker usw.

Für die unmittelbare Programmierung solcher Rechner werden die Maschinenbefehle mit kurzen und leicht eingängigen Namen bezeichnet. Eine Folge solcher Befehle bilden ein *Assemblerprogramm*. Es wird durch einen *Assembler* in Maschinencode (Bitfolgen) übersetzt.

Um das Prinzip sichtbar zu machen, soll ein einfacher Algorithmus in einer Art Assembler-Sprache formuliert werden. Und zwar nehmen wir den *Euklidischen Algorithmus* zur Berechnung des größten gemeinsamen Teilers zweier Zahlen p und q als Beispiel.

Zunächst ein paar Bezeichnungen: a div b bezeichne das Ergebnis der *ganzzahligen Division* von a durch b, und a mod b den *Rest der Division*. Dabei wird a als nichtnegativ und b sogar als positiv vorausgesetzt. Offensichtlich gilt $a = b \cdot (a \text{ div } b) + a \bmod b$ und $0 \le a \bmod b < b$.

Beispiel: $55 \text{ div } 6 = 9$ und $55 \bmod 6 = 1$.

Der Euklidische Algorithmus geht so:

Setze $p_0 = p$ und $p_1 = q$
Berechne $p_2 = p_0 \bmod p_1$
Falls p_2 ungleich 0, berechne $p_3 = p_1 \bmod p_2$
Falls p_3 ungleich 0, berechne $p_4 = p_2 \bmod p_3$
usw.

Für irgendeinen Index k muss einmal $p_k = 0$ sein und es geht nicht mehr weiter. Dann ist der größte gemeinsame Teiler von p und q gefunden. Er ist gleich p_{k-1}.

Nun ist es nicht sinnvoll, alle Zwischenergebnisse aufzuheben. Es genügen drei Speicherplätze für die Variablen: zwei für die Größen p und q und einer für die Hilfsgröße r, den Divisionsrest. Allerdings muss es möglich sein, Werte umzuspeichern. Wir schreiben $a := b$ für die *Zuweisung.* Durch eine solche Zuweisung verändert sich nur die linke Seite; a erhält densel-

ben Wert wie *b*. Der Euklidische Algorithmus lässt sich mit dieser Vereinbarung so schreiben:

1. Berechne $r = p \bmod q$
2. Falls $r = 0$: Ergebnis ist gleich q; Ende
   Falls $r>0$: $p := q$; $q := r$; Gehe zu 1.

Für die Programmierung wollen wir davon ausgehen, dass die Werte genauso viel Platz benötigen wie die Befehle. Es gibt also eine einheitliche Wortbreite für Befehle und Daten. Die Speicherplätze werden durchnumeriert. Diese Nummern entsprechen den Adressen. Die Variablen *p*, *q* und *r* stehen auf den Plätzen 0, 1 und 2. Ab Speicherplatz 3 kommt das Programm. Die Speicherbelegung zeigt **Bild 1.7-5**.

Die *Befehle* (hier: Einadressbefehle) bestehen aus dem *Operationscode* und der *Adresse* (allgemeiner: *Parameter*). Letztere gibt an,

- woher bzw. wohin Daten zu übertragen sind (bei Datenübertragungsbefehlen)
- welcher Speicherinhalt mit dem Akkumulator zu verknüpfen ist (bei Verknüpfungsbefehlen)
- mit welcher Adresse die Befehlsfolge fortzusetzen ist (bei Sprungbefehlen)

Bei den Sprungbefehlen wird der normale Ablauf insofern unterbrochen, als der Befehlszähler nicht einfach weiterzählt, sondern indem er explizit auf einen bestimmten Wert gesetzt wird.

Literaturhinweise

Die Darstellung des Hardware-Aufbaus des Von-Neumann-Rechners lehnt sich an Schiffmann und Schmitz (1992, Abschnitt „2 von NEUMANN-Rechner“, S. 24 ff.) an. Hinsichtlich der Programmierung folgt die Lektion dem Buch von Roller (1994, Abschnitt „3.6 Steuerwerk“, S. 58 ff.). Über den Aufbau realer Rechner gibt beispielsweise das „*Microsystem Components Handbook*“ der Firma *intel* Auskunft. Zur Assembler-Programmierung siehe beispielsweise das Referenzhandbuch zu *Turbo-Assembler* der Firma *Borland*.

Als Modell für den Von-Neumann-Rechners wurde der Didaktische Computer gewählt:

http://www.oberstufeninformatik.de/dc/index.html

Adresse	***Operati-onscode***	***Parameter***	***Kommentar***
0	JMP	4	PC:= 4
Speicherplätze für die Variablen			
1	DEF	0	Variable p (Wert = 0)
2	DEF	0	Variable q (Wert = 0)
3	DEF	0	Variable r (Wert = 0)
Eingabe der Zahlenwerte			
4	INM	1	Eingabe p (INput to Memory)
5	INM	2	Eingabe q
Befehlsfolge des ersten Schrittes: r := p mod q			
6	LDA	1	AC:= p (LoaD into Accumulator)
7	STA	3	r:= AC (STore Accumulator to memory)
8	SUB	2	AC:= AC-q (SUBtract from accumulator)
9	JNM	7	Falls 0≤AC: PC:= 7 (Jump on Not Minus)
Befehlsfolge des zweiten Schrittes: Falls r = 0: Ausgabe und Ende. Falls r>0: p:= q; q:= r; Gehe zum ersten Schritt.			
10	LDA	3	AC:= r (LoaD into Accumulator)
11	JZE	17	Falls AC=0: PC:= 17 (Jump on ZEro)
12	LDA	2	AC:= q
13	STA	1	p:= AC
14	LDA	3	AC:= r
15	STA	2	q:= AC
16	JMP	6	PC:= 6
Ausgabe und Ende			
17	OUT	2	Ausgabe: q (Ergebnis)
18	END		

Bild 1.7-5 Speicheraufteilung und Programm für den Euklidischen Algorithmus.

Erläuterungen:
AC: Akkumulator (Accumulator)
PC: Befehlszähler (Program Counter)
Start des Programms mit PC = 0

2 Sprachliche Grundlagen

2.1 Syntax der Kurzform-Logik

Wiederholung der Grundlagen der Logik: Aussagenlogik aus sprachlicher Sicht. Betonung der Formalisierung.

Programmieren ist das Konstruieren von Maschinen mittels einer Universalmaschine - dem Rechner. Die Funktion der Maschine wird durch ein *Programm* festgelegt, also durch die Formulierung eines Algorithmus in einer *Programmiersprache*.

Die Konstruktionsaufgabe läuft auf das Schreiben von Texten hinaus. Tatsächlich empfinden sich manche Informatiker eher als Dichter denn als Konstrukteure. Bedenkenswert ist, was Knuth in seinem Buch „Literate Programming“ (CSLI Lecture Notes, 1992) zu diesem Thema zu sagen hat: Programme, die wirklich schön, nützlich und gewinnbringend sein sollen, müssen lesbare Texte sein. Beim Schreiben sollte man mehr an den Leser als an die Maschine denken.

Der gute Software-Konstrukteur ist vor allem aber Ingenieur; und er arbeitet wie ein Ingenieur auf guter wissenschaftlicher Grundlage. Deshalb wollen wir uns zuerst mit den Grundlagen der Programmiersprachen auseinandersetzen.

Der erste Unterricht im Programmieren konfrontiert den Anfänger mit einer Schwierigkeit: Er muss zwei Sorten von Sprachen auseinander halten, nämlich die *Objektsprache* (z.B. die Programmiersprache `Pascal`) und die *Metasprache* (z.B. Deutsch), also die Sprache, in der ihm das Programmieren beigebracht wird.

Die Programmiersprache hat eine wesentlich strengere, einfachere Struktur als die Metasprache. Sie ist eine *formale Sprache*. Prototypen der formalen Sprachen sind die Sprachsysteme der mathematischen Logik: die *Aussagenlogik* und die *Prädikatenlogik.*

Die Prädikatenlogik ist es auch, die es erlaubt, die Anforderungen an ein Programm - und letztlich dessen Funktion - mittels Vor- und Nachbedingung zu beschreiben. Zudem treten in den Programmen selbst logische Ausdrücke auf. Also: Die mathematische Logik ist das A und O der Programmierung.

Anhand der Sprache der Logik wird exemplarisch deutlich, wie man zwischen der Form sprachlicher Gebilde - der *Syntax* oder *Grammatik* - und deren Inhalt bzw. Bedeutung - der *Semantik* - zu unterscheiden hat. Diese Unterscheidung soll hier einigermaßen konsequent herausgearbeitet werden. Insbesondere den Darstellungsmitteln zur Definition der Syntax und der Semantik von Programmiersprachen wird einiger Platz eingeräumt. Dadurch soll dem Lernenden später der Einstieg in andere Programmiersprachen erleichtert werden.

Aber bevor wir uns mit diesen bereits recht komplexen Sprachen beschäftigen, soll uns eine minimale Sprache als Modell dienen, an dem wir den Umgang mit solchen *formalen Sprachen* lernen. Wir wollen die Ausdrücke dieser Modell-Sprache *Kurzform-Ausdrücke* nennen. Kurzform-Ausdrücke zeichnen sich durch extrem konzentrierte Schreibweise aus und wir werden sie wegen dieses Vorteils später bei der Kommentierung von Herleitungen (Deduktionen) verwenden.

Die *Syntax* legt fest, was wir unter einem *wohlgeformten Ausdruck* verstehen wollen. Für die Syntaxdarstellung nehmen wir die *erweiterte Backus-Naur-Form (EBNF)* entsprechend dem `Pascal`-Standard (DIN 66 256). In der Metasprache EBNF werden die Symbole in folgenden Bedeutungen verwendet:

„=“ steht für „ist definiert als“

„|" für „alternativ"
„[x]" für „kein oder ein Auftreten von x"
„{x}" für „kein, ein- oder mehrfaches Auftreten von x"

Nichtterminale Symbole (sie kommen nur in der Metasprache vor) werden kursiv geschrieben. *Terminale Symbole* sind die Elemente der Objektsprache und werden nichtkursiv geschrieben. Sie können in Anführungszeichen stehen. Sie müssen in Anführungszeichen stehen, wenn Verwechslungsgefahr mit einem der Symbole der Metasprache besteht. Die Festlegung der Syntax geschieht durch eine Reihe von *Produktionsregeln* der Gestalt

```
Nichtterminales Symbol = Rechte Seite der Produktionsregel
```

Jedes nichtterminale Symbol kommt in genau einer Produktionsregel auf der linken Seite vor. Es gibt ein *Startsymbol* und alle wohlgeformten Sätze lassen sich aus diesem Startsymbol ableiten. Bei einer solchen Ableitung werden Schritt für Schritt die nichtterminalen Symbole durch die auf der rechten Seite der Produktionsregel aufgeführten Symbole oder Symbolfolgen ersetzt. Das geschieht so lange, bis alle nichtterminalen Symbole verschwunden sind.

Kurzform-Ausdrücke sind Zeichenfolgen, die nach den folgenden *Produktionsregeln* aus dem Startsymbol *Ausdruck* gebildet werden.

```
Konstante = 0 | 1
Variable = A | B | C | D | E | F | G | H | I | J | K | L | M | N |
           O | P | Q | R | S | T | U | V | W | X | Y | Z
Element = Konstante | Variable
Faktor = "¬" Faktor | Element | "(" Ausdruck ")"
Term = Faktor { Faktor }
EinfacherAusdruck = Term { "+" Term}
VergleichsOperator = "=" | "≤"
Ausdruck = EinfacherAusdruck [ VergleichsOperator EinfacherAusdruck ]
```

Ausdrücke, die gemäß diesen Produktionsregeln gebildet worden sind, heißen *wohlgeformt*. Zuweilen nennt man einen wohlgeformten Ausdruck auch eine *Formel*. Kurzform-Ausdrücke, also wohlgeformt nach den Produktionsregeln dieser Syntax, sind:

```
1. A=(B≤¬A)
2. (A=A¬B)≤A+B
3. (A+B)(A≤C)≤(B+C)
4. (A=B)(C=¬D)
5. A=(B=(C=¬(B=C)))
6. (AB+AC+BC)(A≤¬B)(B≤¬C)(C≤A)
7. 0
```

Nicht wohlgeformt hingegen sind

```
8. A+(+B)
9. ¬
```

Jeder wohlgeformte Ausdruck kann durch einen *Syntaxbaum* (auch: Ableitungsbaum) beschrieben werden. Für den ersten der wohlgeformten Ausdrücke ist dieser in **Bild 2.1-1** dargestellt.

Der Syntaxbaum ist eine Darstellung der *Bedeutungsstruktur* eines Ausdrucks - nicht jedoch seiner Bedeutung. Er ist strikt von oben nach unten und von links nach rechts zu lesen! Eine Grammatik heißt *eindeutig*, wenn es zu jedem wohlgeformten Ausdruck genau einen Syntaxbaum gibt.

Die Eindeutigkeit der Kurzform-Grammatik ergibt sich aus folgender Tatsache: Der Syntaxbaum lässt sich schrittweise konstruieren, indem man die Symbole eines Ausdrucks der Reihe nach und jeweils einzeln betrachtet. Bei jeder gemäß *Kurzform-Grammatik* auftretenden Alternative ist es stets möglich, allein unter Betrachtung des nächsten Eingabesymbols zu entscheiden, wie der Baum weiter zu entwickeln ist.

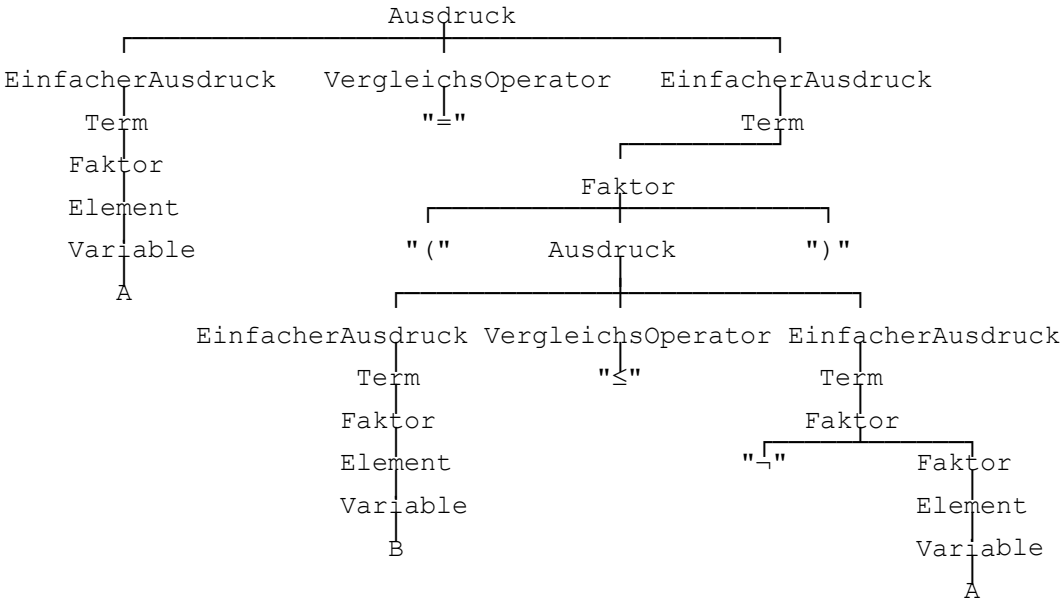


Bild 2.1-1 Syntaxbaum des Kurzform-Ausdrucks A=(B≤¬A)

Das Erschließen der Bedeutungsstruktur - also: die grammatische Satzzerlegung - ist die Aufgabe des *Parsers*. Der Parser liefert als wichtigstes Ergebnis der Satzzerlegung eines wohlgeformten Ausdrucks den Syntaxbaum.

Übung 1: Zeichnen Sie den Syntaxbaum zum 4. der wohlgeformten Ausdrücke.

Übung 2: Versuchen Sie auch Syntaxbäume zum 8. und 9. Ausdruck zu zeichnen. Warum gelingt es nicht?

Wem die Beschränkung der Kurzform-Ausdrücke auf einen endlichen Variablenbereich unbefriedigend erscheint, mache sich klar, dass man durch folgende Produktionsregeln - die die obige für die *Variable* ersetzen - auf beliebig große Variablenbereiche übergehen kann, ohne die gedrängte Form aufgeben zu müssen:

```
Variable = Großbuchstabe { Kleinbuchstabe }
Großbuchstabe = A | B | ... | Z
Kleinbuchstabe = a | b | ... | z
```

Literaturhinweise

Für ein vertiefendes Studium der Logik-Grundlagen werden folgende Bücher empfohlen:

Gries, D.: The Science of Programming. Springer Heidelberg 1981
Hermes, H.: Einführung in die mathematische Logik. Teubner, Stuttgart 1991
Hofstadter, D. R.: Gödel, Escher, Bach - ein Endloses Geflochtenes Band. Klett-Cotta, Stuttgart 1988
Loeckx, J.; Sieber, K.: The Foundations of Program Verification. Wiley-Teubner Series in Computer Science, Stuttgart 1987
Sander, P.; Stucky, W.; Herschel, R.: Automaten, Sprachen, Berechenbarkeit. Teubner, Stuttgart 1992

Dem Buch von Loeckx und Sieber sind viele Anregungen zur Konzeption dieser Lehrveranstaltung entnommen. Es ist zwar elementar in dem Sinne, dass es wenig Voraussetzungen verlangt. Für den Anfänger ist der sehr knappe mathematische Stil dennoch eine ziemlich große Herausforderung. Das Buch von Sander u. a. bietet eine Übersicht über formale Sprachen und deren Klassifizierung. Dort wird auch der Ableitungsbaum genau definiert.

2.2 Semantik

Durch den Syntaxbaum ist die Bedeutungsstruktur von Ausdrücken festgelegt, nicht aber die Bedeutung selbst. Die Bedeutung - also die *Semantik* - ist erst festgelegt, wenn wir wissen, wie wir die Ausdrücke auszuwerten haben. Die Bedeutungsstruktur ist die Anleitung zur Auswertung.

Sei V eine Menge von Variablen, über die wir Aussagen machen wollen. Nur die Variablen dieser Menge mögen in den zur Debatte stehenden Ausdrücken vorkommen. Ein *Zustand* (auch *Wertebelegung*) σ ist dadurch gegeben, dass jeder Variablen ein bestimmter Wert (eine der Konstanten 0 oder 1) zugeordnet ist. Wir bezeichnen den Wert, den eine Variable v im Zustand σ annimmt, mit $\sigma(v)$; σ ist also eine Funktion:

$\sigma: V \rightarrow \{0, 1\}$

Wir wollen nun den Standpunkt ändern und den Zustand ebenfalls als variabel ansehen: Die Menge aller Zustände - also aller möglichen Wertebelegungen der Variablen - bezeichnen wir mit Σ. Offensichtlich gilt $|\Sigma| = 2^{|V|}$.

Die Bedeutung - auch: *Semantik* - eines (Kurzform-)Ausdrucks a ist eine Funktion, die jedem Zustand einen Wert zuordnet:

$a: \Sigma \rightarrow \{0, 1\}$

Diese Funktion ist nun für alle Kurzform-Ausdrücke zu definieren. Für den Wert des Ausdrucks a im Zustand σ schreiben wir $\sigma(a)$ oder $a(\sigma)$. Beide Schreibweisen besagen dasselbe, nämlich „Wert des Ausdrucks a im Zustand σ“. Sowohl a als auch σ sind als variabel anzusehen.

Nun müssen wir noch festlegen, wie ein Kurzform-Ausdruck für einen bestimmten Zustand auszuwerten ist. In der mathematischen Logik nennt man eine solche *Auswertung* auch *Interpretation*.

Für einen bestimmten Zustand $\sigma \in \Sigma$ geschieht die *Auswertung* eines Ausdrucks a - also die Bestimmung des Wertes $a(\sigma)$ bzw. $\sigma(a)$ - anhand des Syntaxbaums nach folgenden *Auswertungsregeln:*

1. Jeder Variablen v wird ihr Wert $\sigma(v)$ zugeordnet.
2. Der Wert eines Elements ergibt sich aus dem Wert des nachfolgenden Zweigs (Konstante oder bewertete Variable).
3. Der Wert eines Faktors ergibt sich folgendermaßen:

3.1 Ist der Faktor Verzweigungspunkt für ein Terminalsymbol „¬“ und einen Faktor mit dem Wert x, dann erhält der Faktor im Verzweigungspunkt den Wert $1-x$.

3.2 Ist der Faktor ein (bereits bewertetes) Element, dann erhält er denselben Wert wie das Element.

3.3 Ist der Faktor Verzweigungspunkt für die Terminalsymbole „(„ und „)“ mit einem eingeschlossenen (bereits bewerteten) Ausdruck, dann erhält der Faktor denselben Wert wie der Ausdruck.

4. Jedem Term wird das Minimum der Werte seiner Faktoren als Wert zugeordnet. Also immer dann, wenn alle Faktoren den Wert 1 haben, hat auch der Term den Wert 1. Falls wenigstens einer der Faktoren den Wert 0 hat, dann hat auch der Term den Wert 0.
5. Jedem einfachen Ausdruck wird das Maximum der Werte seiner Terme zugeordnet. Also immer dann, wenn alle Terme den Wert 0 haben, hat auch der einfache Ausdruck den

Wert 0. Falls wenigstens einer der Terme den Wert 1 hat, dann hat auch der einfache Ausdruck den Wert 1.

6. Der Wert eines Ausdrucks ergibt sich folgendermaßen:

6.1 Ist der Ausdruck ein (bereits bewerteter) einfacher Ausdruck, dann erhält der Ausdruck denselben Wert.

6.2 Ist der Ausdruck Verzweigungspunkt für zwei bewertete einfache Ausdrücke und den Vergleichsoperator „=„ dann wird der Wert des Ausdrucks folgendermaßen bestimmt: 0=0 ergibt 1, 0=1 ergibt 0, 1=0 ergibt 0, 1=1 ergibt 1.

6.3 Ist der Ausdruck Verzweigungspunkt für zwei bewertete einfache Ausdrücke und den Vergleichsoperator „≤“, dann wird der Wert des Ausdrucks folgendermaßen bestimmt: 0≤0 ergibt 1, 0≤1 ergibt 1, 1≤0 ergibt 0, 1≤1 ergibt 1.

Die Regeln werden ausgeführt, solange wenigstens eine der Regeln auf einen noch nicht bewerteten Zweig anwendbar ist. Wegen der Eindeutigkeit der Grammatik von Kurzformausdrücken ist durch diese Auswertungsvorschrift der Wert eines jeden Kurzform-Ausdrucks eindeutig bestimmt.

Beispiel: Der Wert des Ausdrucks (A=A¬B)≤A+B ist gesucht für den Zustand σ, der definiert ist durch σ(A) = 0 und σ(B) = 1. Die systematische Anwendung der Auswertungsregeln liefert folgende Ersetzungen:

(A=A¬B)≤A+B	
→ (0=0¬1)≤0+1	//Regel 1
→ (0=00)≤0+1	//Regel 3.1
→ (0=0)≤0+1	//Regel 4
→ (0=0)≤1	//Regel 5
→ (1)≤1	//Regel 6.2
→1≤1	//Regel 3.3
→1	//Regel 6.3

Also ist σ((A=A¬B)≤A+B) = 1.

Da auch eine Variable *v* ein Ausdruck ist, können wir nun für den Wert dieser Variablen im Zustand σ auch schreiben *v*(σ). Es ist also *v*(σ) = σ(*v*).

Wenn wir 0 und 1 mit den Wahrheitswerten falsch und wahr identifizieren, dann haben wir mit den Kurzform-Ausdrücken ein einfaches Mittel um Aussagen, die durch Variablen repräsentiert werden, zu neuen komplexeren Aussagen zu verbinden, **Bild 2.2-1**.

Wir bezeichnen die Menge der Wertebelegungen (Zustände), für den ein Ausdruck den Wert 1 annimmt, als *logischen Spielraum* oder *Erfüllungsmenge* des Ausdrucks. Der logische Spielraum von Kurzform-Ausdrücken kann mit dem Programm `LogScope` (enthalten im Programm `Logic`) berechnet werden.

Konjunktion (Und-Verknüpfung)

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

Disjunktion (Oder-Verknüpfung)

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Negation

A	¬A
0	1
1	0

Äquivalenz (Bijunktion)

A	B	A=B
0	0	1
0	1	0
1	0	0
1	1	1

Implikation (Subjunktion)

A	B	A≤B
0	0	1
0	1	1
1	0	0
1	1	1

Bild 2.2-1 Wertetabellen für einfache Ausdrücke

Mit Kurzform-Ausdrücken können wir also *Aussagen* über etwas machen und wir können Sie zur Formulierung von Sätzen verwenden. Für die Übertragung deutscher Sätze in Kurzformlogik ist in vielen Fällen eine wörtliche Übersetzung möglich: „und“ meint die Konjunktion, „oder“ die Disjunktion, „nicht“ die Negation; „genau dann ... wenn ...“ gibt die Äquivalenz und „wenn ... dann ...“ gibt die Implikation wieder.

Jede der Variablen steht für eine *elementare Aussage*, die entweder wahr oder falsch sein kann. Die elementaren Aussagen „Es regnet“ und „Die Straße ist nass“ bezeichnen wir mit den Variablen R bzw. N. Diesen Schritt nennen wir Codierung. Der Satz „Wenn es regnet, ist die Straße nass“ sieht in Kurzformlogik dann so aus: R≤N. Diese Aussage ist nur dann falsch, wenn es tatsächlich regnet (R=1) und die Straße nicht nass ist (N=0). In allen anderen Fällen stimmt sie.

Wenn wir einen Kurzform-Ausdruck *a* ohne weiteren Kommentar verwenden, soll das immer heißen, dass wir voraussetzen, dass er einer wahren Aussage entspricht. Folglich ist es egal, ob wir „Es gilt *a*“, „*a* ist wahr“, „*a*=1“ oder nur „*a*“ schreiben. Gemeint ist stets, dass der Ausdruck, für den *a* steht, wahr ist - also den Wert 1 annimmt.

Wenn wir Aussagen über eine „Welt“ machen, heißt das, dass wir Sachverhalte beschreiben, und dass wir das, was nicht ist, ausschließen. Jede Aussagen über diese „Welt“ grenzt insofern den logischen Spielraum ein, reduziert die Menge aller denkbarer Welten. Im Idealfall - wenn sich nichts Neues mehr sagen lässt - bleibt schließlich nur die eine - uns interessierende - „Welt“ übrig. Je kleiner die Erfüllungsmenge einer Aussage ist, desto stärker ist diese Aussage.

Der Satz „Wenn es regnet, ist die Straße nass“ gibt eine Art *Naturgesetz* wieder. Er sagt etwas darüber aus, inwiefern sich unsere Welt von anderen denkbaren Welten unterscheidet. Auch der Satz „Wenn es regnet, gehe ich nicht zum Fußballspiel“ schränkt Möglichkeiten ein. In diesem Fall sagt der Satz etwas über meine Absichten. Es gibt Sätze, die sind in unserer Welt stets wahr. Der Satz „Wenn der Mond aus grünem Käse besteht (K), dann scheint heute Mittag die Sonne (S)“ hat die Grundform K≤S. Er ist stets wahr, unabhängig davon, ob die Sonne scheint oder nicht: Bereits K ist - in unsrer Welt - falsch, hat also den Wert 0, und 0≤S hat

stets den Wert 1, stimmt also immer. Und dann gibt es noch Sätze, die sind in allen denkbaren Welten richtig, zum Beispiel Aussagen der Form A=A oder A+¬A. Solche Aussagen heißen *Tautologien*.

Offensichtlich ist die *Kurzform-Logik* nichts wesentlich anderes als die Boolesche Algebra aus der Lektion 1.3. Es handelt sich ebenfalls um eine Aussagenlogik. Mit beiden Formen der Aussagenlogik lassen sich dieselben Sachverhalte ausdrücken.

Zwei Unterschiede gibt es allerdings; einen eher unwichtigen und einen etwas wichtigeren. Der unwichtige: Bei der Kurzform-Logik steht das „+"-Zeichen für die Disjunktion und nicht das Zeichen „∨„ und das Zeichen für die Konjunktion fehlt ganz. Der wichtige: Die Kurzform-Logik kommt mit weniger Klammern aus; es gibt eine klare Prioritätenregelung für Operationen: Konjunktion vor Disjunktion vor Vergleichungen. Beides dient der Kürze der Darstellung.

Übung 1: Geben Sie den logischen Spielraum des Kurzformausdrucks A=(B≤¬A) an. *Hinweis:* Benutzen Sie die folgende Tabelle. Ergänzen Sie die Einträge.

A	B	¬A	B≤¬A	A=(B≤¬A)
0	0			
0	1			
1	0			
1	1			

Übung 2: Bestimmen Sie die logischen Spielräume folgender Kurzformausdrücke: A≤¬A, (A≤B)(B≤C), A+B+C+D≤ABCD, ¬((A≤B)≤(B≤A))

2.3 Äquivalenztransformationen

Noch einmal: Gesetze der Aussagenlogik. Automatische Generierung der DNF. Minimierung mit elementaren Mitteln.

Nimmt ein Ausdruck für alle möglichen Wertebelegungen den Wert 1 an, sprechen wir von einer *Tautologie. Tautologien dienen zur Formulierung der Gesetze der Logik*. Besonders wichtig sind die *Äquivalenzen:* Die Ausdrücke *a* und *b* heißen äquivalent, wenn *a=b* eine Tautologie ist (Also: sie sind entweder beide wahr oder beide falsch).

In den Äquivalenzen der **Tafel 2.3-1** kann man jede Variable durch eine Formel - also einen wohlgeformten Ausdruck - ersetzen. Dabei ist die Formel gegebenenfalls einzuklammern. Durch solche Ersetzungen ergeben sich wiederum Äquivalenzen. Wichtig ist dabei, dass überall in der Formel die Variable durch den ihr zugeordneten Ausdruck ersetzt wird.

Substitutionsregel: Wenn man in einem beliebigen Ausdruck einen geklammerten Teilausdruck durch einen dazu äquivalenten ersetzt, so erhält man einen zum ursprünglichen Ausdruck äquivalenten Ausdruck.

Es handelt sich also - da wir die operationale Seite der Logik hier etwas betonen wollen - um *Gesetze der Äquivalenztransformation*. Die Äquivalenzen beweist man mittels Wertetabelle: für jede Wertebelegung muss sich das Ergebnis 1 ergeben.

In vielen Fällen reichen die Gesetze der Äquivalenztransformation für logische Schlussfolgerungen aus. Dabei ist zu bedenken: Will man Teilausdrücke von Ausdrücken durch äquivalente Ausdrücke ersetzen, geht das nur, wenn dieser Teilausdruck ohne Bedeutungsänderung geklammert werden darf. Die Assoziativgesetze zeigen, dass die Faktoren eines Terms beliebig geklammert werden dürfen - ebenso die Terme einfacher Ausdrücke. Außerdem kann man bei mehrfachen Verneinungen Klammern setzen: ¬¬A=¬(¬A).

Offensichtlich lassen sich Äquivalenzen und Implikationen mittels Äquivalenz-Transformationen aus den Kurzform-Ausdrücken eliminieren: Jeder Ausdruck ist einem einfachen Ausdruck äquivalent. Ferner lassen sich aus diesem einfachen Ausdruck sämtliche Klammern eliminieren.

Das Ergebnis dieser Umformungen ist eine Disjunktion von Termen, die nur noch Variablen, Konstante und bzw. deren Negationen enthalten. Man kann ferner dafür sorgen, dass in jedem Term jede Variable oder Konstante höchstens einmal vorkommt (entweder negiert oder nicht negiert) und dass die Anzahl der Terme möglichst klein wird: Das Resultat dieser Umformungen ist eine (minimierte) *disjunktive Normalform* (DNF).

Logikrätsel: Die Welt der Ritter und Schurken

Übung im Umgang mit den Transformationsgesetzen kann man durch das Lösen logischer Rätsel erwerben. Die Lösung läuft darauf hinaus, dass man über den Sachverhalt Aussagen macht und diese geeignet codiert (*Prämissen*). Die Aussagen werden konjunktiv zu einer Gesamtaussage verbunden. Letztere wird dann mittels Äquivalenztransformationen in die minimierte disjunktive Normalform gebracht. Diese liefert dann meist bereits die Antwort.

Die folgenden Fragen legt Smullyan (1986) seinen Lesern als Einführungskurs in die Welt der Ritter und Schurken vor: Die Ritter und Schurken sind die einzigen Bewohner einer Insel. Ritter sagen stets die Wahrheit und Schurken lügen konsequent.

Frage 1: Kann es auf dieser Insel einen Einwohner geben, der behaupten kann, ein Schurke zu sein?

Frage 2: Kann es einen Inselbewohner geben, der behaupten kann, er und sein Bruder seien beide Schurken?

Frage 3: Angenommen, ein Einwohner sagt über sich und seinen Bruder: „Mindestens einer von uns ist ein Schurke." Zu welchem Typ gehören die beiden?

Frage 4: Angenommen, er würde statt dessen sagen: „Genau einer von uns ist ein Schurke." Welche Schlüsse kann man über die beiden ziehen?

Frage 5: Angenommen, er würde statt dessen sagen: „Mein Bruder und ich gehören zum gleichen Typ; wir sind entweder beide Ritter oder beide Schurken." Was könnte man dann über die beiden schließen?

Herleitungen der Antworten zu den Fragen 1 bis 5: Wir *codieren* die Aussagen folgendermaßen: A steht für die Aussage „Der Einwohner ist Ritter" und B steht für die Aussage „Sein Bruder ist Ritter". Der Kurzformausdruck ¬B bedeutet dann beispielsweise: „Der Bruder ist ein Schurke".

Bei den Beweisen werden nur Äquivalenztransformationen ausgeführt. Neben dem Äquivalenzzeichen steht jeweils als Kommentar (ab „//" bis Zeilenende) die angewendete Regel.

Die erste Zeile jeder Folge von Äquivalenztransformationen ist der zu untersuchende Sachverhalt in formaler Schreibweise. Diese Aussagen haben die folgende Grundform: Ist der Einwohner Ritter dann stimmt seine Behauptung, andernfalls nicht. Die Behauptung des Einwohners ist also äquivalent zu A.

In der ersten Frage behauptet der Einwohner, er sei Schurke (¬A). Also lautet der Sachverhalt: A=¬A.

Die Äquivalenztransformationen zielen stets auf die minimierte DNF ab.

Antwort 1: Nein, keiner kann behaupten ein Schurke zu sein. Beweis:

```
A=¬A
A¬A+¬A¬¬A               //Äquivalenz-Elimination
A¬A+¬AA                 //Doppelte Verneinung
0+0                     //Widerspruch
0                       //Oder-Elimination
```

Das ist ein Widerspruch zum behaupteten Sachverhalt.

Antwort 2: Der Einwohner ist Schurke und sein Bruder Ritter. Beweis:

```
A=¬A¬B
A(¬A¬B)+¬A¬(¬A¬B)       //Äquivalenz-Elimination
A(¬A¬B)+¬A(A+B)         //De Morgan
(A¬A)¬B+¬AA+¬AB         //Assoziativ- und Distributivgesetze
0¬B+0+¬AB               //Widerspruch
¬AB                     //Und-, Oder-Elimination
```

Antwort 3: Der Einwohner ist Ritter und sein Bruder Schurke. Beweis:

```
A=(¬A+¬B)
A(¬A+¬B)+¬A¬(¬A+¬B)     //Äquivalenz-Elimination
A(¬A+¬B)+¬AAB           //De Morgan
A¬A+A¬B+¬AAB            //Distributivgesetz
0+A¬B+0B                //Widerspruch
A¬B                     //Und-, Oder-Elimination
```

Antwort 4: Über den Einwohner lässt sich nichts sagen, aber sein Bruder muss ein Schurke sein. Beweis:

```
A=(¬AB+A¬B)
A(¬AB+A¬B)+¬A¬(¬AB+A¬B)                 //Äquivalenz-Elimination
A(¬AB+A¬B)+¬A¬(¬AB)¬(A¬B)               //De Morgan
A(¬AB+A¬B)+¬A(A+¬B)(¬A+B)               //De Morgan
A¬AB+AA¬B+¬AA¬A+¬AAB+¬A¬B¬A+¬A¬BB       //Distributivgesetze
0B+AA¬B+0¬A+0B+¬A¬B¬A+¬A0               //Widerspruch
A¬B+¬A¬B                                //Und-, Oder-Elimination
(A+¬A)¬B                                //Distributiv-Gesetz
1¬B                                     //Tertium non datur
¬B                                      //Und-Elimination
```

Antwort 5: Über den Einwohner lässt sich nichts sagen, aber sein Bruder muss Ritter sein. Beweis:

```
A=(A=B)
A=(AB+¬A¬B)                             //Äquivalenz-Elimination
A(AB+¬A¬B)+¬A¬(AB+¬A¬B)                 //Äquivalenz-Elimination
A(AB+¬A¬B)+¬A¬(AB)¬(¬A¬B)               //De Morgan
A(AB+¬A¬B)+¬A(¬A+¬B)(A+B)               //De Morgan
AAB+A¬A¬B+¬A¬AA+¬A¬AB+¬A¬BA+¬A¬BB       //Distributivgesetze
AB+¬AB                                  //Und-, Oder-Elimination, Widerspruch
(A+¬A)B                                 //Distributivgesetz
B                                       //Tertium non datur
```

Die Äquivalenztransformation in die minimierte DNF lässt sich offenbar vollständig automatisieren. Da solche Umformungen bei der Programmierung immer wieder vorkommen, empfiehlt sich der Einsatz eines Programms. Ein solches - äußerst einfach gehaltenes - Programm ist `LogTrans`; es ist - wie das Programm `LogScope` - in das Programm `Logic` integriert.

Natürlich hätte man auch Wertetabellen bzw. das Programm `LogScope` für die Beantwortung der Fragen heranziehen können. Obiges Vorgehen zeichnet sich dadurch aus, dass bei den Umformungen allein die Syntax entscheidend ist. Die Semantik der Ausdrücke spielt nur bei der Codierung der Prämissen und bei der Interpretation des Ergebnisses eine Rolle. Das hat große praktische Bedeutung, was man beispielsweise bei der Behandlung quasi-boolescher Ausdrücke (teilweise undefinierte Ausdrücke) und bei dem weiteren Ausbau der Logik zur Prädikatenlogik erkennen wird.

Übung: Inspektor Craig muss auf der Insel der Ritter und Schurken einen Fall lösen. Ein Verdächtiger wird von Craig befragt. Hier das Protokoll:

Craig: Was wissen Sie von Arthur York?

Angeklagter: Arthur York hat einmal behauptet, ich sei ein Schurke.

Craig: Sind Sie vielleicht Arthur York?

Angeklagter: Ja.

Stellen Sie fest, ob der Angeklagte Arthur York ist. Tun Sie das, indem Sie die Basisaussagen („Angeklagter ist Ritter“, „York ist Ritter“, „Angeklagter ist York“) codieren und - dem Protokoll entsprechend - zu neuen Aussagen verknüpfen. Bilden sie die konjunktive Verknüpfung der Einzelaussagen. Führen Sie Äquivalenztransformationen durch, so dass schließlich der Sachverhalt als minimierte DNF vorliegt.

Literaturhinweise

In den Büchern von Smullyan sind viele Logik-Aufgaben zu finden, beispielsweise in:

Smullyan, R.: Dame oder Tiger? Logische Denkspiele und eine mathematische Novelle über Gödels große Entdeckung. Fischer, Frankfurt/M. 1983

Smullyan, R.: Spottdrosseln und Metavögel. Computerrätsel, Mathematische Abenteuer und ein Ausflug in die vogelfreie Logik. Fischer Logo, Frankfurt/M. 1986

Konforowitsch, A. G.: Logischen Katastrophen auf der Spur. Fachbuchverlag Leipzig-Köln 1992

Tafel 2.3-1 Gesetze der Äquivalenztransformation

1. Kommutativgesetze

```
AB = BA
A+B = B+A
(A=B)=(B=A)
```

2. Assoziativgesetze

```
A(BC)=(AB)C
A+(B+C)=(A+B)+C
```

3. Distributivgesetze

```
A+(BC)=(A+B)(A+C)
A(B+C)=AB+AC
```

4. De Morgansche Gesetze

```
¬(AB)=¬A+¬B
¬(A+B)=¬A¬B
```

5. Gesetz der doppelten Verneinung

```
¬¬A=A
```

6. Gesetz vom ausgeschlossenen Dritten (tertium non datur)

```
A+¬A=1
```

7. Gesetz vom Widerspruch

```
A¬A=0
```

8. Gesetz der Implikations-*Elimination*

```
(A≤B)=¬A+B
```

9. Gesetze der Äquivalenz-Elimination

```
(A=B)=(A≤B)(B≤A)
(A=B)=AB+¬A¬B
```

10. Gesetze der Oder-Elimination:

```
A+A=A
A+1=1
A+0=A
A+AB=A
```

11. Gesetze der Und-Elimination

```
AA=A
A1=A
A0=0
A(A+B)=A
```

12. Gesetz der Gleichheit

```
A=A
```

2.4 Der Kellerspeicher (Stack)

Schachtelsätze und rekursive Grammatik. Sequentielle Auswertung von Kurzform-Ausdrücken.

Eine Eigenheit der Kurzform-Ausdrücke erschwert ihre einfache sequentielle Auswertung: Geklammerte Ausdrücke können Teile von Ausdrücken sein. Ausdrücke sind *rekursiv* definiert. „Schachtelsätze" sind also erlaubt.

Schachtelsätze zeichnen die deutsche Sprache aus - und machen sie nicht gerade einfach. Kurt Tucholsky hat in seinen „Ratschlägen für einen schlechten Redner" ein schönes Beispiel geliefert: „Sprich mit langen, langen Sätzen - solchen, bei denen Du, der Du Dich zu Hause, wo Du ja die Ruhe, deren Du so sehr benötigst, Deiner Kinder ungeachtet, hast, vorbereitest, genau weißt, wie das Ende ist, die Nebensätze schön ineinandergeschachtelt, so daß der Hörer, ungeduldig auf seinem Sitz hin und her träumend, sich in einem Kolleg wähnend, in dem er früher so gern geschlummert hat, auf das Ende solcher Periode wartet ... nun, ich habe Dir eben ein Beispiel gegeben. So mußt Du sprechen."

Durch eine *Einzugstechnik* (englisch: Indentation) - wir werden sie später beim Programmieren noch ausgiebig nutzen - lässt sich die Satzstruktur dieses Monstrums verdeutlichen:

```
Sprich mit langen, langen Sätzen - solchen, bei denen Du,
  der Du Dich zu Hause,
    wo Du ja die Ruhe,
      deren Du so sehr benötigst, Deiner Kinder ungeachtet,
    hast,
  vorbereitest,
genau weißt, wie das Ende ist,
die Nebensätze schön ineinandergeschachtelt,
  so daß der Hörer,
    ungeduldig auf seinem Sitz hin und her träumend,
    sich in einem Kolleg wähnend,
      in dem er früher so gern geschlummert hat,
  auf das Ende solcher Periode wartet ...
nun, ich habe Dir eben ein Beispiel gegeben.
So mußt Du sprechen.
```

Der Hauptsatz und jeder der Nebensätze werden auf der Einzugsebene fortgesetzt, auf der sie begonnen worden sind. Jeder Nebensatz beginnt mit einem größeren Einzug als der durch ihn unterbrochene Satz bzw. Nebensatz. Das verdeutlicht die logischen Zusammenhänge. Diese Form entspricht auch dem Denkablauf: Wenn ein Nebensatz abgeschlossen ist, springt man auf die nächst höhere Ebene zurück und führt den dort unterbrochenen Gedanken weiter.

Wenn man den Satz hört und dabei interpretiert, muss man sozusagen den durch einen Nebensatz unterbrochenen Satz vorübergehend speichern. Sobald der Nebensatz zuende ist, holt man das Gespeicherte zurück und setzt den Gedankengang fort. Der Speicher wirkt wie ein *Keller* oder *Stapel:* Was zuletzt hineinkommt, kommt zuerst wieder heraus. Ein solcher Kellerspeicher ist also nach dem LIFO-Prinzip organisiert: Last in first out. (Ein weiteres wichtiges Speicherprinzip ist das bei Warteschlangen häufig auftretende FIFO-Prinzip: First in first out.)

Was hier die Sätze und Nebensätze sind, das sind bei den modernen Programmiersprachen Blöcke von Anweisungen, die logisch eine Einheit bilden. In diesem Fall spricht man von *Blockschachtelung*.

Der Kellerspeicher

Keller spielen eine zentrale Rolle bei der syntaktischen Analyse und der Interpretation von Ausdrücken und Programmtexten. Wir werden unsere Rechnerarchitektur (Lektion 1.8) um einen solchen Kellerspeicher (englisch: *Stack*) erweitern.

Mit Reg bezeichnen wir eines der Register unseres Rechners. Außerdem statten wir das Leitwerk des Rechners mit einem speziellen Adressregister, dem sogenannten Stackpointer SP, aus. Der Stack selbst ist im Speicher abgelegt. Er möge einen Speicherbereich bis zu einer bestimmten maximalen Adresse - nennen wir sie max - einnehmen.

Falls der Speicher leer ist, zeigt der Stackpointer auf den Platz dahinter: SP = max + 1. Wird ein Element vom Register Reg auf den Stack gelegt, muss zunächst der Stackpointer um 1 verringert und anschließend der Inhalt von Reg auf den Speicherplatz über tragen werden, auf den SP nun zeigt. Diese Operation bezeichnen wir mit PUSH. Die Umkehrung, das Entnehmen des letzten Elements vom Speicher, heißt POP. Die Speicherorganisation zeigt **Bild 2.4-1**. Der Stackpointer zeigt immer auf das oberste (zuletzt abgespeicherte) Element des Stapels.

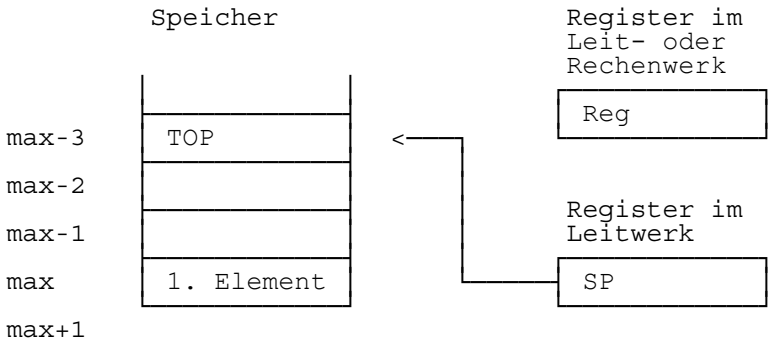


Bild 2.4-1 Kellerspeicher mit Registern
(Der Wert des Stackpointers: SP = max-3)

Für den Stack sind folgende Operationen und Funktionen definiert:

```
PUSH Reg: SP:= SP-1; (SP):= Reg

POP Reg:  Reg:= (SP); SP:= SP+1

TOP = (SP)
```

Dabei bezeichnet (SP) den Inhalt der Speicherzelle, auf die SP zeigt.

Auswertungsalgorithmus

Zunächst sei präzisiert, was unter einem (gerichteten) *Baum* zu verstehen ist: Ein Baum ist der Sonderfall eines (schwach zusammenhängenden) *gerichteten Graphen:* Er besteht aus einer Knotenmenge und einer Menge von Pfeilen, den *Zweigen* des Baums. Jeder Zweig ist eine Verbindung zwischen zwei Knoten. Im Baum gibt es genau einen Knoten ohne hinführenden Zweig. Das ist die *Wurzel* des Baums. Alle anderen Knoten des Baums haben genau einen hinführenden Zweig und eine beliebige Anzahl wegführender Zweige. Die Knoten ohne wegführenden Zweig werden *Blätter* genannt. Der Anfangsknoten eines Zweiges heißt *Elter* des Endknotens - und umgekehrt ist der Endknoten das *Kind* des Anfangsknotens. Jedes Blatt ist von der Wurzel aus über Pfeile und Knoten erreichbar.

In Lektion 2.1, Bild 2.1-1, ist der Syntaxbaum des Ausdrucks A=(B≤¬A) zu sehen. Die Wurzel des Baumes ist ganz oben. Ihr ist das Startsymbol zugeordnet. Den Blättern sind Terminalsymbole zugeordnet. Zu allen anderen Knoten gehören nichtterminale Symbole.

Wir komprimieren das Syntaxdiagramm jetzt so, dass alle Informationen für die Auswertung erhalten bleiben und alle für die Auswertung unnötigen Teile wegfallen. Die Funktions- bzw. Verknüpfungszeichen *f* können direkt den Elterknoten der Operanden zugeordnet werden. (Knoten werden hier durch Rechtecke symbolisiert und das Verknüpfungszeichen wird dort eingetragen. Hinsichtlich der Pfeilrichtungen wird die Konvention für Blockschaltbilder übernommen: Geht der Pfeil von oben nach unten bzw. von links nach rechts, dann unterbleibt eine besondere Kennzeichnung der Richtung.)

Die nichtterminalen Symbole werden für die Auswertung nicht gebraucht. Auch werden Knoten mit nur einem Kind, die einzig „Durchreichefunktion“ haben, gestrichen. Der vereinfachte Syntaxbaum ist in **Bild 2.4-2** links zu sehen.

komprimierter Syntaxbaum

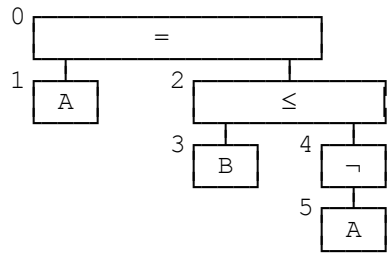


Tabelle der Funktionen Grad und f

N	Grad	f
0	2	=
1	0	A
2	2	≤
3	0	B
4	1	¬
5	0	A

Bild 2.4-2 Komprimierter Syntaxbaum zu Bild 2.1-1

In Bild 2.4-2 sind die Knoten des Baumes durchnumeriert (Zahl an der linken oberen Ecke). Wir bezeichnen die Anzahl der Kinder eines Knotens *N* mit Grad(*N*) und die Verknüpfung mit *f*(*N*). Die entsprechende Tabelle ist ebenfalls in Bild 2.4-2 angegeben.

Die Auswertung des Syntaxbaums geschieht im *Postorder-Durchlauf*. Dabei werden alle Knoten des Baumes genau einmal besucht. Die Reihenfolge, in der die Knoten dabei besucht werden, ist durch die *Regel für den Postorder-Durchlauf* festgelegt: Bevor ein Knoten besucht wird, werden alle Kinder eines Knotens, und zwar von links nach rechts, nacheinander besucht - erst dann der Knoten selbst.

Wenn im Postorder-Durchlauf der Wurzelknoten erreicht ist, dann sind alle Knoten des Baums einmal besucht worden. Denn: Dann hat man vorher alle Kinder des Wurzelknotens besucht, und deren Kinder, und deren Kinder, ...

Für den Baum in Bild 2.4-2 ergibt sich im Postorder-Durchlauf folgende Reihenfolge der Knoten: 1, 3, 5, 4, 2, 0.

Übung: Begründen Sie, dass die Postorder-Folge der Knoten eines Baumes zusammen mit ihren Graden sämtliche Informationen eines Baumes beinhalten, und dass sich daraus der Baum leicht rekonstruieren lässt. Zeigen Sie das für die Knotenfolge 1, 3, 5, 4, 2, 0 und die zugehörige Folge der Grade 0, 0, 0, 1, 2, 2.

Die Postorder-Folge der Knoten zusammen mit der zugehörigen Folge der Grade der Knoten ist also eine mögliche Repräsentation eines Baumes - sogar eine sehr praktische, wie wir sehen werden.

Der Postorder-Durchlauf gestattet auf einfache Weise die Bestimmung des Wertes eines Ausdrucks, wenn für jeden Knoten noch die Funktion gegeben ist.

Wir gehen also von einer Tabelle aus, in der die Knoten in Postorder-Folge zusammen mit ihrem Grad und ihrer Funktion aufgeführt sind. Im Beispiel des Bildes 2.4-2 sieht die *Postorder-Darstellung des Syntaxbaums* so aus:

N	Grad	f
1	0	A
3	0	B
5	0	A
4	1	¬
2	2	≤
0	2	=

Sind beispielsweise die Werte der Vorgängerknoten eines Knotens *N* bekannt, dann lässt sich auch der Wert dieses Knotens bestimmen. Denn: Die Werte der Kinder sind alle bekannt - sie liegen nach Postorder-Regel ja vorher - und außerdem kennt man die Funktion des Knotens *N*.

Hat ein Knoten keine Kinder - ist er also ein Blatt - dann ist ihm eine Konstante oder eine Variable eingetragen. Diese haben bei gegebenem Zustand stets einen Wert.

Da der erste Knoten in Postorder keinen Vorgänger hat - er muss ein Blatt sein - haben wir also einen Wert für den ersten Knoten (in Postorder). Damit auch für den zweiten (in Postorder), usw.

Da alle Kinder, die in die Funktion eines Knotens eingehen, in Postorder-Reihenfolge unmittelbar vor dem Knoten stehen, ist mit der Postorder-Folge und den Funktionen Grad und *f* die Auswertung nach folgendem *Auswertungsalgorithmus* möglich. Dieser Algorithmus benutzt einen Stack.

1. *Initialisierung:* Setze den Stack zurück (Stack ist leer). Setze *N* auf den ersten Knoten in Postorder
2. Nimm die letzten Grad(*N*) Werte des Stacks x_1, x_2, ..., $x_{\mathrm{Grad}(N)}$ und berechne mittels der Funktion *f*(*N*) den Wert des Knotens
3. Nimm die letzten Grad(*N*) Werte vom Stack (`POP`) und füge dem Stack den Wert des Knotens *N* an (`PUSH`).
4. Falls *N* der letzte Knoten in Postorder ist, dann ist die Wurzel erreicht und ihr Wert bestimmt - und damit auch der Wert des Ausdrucks. Er steht jetzt - als einziger Eintrag - auf dem Stack. Ansonsten: Setze *N* auf den Nachfolger (in Postorder) von *N* und fahre mit dem 2. Schritt fort.

Die Postorder-Darstellung des Syntaxbaums wird also zeilenweise durchgegangen. In **Bild 2.4-3** ist dieser Durchlauf für den Syntaxbaum des Bildes 2.4-2 dargestellt. Die Auswertung des Ausdrucks wird für den Zustand σ, der definiert ist durch σ(A) = 0 und σ(B) = 1, vorgenommen.

N	Grad	f	Stack	Kommentar
1	0	A	0	AC:= 0; PUSH AC
3	0	B	01	AC:= 1; PUSH AC
5	0	A	010	AC:= 0; PUSH AC
4	1	¬	011	POP AC; AC:= ¬AC; PUSH AC
2	2	≤	01	POP Reg; POP AC; AC:= AC≤Reg; PUSH AC
0	2	=	0	POP Reg; POP AC; AC:= AC=Reg; PUSH AC

Bild 2.4-3 Anwendung des Auswertungsalgorithmus auf ein Beispiel

Bei der Auswertung spielt übrigens der Knotenname keine Rolle. Die erste Spalte der Postorder-Darstellung kann auch weggelassen werden, wenn der Name nicht aus anderen Gründen benötigt wird.

Der Auswertungsalgorithmus ist eine ganz typische Anwendung eines Kellerspeichers.

Neben der Postorder-Reihenfolge ist noch die *Preorder*-Reihenfolge wichtig: Beim Preorder-Durchgang wird das Elter vor den Kindern besucht. Die Reihenfolge bei unserem Beispiel-

baum wäre dann: 0, 1, 2, 3, 4, 5. Also geschah bei diesem Beispiel bereits die ursprüngliche Nummerierung in Preorder-Reihenfolge.

Liegt für die Operatoren die Anzahl der Argumente fest, erhält man aus der Preorder- und Postorder-Darstellungen leicht die so genannte *polnische* bzw. *umgekehrte polnische Notation*. Setzt man beispielsweise die Disjunktion und die Konjunktion als grundsätzlich zweistellig voraus - anstelle von „*x*+*y*+*z*“ wäre dann etwa „*x*+(*y*+*z*)“ zu schreiben -, dann lautet der Ausdruck „A=(B≤¬A)“ in umgekehrter polnischer Notation „ABA¬≤=“; das ist die dritte Spalte in Bild 2.4-3.

Übung: Übertragen Sie die Auswertungstechnik für Ausdrücke auf die üblichen arithmetischen Ausdrücke. Setzen Sie auch hier voraus, dass alle Operationen zweistellig sind. Führen Sie das Programm für die folgenden Arithmetikausdrücke aus: a(b+c(a-b)), (a+b)(a+c). Führen Sie die Auswertung für (a, b, c) = (0, 1, 2) und für (a, b, c) = (2, 1, 0) durch.

Literaturhinweise

Die Organisation des Kellerspeichers ist in dem Büchern von Knuth (Band 1, 1973, Abschnitt 2.2.1, S. 234 ff.) und Schiffmann/Schmitz (Band 2, 1992, Abschnitt 2.3.1, S. 30 f.) dargestellt.

Der Auswertungsalgorithmus ist der Knuths Algorithmus F „Evaluate a locally defined function in a tree“ (Band 1, 1973, S. 351).

Bei den graphentheoretischen Begriffen wurde außerdem das folgende Werk zu Rate gezogen:

Neumann, K. Morlock, M.: Operations Research. Hanser, München, Wien 1993

2.5* Quasi-boolesche Ausdrücke

Ein ernstes - und meist verdrängtes - Problem: teilweise undefinierte boolesche Ausdrücke. Eine elegante Lösung.

Boolesche Ausdrücke treten häufig in Verbindung mit Programmverzweigungen auf: In Abhängigkeit vom Wahrheitswert wird der eine oder der andere Zweig durchlaufen. Es ist also für den reibungslosen Ablauf eines Programms von großer Bedeutung, dass diese Ausdrücke stets einen definierten Wert annehmen.

Aber genau das ist nicht immer gegeben. Nehmen wir den `Pascal`-Ausdruck

```
(j<1) OR (i MOD j = 0)
```

Falls j=0 ist dieser Ausdruck undefiniert: Er beinhaltet eine Division durch null. Man sieht dem Ausdruck allerdings an, dass genau dieser Fall durch den Term j<1 bereits abgedeckt werden sollte: Der Ausdruck sollte eigentlich auch in diesem Fall den Wert TRUE erhalten.

Es wäre also durchaus praktisch und würde die Programmierung wesentlich erleichtern, wenn teilweise undefinierten Ausdrücke ein definiertes Ergebnis hätten.

Alles Mögliche ist schon versucht worden, dieses Problem zu lösen. Meist wurde versucht, mit einer dreiwertigen Logik zurecht zu kommen - einer Logik also, die neben den Wahrheitswerten TRUE und FALSE noch einen dritten Wert kennt. Dieser dritte Wert wird hier mit UNDEFINED bezeichnet.

Dreiwertige Logiken basieren - wie die zweiwertigen - auf der Definition der elementaren Operationen mittels Wertetabelle. Auch für diese dreiwertigen Logiken gelten jeweils einige Regeln der Äquivalenztransformation. Aber leider eben nicht alle.

Einzusehen ist das unmittelbar am Gesetz des Tertium non datur: Es ist in jeder dieser dreiwertigen Logiken nicht richtig, dass stets A OR NOT A = TRUE ist, egal welchen Wert A hat. Zur Vermeidung von Widersprüchen muss nämlich bei undefiniertem A auch NOT A undefiniert sein. Und dass die Disjunktion von undefinierten Termen einen definierten Wahrheitswert ergeben soll, ist in keinem der Logiksysteme denkbar.

Soll die Gültigkeit des Gesetzes „Tertium non datur“ mit Biegen und Brechen durch geeignete Festlegung der Elementaroperationen aufrechterhalten werden, dann kneift es woanders: Es entsteht der Zwang, andere wichtige Gesetze der Logik aufzugeben.

Aber es gibt sowohl praktische Lösungen des Problems als auch eine theoretische. Und - das ist das Schöne - überall wo die praktischen Lösungen funktionieren, sagt die Theorie warum!

Für die Auswertung teilweise undefinierter Ausdrücke stellen die meisten Programmiersprachen einfache und wirkungsvolle Mechanismen zur Verfügung, beispielsweise die Operationen des *bedingten Und* und des *bedingten Oder*: CAND und COR.

Beide werden durch das *Kurzschlussverfahren* von Turbo-Pascal realisiert: Im Kurzschlussverfahren wirkt AND wie CAND und OR wie COR. Die Operationen sind über die Wertetabellen in **Bild 2.6-1** definiert. T steht für TRUE, F für FALSE und U für UNDEFINED. Anstelle des Symbols „-“ kann in der Tabelle jeder dieser Werte stehen.

Allgemein ist das Kurzschlussverfahren so definiert, dass es Terme und einfache Ausdrücke nur soweit auswertet - und zwar von links nach rechts -, bis klar ist, welches Ergebnis herauskommen muss. Dabei wird so getan, als seien die restlichen - nicht mehr ausgewerteten - Faktoren bzw. Terme alle definiert. Es wird gar nicht mehr bemerkt, wenn einer dieser nachfolgenden Faktoren oder Terme undefiniert ist.

a	b	a CAND b
F	-	F
U	-	U
T	-	b

a	b	a COR b
F	-	b
U	-	U
T	-	T

Bild 2.6-1 Die Junktoren CAND und COR

Falls ein Ausdruck keine undefinierten Werte enthält, ändert sich durch die Verwendung von CAND und COR anstelle von AND und OR gar nichts.

Aber: Wir haben hier ein System der dreiwertigen Logik - mit der unangenehmen Konsequenz, dass die einfachen Umformungsregeln der zweiwertigen Logik nicht mehr gelten. Für CAND und COR sind beispielsweise bereits die Kommutativgesetze nicht erfüllt: FALSE CAND UNDEFINED ist etwas anderes als UNDEFINED CAND FALSE. Das Gesetz vom ausgeschlossenen Dritten (Tertium non datur) kann man natürlich auch vergessen.

Wir bleiben bei AND und OR - zumindest in der Theorie.

Es gibt eine elegante Möglichkeit, die Rechenregeln zu retten und dennoch undefinierte Faktoren und Terme zuzulassen. Wir definieren den Wert eines Ausdrucks allerdings nicht über das Kurzschlussverfahren, verlangen aber, dass immer dann, wenn das Kurzschlussverfahren einen definierten Wert ergeben würde, nach dieser Definition derselbe Wert herauskommen muss.

Auswertung quasi-boolescher Ausdrücke (nach Bijlsma):

> Für alle definierten Faktoren setzen wir deren jeweiligen Wert ein. Für undefinierte Faktoren vom Typ BOOLEAN führen wir Variablen ein. Wenn ein bestimmter undefinierter Faktor an mehreren Stellen auftritt, dann ist er jedesmal durch dieselbe Variable zu ersetzen. Zur Auswertung ersetzen wir diese Variablen wahlweise durch TRUE oder FALSE. Falls alle möglichen Wertebelegungen (natürlich bleiben die definierten Faktoren unverändert) immer wieder zu demselben Ergebnis führen, weist man dem Ausdruck genau dieses Ergebnis zu. Andernfalls ist der Ausdruck undefiniert.

Diese Methode lässt sich auch so beschreiben: Man ersetzt in einem Ausdruck die definierten Faktoren durch ihre Werte und führt für die undefinierten Faktoren Variablen ein. Diese eingeführten Variablen sind die einzigen in diesem Ausdruck. Nun kann man den logischen Spielraum dieses Ausdrucks bestimmen. Drei Fälle sind zu unterscheiden:

1. Der logische Spielraum umfasst sämtliche Wertebelegungen der Variablen. Anders ausgedrückt: Für jede Wertebelegung ergibt der Ausdruck den Wert TRUE.
2. Der logische Spielraum ist leer. Anders ausgedrückt: Für jede Wertebelegung ergibt der Ausdruck den Wert FALSE.
3. Es gibt Wertebelegungen mit dem Ergebnis TRUE und auch welche mit dem Ergebnis FALSE.

Im ersten Fall erhält der Ausdruck den Wert TRUE, im zweiten den Wert FALSE und im dritten ist der Ausdruck undefiniert.

Durch diese Festlegungen erhält man folgende Wertetabellen für einfach Ausdrücke:

a	b	a AND b
F	–	F
–	F	F
T	–	b
–	T	a
U	U	U

a	b	a OR b
F	–	b
–	F	a
T	–	T
–	T	T
U	U	U

Man erhält immer dann definierte Werte, wenn auch CAND und COR welche liefern. Und die Werte stimmen überein.

Die neuen Wertetabellen darf man *nicht* als Ausgangspunkt für die Definition einer „besseren“ dreiwertigen Logik missverstehen! Man kann nicht einfach diese Verknüpfungsregeln bei der Auswertung quasi-boolescher Ausdrücke verwenden.

Die Wertebelegung (a, UNDEFINED) und (b, FALSE) würde nämlich nach Anwendung dieser Tabelle für den Ausdruck

a AND (NOT a OR b)

den Wert UNDEFINED liefern. Wendet man dagegen die oben festgelegte Methode der Auswertung quasi-boolescher Ausdrücke an, erhält man das (definierte!) Ergebnis FALSE.

Bei der hier gewählten Methode der Auswertung quasi-boolescher Ausdrücke bleiben die Gesetze der Logik, also insbesondere das Äquivalenztransformationssystem gültig. Denn Tautologien bleiben ja für alle möglichen Wertebelegungen gleichermaßen wahr.

Und damit ist schon viel gewonnen: Man kann jeden quasi-booleschen Ausdruck in eine minimierte disjunktive Normalform transformieren. Lassen sich die Terme und Faktoren dieser minimierte DNF noch so umsortieren, dass sich mit dem Kurzschlussverfahren ein definierter Wert ergibt, dann ist der das Ergebnis des Ausdrucks.

Übung: Ergänzen Sie die folgenden Wertetabellen, indem Sie die Ausdrücke nach der Methode von Bijlsma auswerten:

A	B	C	D	(A=B)(B=D)(C=¬D)
0	0	0	?	
0	0	1	?	
1	1	0	?	
1	1	1	?	
0	?	0	1	
1	?	0	1	
?	0	0	1	
?	1	0	1	

A	B	(A=A¬B)≤A+B
0	?	
1	?	
?	0	
?	1	

A	B	C	(A=B)(B=C)+(B=¬C)
0	?	0	
0	?	1	
1	?	0	
1	?	1	

Literaturhinweise

Baber, R. L.: The Spine of Software. John Wiley 1987
Bijlsma, A.: Semantics of Quasi-Boolean Expressions. Aus: Beauty is Our Business (Feijen u. a., 1990, 27-35)
Feijen, W. H. J.; van Gasteren, A. J. M.; Gries, D.; Misra, J.: Beauty is Our Business. Springer, New York 1990

Im Buch von Baber sind einige Systeme der dreiwertigen Logik zu finden. Die hier vorgestellte Methode zur Auswertung quasi-boolescher Ausdrücke folgt dem Aufsatz von Bijlsma.

2.6* Deduktion

Logisches Schließen. Deduktionssysteme. Automatisierung des Denkens, Künstliche Intelligenz. (Diese Lektion bringt keinen neuen prüfungsrelevanten Stoff. Zweck ist das Einüben von Fertigkeiten im Umgang mit Logik und das Kennenlernen von Denkweisen der heutigen Informatik.)

Deduktionssysteme haben in der heutigen Informatik einige Bedeutung erlangt: Ein Teil der sogenannten KI-Forschung (KI steht für Künstlichen Intelligenz) gilt der „Automatisierung des logischen Denkens". Praktische Ergebnisse dieser Forschungsaktivitäten sind Sprachen der *logischen Programmierung* (`PROLOG`), automatisches Beweisen mathematischer Theoreme und Werkzeuge für Programmkorrektheitsbeweise.

Bei der *Deduktion* handelt es sich um das Erschließen von wahren Aussagen aus bereits als wahr erkannten - oder als wahr vorausgesetzten - Aussagen. Zur Erinnerung: Wahre Aussagen sind Formeln, die wahr sind. (Eine Formel ist wahr, wenn die Auswertung den Wahrheitswert TRUE liefert.) Wenn wir wahre Aussagen als *Sätze* bezeichnen, geht es also um das Erschließen neuer Sätze aus bereits bekannten.

Für Schlussfolgerungen dieser Art sind die folgenden Sprech- und Schreibweisen gleichbedeutend:

1. E lässt sich aus $E_1, E_2, ..., E_n$ ableiten

2. Von $E_1, E_2, ..., E_n$ lässt sich auf E schließen

3. $E_1, E_2, ..., E_n \rightarrow E$

In allen Fällen soll das bedeuten, dass die E_i den Ausdruck E implizieren, das heißt, dass folgender Ausdruck wahr ist

4. $E_1 \wedge E_2 \wedge ... \wedge E_n \leq E$

Die E_i heißen *Prämissen* und E ist die *Konklusion*. Eine *Ableitung* oder Deduktion ist immer so zu verstehen: Eine Konklusion ist in all den Zuständen (Wertebelegungen der Variabeln) wahr, in denen auch sämtliche Prämissen wahr sind. Eine zu 1. bis 4. äquivalente Formulierung ist die *Widerspruchsregel*, die in der Beweistechnik - insbesondere der Technik der automatischen Beweise - eine große Rolle spielt:

5. $E_1, E_2, ..., E_n, \neg E \rightarrow 0$

Für die Deduktion kann man - ähnlich wie bei den Äquivalenztransformationen - Umformungsregeln aufstellen. Einige dieser *Deduktionsregeln* (auch: *Schlussregeln*) sind:

1. Der *Modus ponens:* $A, A \leq B \rightarrow B$
2. *Modus tollens:* $A \leq B, \neg B \rightarrow \neg A$
3. Gesetz zum Aufbau von *Schlussketten:* $A \leq B, B \leq C \rightarrow A \leq C$

Die Deduktionsregeln kann man beweisen, in dem man auf die 4. Schreibweise als Implikation übergeht und dann zeigt, dass dieser Ausdruck eine Tautologie ist. Der Nachweis wird mittels Äquivalenztransformationen oder über den logischen Spielraum, also mittels Wertetabelle, geführt.

Für den Modus ponens wird das einmal in Kurzform-Schreibweise vorgeführt. Die Anwendung von Assoziativgesetzen und Kommutativgesetzen wird nicht kommentiert.

```
(A(A≤B)≤B)
=                          //Implikations-Elimination
¬(A(¬A+B))+B
```

```
=                          //De Morgan
¬A+¬(¬A+B)+B
=                          //De Morgan, doppelte Verneinung
¬A+A¬B+B
=                          //Und-Elimination (rückwärts)
¬A+A¬B+1B
=                          //Tertium non datur
¬A+A¬B+(A+¬A)B
=                          //Distributivgesetz
¬A+A¬B+AB+¬AB
=                          //Distributivgesetz
¬A+A(¬B+B)+¬AB
=                          //Tertium non datur
¬A+A1+¬AB
=                          //Und-Elimination
¬A+A+¬AB
=                          //Oder-Elimination
¬A+A
=                          //Tertium non datur
1
```

Nun sei kurz charakterisiert, was von dem bisher über Logik Gesagten wir in dieser Lektion beibehalten und was wir anders machen wollen:

1. Die Syntax der Formeln wird als gegeben vorausgesetzt.
2. Wir nehmen die Semantik ebenfalls als gegeben an. Aber:
3. Die Semantik wird bei den logischen Schlussfolgerungen nicht benutzt. Die Konstanten 0 und 1 (bzw. FALSE und TRUE) treten nicht in Erscheinung.
4. Alle Umformungen sind reine *Symbolmanipulationen:* Aus Formeln werden - unter Verwendung gewisser *Schlussregeln* (Deduktionsregeln) - neue Formeln abgeleitet. Alles spielt sich also auf der Ebene der Syntax ab.
5. Es werden nur solche Schlussregeln zugelassen, mit denen sich aus wahren Aussagen grundsätzlich nur ebenfalls wahre Aussagen erschließen lassen - immer bezogen auf die gegebene Semantik. (Das heißt nicht, dass als Prämissen nur Sätze - also wahre Aussagen - zugelassen sind.)

In dieser Lektion wird für Formeln die mathematische Notation gewählt mit folgenden Ausnahmen: Da der Pfeil „$\rightarrow$„ bereits für die Deduktion vergeben ist, wollen wir für die Inklusion nach wie vor das „$\leq$“-Zeichen verwenden. Und für die Gleichheit bzw. Äquivalenz schreiben wir nach wie vor das „=„-Zeichen. Ansonsten steht „$\wedge$„ für die Konjunktion, „$\vee$„ für die Disjunktion und „$\neg$“ für die Negation,.

Nun fragen wir nach einem vollständigen System der natürlichen Deduktion, mit dem sich alle möglichen Formeln, also insbesondere das Äquivalenztransformationssystem, ableiten lassen. Dieses System soll möglichst nur wenige und einfache Schlussregeln enthalten.

Die *Schlussregeln* der **Tafel 2.7-1** bilden die Basis des Systems der *natürlichen Deduktion.* Die Prämissen stehen oberhalb des Striches und die Konklusionen unterhalb.

Merkhilfe für die Benennungen: „$\wedge$-i“ heißt soviel wie „$\wedge$-Introduktion“ und „$\wedge$-e“ soviel wie „$\wedge$-Elimination“.

Diese Deduktionsregeln kann man allesamt beweisen, indem man auf die 4. Schreibweise als Implikation übergeht und dann zeigt, dass dieser Ausdruck eine Tautologie ist. Den Nachweis führt man mittels Äquivalenztransformationen oder auch mittels Wertetabelle.

Damit ist klar, dass das Deduktionssystem auch nicht mehr leisten kann als das Äquivalenztransformationssystem. Aber genauso gut kann man das Deduktionssystem als Ausgangs-

punkt wählen und sämtliche Äquivalenzen des Äquivalenztransformationssystems herleiten! Das ist der Grundgedanke der *Kalkülisierung der Logik* (Gentzen-Kalküle). Hier ist also die Antwort auf die Frage: Wie lässt sich Logik erklären, ohne dabei logische Schlussfolgerungen zu verwenden?

Wie die Beweise durchzuführen sind, zeigt das nachfolgende beschriebene *Beweisschema*[1]. Dabei handelt es sich um eine verschachtelte Struktur analog der *Blockstruktur* bei höheren Programmiersprachen wie Pascal.

```
E1, E2, ..., En → E
──┬──────────────
  │ E1
  │ E2
  │ ...
  │ En
  │ S1                              Ab hier erscheinen Konklusionen,
  │ S2                                die sich aus den Prämissen er-
  │ ...                               schließen lassen. Eine davon ist
  │ Sm                                            E (wenn's gut geht)
  │ E
```

Ab jetzt ist die Schlussfolgerung „E_1, E_2, ..., $E_n \rightarrow E$“ verwendbar - aber auch der Satz „$E_1 \wedge E_2 \wedge \ldots \wedge E_n \leq E$“.

[1] Bislang wurde die Bedeutung logischer Ausdrücke durch Auswertungsregeln definiert. Die sich daraus ergebenden Regeln der Äquivalenztransformation wollen wir nun dafür nutzen, das Beweisschema der natürlichen Deduktion zu begründen.

Die Deduktion findet in einem Kontext von gültigen Sätzen (Aussagen) statt. Mit c bezeichne ich die Konjunktion der Sätze des Kontexts. Sei nun $ca \rightarrow b$ eine gültige Deduktion (typischerweise geführt als Unterbeweis und durch Einrückung gekennzeichnet). Also ist $ca \leq b$ eine wahre (tautologische) Aussage. Daraus folgt durch Äquivalenztransformation $c \leq (a \leq b)$. Das heißt: Die Implikation $a \leq b$ kann wegen $c = c(c \leq (a \leq b)) = c(a \leq b)$ den Sätzen des Kontexts hinzugefügt werden. Führt die Implikation auf einen Widerspruch: $b=0$, dann geht $c(a \leq b)$ über in $c \neg a$.

Die Schachtelung lässt sich beliebig tief staffeln:

```
E1, E2, ..., En → E
┌──────────────────────
│ E1
│ E2
│ ...
│ En
│ S1
│ S2
│ ...
│ Sm
│ F1, F2, ..., Fn → F
│ ┌────────────────────
│ │ F1                                          Unterbeweis
│ │ F2
│ │ ...
│ │ Fn
│ │ T1
│ │ T2
│ │ ...
│ │ Tm
│ │ F
│ F1∧F2∧ ... ∧Fn ≤ F
│ E                              Aussage, zu deren Erschließung
│                               die Aussage „F1∧F2∧ ... ∧Fn ≤ F"
│                                              benötigt wurde.
└
E1∧E2∧ ... ∧En ≤ E
```

Unterbeweise finden in einem bestimmten Kontext, der durch die übergeordneten Beweisblöcke gegeben ist, statt: Alle Aussagen aus den übergeordneten Blöcken dürfen verwendet werden.

Auch das Beweisschema lässt sich mit Hilfe der Äquivalenztransformationen rechtfertigen (siehe den Hinweis zur Übung 2). Nun beweisen wir die *Verneinungsregel:*

```
p → ¬¬p
┌──────
│ p
│ ¬p → p ∧ ¬p
│ ┌────────────
│ │ ¬p
│ │ p ∧ ¬p
│ │
│ ¬¬p
└
p ≤ ¬¬p                                              (¬-i)
¬¬p → p
┌──────
│ ¬¬p
│ ¬p → ¬p ∧ ¬¬p
│ ┌────────────
│ │ ¬p
│ │ ¬p ∧ ¬¬p
│ p
¬¬p ≤ p
p = ¬¬p
```

Interessant ist die Herleitung der Tautologien b ∨ ¬b und ¬(b ∧ ¬b). Zunächst wird gezeigt, dass ¬(b ∧ ¬b) eine Tautologie ist:

```
→ ¬(b ∧ ¬b)
┌─────────
│ b ∧ ¬b → b ∧ ¬b
├─────────────────
│ │ b ∧ ¬b
│ ¬(b ∧ ¬b)
```

Nun folgt der Beweis, dass b ∨ ¬b eine Tautologie ist:

```
→ b ∨ ¬b
┌────────
│ ¬(b ∨ ¬b) → b ∧ ¬b
├──────────────────────
│ │ ¬(b ∨ ¬b)
│ │ b → (b ∨ ¬b) ∧ ¬(b ∨ ¬b)
│ ├──────────────────────────
│ │ │ b
│ │ │ b ∨ ¬b                                  (∨-i)
│ │ │ (b ∨ ¬b) ∧ ¬(b ∨ ¬b)                    (∧-i)
│ │ ¬b                                        (¬-i)
│ │ ¬b → (b ∨ ¬b) ∧ ¬(b ∨ ¬b)
│ ├──────────────────────────
│ │ │ ¬b
│ │ │ b ∨ ¬b                                  (∨-i)
│ │ │ (b ∨ ¬b) ∧ ¬(b ∨ ¬b)                    (∧-i)
│ │ b                                         (¬-e)
│ │ b ∧ ¬b                                    (∧-i)
│ b ∨ ¬b                                      (¬-e)
```

Übung 1: Im Deduktionssystem gibt es keine Konstanten für falsch und wahr. Eine Aussage, aus der sich jede beliebige Aussage folgern lässt - z. B. a ∧ ¬a - heißt *logisch falsch*. Und eine *logisch wahre Aussage* ist eine Aussage, die aus jeder beliebigen Aussage abgeleitet werden kann - z. B. a ∨ ¬a. Zeigen Sie, dass im Deduktionssystem die logisch falschen Aussagen die Rolle der Konstanten 0 (bzw. FALSE) und die logisch wahren Aussagen die Rolle der Konstanten 1 (bzw. TRUE) spielen. Es genügt, die Formeln a ∧ 0 = 0, a ∧ 1 = a, a∨ 0 = a und a∨ 1 = 1 abzuleiten.

Bei der natürlichen Deduktion können Umformungsprogramme hilfreich sein. Zum Lösen von Logik-Puzzles nehme ich das Programm `LogTrans`.

Übung 2 (The Tardy Bus Problem): Gegeben sind folgende Prämissen:

(a) Wenn Bill den Bus nimmt, wird er seine Verabredung verpassen, falls der Bus zu spät kommt.

(b) Bill sollte nicht nach Hause gehen, falls er die Verabredung verpasst und deprimiert ist.

(c) Wenn Bill den Job nicht kriegt, ist er deprimiert und er sollte nicht nach Hause gehen.

Welche der folgenden *Vermutungen* stimmen (sind Konklusionen aus den Prämissen)?

1. Wenn Bill den Bus nimmt, dann kriegt er den Job, falls der Bus zu spät kommt.
2. Bill kriegt den Job, falls er seine Verabredung verpasst und nach Hause gehen sollte.

3. Falls der Bus zu spät kommt, dann nimmt Bill den Bus nicht oder er verpasst seine Verabredung, falls er den Job nicht kriegt. (Achtung: Diese Vermutung ist zweideutig formuliert. Eindeutig wird's, wenn Sie nach dem erste Komma eine Klammer aufmachen und diese mit dem zweiten Komma wieder schließen.)
4. Bill nimmt nicht den Bus, falls der Bus zu spät kommt und er den Job nicht kriegt.
5. Falls Bill seine Verabredung nicht verpasst, bekommt er den Job nicht und er sollte nicht nach Hause gehen.
6. Bill ist deprimiert, falls der Bus zu spät kommt oder er seine Verabredung verpasst.
7. Falls Bill den Job kriegt, dann ist Bill nicht deprimiert oder er sollte nicht nach Hause gehen.
8. Falls Bill nach Hause gehen sollte und er den Bus nimmt, dann ist er nicht deprimiert, falls der Bus zu spät kommt.

Hinweis: Codieren Sie die Teilaussagen „Bill nimmt den Bus“ (B), „Bill verpasst seine Verabredung“ (A), „der Bus kommt zu spät“ (L), „Bill sollte nach Hause gehen“ (H), „Bill kriegt den Job“ (J) und „Bill ist deprimiert“ (D) und formulieren Sie damit die Prämissen. Zum Beweis der Konklusion der Form $a \leq b$ gehen Sie vor wie im Beweisschema: Bezeichne p die Konjunktion der Prämissen, dann ist $p \leq (a \leq b)$ gleichbedeutend mit $p \wedge a \leq b$. Bilden Sie also erst die minimierte DNF von $p \wedge a$ und zeigen Sie dann, dass diese b impliziert. Zur Widerlegung einer Vermutung genügt es, einen Fall anzugeben, in dem die Prämissen richtig sind und die Vermutung gleichzeitig falsch ist. Die Vermutung ist also widerlegt, wenn es einen Fall gibt (dargestellt als Term), aus dem sich die Ausdrücke p, a und $\neg b$ ableiten lassen; damit ist dann gezeigt, dass der Ausdruck $p \wedge a \wedge \neg b$ einen nichtleeren Spielraum hat. Übrigens: Die Vermutungen 2, 3 und 8 lassen sich ableiten, alle anderen sind zu widerlegen. Sie dürfen das Äquivalenztransformationssystem als bewiesen annehmen.

Literaturhinweise

Das System der natürlichen Deduktion wird hier ganz in Anlehnung an das vorzügliche Lehrbuch von David Gries entwickelt. Auch Douglas Hofstadter stellt die wichtigsten Dinge in seinem populären „GEB“ präzise dar (Kapitel „Die Aussagenlogik“, S. 198 ff.):

Gries, D.: The Science of Programming. Springer-Verlag Heidelberg 1981

Hofstadter, D. R.: Gödel, Escher, Bach - ein Endloses Geflochtenes Band. Klett-Cotta Stuttgart 1988

Übersichtscharakter hat der Abschnitt über Logik im Mathematik-Lehrbuch von Dörfler/Peschek. Dasselbe gilt für das gesamte - breit angelegte - Buch von Bläsius/Bürckert, das Beiträge aus vielen Gebieten und Forschungsrichtungen enthält. Das Buch von Schmid und Kindsmüller liegt im Schnittbereich von KI und kognitiver Psychologie:

Bläsius, K. H.; Bürckert, H.-J. (Herausgeber): Deduktionssysteme - Automatisierung des logischen Denkens. Oldenbourg, München 1992

Dörfler, W.; Peschek, W.: Einführung in die Mathematik für Informatiker. Hanser, München, Wien 1988

Schmid, U.; Kindsmüller, M. Ch.: Kognitive Modellierung. Eine Einführung in die logischen und algorithmischen Grundlagen. Spektrum Akademischer Verlag. Hochschultaschenbuch. Heidelberg, Berlin, Oxford 1996

Die Werke von Loeckx/Sieber, Lorenzen und Hermes bringen die Grundlagen der Logik auf hohem mathematischen Niveau und sind in sich abgeschlossen. Sie können für diejenigen interessant werden, die ihre Diplomarbeit auf dem Gebiet der Informationstechnik schreiben wollen:

Hermes, H.: Einführung in die mathematische Logik. Teubner, Stuttgart 1991

Loeckx, J.; Sieber, K.: The Foundations of Program Verification. Wiley-Teubner Series in Computer Science, Stuttgart 1987

Lorenzen, P.: Formale Logik. Walter de Gruyter, Berlin 1970

Tafel 2.7-1 Die grundlegenden Schlussregeln der natürlichen Deduktion

Erläuterung: Die Bezeichner E bzw. E1, E2, E3, usw. stehen für wohlgeformte Ausdrücke (Expressions)

$\dfrac{E1,\ E2,\ \ldots,\ En}{E1 \wedge E2 \wedge \ldots \wedge En}$	(∧-i)
$\dfrac{E1 \wedge E2 \wedge \ldots \wedge En}{Ei}$	(∧-e)
$\dfrac{Ei}{E1 \vee E2 \vee \ldots \vee En}$	(∨-i)
$\dfrac{E1 \vee E2 \vee \ldots \vee En,\ E1 \leq E,\ E2 \leq E,\ \ldots,\ En \leq E}{E}$	(∨-e)
$\dfrac{E \rightarrow E1 \wedge \neg E1}{\neg E}$	(¬-i)
$\dfrac{\neg E \rightarrow E1 \wedge \neg E1}{E}$	(¬-e)
$\dfrac{E1 \leq E2,\ E2 \leq E1}{E1 = E2}$	(=-i)
$\dfrac{E1 = E2}{E1 \leq E2,\ E2 \leq E1}$	(=-e)
$\dfrac{E1,\ E2,\ \ldots,\ En \rightarrow E}{E1 \wedge E2 \wedge \ \ldots\ \wedge En \leq E}$	(≤-i)
$\dfrac{E1 \leq E2,\ E1}{E2}$	(≤-e)

Sachverzeichnis

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z