

Skript
Vorkurs Informatik
Sommersemester 2024

PD Dr. David Sabel
Conrad Rau
Ronja Düffel
Stand: 22. März 2024

Inhaltsverzeichnis

I. Digitaltechnik	5
1. Codes	9
1.1. Dualcode	9
1.2. Alphanumerische Codes	10
2. Schaltalgebra	13
2.1. Schaltvariablen und Schaltfunktionen	13
2.1.1. Grundverknüpfungen und ihre Darstellung	14
2.2. Darstellung von Schaltfunktionen	16
2.2.1. Wahrheitstabelle	16
2.2.2. Funktionsgleichung	17
Vorrang- und Klammerregeln	17
2.2.3. Darstellung mit Schaltsymbolen	18
Symbolische Darstellung logischer Gatter	18
2.3. Gesetze der Schaltalgebra	20
2.4. Weitere Verknüpfungen	22
2.4.1. Vollständige Operatorensysteme	24
2.5. Normalformen	25
2.5.1. Min- und Maxterme	26
2.6. KV-Diagramme	29
2.7. Minimierung von Schaltfunktionen	32
2.7.1. Minimierung mit den Gesetzen der Schaltalgebra	34
2.7.2. Minimierung mit KV-Diagrammen	34
Aufstellung einer minimalen DNF	38
2.8. Unvollständig spezifizierte Funktionen	39
2.9. Schaltungsentwurf	40
II. Benutzung von Unix-Systemen, Einführung in das Funktionale Programmieren	43
3. Einführung in die Bedienung von Unix-Systemen	45
3.1. Unix und Linux	45
3.1.1. Dateien und Verzeichnisse	46
3.1.2. Login und Shell	47
3.1.3. Befehle	48
3.1.4. History und Autovervollständigung	51
3.2. Editieren und Textdateien	52

4. Programmieren und Programmiersprachen	55
4.1. Programme und Programmiersprachen	55
4.1.1. Imperative Programmiersprachen	56
4.1.2. Deklarative Programmiersprachen	57
4.2. Haskell: Einführung in die Benutzung	58
4.2.1. GHCi auf den Rechnern der RBI	59
4.2.2. GHCi auf dem eigenen Rechner installieren	59
4.2.3. Bedienung des Interpreters	60
4.2.4. Quelltexte erstellen und im GHCi laden	61
4.2.5. Kommentare in Quelltexten	63
4.2.6. Fehler	64
5. Grundlagen der Programmierung in Haskell	67
5.1. Ausdrücke und Typen	67
5.2. Basistypen	69
5.2.1. Wahrheitswerte: Der Datentyp <code>Bool</code>	69
5.2.2. Ganze Zahlen: <code>Int</code> und <code>Integer</code>	71
5.2.3. Gleitkommazahlen: <code>Float</code> und <code>Double</code>	71
5.2.4. Zeichen und Zeichenketten	72
5.3. Funktionen und Funktionstypen	73
5.4. Einfache Funktionen definieren	78
5.5. Rekursion	84
5.6. Listen	91
5.6.1. Listen konstruieren	92
5.6.2. Listen zerlegen	94
5.6.3. Einige vordefinierte Listenfunktionen	96
5.6.4. Nochmal Strings	97
5.7. Paare und Tupel	98
5.7.1. Die Türme von Hanoi in Haskell	99
A. Anmeldung an RBI-Rechner	101
A.1. Voraussetzung	101
A.2. Vom PC in den Poolräumen	101
A.3. Vom eigenen PC aus	102
A.4. Vom Vorkurs-Laptop aus	103

Teil I.

Digitaltechnik

Ronja Düffel

Als eines der Hauptgebiete der Informatik beschäftigt sich die Technische Informatik mit Architektur, Entwurf, Realisierung, Bewertung und Betrieb von Rechnersystemen auf der Ebene der Hardware und der systemnahen Software.

Computer sind digitale Systeme. Das heißt, dass Informationen in wertdiskreten Zuständen codiert und verarbeitet werden. In der binären Digitaltechnik wie sie in Computern zur Anwendung kommt, gibt es nur zwei Zustände, nämlich 0 und 1 oder auch *false* und *true*. Technisch sind diese im Computer als unterschiedliche Spannungspegel realisiert. Liegt die Spannung im hohen Bereich, so liegt Zustand *H* vor, im unteren Bereich Zustand *L*. Ist der hohe Spannungspegel *H* dem 1-Zustand zugeordnet ($H \leftarrow 1$) und der niedrige 0 ($L \leftarrow 0$), so spricht man von *positiver Logik*. Der umgekehrte Fall, $H \leftarrow 0, L \leftarrow 1$ heißt *negative Logik*

Das heißt, für die Darstellung von Informationen im Computer stehen lediglich zwei Werte zur Verfügung. Um also nun Zahlen und Schriftzeichen darstellen zu können, müssen diese codiert werden.

positive
und
negative
Logik

1. Codes

Ein Code bildet Zeichen eines Zeichenvorrats auf die Zeichen eines anderen Zeichenvorrats ab. Ein bekanntes Beispiel ist der Morsecode, der Buchstaben auf Kombinationen von kurzen und langen Signalen (Licht, Ton, etc) abbildet. In digitalen Systemen kommen Dualcodes zum Einsatz.

1.1. Dualcode

Dualcodes beruhen auf nur zwei Zeichen, nämlich 0 und 1. Daher spricht man auch von *Binärcode*. Wie im Dezimalsystem auch, werden größere Zahlen durch Aneinanderreihung der zur Verfügung stehenden Zeichen gebildet, wobei die Position des Zeichens bestimmt, wieviel es zum Gesamtwert beiträgt. Man spricht auch von gewichteten Codes, da weiter links stehende Zeichen mehr Gewicht haben.

Binärcode

Eine Dual- oder Binärzahl Z_2 besteht aus einer Aneinanderreihung von Zeichen ($z_i \in \{0, 1\}$). Die Zeichen z_i eines Wortes werden in der Digitaltechnik *Bits* genannt. Ein Bit ist die kleinste Informationseinheit. Die nächst größere ist ein *Byte*. Ein Byte besteht aus 8 Bit.

Bit
Byte

Zurück zu den Binärzahlen. Eine Binärzahl mit n Stellen, sieht also wie folgt aus:

$$Z_2 = z_{n-1}z_{n-2}z_{n-3} \dots z_1z_0$$

Die Position zählt also von rechts nach links hoch. Da das erste (rechtste!) Zeichen die Position 0 hat, hat das n -te Zeichen die Position $n - 1$.

Den einzelnen Bits werden entsprechend ihrer Position i in der Zahl Gewichte 2^i zugeordnet. Damit kann die äquivalente Dezimalzahl Z_{10} berechnet werden:

$$Z_{10} = g(Z_2) = z_{n-1} \cdot 2^{n-1} + z_{n-2} \cdot 2^{n-2} + \dots + z_1 \cdot 2^1 + z_0 \cdot 2^0$$

Das sieht für manche jetzt vielleicht, wegen der mathematischen Notation, kompliziert aus. Es ist aber eigentlich nichts anderes, als das was wir von den Dezimalzahlen, mit denen wir alltäglich umgehen, kennen. Der Umgang mit Dezimalzahlen ist uns nur so geläufig, dass wir uns dieser Interpretation der Zahlen nicht mehr bewusst sind.

Beispiel 1.1.

Die Dezimalzahl 243_{10} , die durch den Index 10 als solche gekennzeichnet ist, hat den Wert $2 \cdot 10^2 + 4 \cdot 10^1 + 3 \cdot 10^0 = 2 \cdot 100 + 4 \cdot 10 + 3 \cdot 1 = 243$. Die Dezimalzahl 324_{10} besteht zwar aus den gleichen Zeichen, repräsentiert aber einen anderen Wert, nämlich $3 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 = 300 + 20 + 4 = 324$.

1. Codes

Im Dezimalsystem stehen zehn Zeichen zur Verfügung: 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9. Die Basis ist daher 10. Das garantiert, dass alle Werte dargestellt werden können und die Darstellung eindeutig ist. Eindeutigkeit heißt, dass eine Zahl nur auf eine Art und Weise dargestellt werden kann.

Im Dual- oder Binärsystem stehen zwei Zeichen zur Verfügung und die Basis ist dementsprechend 2. Um eine Dualzahl in eine Dezimalzahl umzurechnen, interpretieren wir sie genau so, wie zuvor die Dezimalzahl, allerdings zur Basis 2.

Beispiel 1.2.

Die Dualzahl 11110011_2 , gekennzeichnet durch den Index 2, steht also für den Dezimalwert $1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 243_{10}$

Um eine Dezimalzahl in eine Dualzahl umzuwandeln, teilt man durch 2 und notiert die Reste, von rechts nach links (**Achtung:** das ist entgegen der gewohnten Schreibrichtung!). Für die Dezimalzahl 324_{10} ergibt sich z.B.:

$$\begin{array}{rll} 324 : 2 = & 162 & \text{Rest 0} \\ & 162 : 2 = & 81 & \text{Rest 0} \\ & & 81 : 2 = & 40 & \text{Rest 1} \\ & & & 40 : 2 = & 20 & \text{Rest 0} \\ & & & & 20 : 2 = & 10 & \text{Rest 0} \\ & & & & & 10 : 2 = & 5 & \text{Rest 0} \\ & & & & & & 5 : 2 = & 2 & \text{Rest 1} \\ & & & & & & & 2 : 2 = & 1 & \text{Rest 0} \\ & & & & & & & & 1 : 2 = & 0 & \text{Rest 1} \end{array}$$

Die Dualzahl $101000100_2 = 324_{10}$

1.2. Alphanumerische Codes

Um nun Buchstaben, Ziffern und Sonderzeichen darstellen zu können, gibt es alphanumerische Codes. Ein sehr bekannter ist der ASCII-Code. Hierbei wird jedes Zeichen mit 7 Bit ($a_6 a_5 a_4 a_3 a_2 a_1 a_0$) codiert. Die ersten drei Bit a_6, a_5 und a_4 geben die Spalte, die letzten vier Bit a_3, a_2, a_1 und a_0 geben die Zeile der Codierungstabelle an (siehe Abb.: 1.1). Die ASCII-Tabelle hat $2^7 = 128$ Zeichen und umfasst die Zeichen einer englischsprachigen Tastatur. D.h. deutsche Umlaute können mit ASCII nicht codiert werden. Im Folgenden interessieren uns aber erstmal nur Zahlen.

1.2. Alphanumerische Codes

Bits					0	0	0	0	1	1	1	1				
b ₇	b ₆	b ₅	b ₄	b ₃	0	0	0	0	1	1	1	1				
b ₄	b ₃	b ₂	b ₁	Column	0	1	2	3	4	5	6	7				
Row	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p				
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q				
0	0	1	0	2	STX	DC2	"	2	B	R	b	r				
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s				
0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t				
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u				
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v				
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w				
1	0	0	0	8	BS	CAN	(8	H	X	h	x				
1	0	0	1	9	HT	EM)	9	I	Y	i	y				
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z				
1	0	1	1	11	VT	ESC	+	;	K	[k	{				
1	1	0	0	12	FF	FS	,	<	L	\	l					
1	1	0	1	13	CR	GS	-	=	M]	m	}				
1	1	1	0	14	SO	RS	.	>	N	^	n	~				
1	1	1	1	15	SI	US	/	?	O	_	o	DEL				

Abbildung 1.1.: ASCII-Tabelle, von an unknown officer or employee of the United States Government - MIL-STD-188-100, pg. B-2, Fig 1, 1972. (different scan), Gemein- frei, <https://commons.wikimedia.org/w/index.php?curid=63485656>

2. Schaltalgebra

Unter einer Algebra versteht man in der Mathematik eine Menge von Elementen und Verknüpfungen auf dieser Menge (z.B. die Menge der ganzen Zahlen und die Verknüpfungen der Multiplikation und Addition $(\mathbb{Z}, \cdot, +)$). Die Schaltalgebra als Spezialfall der *bool'schen Algebra*, liefert die theoretische Grundlage der Digitaltechnik. Mit ihr können digitale Schaltungen und Schaltnetze entworfen und beschrieben werden.

bool'sche Algebra

Die Schaltalgebra ist auf der zweielementigen Menge $\mathcal{B} = \{0, 1\}$ definiert. Es existieren die drei Grundverknüpfungen (*Operatoren*) \wedge , \vee und \neg , für die entsprechende Schaltzeichen (Logik-Gatter) eingeführt werden (siehe Tab. 2.1).

Operatoren
Logik-
Gatter

Operator	Schaltzeichen	Benennung
\neg		NICHT - Gatter
\wedge		UND - Gatter
\vee		ODER - Gatter

Tabelle 2.1.: Operatoren und Schaltzeichen nach IEC 60617-12 : 1997

2.1. Schaltvariablen und Schaltfunktionen

Eine Schaltvariable ist eine binäre Variable, d.h. sie kann einen von zwei Werten annehmen, nämlich 0 oder 1. Mit den Verknüpfungen \wedge , \vee , \neg können Funktionen aufgestellt werden. Wie in der (allgemeinen) Algebra, die man aus der Schule kennt, beschreibt eine Funktion der Schaltalgebra die Abhängigkeit einer Ausgangsvariablen y von einer oder mehreren unabhängigen, binären Schaltvariablen a, b, c, \dots

Schalt-
variable

Beispiel 2.1.

Funktion der allgemeinen Algebra:

$$f(x) = 5x + 3 \text{ oder } y = 5x + 3$$

Funktion der Schaltalgebra:

$$f(a, b) = a \wedge b \text{ oder } y = a \wedge b$$

2. Schaltalgebra

Definition 2.2 (Schaltfunktion).

Schaltfunktion

Eine Schaltfunktion ist eine Gleichung der Schaltalgebra, die die Abhängigkeit einer oder mehrerer, binärer Schaltvariablen y (Ausgangsvariable(n)) von einer oder mehreren, unabhängigen binären Schaltvariablen x beschreibt. Handelt es sich um mehrere Ein- und Ausgangsvariablen, so sind x und y Vektoren. Für mehrere Eingangsvariablen schreiben wir $x = (a, b, c, \dots)$, für mehrere Ausgangsvariablen $y = (y_1, y_2, y_3, \dots)$.

Beispiel 2.3.

Die Addition zweier 4 Bit Zahlen kann als Schaltfunktion mit einem Eingabevektor der Länge 8 und einem Ausgabevektor der Länge 5 realisiert werden. Hierbei entsprechen die ersten 4 Bit des Eingabevektors dem ersten Summanden, und die zweiten 4 Bit entsprechen dem zweiten Summanden.

Bemerkung: Im Vorsemesterkurs werden wir uns auf Schaltfunktionen mit nur einer Ausgangsvariablen beschränken.

Definition 2.4 (Stelligkeit einer Schaltfunktion).

Stelligkeit

Die *Stelligkeit* einer Schaltfunktion ist die Anzahl ihrer Eingangsvariablen.

Die Funktion $f(a) = \bar{a}$ beispielsweise ist eine einstellige Funktion. Bei der Funktion $g(a, b, c)$ handelt es sich um eine dreistellige Funktion, $h(a, b, c, d, e)$ ist eine 5-stellige Funktion.

Definition 2.5 (Belegung einer Schaltvariablen).

Belegung

Unter der *Belegung einer Schaltvariablen* versteht man die Zuweisung eines konkreten Wertes (0 oder 1) an eine Schaltvariable.

Die Belegung einer mehrstelligen Funktion kann als Vektor angegeben werden. Beispielsweise bedeutet die Belegung $(1, 1, 0)$ für die Schaltfunktion $g(a, b, c)$, dass a der Wert 1, b der Wert 1 und c der Wert 0 zugewiesen wird.

2.1.1. Grundverknüpfungen und ihre Darstellung

Für die Grundverknüpfungen der Schaltalgebra \wedge , \vee und \neg gibt es drei gleichwertige Darstellungen.

- Wahrheitstabelle,
- Schaltzeichen und
- Funktion

Name	Wahrheitstabelle	Schaltzeichen	Funktion															
NICHT, NOT, Komplement, Negation	<table border="1"> <tr> <td>a</td> <td>\bar{a}</td> </tr> <tr> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> </tr> </table>	a	\bar{a}	0	1	1	0		$y = f(a) = \bar{a}$									
a	\bar{a}																	
0	1																	
1	0																	
UND, AND, Konjunktion	<table border="1"> <tr> <td>a</td> <td>b</td> <td>$a \wedge b$</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	a	b	$a \wedge b$	0	0	0	0	1	0	1	0	0	1	1	1		$y = f(a, b) = a \wedge b$
a	b	$a \wedge b$																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
ODER, OR, Disjunktion	<table border="1"> <tr> <td>a</td> <td>b</td> <td>$a \vee b$</td> </tr> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </table>	a	b	$a \vee b$	0	0	0	0	1	1	1	0	1	1	1	1		$y = f(a, b) = a \vee b$
a	b	$a \vee b$																
0	0	0																
0	1	1																
1	0	1																
1	1	1																

Tabelle 2.2.: Grundverknüpfungen und ihre Darstellung

Definition 2.6 (Gleichheit von Schaltfunktionen).

Wir bezeichnen zwei Schaltfunktionen als gleich, wenn sie für dieselbe Belegung der Eingangsvariablen, dasselbe Ergebnis in Bezug auf den Wert der Ausgangsvariablen y haben.

Beispiel 2.7.

Die Funktionen $f(a, b) = a \wedge b$ und $g(a, b) = \overline{\bar{a} \vee \bar{b}}$ sind gleich. Das lässt sich leicht mit einer Wahrheitstabelle überprüfen.

a	b	$f(a, b) = a \wedge b$	\bar{a}	\bar{b}	$\bar{a} \vee \bar{b}$	$g(a, b) = \overline{\bar{a} \vee \bar{b}}$
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

Für alle Belegungen von a und b sind die Funktionswerte der Schaltfunktionen f und g gleich. $f(a, b) = 0$ genau dann, wenn $g(a, b) = 0$ und $f(a, b) = 1$ genau dann, wenn $g(a, b) = 1$.

Die Funktionen $f(a, b) = a \wedge b$ und $g(a, b) = a \vee b$ sind nicht gleich, denn für die Belegung $a = 0$ und $b = 1$ gilt: $f(a, b) = 0$ und $g(a, b) = 1$, wie man folgender Wahrheitstabelle entnehmen kann.

2. Schaltalgebra

a	b	$f(a, b) = a \wedge b$	$g(a, b) = a \vee b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

2.2. Darstellung von Schaltfunktionen

Es gibt vier gleichwertige Darstellungsformen von Schaltfunktionen:

- Wahrheitstabelle
- Funktionsgleichung
- Schaltzeichen
- KV-Diagramm

2.2.1. Wahrheitstabelle

Wahrheitstabellen haben wir oben bereits benutzt, um Schaltfunktionen darzustellen, hier führen wir sie ausführlicher ein. Die Wahrheitstabelle ist eine Tabelle, die alle möglichen Belegungskombinationen der Eingangsvariablen und den, durch die Funktion zugeordneten Wert der Ausgangsvariablen y enthält. Eine Wahrheitstabelle gibt also für jede Wertekombination der Eingangsvariablen den zugehörigen Funktionswert der Ausgangsvariablen y an. Da es bei n binären Eingangsvariablen 2^n Kombinationsmöglichkeiten gibt, hat die Wahrheitstabelle einer Funktion mit n Eingangsvariablen 2^n Zeilen. So hat die Wahrheitstabelle der 3-stelligen Funktion $f(a, b, c)$ in Beispiel 2.8, $n = 3$ Eingangsvariablen (a, b und c) und somit $2^3 = 8$ Zeilen.

Ist die Schaltfunktion, für die man eine Wahrheitstabelle aufstellt, komplexer, kann es sinnvoll sein, die Wahrheitswerte einzelner Terme ebenfalls in der Wahrheitstabelle anzugeben (siehe Bsp. 2.9).

Beispiel 2.8.

Die Wahrheitstabelle der 3-stelligen Schaltfunktion $f(a, b, c)$, die genau dann den Wert 1 hat, wenn eine ungerade Anzahl an Eingangsvariablen den Wert 1 hat, sieht folgendermaßen aus:

a	b	c	$y = f(a, b, c)$
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	1

Tabelle 2.3.: Wahrheitstabelle für 3-stellige Schaltfunktion $f(a, b, c)$.**2.2.2. Funktionsgleichung**

Jede binäre Schaltfunktion kann allein durch die Grundverknüpfungen \wedge , \vee und $\bar{}$ dargestellt werden.

Beispiel 2.9.

$$y = g(a, b, c) = (a \vee b) \wedge (\overline{a \wedge \bar{c}}) \vee (b \wedge c)$$

a	b	c	\bar{c}	$a \vee b$	$a \wedge \bar{c}$	$\overline{a \wedge \bar{c}}$	$(a \vee b) \wedge (\overline{a \wedge \bar{c}})$	$b \wedge c$	$g(a, b, c)$
0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	0	0	0
0	1	0	1	1	0	1	1	0	1
0	1	1	0	1	0	1	1	1	1
1	0	0	1	1	1	0	0	0	0
1	0	1	0	1	0	1	1	0	1
1	1	0	1	1	1	0	0	0	0
1	1	1	0	1	0	1	1	1	1

Vorrang- und Klammerregeln

Wie in der allgemeinen Algebra, gibt es auch in der Schaltalgebra bei der Auswertung der Ausdrücke Vorrangregeln zu beachten. Es gilt Negation, vor Konjunktion, vor Disjunktion, also $\bar{}$, vor \wedge , vor \vee , analog zu ‘‘Punkt- vor Strichrechnung’’ in der allgemeinen Algebra. Wenn Formeln durch die Verknüpfung anderer Formeln entstehen, sollten Klammern gesetzt werden. Manchmal ist das Setzen von Klammern unerlässlich (siehe Beispiel 2.10)

Bei direkten UND-Verknüpfungen einzelner Variablen, wird häufig das Verknüpfungszeichen \wedge der besseren Lesbarkeit wegen weggelassen. Statt $a \wedge \bar{b} \wedge c$ schreibt man $a\bar{b}c$.

2. Schaltalgebra

Beispiel 2.10.

Betrachten wir die drei Schaltfunktionen f, g und h und die zugehörige Wahrheitstabelle (Tab. 2.4):

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

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

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

a	b	c	$b \wedge c$	$f(a, b, c) = a \vee b \wedge c$	$g(a, b, c) = a \vee (b \wedge c)$	$a \vee b$	$h(a, b, c) = (a \vee b) \wedge c$
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	1	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	0
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	0
1	1	1	1	1	1	1	1

Tabelle 2.4.: Wahrheitstabelle für die Schaltfunktionen $f(a, b, c)$, $g(a, b, c)$ und $h(a, b, c)$.

Die Schaltfunktionen f und g sind gleich, da die Auswertung von f aufgrund der Vorrangregeln sowieso so erfolgt, wie in g durch die Klammern vorgegeben. Insofern sind die Klammern in g überflüssig, sollten aber trotzdem gesetzt werden, um Fehler zu vermeiden. Funktion h jedoch ist nicht gleich zu den Funktionen f und g , da hier durch die Klammerung die ODER-Verknüpfung vor der UND-Verknüpfung ausgewertet werden muss.

Ein analoges Beispiel aus der allgemeinen Algebra wären die Gleichungen

$$x = 3 + (5 \cdot 2) = 3 + 10 = 13 \text{ und}$$

$$y = (3 + 5) \cdot 2 = 8 \cdot 2 = 16.$$

2.2.3. Darstellung mit Schaltsymbolen

Alle Verknüpfungen können auch durch Schaltsymbole dargestellt werden (vergl. Tab. 2.1). Also lassen sich auch alle Schaltfunktionen mit Hilfe von Schaltsymbolen als Schaltung darstellen. Die technischen Bauteile die die jeweiligen Verknüpfungen darstellen, nennt man *Gatter*. Sie können technisch auf unterschiedliche Arten realisiert werden. Wie, erfahren Sie in der Lehrveranstaltung "Rechnertechnologie und kombinatorische Schaltungen" (RTKS).

Gatter

Symbolische Darstellung logischer Gatter

Die Darstellung logischer Gatter ist in IEC 60617-12¹ geregelt. Logische Gatter werden durch ein hochkant stehendes Rechteck dargestellt. Die Funktion des Gatters wird durch ein Symbol innerhalb des Rechtecks angegeben. $\&$ steht für UND (Konjunktion, \wedge), ≥ 1 für ODER (Disjunktion, \vee). Das Zeichen für ODER behält man besser, wenn man sich klar macht, dass

¹International Electrotechnical Commission Standard

2.2. Darstellung von Schaltfunktionen

eine Disjunktion immer dann zu 1 auswertet, wenn mindestens eine der Eingangsvariablen den Wert 1 hat.

Das Gatter für die Negation (\neg) hat als Symbol im Rechteck eine 1 und am Ausgang einen kleinen Kreis.

Die Eingänge sind links, die Ausgänge rechts der Symbole angeordnet (siehe Abb. 2.1).

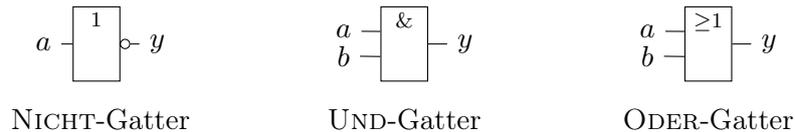


Abbildung 2.1.: Schaltsymbole der Grundverknüpfungen \neg , \wedge und \vee .

Bemerkung: Dies sind die Symbole, wie sie im deutschen Sprachraum üblich sind. In englischsprachigen Texten finden sich häufig die amerikanischen Symbole, wie sie in ANSI/IEEE Std 91/91a-1991 geregelt sind.²

Durch Kombination der Symbole lassen sich nun Schaltfunktionen darstellen

Beispiel 2.11.

Die Schaltfunktion $f(a, b) = (\bar{a} \vee b)$ kann durch Kombination eines NICHT-Gatters und eines ODER-Gatters dargestellt werden (Abb.: 2.2).

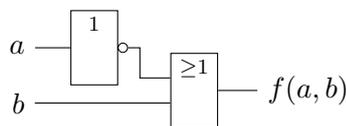


Abbildung 2.2.: Schaltung für $f(a, b) = (\bar{a} \vee b)$

Beispiel 2.12.

Die Schaltfunktion $g(a, b, c) = (a \wedge \bar{c}) \vee (\bar{b} \wedge c)$ kann durch Kombination zweier NICHT-Gatter, zweier UND-Gatter und eines ODER-Gatters dargestellt werden (siehe Abb.: 2.3).

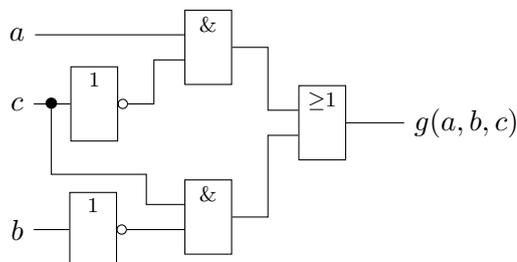


Abbildung 2.3.: Schaltung für $g(a, b, c) = (a \wedge \bar{c}) \vee (\bar{b} \wedge c)$

²Einen guten Überblick liefert die Wikipediaseite: <https://de.wikipedia.org/wiki/Logikgatter>

2. Schaltalgebra

Bei der Darstellung der Schaltfunktion als Schaltung, muss natürlich ebenfalls auf die Auswertungsreihenfolge der Verknüpfungen geachtet werden. Die Auswertung der Signale in einer logischen Schaltung erfolgt von links nach rechts. Also müssen Terme, die zuerst ausgewertet werden müssen, weiter links stehen, als solche, die später ausgewertet werden.

Beispiel 2.13.

Betrachten wir die Schaltfunktionen g und h aus Beispiel 2.10.

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

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

In Funktion g muss der Term $b \wedge c$ vor der ODER-Verknüpfung mit a ausgewertet werden. Die Schaltung dazu zeigt Abbildung 2.4.

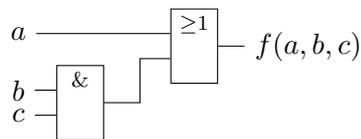


Abbildung 2.4.: Schaltung für $g(a, b, c) = a \vee (b \wedge c)$

In Funktion h muss der Term $a \vee b$ vor der UND-Verknüpfung mit c ausgewertet werden. Abbildung 2.5 zeigt die Schaltung zu Funktion $h(a, b, c)$.

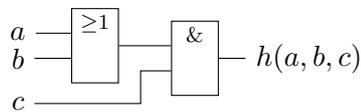


Abbildung 2.5.: Schaltung für $h(a, b, c) = (a \vee b) \wedge c$

2.3. Gesetze der Schaltalgebra

Wie für die allgemeine Algebra gibt es auch für die Schaltalgebra einige Umformungsregeln und Gesetze.

Tip: Wenn man sich \wedge als Multiplikation und \vee als Addition vorstellt, kommen einem viele der Gesetze gar nicht neu vor. (**Achtung:** Das gilt nicht für alle Gesetze!)

Für Schaltvariablen a, b und c , gelten folgende Gesetze:

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. Absorptionsgesetze:

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

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

4. Distributivgesetze:

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

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

(Gilt nicht für \cdot und $+$)

5. Neutrale Elemente:

$$a \wedge 1 = a$$

$$a \vee 0 = a$$

6. Inverse Elemente:

$$a \wedge \bar{a} = 0$$

$$a \vee \bar{a} = 1$$

7. Reduktionsgesetze:

$$a \wedge 0 = 0$$

$$a \vee 0 = a$$

$$a \vee 1 = 1$$

$$a \wedge 1 = a$$

8. Idempotenzgesetze:

$$a \wedge a = a$$

$$a \vee a = a$$

9. Resolutionsregeln:

$$(a \wedge b) \vee (a \wedge \bar{b}) = a$$

$$(a \vee b) \wedge (a \vee \bar{b}) = a$$

10. De Morgansche Regeln:

$$\overline{a \wedge b} = \bar{a} \vee \bar{b}$$

$$\overline{a \vee b} = \bar{a} \wedge \bar{b}$$

2.4. Weitere Verknüpfungen

Außer der bereits genannten Grundverknüpfungen AND, OR und NOT, kommen in der Digitaltechnik auch die Verknüpfungen NAND, NOR, XOR und XNOR vor. Die Bedeutung der jeweiligen Verknüpfungen lässt sich an der Wahrheitstabelle sehr gut ablesen. Besinnt man sich darauf, dass 1 für *true* und 0 für *false* stehen könnte, dann ist die Belegung der Funktionswerte auch sehr intuitiv.

- NICHT, NOT: Die Ausgangsvariable y ist genau dann 1, wenn die Eingangsvariable 0 ist.
- UND, AND: Die Ausgangsvariable y ist genau dann 1, wenn alle Eingangsvariablen 1 sind.
- ODER, OR: Die Ausgangsvariable y ist genau dann 1, wenn mindestens eine der Eingangsvariablen 1 ist.
- NAND: Steht für **Not And**, und y ist genau dann 1, wenn $a \wedge b = 0$ ist. Also, wenn mindestens eine Eingangsvariable 0 ist.
- NOR: Steht für **Not Or**, und y ist genau dann 1, wenn $a \vee b = 0$ ist. Also, wenn alle Eingangsvariablen 0 sind.
- XOR: Steht für **Exclusive Or**, und y ist genau dann 1, wenn eine ungerade Anzahl der Eingangsvariablen 1 ist. Gibt es nur zwei Eingangsvariablen, heißt das, dass genau eine der beiden 1 sein muss, damit y den Wert 1 hat.
- XNOR: Steht für **Exclusive Not Or**, und y ist genau dann 1, wenn eine gerade Anzahl an Eingangsvariablen 1 ist. Gibt es nur zwei Eingangsvariablen, kann man auch sagen, dass y genau dann 1 ist, wenn der Wert der beiden Eingangsvariablen identisch ist, also beide 1, oder beide 0.

2.4. Weitere Verknüpfungen

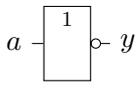
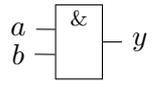
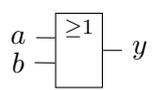
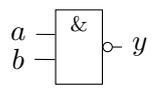
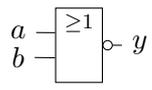
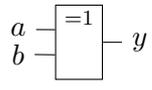
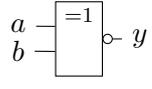
Name	Wahrheitstabelle	Schaltzeichen	Funktion															
NICHT, NOT, Komplement, Negation	<table border="1"> <tr><td>a</td><td>\bar{a}</td></tr> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </table>	a	\bar{a}	0	1	1	0		$y = f(a) = \bar{a}$									
a	\bar{a}																	
0	1																	
1	0																	
UND, AND, Konjunktion	<table border="1"> <tr><td>a</td><td>b</td><td>$a \wedge b$</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	a	b	$a \wedge b$	0	0	0	0	1	0	1	0	0	1	1	1		$y = f(a, b) = a \wedge b$
a	b	$a \wedge b$																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
ODER, OR, Disjunktion	<table border="1"> <tr><td>a</td><td>b</td><td>$a \vee b$</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	a	b	$a \vee b$	0	0	0	0	1	1	1	0	1	1	1	1		$y = f(a, b) = a \vee b$
a	b	$a \vee b$																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NAND	<table border="1"> <tr><td>a</td><td>b</td><td>$\overline{a \wedge b}$</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	$\overline{a \wedge b}$	0	0	1	0	1	1	1	0	1	1	1	0		$y = f(a, b) = \overline{a \wedge b}$
a	b	$\overline{a \wedge b}$																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR	<table border="1"> <tr><td>a</td><td>b</td><td>$\overline{a \vee b}$</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	$\overline{a \vee b}$	0	0	1	0	1	0	1	0	0	1	1	0		$y = f(a, b) = \overline{a \vee b}$
a	b	$\overline{a \vee b}$																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
XOR	<table border="1"> <tr><td>a</td><td>b</td><td>$a \oplus b$</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	a	b	$a \oplus b$	0	0	0	0	1	1	1	0	1	1	1	0		$y = f(a, b) = a \oplus b$
a	b	$a \oplus b$																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
XNOR	<table border="1"> <tr><td>a</td><td>b</td><td>$\overline{a \oplus b}$</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	a	b	$\overline{a \oplus b}$	0	0	1	0	1	0	1	0	0	1	1	1		$y = f(a, b) = \overline{a \oplus b}$
a	b	$\overline{a \oplus b}$																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

Tabelle 2.5.: Wichtige Verknüpfungen in der Digitaltechnik

2. Schaltalgebra

2.4.1. Vollständige Operatorensysteme

Ein vollständiges Operatorensystem ist eine Menge von Operatoren, die es erlauben, jede beliebige Schaltfunktion darzustellen. Die Operatoren \wedge , \vee und $\bar{}$ bilden ein vollständiges Operatorensystem. Ferner bilden sowohl die Verknüpfung NAND, als auch die Verknüpfung NOR ein vollständiges Operatorensystem.

So können Schaltungen mit nur einem Gattertyp (NAND oder NOR) realisiert werden. Die Verwendung eines einheitlichen Gattertyps hat Vorteile (keine Fehler aufgrund von Verwechslung, Lagerung, Einsparungen durch Mengenrabatt) aber auch Nachteile (Umwandlung der Schaltung, höhere Anzahl an Gattern in Schaltung, Schaltung schwieriger zu lesen).

Operatorensystem	Negation	Konjunktion	Disjunktion
$\wedge, \vee, \bar{}$	\bar{a}	$a \wedge b$	$a \vee b$
$\bar{\wedge}$	$\overline{a \wedge a}$	$\overline{(a \wedge b) \wedge (a \wedge b)}$	$\overline{(\bar{a} \wedge \bar{a}) \wedge (\bar{b} \wedge \bar{b})}$
$\bar{\vee}$	$\overline{a \vee a}$	$\overline{(a \vee a) \vee (b \vee b)}$	$\overline{(a \vee b) \vee (a \vee b)}$

Tabelle 2.6.: Übersicht Operatorensysteme und Darstellung der Grundverknüpfungen UND, ODER und NICHT.

Beweis 2.14.

Negation

a	\bar{a}	$\overline{a \wedge a}$	$\overline{a \vee a}$
0	1	1	1
1	0	0	0

Konjunktion

a	b	$a \wedge b$	$x = \overline{a \wedge b}$	$\overline{x \wedge x}$	$y = \overline{a \vee a}$	$z = \overline{b \vee b}$	$\overline{y \vee z}$
0	0	0	1	0	1	1	0
0	1	0	1	0	1	0	0
1	0	0	1	0	0	1	0
1	1	1	0	1	0	0	1

Disjunktion

a	b	$a \vee b$	$x = \overline{a \wedge a}$	$y = \overline{b \wedge b}$	$\overline{x \wedge y}$	$z = \overline{a \vee b}$	$\overline{z \vee z}$
0	0	0	1	1	0	1	0
0	1	1	1	0	1	0	1
1	0	1	0	1	1	0	1
1	1	1	0	0	1	0	1

Umwandlung in NAND- und NOR-Form

Um die Funktion mit den Operatoren NOT, AND und OR ausschließlich mit NAND- bzw. NOR-Operatoren darzustellen, können die Äquivalenzen in Tabelle 2.6 genutzt werden. Meist ist es aber einfacher und effizienter, wenn doppelte Negation und die De Morgan'sche Regel (vergl. 2.3) anwendet. Negationen einzelner Eingangsvariablen können dann mit der Äquivalenz aus Tabelle 2.6 ersetzt werden.

Beispiel 2.15.

Betrachten wir die vierstellige Schaltfunktion $f(a, b, c, d) = (\bar{a} \wedge \bar{b}) \vee (a \wedge b) \vee (c \wedge d)$.

Umwandlung in NAND-Form:

$$\begin{aligned}
 f(a, b, c, d) &= (\bar{a} \wedge \bar{b}) \vee (a \wedge b) \vee (c \wedge d) && \text{doppelte Negation} \\
 &= \overline{\overline{(\bar{a} \wedge \bar{b}) \vee (a \wedge b) \vee (c \wedge d)}} && \text{De Morgan} \\
 &= \overline{(\bar{a} \wedge \bar{b}) \wedge (a \wedge b) \wedge (c \wedge d)} && \text{Negation ersetzen} \\
 &= \overline{((\bar{a} \wedge \bar{a}) \wedge (\bar{b} \wedge \bar{b})) \wedge (a \wedge b) \wedge (c \wedge d)}
 \end{aligned}$$

Umwandlung in NOR-Form:

$$\begin{aligned}
 f(a, b, c, d) &= (\bar{a} \wedge \bar{b}) \vee (a \wedge b) \vee (c \wedge d) && \text{doppelte Negation} \\
 &= \overline{\overline{(\bar{a} \wedge \bar{b}) \vee (a \wedge b) \vee (c \wedge d)}} && \text{De Morgan} \\
 &= \overline{(\bar{a} \vee \bar{b}) \wedge (a \vee b) \wedge (c \vee d)} && \text{vereinfachen} \\
 &= \overline{(a \vee b) \wedge (\bar{a} \vee \bar{b}) \wedge (\bar{c} \vee \bar{d})} && \text{OR zu NOR} \\
 &= \overline{a \vee b \vee (\bar{a} \vee \bar{b}) \vee (\bar{c} \vee \bar{d})} && \text{Negation ersetzen} \\
 &= \overline{a \vee b \vee ((\bar{a} \vee \bar{a}) \vee (\bar{b} \vee \bar{b})) \vee ((\bar{c} \vee \bar{c}) \vee (\bar{d} \vee \bar{d}))}
 \end{aligned}$$

2.5. Normalformen

Für die Darstellung von Schaltfunktionen als Funktionsgleichung, gibt es standardisierte Formen, sog. Normalformen.

Definition 2.16 (Disjunktive und Konjunktive Normalform).

Die *disjunktive Normalform* (DNF) einer Schaltfunktion ist eine VerODERung von UND-Termen, bzw. eine Disjunktion von Konjunktionstermen (vergl. Bsp. 2.17, Gleichung 2.1).
 DNF

Enthalten die Konjunktionsterme *jede* Eingangsvariable in negierter oder nicht negierter Form, spricht man von einer *kanonischen disjunktiven Normalform* (KDNF) (siehe Bsp. 2.17, Gl. 2.2).
 KDNF

Die *konjunktive Normalform* (KNF) einer Schaltfunktion ist eine VerUNDung von ODER-Termen, bzw. eine Konjunktion von Disjunktionstermen (Bsp. 2.17, Gl. 2.3).
 KNF

Kommen in allen Disjunktionstermen alle Eingangsvariablen in negierter oder nicht negierter Form vor, so handelt es sich um eine *kanonische konjunktive Normalform* (KKNF) (Bsp. 2.17, Gl. 2.4).
 KKNF

2. Schaltalgebra

Beispiel 2.17.

Betrachten wir die Schaltfunktion aus Beispiel 2.9, $g(a, b, c) = (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c)$:

Eine DNF wäre:

$$g(a, b, c) = (a \wedge c) \vee (\overline{a} \wedge b) \vee (b \wedge c) = (ac) \vee (\overline{a}b) \vee (bc) \quad (2.1)$$

Die KDNF ist:

$$g(a, b, c) = (\overline{a} \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge b \wedge c) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge c) = (\overline{a}b\overline{c}) \vee (\overline{a}bc) \vee (a\overline{b}c) \vee (abc) \quad (2.2)$$

Eine KNF der Funktion wäre:

$$g(a, b, c) = (\overline{a} \vee c) \wedge (a \vee b) \wedge (\overline{a} \vee b \vee c) \quad (2.3)$$

Die KKNF lautet:

$$g(a, b, c) = (a \vee b \vee c) \wedge (a \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee \overline{b} \vee c) \quad (2.4)$$

2.5.1. Min- und Maxterme

Definition 2.18 (Min- und Maxterme).

Ein *Minterm* ist ein Konjunktionsterm, der alle Eingangsvariablen einer Schaltfunktion enthält. Bei n Eingangsvariablen gibt es 2^n verschiedene Minterme. Jeder Minterm hat nur bei *einer einzigen* Wertekombination der Eingangsvariablen den Wert 1. Den Minterm, der für eine bestimmte Wertekombination den Wert 1 hat, erhält man, indem man die Eingangsvariablen mit UND verknüpft und die Eingangsvariablen negiert, deren Wert 0 ist. So ist der zugehörige Minterm zur Wertekombination 1 0 der Minterm $a \wedge \overline{b}$ (siehe Tab 2.7).

a	b	$m_0 = \overline{a} \wedge \overline{b}$	$m_1 = \overline{a} \wedge b$	$m_2 = a \wedge \overline{b}$	$m_3 = a \wedge b$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Tabelle 2.7.: Wertekombinationen von zwei Schaltvariablen mit möglichen Mintermen und zugehörigen Werten.

Ein *Maxterm* ist ein Disjunktionsterm, der alle Eingangsvariablen einer Schaltfunktion enthält. Bei n Eingangsvariablen gibt es 2^n verschiedene Maxterme. Jeder Maxterm hat nur bei *einer* Wertekombination der Eingangsvariablen den Wert 0, bei allen anderen ist der Wert 1. Den Maxterm, der für eine bestimmte Wertekombination den Wert 0 hat, erhält man, indem man die Eingangsvariablen mit ODER verknüpft und die Eingangsvariablen negiert, deren Wert 1 ist. So ist der zugehörige Maxterm zur Wertekombination 1 0 der Maxterm $\overline{a} \vee b$ (siehe Tab 2.8).

a	b	$M_0 = a \vee b$	$M_1 = a \vee \bar{b}$	$M_2 = \bar{a} \vee b$	$M_3 = \bar{a} \vee \bar{b}$
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Tabelle 2.8.: Wertekombinationen von zwei Schaltvariablen mit möglichen Maxtermen und zugehörigen Werten.

Die Indizes der Min- und Maxterme ergeben sich direkt aus der Belegung der Schaltvariablen, wenn man diese als Dualzahl auffasst. $00 \rightarrow$ Index 0 und somit $m_0 = \bar{a} \wedge \bar{b}$ bzw. $M_0 = a \vee b$.

Merke: Für den Minterm verknüpfe alle Eingangsvariablen mit UND (\wedge) und negiere alle, deren Wert 0 ist.

Für den Maxterm verknüpfe alle Eingangsvariablen mit ODER (\vee) und negiere alle, deren Wert 1 ist.

Man kann eine Schaltfunktion also auch durch Angabe der Min-bzw. Maxterme darstellen.

Beispiel 2.19.

Betrachten wir die Funktion $f_1(a, b) = m_0 \vee m_2$.

Minterm m_0 ist in diesem Fall $(\bar{a} \wedge \bar{b})$, Minterm m_2 ist $(a \wedge \bar{b})$ also gilt:

$$f(a, b) = m_0 \vee m_2 = (\bar{a} \wedge \bar{b}) \vee (a \wedge \bar{b})$$

Achtung: Hier ist ganz wichtig, in welcher Reihenfolge die Eingangsvariablen in der Funktionsdefinition vorkommen. Die Funktion $f_2(b, a) = m_0 \vee m_2$ beschreibt nämlich eine ganz andere Funktion als $f_1(a, b)$. Der Minterm m_0 ist hier zwar auch $(\bar{b} \wedge \bar{a})$, allerdings nur, weil beide Eingangsvariablen negiert sind und in der bool'schen Algebra das Kommutativgesetz gilt. Der Minterm m_2 ist hier $(b \wedge \bar{a})$, denn b ist in dieser Funktion die höherwertige Eingangsvariable, steht weiter links und bekommt bei der Belegung $(1, 0)$ die 1 zugewiesen. Für $f_2(b, a)$ gilt also:

$$f_2(b, a) = m_0 \vee m_2 = (\bar{b} \wedge \bar{a}) \vee (b \wedge \bar{a})$$

Wie erhält man aus einer Wertetabelle oder Funktionsgleichung eine DNF bzw. KNF?

Um eine gegebene Funktion als DNF bzw. KNF angeben zu können, gibt es mehrere Möglichkeiten:

- 1. Umformen der Gleichung:** Jede Funktionsgleichung kann durch Umformungen nach den Gesetzen der Schaltalgebra (Abschn. 2.3) in eine DNF, bzw. KNF gebracht werden.

2. Schaltalgebra

Beispiel 2.20.

Die Funktion aus Beispiel 2.8 kann folgendermaßen in eine DNF umgeformt werden.

$$\begin{aligned}
 g(a, b, c) &= (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c) && \text{De Morgan} \\
 &\stackrel{10}{=} (a \vee b) \wedge (\overline{a} \vee \overline{\overline{c}}) \vee (b \wedge c) \\
 &= (a \vee b) \wedge (\overline{a} \vee c) \vee (b \wedge c) && \text{Distributivgesetz} \\
 &\stackrel{4}{=} ((a \vee b) \wedge \overline{a}) \vee ((a \vee b) \wedge c) \vee (b \wedge c) && \text{Distributivgesetz} \\
 &\stackrel{4}{=} ((\overline{a} \wedge a) \vee (\overline{a} \wedge b)) \vee ((a \wedge c) \vee (b \wedge c)) \vee (b \wedge c)
 \end{aligned}$$

Dies ist bereits eine DNF. Jetzt kann sie noch ein wenig vereinfacht werden:

$$\begin{aligned}
 &((\overline{a} \wedge a) \vee (\overline{a} \wedge b)) \vee ((a \wedge c) \vee (b \wedge c)) \vee (b \wedge c) && \text{Inverses Element} \\
 &\stackrel{6}{=} 0 \vee (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c) \vee (b \wedge c) && \text{Reduktionsgesetz} \\
 &\stackrel{7}{=} (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c) \vee (b \wedge c) && \text{Idempotenz} \\
 &\stackrel{8}{=} (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c)
 \end{aligned}$$

Die Umformung in eine KNF funktioniert z.B. wie folgt:

$$\begin{aligned}
 g(a, b, c) &= (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c) && \text{De Morgan} \\
 &\stackrel{10}{=} (a \vee b) \wedge (\overline{a} \vee \overline{\overline{c}}) \vee (b \wedge c) \\
 &= (a \vee b) \wedge (\overline{a} \vee c) \vee (b \wedge c) && \text{Distributivgesetz} \\
 &\stackrel{4}{=} (a \vee b) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee c \vee c) && \text{Idempotenz} \\
 &\stackrel{8}{=} (a \vee b) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee c)
 \end{aligned}$$

2. Direktes Ablesen aus der Wahrheitstabelle: Liegt für die Schaltfunktion eine vollständige Wahrheitstabelle vor, dann können KDNF und KKNF direkt aus der Wahrheitstabelle abgelesen werden. In der KDNF treten genau die Minterme auf, bei denen die Ausgangsvariable y der Schaltfunktion den Wert 1 hat. In der KKNF treten genau die Maxterme auf, für die die Ausgangsvariable der Schaltfunktion den Wert 0 hat.

Beachte: Jede KDNF ist eine DNF. Die KDNF ist nur eine spezielle DNF, nämlich die, bei der die Konjunktionsterme alle Eingangsvariablen enthalten (siehe Def. 2.16). Das gilt analog auch für die KKNF.

Beispiel 2.21.

Die Wahrheitstabelle für die Schaltfunktion aus Beispiel 2.8, $g(a, b, c) = (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c)$ mit den zugehörigen Min- und Maxtermen sieht wie folgt aus:

a	b	c	$y = g(a, b, c)$	Minterme	m_i	Maxterme	M_i
0	0	0	0	$\overline{a} \wedge \overline{b} \wedge \overline{c}$	m_0	$a \vee b \vee c$	M_0
0	0	1	0	$\overline{a} \wedge \overline{b} \wedge c$	m_1	$a \vee b \vee \overline{c}$	M_1
0	1	0	1	$\overline{a} \wedge b \wedge \overline{c}$	m_2	$a \vee \overline{b} \vee c$	M_2
0	1	1	1	$\overline{a} \wedge b \wedge c$	m_3	$a \vee \overline{b} \vee \overline{c}$	M_3
1	0	0	0	$a \wedge \overline{b} \wedge \overline{c}$	m_4	$\overline{a} \vee b \vee c$	M_4
1	0	1	1	$a \wedge \overline{b} \wedge c$	m_5	$\overline{a} \vee b \vee \overline{c}$	M_5
1	1	0	0	$a \wedge b \wedge \overline{c}$	m_6	$\overline{a} \vee \overline{b} \vee c$	M_6
1	1	1	1	$a \wedge b \wedge c$	m_7	$\overline{a} \vee \overline{b} \vee \overline{c}$	M_7

Die aus der Tabelle abgelesene KDNF lautet:

$$g(a, b, c) = (\overline{a} \wedge b \wedge \overline{c}) \vee (\overline{a} \wedge b \wedge c) \vee (a \wedge \overline{b} \wedge c) \vee (a \wedge b \wedge c)$$

Die abgelesene KKNF lautet:

$$g(a, b, c) = (a \vee b \vee c) \wedge (a \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee \overline{b} \vee c)$$

Beachte: Die kanonische Normalform ist bis auf die Reihenfolge der Min- bzw Maxterm eindeutig.

Ist keine vollständige Wahrheitstabelle gegeben, sondern nur eine Funktionsgleichung, kann für die gegebene Funktionsgleichung natürlich auch zunächst die Wahrheitstabelle aufgestellt und DNF, bzw. KNF dann daraus ablesen werden. Das funktioniert allerdings nur, wenn die Schaltfunktion nicht zu viele Eingangsvariablen hat, da sich die Anzahl der Zeilen einer Wahrheitstabelle mit jeder Eingangsvariablen verdoppelt (vergl. 2.2.1).

2.6. KV-Diagramme

Das **Karnaugh-Veitch-Diagramm** ist eine graphische Darstellung einer Wahrheitstabelle oder Schaltfunktion. Es wurde Anfang der 50'er Jahre von Edward W. Veitch und Maurice Karnaugh entwickelt. Das Diagramm ist eine Matrix, in der jedem Feld eine Kombination der Eingangsvariablen zugeordnet ist. Dabei sind die Felder so zugeordnet, dass sich beim Übergang von einem Feld in ein benachbartes Feld der Wert nur *einer* Eingangsvariablen ändert.

Mit Balken an Spalten und Zeilen der Matrix wird angezeigt, in welchen Feldern der Wert der entsprechenden Variablen 1 ist. In Felder die nicht in der entsprechenden Spalte bzw. Zeile liegen, ist der Wert der Variable damit implizit 0.

So ist dem Feld mit Index 0 im KV-Diagramm für eine Eingangsvariable (a) (Abb.: 2.6) die Belegung 0, bzw. \overline{a} zugeordnet und dem Feld mit Index 1 die Belegung 1 bzw. a . Im KV-Diagramm für zwei Eingangsvariablen (a, b) ist Feld 0 die Belegung 00, bzw. $\overline{a}\overline{b}$, Feld 1 die Belegung 01, bzw. $\overline{a}b$, Feld 2 die Belegung 10, bzw. $a\overline{b}$ und Feld 3 die Belegung 11, bzw. ab

2. Schaltalgebra

zugeordnet. Die Indizes der Felder im KV-Diagramm sind also identisch mit den Indizes der Min- und Maxterme, die ihnen zugeordnet sind (vergl. Tab. 2.9).

Bemerkung: Im Skript sind die KV-Diagramme mit entsprechenden Indizes dargestellt, um im Text besser Bezug auf bestimmte Felder nehmen zu können. Auf den Übungszetteln sind die Indizes nicht abgebildet, da es besser ist, sich an die äußeren Beschriftungen zu halten. Die Felder eines KV-Diagramms müssen nicht zwangsläufig so angeordnet sein, dass z.B. 00 bzw. $\bar{a}\bar{b}$ in der linken, oberen Ecke ist. Es ist lediglich gefordert, dass sich beim Übergang in ein benachbartes Feld, die Belegung nur *einer* Eingangsvariablen ändert. 00 könnte jeder anderen Ecke zugeordnet sein. Dann müssten die Balken an den Rändern des Diagramms entsprechend anders positioniert sein.

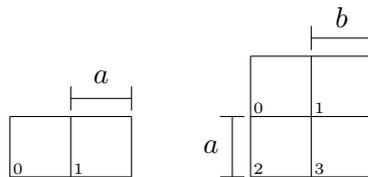


Abbildung 2.6.: KV-Diagramme für eine (a) und für zwei (a, b) Eingangsvariablen

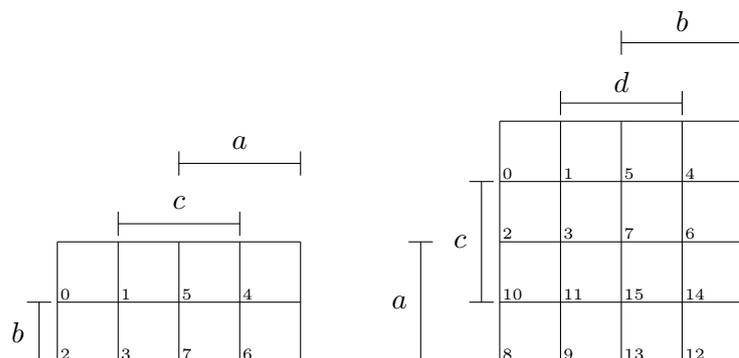


Abbildung 2.7.: KV-Diagramme für drei (a, b, c) und für vier (a, b, c, d) Eingangsvariablen

Feldindex	Dualzahl	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	Minterm	Maxterm
0	0000	0	0	0	0	$\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge \bar{d}$	$a \vee b \vee c \vee d$
1	0001	0	0	0	1	$\bar{a} \wedge \bar{b} \wedge \bar{c} \wedge d$	$a \vee b \vee c \vee \bar{d}$
2	0010	0	0	1	0	$\bar{a} \wedge \bar{b} \wedge c \wedge \bar{d}$	$a \vee b \vee \bar{c} \vee d$
3	0011	0	0	1	1	$\bar{a} \wedge \bar{b} \wedge c \wedge d$	$a \vee b \vee \bar{c} \vee \bar{d}$
4	0100	0	1	0	0	$\bar{a} \wedge b \wedge \bar{c} \wedge \bar{d}$	$a \vee \bar{b} \vee c \vee d$
5	0101	0	1	0	1	$\bar{a} \wedge b \wedge \bar{c} \wedge d$	$a \vee \bar{b} \vee c \vee \bar{d}$
6	0110	0	1	1	0	$\bar{a} \wedge b \wedge c \wedge \bar{d}$	$a \vee \bar{b} \vee \bar{c} \vee d$
7	0111	0	1	1	1	$\bar{a} \wedge b \wedge c \wedge d$	$a \vee \bar{b} \vee \bar{c} \vee \bar{d}$
8	1000	1	0	0	0	$a \wedge \bar{b} \wedge \bar{c} \wedge \bar{d}$	$\bar{a} \vee b \vee c \vee d$
9	1001	1	0	0	1	$a \wedge \bar{b} \wedge \bar{c} \wedge d$	$\bar{a} \vee b \vee c \vee \bar{d}$
10	1010	1	0	1	0	$a \wedge \bar{b} \wedge c \wedge \bar{d}$	$\bar{a} \vee b \vee \bar{c} \vee d$
11	1011	1	0	1	1	$a \wedge \bar{b} \wedge c \wedge d$	$\bar{a} \vee b \vee \bar{c} \vee \bar{d}$
12	1100	1	1	0	0	$a \wedge b \wedge \bar{c} \wedge \bar{d}$	$\bar{a} \vee \bar{b} \vee c \vee d$
13	1101	1	1	0	1	$a \wedge b \wedge \bar{c} \wedge d$	$\bar{a} \vee \bar{b} \vee c \vee \bar{d}$
14	1110	1	1	1	0	$a \wedge b \wedge c \wedge \bar{d}$	$\bar{a} \vee \bar{b} \vee \bar{c} \vee d$
15	1111	1	1	1	1	$a \wedge b \wedge c \wedge d$	$\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d}$

Tabelle 2.9.: Feldindizes und zugeordnete Belegung der Eingangsvariablen sowie Min- und Maxterme bei KV-Diagrammen für vier Eingangsvariablen.

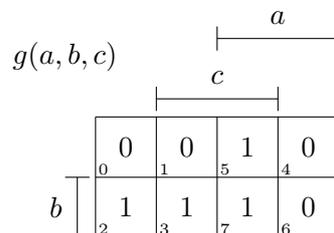
Soll eine gegebene Schaltfunktion, die in KDNF vorliegt, in einem KV-Diagramm dargestellt werden, so wird in die Felder mit den Indizes, deren zugeordnete Minterme in der Schaltfunktion auftauchen, eine 1 geschrieben. In alle Felder, deren zugeordnete Minterme nicht in der gegebenen KDNF auftauchen, wird eine 0 eingetragen.

Beispiel 2.22.

Betrachten wir wieder die Funktion aus Beispiel 2.8 in KDNF (vergl. Bsp. 2.21),

$$g(a, b, c) = (\bar{a} \wedge b \wedge \bar{c}) \vee (\bar{a} \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c) \vee (a \wedge b \wedge c)$$

Die Funktion hat drei Eingangsvariablen und die aufgeführten Minterme entsprechen den Belegungen (0, 1, 0), (0, 1, 1), (1, 0, 1) und (1, 1, 1). Demnach erhalten die Felder mit Indizes 2, 3, 5 und 7 eine 1, in alle anderen Felder muss eine 0 eingetragen werden.



Liegt eine gegebene Schaltfunktion in KKNF vor, so muss in alle Felder, deren zugeordnete Maxterme in der KKNF auftauchen, eine 0 eingetragen werden. In alle anderen Felder muss eine 1 eingetragen werden.

2. Schaltalgebra

Beispiel 2.23.

Betrachten wir die vierstellige Funktion $f(a, b, c, d)$ in KKNF:

$$f(a, b, c, d) = (a \vee b \vee \bar{c} \vee \bar{d}) \wedge (a \vee \bar{b} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee b \vee \bar{c} \vee d) \wedge (\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d})$$

Die Maxterme entsprechen den Belegungen $(0, 0, 1, 1)$, $(0, 1, 1, 1)$, $(1, 0, 1, 0)$ und $(1, 1, 1, 1)$. Dementsprechend müssen in die Felder mit Indizes 3, 7, 10 und 15 eine 0 und in alle anderen Felder eine 1 eingetragen werden.

$f(a, b, c, d)$

		----- b		
		----- d		
		1	1	1
	0	1	0	1
	2	1	0	1
	10	0	1	1
	8	1	1	1
c	-----	1	0	1
a	-----	1	1	1

2.7. Minimierung von Schaltfunktionen

Schaltfunktionen, die in kanonischer Normalform oder als Wahrheitstabelle gegeben sind, enthalten oft redundante Terme und können vereinfacht werden.

Beispiel 2.24.

Die Funktionsgleichung

$$f(a, b, c) = (a \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c)$$

Kann vereinfacht werden zu

$$f(a, b, c) = (a \wedge \bar{c}) \vee (a \wedge b)$$

Mit Vereinfachung ist hier gemeint, dass die Funktion mit möglichst wenig Verknüpfungen und möglichst wenig Variablen dargestellt wird. Eine vereinfachte, oder *minimierte* Funktion lässt sich mit weniger Bauteilen realisieren.

Beispiel 2.25.

Betrachten wir die Funktion $f(a, b, c)$ aus Beispiel 2.24. Abbildung 2.8 zeigt die KDNF der Funktion als Schaltung realisiert.

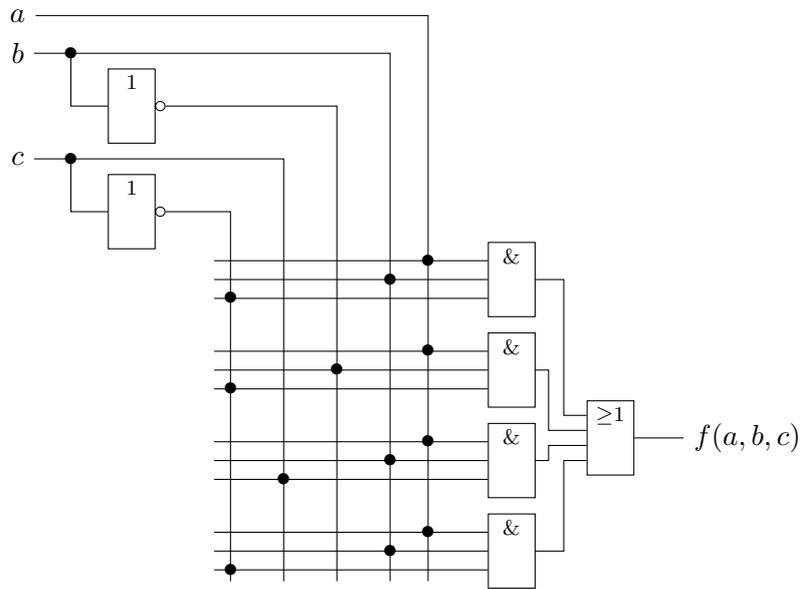


Abbildung 2.8.: Schaltung für $f(a, b, c) = (a \wedge b \wedge \bar{c}) \vee (a \wedge \bar{b} \wedge \bar{c}) \vee (a \wedge b \wedge c) \vee (a \wedge \bar{b} \wedge c)$.
Abbildung 2.9 zeigt die Schaltung der minimierten Funktion.

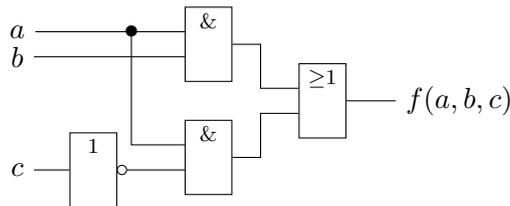


Abbildung 2.9.: Schaltung für $f(a, b, c) = (a \wedge \bar{c}) \vee (a \wedge b)$

Es gibt verschiedene Verfahren um Schaltungen zu minimieren. Die Grundlage aller Verfahren sind die Resolutionsregeln (siehe Kap. 2.3).

$$\begin{aligned}
 (a \wedge b) \vee (a \wedge \bar{b}) &= a \wedge (b \vee \bar{b}) && \text{Distributivgesetz} \\
 &= a \wedge 1 && \text{Inverse Elemente} \\
 &= a && \text{Reduktionsgesetze}
 \end{aligned}$$

$$\begin{aligned}
 (a \vee b) \wedge (a \vee \bar{b}) &= a \vee (b \wedge \bar{b}) && \text{Distributivgesetz} \\
 &= a \vee 0 && \text{Inverse Elemente} \\
 &= a && \text{Reduktionsgesetze}
 \end{aligned}$$

Das heißt, unterscheiden sich UND- bzw. ODER-Terme nur in der Negation einer einzigen Variablen, so können sie zu einem Term verschmolzen werden, bei dem diese Variable weggelassen wird.

2. Schaltalgebra

2.7.1. Minimierung mit den Gesetzen der Schaltalgebra

Beispiel 2.26.

Bei der KDNF der Funktion

$$f(a, b, c, d) = (\bar{a}\bar{b}cd) \vee (abcd) \vee (ab\bar{c}d) \vee (\bar{a}bcd) \vee (\bar{a}\bar{b}\bar{c}d)$$

können z.B. die Terme 1 und 2, die Terme 2 und 3, sowie 4 und 5 zusammengefasst werden zu

$$f(a, b, c, d) = (acd) \vee (abd) \vee (\bar{a}bd)$$

Nun können noch die letzten beiden Terme zusammen gefasst werden, und man erhält

$$f(a, b, c) = (acd) \vee (bd)$$

2.7.2. Minimierung mit KV-Diagrammen

Das Verfahren mit KV-Diagrammen ist ein graphisches Verfahren. Im KV-Diagramm ist jedem Feld ein Min- bzw. Maxterm zugeordnet, und zwar so, dass sich Min- bzw. Maxterme benachbarter Felder nur in der Belegung einer Variablen unterscheiden (vergl. Kap. 2.6). Daher können Felder mit benachbarten 1-en bzw. 0-en zusammengefasst werden.

Beispiel 2.27.

Betrachten wir die Funktion aus Beispiel 2.26. In ein KV-Diagramm eingetragen, sieht die Funktion wie folgt aus:

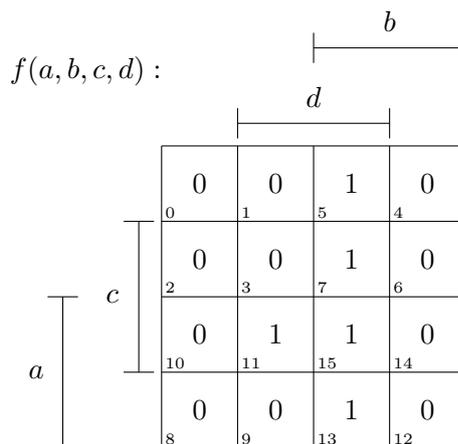
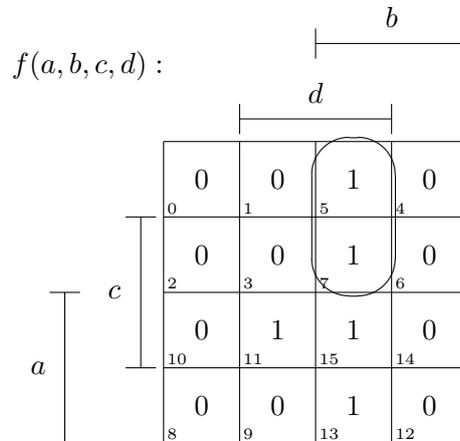


Abbildung 2.10.: KV-Diagramm für Funktion $f(a, b, c, d) = (\bar{a}\bar{b}cd) \vee (abcd) \vee (ab\bar{c}d) \vee (\bar{a}bcd) \vee (\bar{a}\bar{b}\bar{c}d)$

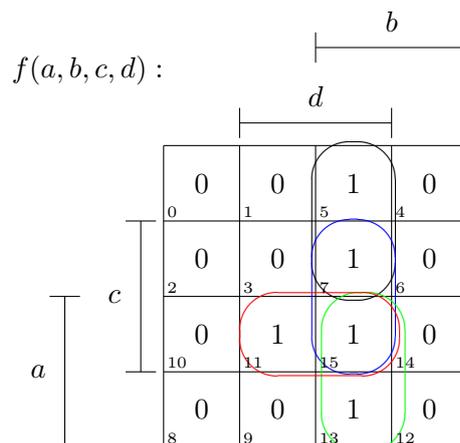
Nun können z.B. die 1-en der Felder mit Indizes 5 und 7 zusammengefasst werden.

2.7. Minimierung von Schaltfunktionen



Dem zusammengefassten Block ist der Term $(\bar{a} \wedge b \wedge d)$ oder $(\bar{a}bd)$ zugeordnet. Wenn wir uns zurück erinnern an Beispiel 2.26, entsprechen die Felder mit Indizes 5 und 7 den Mintermen $(\bar{a}\bar{b}\bar{c}d)$ und $(\bar{a}bcd)$, welche zu $(\bar{a}bd)$ zusammengefasst wurden, da sie sich lediglich in der Belegung der Variablen c unterscheiden.

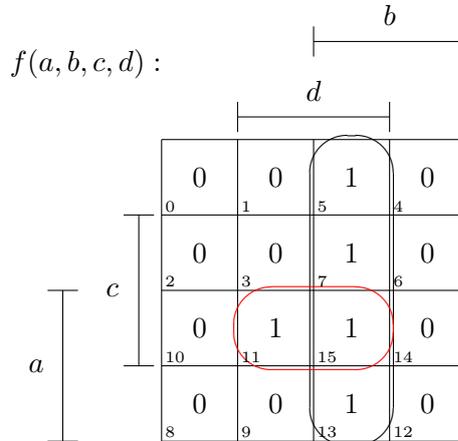
Es können weiterhin noch die Felder mit den Indizes 7 und 15 (blau), 15 und 13 (grün), sowie 11 und 15 (rot) zusammen gefasst werden



Dies entspräche den Termen (cbd) , (abd) und (acd) .

Allerdings können wir die Felder mit Indizes 5, 7, 15 und 13 auch in einem Block zusammenfassen, da sie alle benachbart sind.

2. Schaltalgebra



Der neue Block wird durch den Term (bd) abgedeckt. Zusammen mit dem anderen Block (acd) , ergibt sich nun für die minimierte Funktionsgleichung

$$f(a, b, c) = (bd) \vee (acd)$$

Das entspricht dem Ergebnis, welches wir bei der Minimierung durch Anwendung der Reduktionsregeln erhalten haben.

Bemerkung: Das muss nicht immer so sein. Die minimierte DNF muss nicht eindeutig sein. Bei manchen Funktionen gibt es mehrere Möglichkeiten der Minimierung, wie wir sehen werden.

Definition 2.28 (Implikant, Primimplikant).

Implikant

Ein UND-Term m einer Bool'schen Funktion f , heißt *Implikant*, wenn er Teil der Funktion f ist. D.h. wenn gilt: falls $m(x) = 1$ dann $f(x) = 1$ für alle möglichen Belegungen der Eingangsvariablen $x = (a, b, c, \dots)$. Der kleinste Implikant einer Funktion f ist ein Minterm, der in der KDNF vorkommt. Für das KV-Diagramm heißt das, dass ein kleinster Implikant aus genau einem Feld besteht, in dem eine 1 steht.

Primimplikant

Ein *Primimplikant* von f ist ein Implikant m von f , der nicht weiter verkürzt werden kann. D.h. es ist ein Implikant minimaler Länge. Im KV-Diagramm bedeutet das, dass es keinen größeren Block von 1-en gibt, der m enthält.

In Beispiel 2.27 sind $(\bar{a}bd)$, (cbd) , (abd) , (acd) und (bd) Implikanten, (acd) und (bd) sind Primimplikanten. $(\bar{a}bd)$, (cbd) und (abd) sind keine Primimplikanten, da (bd) größer ist und die anderen Implikanten enthält.

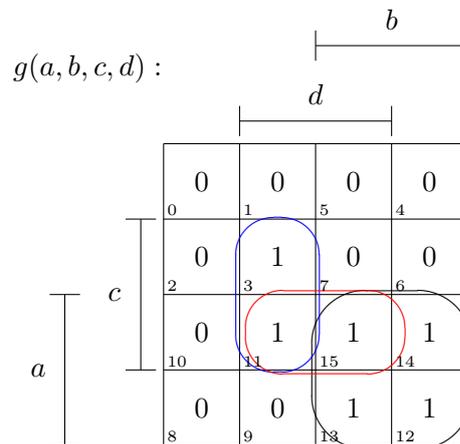
Je größer die Fläche wird, die ein Implikant im KV-Diagramm abdeckt, desto weniger Variablen tauchen im Implikanten auf. Die Blockgröße wächst in Zweierpotenzen. Es dürfen immer nur Blöcke von 2, 4, oder 8 usw. Feldern zusammengefasst werden.

Definition 2.29 (Kernprimimplikant).

Ein *Kernprimimplikant* ist ein Primimplikant, der eine 1 enthält, die von keinem andern Primimplikanten abgedeckt wird.

Beispiel 2.30.

Im KV-Diagramm der Funktion



Sind (ab) , (acd) und $(\bar{b}cd)$ Primimplikanten. $(\bar{b}cd)$ ist Kernprimimplikant, da er als einziger Primimplikant die 1 in dem Feld mit Index 3 abdeckt. (bd) ist Kernimplikant, da er als einziger Primimplikant die 1-en in den Feldern mit Indizes 12, 13 und 14 abdeckt. (acd) ist *kein* Kernprimimplikant, da alle 1-en, die er abdeckt auch von andern Primimplikanten abgedeckt werden.

Definition 2.31 (relativ und absolut eliminierbar).

Ein Primimplikant heißt *absolut eliminierbar*, wenn alle 1-en, die er abdeckt, auch durch Kernprimimplikanten abgedeckt werden. (acd) aus Beispiel 2.30 ist also ein absolut eliminierbarer Primimplikant.

absolut eliminierbar

Alle Primimplikanten, die weder Kernprimimplikanten, noch absolut eliminierbare Primimplikanten sind, sind *relativ eliminierbare* Primimplikanten.

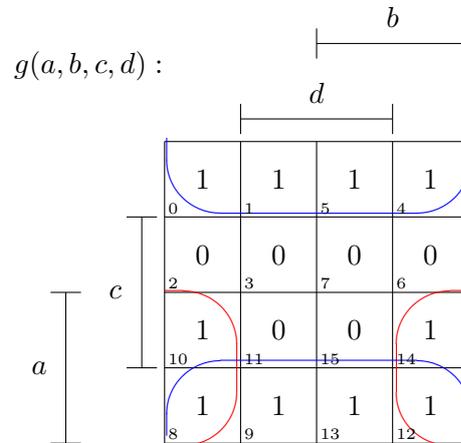
relativ eliminierbar

Achtung! Die Ränder der KV-Diagramme sind auch benachbart. D.h. Felder am linken und rechten Rand, sowie oben und unten können auch in einem Block zusammengefasst werden.

Beispiel 2.32.

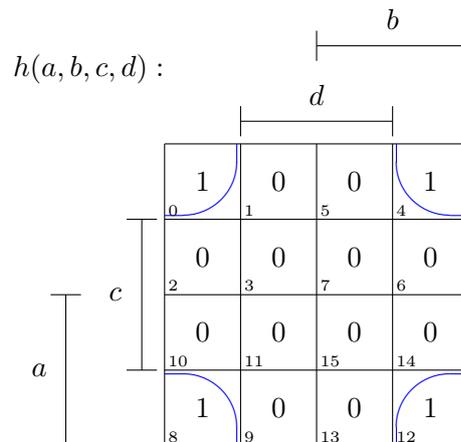
Hier bilden die Felder 0, 1, 4, 5, 8, 9, 12 und 13 den Primimplikanten \bar{c} . Die Felder 8, 10, 12 und 14 werden zum Primimplikanten $(a\bar{d})$

2. Schaltalgebra



Auch die Ecken eines KV-Diagramms sind benachbart!

Die Felder mit Indizes 0, 4, 8 und 12 werden zum Primimplikanten $(\bar{c}\bar{d})$ zusammengefasst.



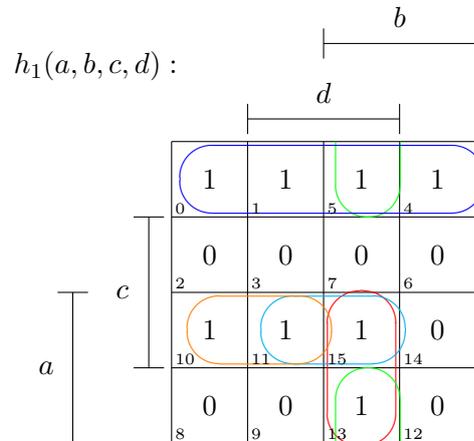
Aufstellung einer minimalen DNF

Um aus dem KV-Diagramm die minimale DNF aufzustellen, werden zunächst graphisch alle Primimplikanten eingezeichnet. Die minimale DNF muss alle 1-Terme der Schaltfunktion überdecken. Also sind alle Kernprimimplikanten in der minimalen DNF vertreten. Die absolut eliminierbaren Primimplikanten benötigt man nicht, denn ihre 1-en sind bereits von den Kernprimimplikanten überdeckt. Aus den relativ eliminierbaren Primimplikanten wählt man dann die aus, die die restlichen 1-en überdecken.

Beispiel 2.33.

Im KV-Diagramm der Funktion $h_1(a, b, c, d)$ sind bereits alle Primimplikanten eingezeichnet:

2.8. Unvollständig spezifizierte Funktionen



Kernprimimplikanten sind $(\bar{a}\bar{c})$, und $(a\bar{b}c)$. Alle anderen sind relativ eliminierbare Primimplikanten. Um nun die fehlenden 1-en in den Feldern 13 und 15 abzudecken, wählen wir Primimplikant (abd) , denn das ist kleiner, als die Alternative, die beiden Felder mit den beiden anderen Primimplikanten $(b\bar{c}d)$ und (acd) abzudecken.

Nun werden die gewählten Primimplikanten zu einer DNF zusammengefügt. Die minimale DNF lautet also

$$h_1(a, b, c, d) = (\bar{a}\bar{c}) \vee (a\bar{b}c) \vee (abd)$$

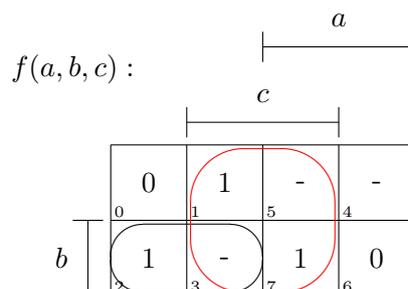
Bemerkung: Das Aufstellen der minimalen KNF funktioniert nach demselben Prinzip. Statt mit Mintermen wird mit Maxtermen gearbeitet und die Blöcke heißen nicht Implikanten sondern *Prim-implikate*.

2.8. Unvollständig spezifizierte Funktionen

Es kommt vor, dass Schaltfunktionen nicht vollständig spezifiziert sind. In dem Fall können manche Funktionswerte beliebig gewählt werden. Im KV-Diagramm wird das mit $-$ oder d (don't care) markiert. Die don't care-Terme können zur Minimierung der Funktion genutzt werden, müssen aber nicht überdeckt werden, wenn sie die Funktion "vergrößern" würden. Man kann sich also aussuchen, ob die Funktion an der Stelle zu 1 oder zu 0 auswerten soll.

Beispiel 2.34.

Funktion $f(a, b, c)$ ist durch folgendes KV-Diagramm gegeben.



2. Schaltalgebra

Die Primimplikanten können unter Einbeziehung der don't care-Felder so eingezeichnet werden, dass sie möglichst groß werden.

So kommt man auf die minimierte Form

$$f(a, b, c) = (\bar{a}b) \vee c$$

Ohne die Verwendung der don't care-Terme, hätte man folgende minimierte Form gefunden:

$$f(a, b, c) = (\bar{a}b\bar{c}) \vee (\bar{a}bc) \vee (abc)$$

2.9. Schaltungsentwurf

Um nun eine Schaltung für ein konkretes Problem zu entwerfen, müssen verschiedene Schritte durchlaufen werden.

1. Problemanalyse und Aufstellung einer Wahrheitstabelle für die zu implementierende Funktion
2. Minimierung der Funktion, z.B. mit Hilfe von KV-Diagramm
3. Implementierung der Schaltung

Beispiel 2.35.

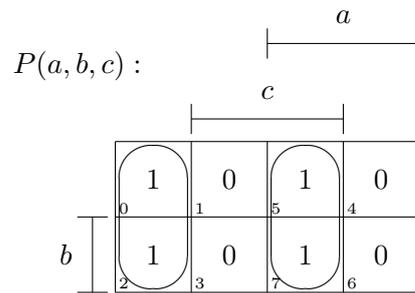
Wenn wir z.B. eine Schaltung $P(a, b, c)$ entwerfen wollen, die immer dann eine 1 ausgibt wenn die Eingangsbelegung ein Palindrom³ ist.

Wie sieht die Wahrheitstabelle für solch eine Funktion aus?

a	b	c	$P(a, b, c)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Das zugehörige KV-Diagramm sieht so aus:

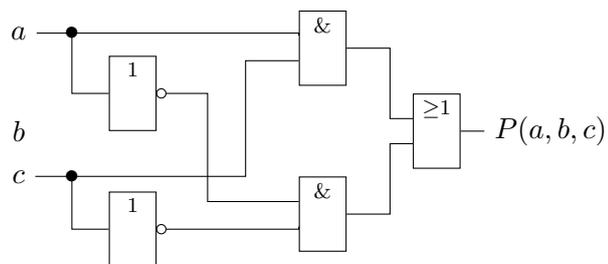
³Ein Palindrom ist ein Wort, das von rechts und von links gelesen, gleich ist. z.B. die Wörter Rentner oder Ebbe



Die minimierte DNF lautet dann:

$$P(a, b, c) = (\bar{a}\bar{c}) \vee (ac)$$

Diese Funktion könnten wir nun mit zwei NICHT-Gattern, zwei UND-Gattern und einem ODER-Gatter realisieren.



Teil II.

Benutzung von Unix-Systemen, Einführung in das Funktionale Programmieren

Prof. Dr. David Sabel und Conrad Rau
überarbeitet von Ronja Düffel

3. Einführung in die Bedienung von Unix-Systemen

In diesem Kapitel werden wir vorwiegend die Grundlagen der Bedienung von Unix-Systemen und dabei insbesondere die Benutzung der für Informatikstudierende zur Verfügung stehenden Linux-Rechner an der Goethe-Universität erläutern.

3.1. Unix und Linux

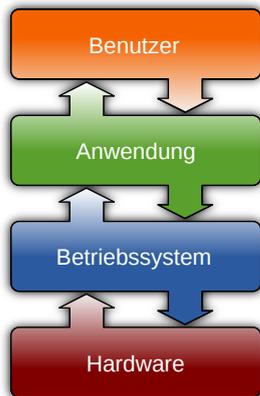


Abbildung 3.1.:

Unix ist ein Betriebssystem und wurde 1969 in den *Bell Laboratories* (später *AT&T*) entwickelt. Als Betriebssystem verwaltet Unix den Arbeitsspeicher, die Festplatten, CPU, Ein- und Ausgabegeräte eines Computers und bildet somit die Schnittstelle zwischen den Hardwarekomponenten und der Anwendungssoftware (z.B. Office) des Benutzers (Abb. 3.1). Da seit den 80er Jahren der Quellcode von Unix nicht mehr frei verfügbar ist und bei der Verwendung von Unix hohe Lizenzgebühren anfallen, wurde 1983 das GNU-Projekt (**GNU's Not Unix**) ins Leben gerufen, mit dem Ziel, ein freies Unix-kompatibles Betriebssystem zu schaffen. Dies gelang 1991 mithilfe des von Linus Torvalds programmierten Kernstücks des Betriebssystems, dem Linux-Kernel. GNU Linux ist ein vollwertiges, sehr mächtiges Betriebssystem.

Unix

GNU
Linux

Da der Sourcecode frei zugänglich ist, können alle die wollen, ihre eigenen Anwendungen und Erweiterungen programmieren und diese veröffentlichen. Es gibt unzählige Linux-Distributionen (Red Hat, SuSe, Ubuntu,...), welche unterschiedliche Software Pakete zur Verfügung stellen. Auf den Rechnern der **Rechnerbetriebsgruppe**

Informatik der Goethe Universität (RBI) ist Fedora Linux, der Nachfolger von Red Hat Linux installiert.

Linux stellt den Benutzer:innen sog. *Terminals* zur Verfügung, an denen gearbeitet werden kann. Ein Terminal ist eine Schnittstelle zwischen Mensch und Computer. Es gibt *textbasierte* und *graphische* Terminals.

Terminal

Textbasierte Terminals stellen eine Kommandozeile zur Verfügung. Über diese kann mit Programmen die über ein CLI (**command line interface**) verfügen, durch Eingabe von Befehlen, mithilfe der Computertastatur, interagiert werden. Einige solcher Programme werden wir gleich kennenlernen. Das Terminal stellt Nachrichten und Informationen der Programme in Textform auf dem Bildschirm dar. Der Nachteil textbasierter Terminals ist für Anfänger:innen meist, dass die Kommandozeile auf eine Eingabe wartet. Man muss den Befehl kennen, den

textbasiertes
Terminal
CLI

3. Einführung in die Bedienung von Unix-Systemen

man benutzen möchte, oder wissen, wie man ihn nachschauen kann. Es gibt nicht die Möglichkeit sich mal irgendwo „durchzuklicken“. Der Vorteil textbasierter Terminals ist, dass die Programme mit denen man arbeiten kann, häufig sehr mächtig sind. Ein textbasiertes Terminal bietet sehr viel mehr Möglichkeiten, als ein graphisches Terminal.

graphisches Terminal

Graphische Terminals sind das, was die meisten Menschen, die heutzutage Computer benutzen, kennen. Ein graphisches Terminal lässt sich mit einer Computermaus bedienen. Die Programme werden durch Klicken auf bestimmte Teile des Bildschirms bedient, welche durch Icons (kleine Bilder) oder Schriftzüge gekennzeichnet sind. Damit das funktioniert, benötigen die Programme eine graphische Benutzeroberfläche, auch GUI (**g**raphical **u**ser **i**nterface) genannt. Auf den Rechnern der RBI gibt es, unter anderem, die graphischen Benutzeroberflächen Gnome und KDE.

GUI

Ein einzelner Rechner stellt sieben, voneinander unabhängige Terminals zur Verfügung. Mit den Tastenkombinationen **Strg** + **Alt** + **F1**, **Strg** + **Alt** + **F2** bis **Strg** + **Alt** + **F7** kann zwischen den sieben Terminals ausgewählt werden. Tastatureingaben werden immer an das angezeigte Terminal weitergeleitet. In der Regel ist lediglich das durch die Tastenkombination **Strg** + **Alt** + **F7** erreichbare Terminal graphisch. Alle anderen Terminals sind textbasiert. Auf den RBI-Rechnern ist das graphische Terminal als das aktive Terminal eingestellt, sodass Benutzer:innen die nicht **Strg** + **Alt** + **F1**, ..., **Strg** + **Alt** + **F6** drücken, die textbasierten Terminals nicht zu Gesicht bekommen.

3.1.1. Dateien und Verzeichnisse

Eines der grundlegenden UNIX-Paradigmen ist: „Everything is a file“. Die Zugriffsmethoden für Dateien, Verzeichnisse, Festplatten, Drucker, etc. folgen alle den gleichen Regeln, grundlegende Kommandos sind universell nutzbar. Über die Angabe des Pfades im UNIX-Dateisystem lassen sich Quellen unabhängig von ihrer Art direkt adressieren. So ist beispielsweise mit `/home/hans/protokoll.pdf` eine persönliche Datei des Benutzers „hans“ erreichbar. Ein Verzeichnis auf einem Netzwerklaufwerk ist mit `/usr`, eine Festplattenpartition mit `/dev/sda1` und sogar die Maus ist mit `/dev/mouse` erreichbar.

Dateibaum

Das UNIX-Betriebssystem verwaltet einen *Dateibaum*. Dabei handelt es sich um ein virtuelles Gebilde zur Datenverwaltung. Im Dateibaum gibt es bestimmte Dateien, welche *Verzeichnisse* (engl.: *directories*) genannt werden. Verzeichnisse können andere Dateien (und somit auch Verzeichnisse) enthalten. Jede Datei muss einen Namen haben, dabei wird zwischen Groß- und Kleinschreibung unterschieden. `/home/hans/wichtiges` ist ein anderes Verzeichnis als `/home/hans/Wichtiges`. Jede Datei, insbesondere jedes Verzeichnis, befindet sich in einem übergeordneten Verzeichnis, dem *übergeordneten* Verzeichnis (engl.: *parent directory*). Nur das *Wurzelverzeichnis* (engl.: *root directory*) ist in sich selbst enthalten. Es trägt den Namen „/“.

Verzeichnis

übergeordnetes

Verzeichnis

Wurzel-

verzeichnis

Beispiel 3.1 (Ein Dateibaum).

Nehmen wir an, das Wurzelverzeichnis enthält zwei Verzeichnisse **Alles** und **Besser**. Beide Verzeichnisse enthalten je ein Verzeichnis mit Namen **Dies** und **Das**. In Abbildung 3.2 lässt sich der Baum erkennen. In der Informatik stehen Bäume auf dem Kopf. Die Wurzel befindet sich oben, die Blätter unten.

Pfad

Die beiden Verzeichnisse mit Namen **Dies** lassen sich anhand ihrer Position im Dateibaum leicht auseinanderhalten. Der Weg durch den Dateibaum zu dieser Position heißt *Pfad*

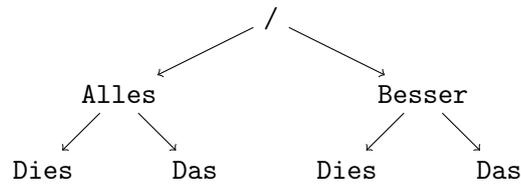


Abbildung 3.2.: Ein Dateibaum

(engl.: *path*). Der Weg von der Wurzel aus, wird als *absoluter* Pfad bezeichnet. Der Weg vom Verzeichnis aus, in dem man sich gerade befindet, wird als *relativer* Pfad bezeichnet. Die absoluten Pfade zu den Verzeichnissen mit Namen **Dies** lauten `/Alles/Dies` und `/Besser/Dies`. Unter UNIX/LINUX dient der Schrägstrich `/` (engl.: *slash*) als Trennzeichen zwischen Verzeichnissen. Im Gegensatz zu Windows, wo dies durch den back-slash `\` geschieht. Wenn wir uns im Verzeichnis `/Besser` befinden, so können die Unterverzeichnisse mit **Dies** und **Das** direkt adressiert werden. Das Symbol `..` bringt uns ins übergeordnete Verzeichnis, in diesem Fall das Wurzelverzeichnis. Somit erreichen wir das Verzeichnis **Alles** aus dem Verzeichnis **Besser** über den relativen Pfad `../Alles`. Befinden wir uns in Verzeichnis `/Alles/Dies` so erreichen wir das Verzeichnis `/Besser/Das` über den relativen Pfad `../../Besser/Das`.

Dateizugriffsrechte

Unter UNIX können auch die Zugriffsrechte einzelner Benutzer:innen auf bestimmte Dateien verwaltet werden. Dabei wird unterschieden zwischen Lese- (*read*), Schreib- (*write*) und Ausführrechten (*x execute*). Für die Vergabe dieser Rechte, wird zwischen Besitzer:in (*owner*) der Datei, einer Gruppe (*group*) von Benutzer:innen und allen Nutzer:innen, die nicht zu der Gruppe gehören (*other*), unterschieden. Um bestimmten Nutzer:innen Zugriffsrechte für bestimmte Dateien zu erteilen, können diese Nutzer:innen zu einer Gruppe zusammengefasst werden. Dann können allen Mitgliedern der Gruppe Zugriffsrechte für diese Dateien erteilt werden.

3.1.2. Login und Shell

Um an einem Terminal arbeiten zu können, müssen sich Benutzer:innen zunächst anmelden. Dies geschieht durch Eingabe eines Benutzernamens und des zugehörigen Passwortes. Diesen Vorgang heißt „sich *Einloggen*“. Erfolgt das Einloggen an einem textbasierten Terminal, so startet danach automatisch eine (Unix)-*Shell*. Dies ist die traditionelle Benutzungsschnittstelle unter UNIX/Linux. Über die Kommandozeile können nun Befehle eingetippt werden, welche der Computer sogleich ausführt. Wenn die Shell bereit ist Kommandos entgegenzunehmen, erscheint eine *Eingabeaufforderung* (engl.: *prompt*). Das ist eine Zeile, die Statusinformationen, wie den Benutzernamen und den Namen des Rechners auf dem man eingeloggt ist, enthält und mit einem blinkenden *Cursor* (Unterstrich) endet.

Wer sich an einem graphischen Terminal einloggt, muss zunächst ein virtuelles textbasiertes Terminal starten, um eine Shell zu Gesicht zu bekommen. Ein virtuelles textbasiertes Terminal kann in der Regel über das Startmenü, Unterpunkt „Konsole“ oder „Terminal“, gestartet werden. Unter der graphischen Benutzeroberfläche KDE kann solch ein Terminal auch gestartet

3. Einführung in die Bedienung von Unix-Systemen

werden, indem man mit der rechten Maustaste auf den Desktop klickt und im erscheinenden Menü den Eintrag „Konsole“ auswählt.

3.1.3. Befehle

Es gibt unzählige Kommandos die die Shell ausführen kann. Wir beschränken uns hier auf einige, die wir für besonders nützlich halten. Um die Shell aufzufordern den eingetippten Befehl auszuführen, muss die *Return*-Taste () betätigt werden. Im Folgenden sind Bildschirm- und -ausgaben in **Schreibmaschinenschrift** gegeben und

in grau hinterlegten Kästen mit durchgezogenen Linien und runden Ecken zu finden.

Später werden wir auch sogenannte Quelltexte darstellen, diese sind

in gestrichelten und eckigen Kästen zu finden.

Passwort
ändern

passwd: ändert das Benutzerpasswort auf dem Rechner auf dem man eingeloggt ist. Nach Eingabe des Befehls, muss zunächst einmal das alte Passwort eingegeben werden. Danach muss zweimal das neue Passwort eingegeben werden. Dieser Befehl ist ausreichend, wenn der Account lediglich auf einem Rechner existiert, z.B. wenn man Linux auf seinem privaten Desktop oder Laptop installiert hat.

```
> passwd   
Changing password for [Benutzername].  
(current) UNIX password:   
Enter new UNIX password:   
Retype new UNIX password:   
passwd: password updated successfully
```

Für den RBI-Account wird folgender Befehl benötigt.

Netzwerk-
passwort

yppasswd: ändert das Passwort im System und steht dann auf allen Rechnern des Netzwerks zur Verfügung.

```
> passwd   
Changing NIS account information for [Benutzername] on [server].  
Please enter old password:   
Changing NIS password for [Benutzername] on [server].  
Please enter new password:   
Please retype new password:   
  
The NIS password has been changed on [server].
```

Arbeits-
verzeichnis

pwd (*print working directory*): gibt den Pfad des Verzeichnisses aus, in dem man sich gerade befindet. Dieses Verzeichnis wird häufig auch als „*aktuelles Verzeichnis*“, oder „*Ar-*

Home-
verzeichnis

beitsverzeichnis“ bezeichnet. Unmittelbar nach dem Login, befindet man sich immer im *Homeverzeichnis*. Der Name des Homeverzeichnis ist der gleiche wie der Benutzername. Hier werden alle persönlichen Dateien und Unterverzeichnisse gespeichert.

```
> pwd ↵
/home/ronja/Vorkurs/sose23/Skript/
```

whoami : gibt den Benutzernamen aus.

```
> whoami ↵
ronja
```

hostname: gibt den Rechnernamen, auf dem man eingeloggt ist, aus.

```
> hostname ↵
nash
```

mkdir: (*make directory*): mit diesem Befehl wird ein neues Verzeichnis (Ordner) angelegt. Dieser Befehl benötigt als zusätzliche Information den Namen, den das neue Verzeichnis haben soll. Dieser Name wird dem Befehl als *Argument* übergeben. Zwischen Befehl und Argument muss immer ein Leerzeichen stehen. Der folgende Befehl legt in dem Verzeichnis, in dem sich der Benutzer gerade befindet, ein Verzeichnis mit Namen „Zeug“ an.

Verzeichnis
erstellen
Argument

```
> mkdir Zeug ↵
```

cd (*change directory*): wechselt das Verzeichnis. Wird kein Verzeichnis explizit angegeben, so wechselt man automatisch in das Homeverzeichnis.

cd ..: wechselt in das nächsthöhere Verzeichnis. Dabei wird `..` als Argument übergeben.

```
> pwd ↵
/home/toni/Vorkurs/
> mkdir Wichtiges ↵
> cd Wichtiges ↵
> pwd ↵
/home/toni/Vorkurs/Wichtiges/
> cd .. ↵
> pwd ↵
/home/toni/Vorkurs/
```

ls (*list*): zeigt eine Liste der Namen der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Dateien die mit einem „.“ anfangen, meist Systemdateien, werden nicht angezeigt.

ls -a : zeigt eine Liste der Namen *aller* (engl.: *all*) Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden an. Auch Dateien die mit einem „.“ anfangen, werden angezeigt. Bei dem `-a` handelt es sich um eine *Option*, die dem Befehl übergeben wird. Optionen werden mit einem oder zwei Bindestrichen eingeleitet. Dabei können mehrer

Option

3. Einführung in die Bedienung von Unix-Systemen

Optionen gemeinsam übergeben werden, ohne dass erneut ein Bindestrich eingegeben werden muss. Wird dem Kommando als Argument der absolute oder relative Pfad zu einem Verzeichnis angegeben, so werden die Namen der in diesem Verzeichnis enthaltenen Dateien angezeigt.

```
> ls ↵
Wichtiges sichtbareDatei1.txt sichtbareDatei2.pdf
> ls -a ↵
. .. Wichtiges sichtbareDatei1.txt sichtbareDatei2.pdf
> ls -a Wichtiges ↵
. ..
```

`ls -l`: zeigt eine Liste der Namen und Zusatzinformationen (l für engl.: *long*) der Dateien und Verzeichnisse, die sich im aktuellen Verzeichnis befinden. Die Einträge ähneln dem Folgenden.

```
> ls -l ↵
-rw-r--r-- 1 alice users 2358 Jan 15 14:23 protokoll.pdf
```

Von rechts nach links gelesen, sagt uns diese Zeile, dass die Datei „protokoll.pdf“ um 14:23 Uhr am 15. Januar dieses Jahres erstellt wurde. Die Datei ist 2358 Byte groß, und gehört der Gruppe „users“, insbesondere gehört sie der Benutzerin „alice“ und es handelt sich um eine (1) Datei. Dann kommen 10 Positionen an denen die Zeichen -, r oder w, stehen. Der Strich (-) an der linkensten Position zeigt an, dass es sich hierbei um eine gewöhnliche Datei handelt. Bei einem Verzeichnis würde an dieser Stelle ein d (für *directory*) stehen. Dann folgen die Zugriffsrechte. Die ersten drei Positionen sind für die Zugriffsrechte der Besitzerin (engl.: *owner*) der Datei. In diesem Fall darf die Besitzerin alice die Datei lesen (*read*) und verändern (*write*). Alice darf die Datei aber nicht ausführen (*x execute*). Eine gewöhnliche .pdf-Datei möchte man aber auch nicht ausführen. Die Ausführungsrechte sind wichtig für Verzeichnisse und Programme. Die mittleren drei Positionen geben die Zugriffsrechte der Gruppe (engl.: *group*) an. Die Gruppe users darf hier die Datei lesen, aber nicht schreiben. Die letzten drei Positionen sind für die Zugriffsrechte aller andern Personen (engl.: *other*). Auch diesen ist gestattet die Datei zu lesen, sie dürfen sie aber nicht verändern.

`chmod` (*change mode*): ändert die Zugriffsberechtigungen auf eine Datei. Dabei muss dem Programm die Datei, deren Zugriffsrechte man ändern will, als Argument übergeben werden. Ferner muss man dem Programm mitteilen, wessen (*user,group,other*, oder *all*) Rechte man wie ändern (+ hinzufügen, - wegnehmen) will.

```
> chmod go +w protokoll.pdf ↵
```

Dieser Befehl gibt der Gruppe g und allen anderen Nutzer:innen o Schreibrechte +w für die Datei protokoll.pdf. Vermutlich ist es keine gute Idee, der ganzen Welt die Erlaubnis zu erteilen die Datei zu ändern.

```
> chmod o -rw protokoll.pdf ↵
```

Zugriffs-
rechte

nimmt allen anderen Nutzer:innen `o` die Schreibrechte wieder weg `-w` und nimmt ihnen auch die Leserechte `-r`.

Alternativ kann die gewünschte Änderung auch als Oktalzahl eingegeben werden. Für die Erklärung dazu verweisen wir auf das Internet, oder die *man-page*, s.u.

`man` (*manual*): zeigt die Hilfe-Seiten zu dem, als Argument übergebenen, Kommando an.

```

MKDIR(1)                                User Commands                                MKDIR(1)

NAME
  mkdir - make directories

SYNOPSIS
  mkdir [OPTION]... DIRECTORY...

DESCRIPTION
  Create the DIRECTORY(ies), if they do not already exist.

  Mandatory arguments to long options are mandatory for short options
  too.

Manual page mkdir(1) line 1 (press h for help or q to quit)

```

Abbildung 3.3.: man page des Befehls `mkdir`

3.1.4. History und Autovervollständigung

Ein sehr nützliches Hilfsmittel beim Arbeiten mit der Shell ist die *history*. Alle in der Shell eingegebenen Befehle werden in der history gespeichert. Mit den Cursortasten `↑` und `↓` kann man in der history navigieren. `↑` holt den zuletzt eingegebenen Befehl in die Eingabezeile, ein erneutes Drücken von `↑` den vorletzten, usw. `↓` arbeitet in die andere Richtung, also z.B. vom vorletzten Befehl zum zuletzt eingegebenen Befehl. Mit den Cursortasten `←` und `→`, kann innerhalb des Befehls navigiert werden, um Änderungen vorzunehmen.

History

Ein weiteres nützliches Hilfsmittel ist die *Autovervollständigung*. Ist der Anfang eines Befehls, oder eines Datei- (oder Verzeichnis-) Namens eingegeben, so kann der Name durch Betätigen der *Tab*-Taste `↵` automatisch vervollständigt werden, solange der angegebene Anfang eindeutig ist. Ist dies nicht der Fall, so wird mit nochmaligem Betätigen der *Tab*-Taste `↵`, eine Liste aller in Frage kommenden Vervollständigungen angezeigt (Abb. 3.4).

Autovervollständigung

3. Einführung in die Bedienung von Unix-Systemen

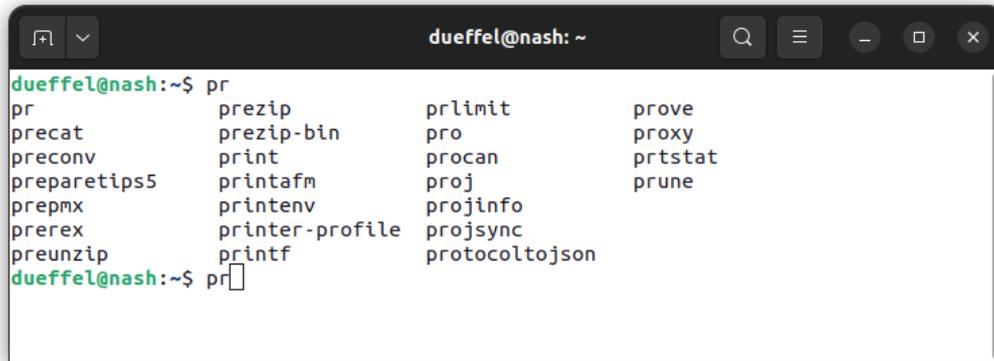


Abbildung 3.4.: Autovervollständigung für die Eingabe `pr`

3.2. Editieren und Textdateien

Bislang ging es um Dateien, Verzeichnisse und das allgemeine Bedienen einer Shell. Im Fokus dieses Abschnittes stehen die Textdateien. Diese werden für uns relevant, da sie unter anderem den *Code* der geschriebenen Programme beherbergen werden. Ein *Texteditor* ist ein Programm, welches das Erstellen und Verändern von Textdateien erleichtert.

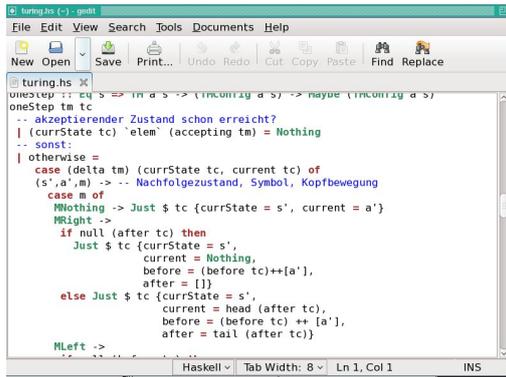
Es gibt unzählige Texteditoren. Unter KDE ist z.B. der Editor `kate` (<http://kate-editor.org/>), unter Gnome der Editor `gedit` (<http://projects.gnome.org/gedit/>) sehr empfehlenswert. Diese können allerdings nur im graphischen Modus ausgeführt werden, dafür ist ihre Bedienung dank vorhandener Mausbedienung recht komfortabel. Beide Editoren unterstützen Syntax-Highlighting für gängige Programmiersprachen. Sie können entweder über die entsprechenden Kommandos aus einer Shell heraus gestartet werden, oder über das Startmenü geöffnet werden. Unter Windows oder MacOS empfiehlt es sich für den Vorkurs einen Editor mit Syntax-Highlighting für die Programmiersprache Haskell zu verwenden, z.B. „Notepad ++“ (<http://notepad-plus-plus.org/>) für Windows und „TextWrangler“ (<http://www.barebones.com/products/textwrangler/>) für Mac OS X . Weitere Allround-Editoren sind Emacs (<http://www.gnu.org/software/emacs/>) und XEmacs (<http://www.xemacs.org/>), deren Bedienung allerdings gewöhnungsbedürftig ist.

Abbildung 3.5 zeigt einige Screenshots von Editoren, wobei eine Quellcode-Datei der Programmiersprache Haskell geöffnet ist.

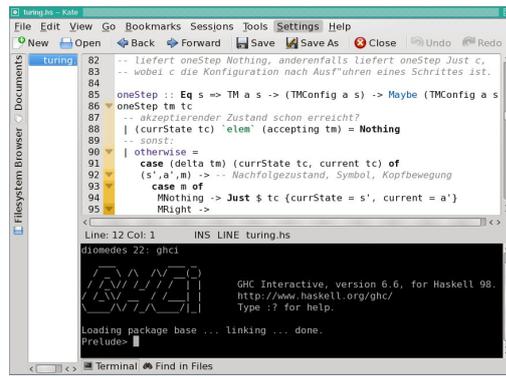
Hat man kein graphisches Interface zur Verfügung, so kann man einen Text-basierten Editor verwenden. Ein solcher weit verbreiteter Texteditor heißt `vi` (sprich: [vi: ai]). Dieser steht nicht nur in der RBI zur Verfügung, er ist auf fast jedem Unix-System vorhanden. Wir werden kurz auf seine Bedienung eingehen. Mit dem Befehl `vi` wird der Editor¹ gestartet. Der Befehl `vi /tmp/irgendeinedatei` startet `vi` und öffnet sogleich eine Sicht auf die angegebene Datei. Diese Sicht nennt man *Buffer*, hier findet das Editieren statt. Nach dem Öffnen und nach dem Speichern stimmt der Inhalt des Buffers mit dem der korrespondierenden Datei überein. Der

¹Oftmals handelt es sich schon um eine verbesserte Variante `vim` (für `vi improved`). Diese Unterscheidung soll uns hier nicht kümmern.

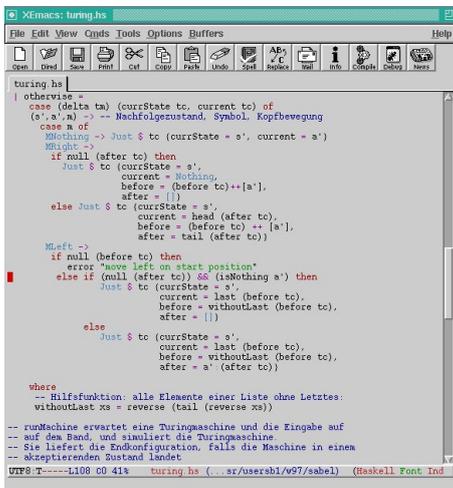
3.2. Editieren und Textdateien



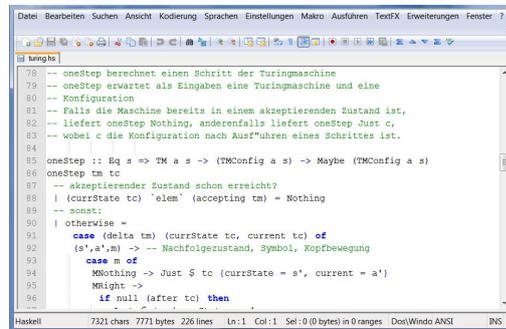
Screenshot gedit



Screenshot kate



Screenshot xemacs



Screenshot Notepad++ (MS Windows)

Abbildung 3.5.: Screenshots verschiedener Editoren

vi unterscheidet einige Betriebsmodi, die sich wie folgt beschreiben lassen. Wichtig dabei ist die Position des *Cursors*, auf die sich die meisten Aktionen beziehen.

1. Im *Befehlsmodus* werden Tastatureingaben als Befehle aufgefasst. Unter einem Befehl kann man sich das vorstellen, was man einem Herausgeber (engl.: editor) zurufen würde, bäte man ihn um Änderungen an einem Text. In diesem Modus befindet sich vi nach dem Starten. vi versteht Befehle wie „öffne/speichere diese und jene Datei“ (:e diese, :w jene), „Lösche die Zeile, in der sich der Cursor befindet!“ (dd) oder „Tausche den Buchstaben unterm Cursor durch den folgenden aus!“ (r). Natürlich ist es auch möglich, den Cursor zu bewegen, etwa mit den Pfeiltasten oder mit mannigfachen anderen Tasten, die Bewegungen in alle möglichen Positionen erlauben. Viele Neider halten :q! für den wichtigsten aller vi-Befehle. Dieser führt jedoch nur zum Programmabbruch (engl.: quit) ohne vorheriges Speichern.
2. Im *EINFÜGEN-* und *ERSETZEN-Modus* erscheinen die eingegebenen Zeichen direkt auf dem Bildschirm, im Buffer. Im ersteren wird Text neu hinzugefügt, im zweiten werden bereits vorhandene Zeichen überschrieben. In diese Modi gelangt man vom Befehlsmodus aus mit den Tasten I bzw. R. Zwischen ihnen kann mit der Einfg-Taste hin- und

3. Einführung in die Bedienung von Unix-Systemen

hergewechselt werden. Durch Betätigen der  -Taste gelangt man wieder zurück in den Befehlsmodus.

3. Die VISUELLEN Modi erlauben das Markieren eines Bereiches des Buffers, um sodann Befehle abzusetzen, die sich auf den markierten Bereich beziehen. Befindet man sich im Befehlsmodus, so gelangt man mit der  -Taste in den gewöhnlichen visuellen Modus. Dieser ermöglicht das Markieren eines Textbereiches durch Bewegen des Cursors. Nach Eingabe und Ausführung eines Befehls – etwa  , „markierten Bereich löschen“ – findet man sich im Befehlsmodus wieder. Auch die  -Taste führt zurück in den Befehlsmodus.

Wir stellen fest, dass ein Einstieg in `vi` mit dem Lernen einiger Tastenkürzel und Schlüsselworte einhergehen muss. Das lohnt sich aber für alle, die oft mit Text arbeiten. Hat man einmal die Grammatik der Befehle, die `vi` akzeptiert, verstanden, so wird das Editieren von Text zum Kinderspiel und geht schnell von der Hand. Wer das Tippen im Zehnfingersystem beherrscht und einzusetzen vermag, weiss schon, dass sich anfänglicher Mehraufwand auszahlen kann.

Ein Schnelleinstieg in `vi` ist mit Hilfe einer vorbereiteten Anleitung möglich. Diese lässt sich in einer Shell mit dem Befehl `vimtutor` starten. In dieser Anleitung sind einige Funktionen des Editors zum sofortigen Ausprobieren aufbereitet worden. Danach empfiehlt es sich, in einer Kurzreferenz zu stöbern.

4. Programmieren und Programmiersprachen

In diesem Kapitel wird zunächst kurz und knapp erklärt, was eine Programmiersprache ist und wie die verschiedenen Programmiersprachen grob eingeteilt werden können. Anschließend wird kurz erläutert, wie der Interpreter GHCi für die funktionale Programmiersprache Haskell (insbesondere auf den Rechnern der RBI) verwendet wird. Genauere Details zum Programmieren in Haskell werden erst im nächsten Kapitel erläutert.

4.1. Programme und Programmiersprachen

Ein Rechner besteht (vereinfacht) aus dem Prozessor (bestehend aus Rechenwerk, Steuerwerk, Registern, etc.), dem Hauptspeicher, Ein- und Ausgabegeräten (Festplatten, Bildschirm, Tastatur, Maus, etc.) und einem Bus-System über den die verschiedenen Bestandteile miteinander kommunizieren (d.h. Daten austauschen).

Rechner

Ein *ausführbares* Computerprogramm ist eine Folge von *Maschinencodebefehlen*, auch *Maschinenprogramm* genannt. Ein einzelner Maschinencodebefehl ist dabei eine Operation, die der Prozessor direkt ausführen kann (z.B. Addieren zweier Zahlen, Lesen oder Beschreiben eines Speicherregisters), d.h. diese Befehle „versteht“ der Prozessor und kann sie direkt verarbeiten (d.h. ausführen). Die Ausführung eines ganzen Maschinenprogramms besteht darin, die Folge von Maschinencodebefehlen nacheinander abzuarbeiten und dabei den Speicher zu manipulieren (d.h. zu verändern, oft wird auch vom „Zustand“ des Rechners gesprochen, damit ist dann die gesamte Speicherbelegung gemeint).

Maschinenprogramm

Allerdings sind Maschinenprogramme eher schwierig zu erstellen und für Programmier:innen schwer zu verstehen. Deshalb gibt es sogenannte *höhere Programmiersprachen*, die es Menschen erlauben, besser verständliche Programme zu erstellen. Diese Programme versteht der Computer allerdings nicht direkt, d.h. sie sind für den Computer nicht ausführbar. Programme in höheren Programmiersprachen werden auch als *Quelltext* oder *Quellcode* bezeichnet. Damit aus dem Quelltext (geschrieben in einer höheren Programmiersprache) ein für den Computer verständliches (und daher ausführbares) Maschinenprogramm wird, ist eine weitere Zutat erforderlich: Entweder kann ein *Compiler* benutzt werden, oder ein *Interpreter*. Ein Compiler ist ein *Übersetzer*: Er übersetzt den Quelltext in ein Maschinenprogramm. Ein *Interpreter* hingegen führt das Programm schrittweise aus (d.h. der Interpreter ist ein Maschinenprogramm und interpretiert das Programm der höheren Programmiersprache). Neben der Übersetzung (durch den Compiler) bzw. der Ausführung (durch den Interpreter) führt ein solches Programm noch weitere Aufgaben durch. Z.B. wird geprüft, ob der Quelltext tatsächlich ein gültiges Programm der Programmiersprache ist. Ist dies nicht der Fall, so gibt ein

höhere
Program-
mierspra-
che
Quellcode
Compiler
Interpreter

4. Programmieren und Programmiersprachen

guter Compiler/Interpreter eine Fehlermeldung aus, die mitteilt, an welcher Stelle der Fehler steckt. Je nach Programmiersprache und je nach Compiler/Interpreter können hier schon Programmierfehler erkannt und mithilfe der Fehlermeldung korrigiert werden.

Es gibt unzählige verschiedene (höhere) Programmiersprachen. Wir werden gleich auf die Charakteristika eingehen, die Programmiersprachen unterscheiden. Diese Kriterien heißen auch Programmiersprachenparadigmen oder Programmierstile.

Im Allgemeinen werden Programmiersprachen in *imperative* und in *deklarative* Programmiersprachen unterschieden.

4.1.1. Imperative Programmiersprachen

„Imperativ“ stammt vom lateinischen Wort „imperare“ ab, was „befehlen“ bedeutet. Tatsächlich besteht der Programmcode eines imperativen Programms aus einzelnen Befehlen (auch *Anweisungen* genannt), die nacheinander ausgeführt werden und den Zustand (d.h. Speicher) des Rechners verändern. Dies klingt sehr ähnlich zu den bereits erwähnten Maschinenprogrammen. Der Unterschied liegt darin, dass in höheren Programmiersprachen die Befehle komplexer und für Menschen verständlicher sind, und dass meistens vom tatsächlichen Speicher abstrahiert wird, indem sogenannte Programmvariablen verwendet werden. Diese sind im Grunde Namen für Speicherbereiche, wobei im Programm nur die Namen verwendet werden, und die Abbildung der Namen auf den tatsächlichen Speicher durch den Compiler oder den Interpreter geschieht (der:die Programmierer:in braucht sich hierum nicht zu kümmern). Z.B. kann X für eine Variable stehen. Wird in einem imperativen Programm die Variable X beispielsweise im *Ausdruck* $X + 5$ verwendet, so ist die Bedeutung hierfür im Normalfall: Lese den Wert von X (d.h. schaue in die zugehörige Stelle im Speicher) und addiere dann die Zahl 5 dazu. Das Verändern des Speichers geschieht in imperativen Sprachen üblicherweise mithilfe der *Zuweisung*, die oft durch $:=$ dargestellt wird. Hinter dem Befehl (bzw. der Zuweisung) $X := 10$ steckt die Bedeutung: Weise dem Speicherplatz, der durch X benannt ist, den Wert 10 zu.

Ein imperatives Programm besteht aus einer Folge solcher Befehle, die bei der Ausführung sequentiell (d.h. nacheinander) abgearbeitet werden. Hierbei werden noch sogenannte *Kontrollstrukturen* verwendet, die den Ablauf des Programmes steuern können. *Verzweigungen* als auch *Schleifen* werden als solche Kontrollstrukturen bezeichnet. Verzweigungen sind Wenn-Dann-Abfragen, die je nachdem, ob ein bestimmtes Kriterium erfüllt ist, das eine oder das andere Programm ausführen. Schleifen ermöglichen es, eine Befehlsfolge wiederholt (oder sogar beliebig oft) auszuführen.

Es gibt verschiedene Unterklassen von imperativen Programmiersprachen, die sich meist dadurch unterscheiden, wie man den Programmcode strukturieren kann. Z.B. gibt es *prozedurale Programmiersprachen*, die es erlauben den Code durch Prozeduren zu strukturieren und zu gruppieren. Eine *Prozedur* ist dabei ein Teilprogramm, das immer wieder (von verschiedenen anderen Stellen des Programmcodes) aufgerufen werden kann. Das spart Programmcode, da ein immer wieder verwendetes Programmstück nur einmal programmiert werden muss. Bekannte prozedurale Programmiersprachen sind z.B. *C*, *Fortran* und *Pascal*.

Eine weitere Unterklasse der imperativen Programmiersprachen, sind die sogenannten *objektorientierten Programmiersprachen*. In objektorientierten Programmiersprachen werden Programme durch sogenannte *Klassen* dargestellt. Klassen geben das Muster vor, wie Instanzen

prozedurale
Programmiersprache

objektorientierte
Programmiersprache

dieser Klasse (Instanzen werden auch *Objekte* genannt) aussehen. Klassen bestehen im Wesentlichen aus *Attributen* und *Methoden*. Attribute legen die Eigenschaften fest, die ein Objekt haben muss (z.B. eine Klasse `Auto` vorstellen, welche die Attribute `Höchstgeschwindigkeit`, `Gewicht` und `Kennzeichen` hat). Methoden definieren, ähnlich wie Prozeduren, Programme, die das Objekt verändern können (z.B. verändern des Kennzeichens). Über die Methoden können Objekte jedoch auch miteinander kommunizieren, indem eine Methode eines Objekts eine andere Methode eines anderen Objekts aufruft. Man sagt dazu auch: „die Objekte versenden Nachrichten untereinander“.

Die Strukturierungsmethode in objektorientierten Programmiersprachen ist die *Vererbung*. Hierdurch können Unterklassen erzeugt werden, dabei übernimmt die Unterklasse sämtliche Attribute und Methoden der Oberklasse und kann noch eigene hinzufügen.

Wird ein objektorientiertes Programm ausgeführt, so werden Objekte als Instanzen von Klassen erzeugt und durch Methodenaufrufe werden Nachrichten zwischen den Objekten ausgetauscht. Objekte werden dabei im Speicher des Rechners abgelegt. Da der Zustand der Objekte bei der Ausführung des System verändert wird, wirken die Methodenaufrufe wie Befehle (die den Zustand des Systems bei der Ausführung des Programms verändern). Deshalb zählen objektorientierte Sprachen zu den imperativen Programmiersprachen. Bekannte objektorientierte Programmiersprache sind z.B. *Java*, *C++* und *C#*, aber die meisten modernen imperativen Sprachen unterstützen auch die objektorientierte Programmierung (z.B. *Modula-3*, *Python*, *Ruby*, ...).

4.1.2. Deklarative Programmiersprachen

„Deklarativ“ stammt vom lateinischen Wort „declarare“ was „erklären“ oder auch „beschreiben“ heißt. Programme in deklarativen Programiersprachen beschreiben das Ergebnis des Programms. Dafür wird jedoch im Allgemeinen nicht genau festgelegt, *wie* das Ergebnis genau berechnet wird. Es wird eher beschrieben, *was* berechnet werden soll. Hierin liegt ein großer Unterschied zu imperativen Sprachen, denn diese geben genau an, wie der Speicher manipuliert werden soll, um das gewünschte Ergebnis zu erhalten. Programme deklarativer Programmiersprachen beschreiben im Allgemeinen nicht die Speicheroperationen, sondern bestehen aus (oft mathematischen) *Ausdrücken*. Zur Ausführung des Programms werden diese Ausdrücke *ausgewertet*. In der Schule wird eine solche Auswertung arithmetischer Ausdrücke oft per Hand ausgeführt. Zur Bestimmung des Wertes des Ausdrucks $(5 \cdot 10 + 3 \cdot 8)$, wird der Ausdruck ausgewertet (in der Schule meist als „ausrechnen“ bezeichnet), z.B. durch die Rechnung $(5 \cdot 10 + 3 \cdot 8) = (50 + 3 \cdot 8) = (50 + 24) = 74$.

In deklarativen Programmiersprachen gibt es wie in imperativen Sprachen auch Variablen, diese meinen aber meistens etwas anderes: Während in imperativen Sprachen Variablen veränderbare Speicherbereiche bezeichnen, bezeichnen Variablen in deklarativen Programmiersprachen im Allgemeinen bestimmte, feststehende Ausdrücke, d.h. insbesondere ist ihr Wert *unveränderlich*. Z.B. kann man in der deklarativen Sprache Haskell schreiben `let x = 5+7 in x*x`. Hierbei ist die Variable `x` nur ein Name für den Ausdruck `5+7` und ihr Wert ist stets 12^1 .

Da deklarativen Programmiersprachen i.A. keine Speicheroperationen direkt durchführen, sind meist weder eine Zuweisung noch Schleifen zur Programmierung vorhanden (diese mani-

¹Der Operator `*` bezeichnet in fast allen Computeranwendungen und Programmiersprachen die Multiplikation.

4. Programmieren und Programmiersprachen

pulieren nämlich den Speicher). Um jedoch Ausdrücke wiederholt auszuwerten, wird *Rekursion* bzw. werden *rekursive Funktionen* verwendet. Diese werden wir später genauer betrachten. Hier sei nur kurz erwähnt, dass eine Funktion rekursiv ist, wenn sie sich selbst aufrufen kann.

Deklarative Sprachen lassen sich grob aufteilen in *funktionale Programmiersprachen* und *logische Programmiersprachen*.

logische Programmiersprache Bei logischen Programmiersprachen besteht ein Programm aus einer Menge von logischen Formeln und Fakten (wahren Aussagen). Zur Laufzeit werden mithilfe logischer Folgerungen (sogenannter Schlussregeln) neue wahre Aussagen hergeleitet, die dann das Ergebnis der Ausführung darstellen. Die bekannteste Vertreterin der logischen Programmiersprachen ist die Sprache *Prolog*.

funktionale Programmiersprache Ein Programm in einer *funktionalen Programmiersprache* besteht aus einer Menge von Funktionsdefinitionen (im engeren mathematischen Sinn) und evtl. selbstdefinierten Datentypen. Das Ausführen eines Programms entspricht dem Auswerten eines Ausdrucks, d.h. das Resultat ist ein einziger Wert. In rein funktionalen Programmiersprachen wird der Zustand des Rechners nicht explizit durch das Programm manipuliert, d.h. es treten bei der Ausführung keine sogenannten *Seiteneffekte* (d.h. sichtbare Speicheränderungen) auf. Tatsächlich wäre zur Auswertung eigentlich gar kein Rechner notwendig. Das Ergebnis könnte auch stets per Hand mit Zettel und Stift berechnet werden (was jedoch oft sehr mühsam wäre). In rein funktionalen Programmiersprachen gilt das Prinzip der *referentiellen Transparenz*: Das Ergebnis der Anwendung einer Funktion auf Argumente, hängt ausschließlich von den Argumenten ab, oder umgekehrt: Die Anwendung einer gleichen Funktion auf gleiche Argumente liefert stets das gleiche Resultat.

Aus dieser Sicht klingen funktionale Programmiersprachen sehr mathematisch und es ist zunächst nicht klar, was außer arithmetischen Berechnungen damit anzufangen ist. Die Mächtigkeit der funktionalen Programmierung ergibt sich erst dadurch, dass die Funktionen nicht nur auf Zahlen, sondern auf beliebig komplexen Datenstrukturen (z.B. Listen, Bäumen, Paaren, usw.) operieren dürfen. So können z.B. Funktionen definiert werden, die als Eingabe einen Text erhalten und Schreibfehler im Text erkennen und den verbesserten Text als Ausgabe zurückliefern. Oder es können Funktionen programmiert werden, die Webseiten nach bestimmten Schlüsselwörtern durchsuchen, etc.

Prominente Vertreter von funktionalen Programmiersprachen sind *Standard ML*, *OCaml*, Microsofts *F#* und *Haskell*. Im Vorkurs und in der Veranstaltung „Programmierparadigmen und Compilerbau“ werden wir die Sprache Haskell behandeln und benutzen.

4.2. Haskell: Einführung in die Benutzung

Die Programmiersprache *Haskell* ist eine pure funktionale Programmiersprache. Der Name „Haskell“ stammt von dem amerikanischen Mathematiker und Logiker Haskell B. Curry. Haskell ist eine relativ neue Programmiersprache, der erste Standard wurde 1990 festgelegt. Damals gab es bereits einige andere funktionale Programmiersprachen. Um eine einheitliche Sprache festzulegen wurde ein Komitee gegründet, das die standardisierte funktionale Programmiersprache Haskell



Das aktuelle Haskell-Logo

entwarf. Inzwischen wurden mehrere Revisionen des Standards veröffentlicht (1999 und leicht verändert 2003 wurde *Haskell 98* veröffentlicht, im Juli 2010 wurde der *Haskell 2010*-Standard veröffentlicht).

Die wichtigste Informationsquelle zu Haskell ist die Homepage von Haskell:

`http://www.haskell.org`

Dort findet man neben dem Haskell-Standard zahlreiche Links, die auf Implementierungen der Sprache Haskell (d.h. Compiler und Interpreter), Bücher, Dokumentationen, Anleitungen, Tutorials, Events, etc. verweisen.

Es gibt einige Implementierungen der Sprache Haskell. Wir werden im Vorkurs den am weitesten verbreiteten *Glasgow Haskell Compiler* (kurz GHC) benutzen. Dieser stellt neben einem Compiler für Haskell auch einen Interpreter (den sogenannten GHCi) zur Verfügung. Für den Vorkurs und auch für die Veranstaltung „Programmierparadigmen und Compilerbau“ ist die Benutzung des Interpreters ausreichend. Die Homepage des GHC ist `http://www.haskell.org/ghc`.

GHC,GHCi

4.2.1. GHCi auf den Rechnern der RBI

Auf den RBI-Rechnern ist der Compiler GHC und der Interpreter GHCi bereits installiert. Der Haskell-Interpreter GHCi ist unter `/opt/rbi/bin/ghci` zu finden, d.h. er kann mit dem Kommando `ghci` (ausgeführt in einer Shell) gestartet werden².

4.2.2. GHCi auf dem eigenen Rechner installieren

Für die Installation des GHC und GHCi sollte GHCup benutzt werden, diese beinhaltet neben GHC und GHCi einige nützliche Werkzeuge und Programmbibliotheken für Haskell. Unter

`https://www.haskell.org/ghcup/`

stehen installierbare Versionen für verschiedene Betriebssysteme (MS Windows, Mac OS) zum Download zur Verfügung.

Informationen zur Installation von Haskell und eines Editors der Syntax-Highlighting etc. für Haskell unterstützt, sowie erste Beispiele mit dem GHCi, sind unter

`https://www.haskell.org/get-started/`

zu finden. Im Vorkurs wird ausschließlich mit dem GHCi gearbeitet, der auch für die Veranstaltung PPDC ausreichend ist.

Nach der Installation ist es i.A. möglich den GHCi zu starten, indem man das Kommando `ghci` in eine Shell eintippt.

²wenn die Umgebungsvariable `PATH` nicht richtig gesetzt ist, muss das Kommando `/opt/rbi/bin/ghci` benutzt werden

4. Programmieren und Programmiersprachen

4.2.3. Bedienung des Interpreters

Nachdem wir den Interpreter gestartet haben (siehe Abschnitt 4.2.1) erhalten wir auf dem Bildschirm

```
ghci ↵
GHCi, version 8.10.7: http://www.haskell.org/ghc/  :? for help
Prelude>
```

In neueren Versionen des GHCi steht statt `Prelude>` der Eingabeprompt `ghci>`

Wir können nun Haskell-Ausdrücke eingeben und auswerten lassen, z.B. einfache arithmetische Ausdrücke:

```
Prelude> 1+1 ↵
2
Prelude> 3*4 ↵
12
Prelude> 15-6*3 ↵
-3
Prelude> -3*4 ↵
-12
```

Werden ültige Ausdrücke eingegeben (also Ausdrücke, die keine Haskell-Ausdrücke sind), so zeigt der Interpreter eine Fehlermeldung an, z.B.

```
Prelude> 1+2+3+4+ ↵ <interactive>:1:9:
  parse error (possibly incorrect indentation or mismatched brackets)
```

Es ist hilfreich, sich die Fehlermeldung durchlesen, bevor man sich auf die Suche nach dem Fehler macht. Sie enthält zumindest die Information, an welcher Stelle des Ausdrucks der Interpreter einen Fehler vermutet: Die Zahlen 1:9 verraten, dass der Fehler in der 1. Zeile und der 9. Spalte ist. Tatsächlich ist dort das `+`-Zeichen, dem keine Zahl folgt die Fehlerursache. Wir werden später noch weitere Fehlermeldungen betrachten.

Neben Haskell-Ausdrücken können wir im Interpreter auch Kommandos zur Steuerung des Interpreters absetzen. Diese werden stets mit einem Doppelpunkt eingeleitet (damit der Interpreter selbst sie von Haskell-Programmen unterscheiden kann). Einige wichtige Kommandos sind:

- | | |
|-------------------------------------|--|
| <code>:quit</code> | Verlassen des Interpreters. Der Interpreter wird gestoppt, und es wird zur Shell zurück gekehrt. |
| <code>:help</code> | Der Interpreter zeigt einen Hilfetext an. Insbesondere wird eine Übersicht über die verfügbaren Kommandos gegeben. |
| <code>:load <i>Dateiname</i></code> | Lädt den Haskell-Quellcode der entsprechenden Datei, die Dateinendung von <i>Dateiname</i> muss <code>.hs</code> lauten. |
| <code>:reload</code> | Lädt die aktuelle geladene Datei erneut (hilfreich, wenn die aktuell geladene Datei gerade im Editor geändert wurde). |

4.2.4. Quelltexte erstellen und im GHCi laden

Normalerweise wird der Quelltext eines Haskell-Programms in einem Editor (siehe Kapitel 3) erstellt und als Haskell-Programm in einer Datei abgespeichert. Die Datei-Endung muss hierbei `.hs` lauten. Abbildung 4.1 zeigt den Inhalt eines ganz einfachen Programms. Es definiert für den Namen `wert` (eigentlich ist `wert` eine Funktion, die allerdings keine Argumente erhält) die Zeichenfolge „Hallo Welt!“ (solche Zeichenfolgen heißen auch *String*).

```
wert = "Hallo Welt!"
```

Abbildung 4.1.: Inhalt der Datei `hallowelt.hs`

Nach dem Erstellen der Datei können wir sie im GHCi mit dem Kommando `:load hallowelt.hs` laden:

```
> ghci ↵
GHCi, version 8.10.7: http://www.haskell.org/ghc/  :? for help
Prelude> :load hallowelt.hs ↵
[1 of 1] Compiling Main ( hallowelt.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Das funktioniert aber nur, wenn der GHCi aus dem Verzeichnis heraus gestartet wurde, das die Datei `hallowelt.hs` enthält. Angenommen wir starten den GHCi aus einem Verzeichnis, aber `hallowelt.hs` liegt in einem Unterverzeichnis namens `programme`, dann müssen wir dem `:load`-Kommando nicht nur den Dateinamen (`hallowelt.hs`), sondern auch den Verzeichnispfad mit übergeben (also `:load programme/hallowelt.hs`). Im folgenden zeigen wir was passiert, wenn der Verzeichnispfad vergessen wird, und wie es richtig ist:

```
> ghci ↵
GHCi, version 8.10.7: http://www.haskell.org/ghc/  :? for help
Prelude> :load hallowelt.hs ↵

<no location info>: can't find file: hallowelt.hs
Failed, modules loaded: none.
Prelude> :load programme/hallowelt.hs ↵
[1 of 1] Compiling Main ( programme/hallowelt.hs, interpreted )
Ok, modules loaded: Main.
```

Nach dem Laden des Quelltexts, können wir die dort definierten Funktionen im Interpreter auswerten lassen. Wir haben nur die Parameter-lose Funktion `wert` definiert, also lassen wir diese mal auswerten:

4. Programmieren und Programmiersprachen

```
*Main> wert ↵  
"Hallo Welt!"
```

Wir erhalten als Ergebnis der Berechnung gerade den String „Hallo Welt“. Wirklich rechnen musste der Interpreter hierfür nicht, da der `wert` schon als dieser String definiert wurde.

Abbildung 4.2 zeigt den Inhalt einer weiteren Quelltextdatei (namens `einfacheAusdruecke.hs`). Dort werden wieder nur parameter-lose Funktionen definiert,

```
zwei_mal_Zwei = 2 * 2  
  
oft_fuenf_addieren = 5 + 5 + 5 + 5 +5 + 5 + 5 + 5 + 5 + 5 + 5  
  
beides_zusammenzaehlen = zwei_mal_Zwei + oft_fuenf_addieren
```

Abbildung 4.2.: Inhalt der Datei `einfacheAusdruecke.hs`

aber rechts vom `=` stehen *Ausdrücke*, die keine Werte sind. Die Funktion `zwei_mal_Zwei` berechnet das Produkt $2 \cdot 2$, die Funktion `oft_fuenf_addieren` addiert elf mal 5 und die Funktion `beides_zusammenzaehlen` addiert die Werte der beiden anderen Funktionen. Dafür *ruft* sie die beiden anderen Funktionen *auf*.

Nach dem Laden des Quelltexts, kann man die Werte der definierten Funktionen berechnen lassen:

```
> ghci ↵  
GHCi, version 8.10.7: http://www.haskell.org/ghc/  :? for help  
Prelude> :load einfacheAusdruecke.hs ↵  
[1 of 1] Compiling Main ( einfacheAusdruecke.hs, interpreted )  
Ok, modules loaded: Main.  
*Main> zwei_mal_Zwei ↵  
4  
*Main> oft_fuenf_addieren ↵  
55  
*Main> beides_zusammenzaehlen ↵  
59  
*Main> 3*beides_zusammenzaehlen ↵  
177
```

Funktions-
aufruf

Funktions-
namen

Beachte, dass Funktionsnamen in einer Quelltextdatei mit einem Kleinbuchstaben oder dem Unterstrich `_` beginnen *müssen*, anschließend können Großbuchstaben und verschiedene Sonderzeichen folgen. Eine Missachtung dieser Regel führt beim Laden des Quelltexts zu einer Fehlermeldung. Ist `zwei_mal_Zwei` z.B. durch `Zwei_mal_Zwei` ersetzt, passiert Folgendes:

```
Prelude> :load grossKleinschreibungFalsch.hs
[1 of 1] Compiling Main ( grossKleinschreibungFalsch.hs, interpreted )

grossKleinschreibungFalsch.hs:3:1:
    Not in scope: data constructor 'Zwei_mal_Zwei'
Failed, modules loaded: none.
```

Zur Erklärung der Fehlermeldung: Der GHCi nimmt aufgrund der Großschreibung an, dass `Zwei_mal_Zwei` ein Datenkonstruktor ist (und daher keine Funktion). Was Datenkonstruktoren sind, erläutern wir an dieser Stelle nicht, wichtig ist nur, dass der GHCi die Funktion nicht als eine solche akzeptiert.

4.2.5. Kommentare in Quelltexten

In Quelltexten sollten neben dem eigentlichen Programmcode auch Erklärungen und Erläuterungen stehen, die insbesondere umfassen: Was macht jede der definierten Funktionen? Wie funktioniert die Implementierung, bzw. was ist die Idee dahinter? Der Ausdruck dafür ist, dass der Quelltext *dokumentiert* sein soll. Der Grund hierfür ist, dass man selbst nach einiger Zeit den Quelltext wieder verstehen, verbessern und ändern kann, oder auch andere Programmier:innen den Quelltext verstehen. Um dies zu bewerkstelligen, gibt es in allen Programmiersprachen die Möglichkeit *Kommentare* in den Quelltext einzufügen, wobei diese speziell markiert werden müssen, damit der Compiler oder Interpreter zwischen Quellcode und Dokumentation unterscheiden kann. In Haskell gibt es zwei Formen von Kommentaren:

Zeilenkommentare: Das Einfügen von zwei Minuszeichen gefolgt von einem Leerzeichen, d.h. „--“, führt dazu, dass alle Zeichen danach bis zum Zeilenende als Kommentar erkannt und dementsprechend vom GHCi ignoriert werden. Zum Beispiel:

```
wert = "Hallo Welt" -- ab hier ist ein Kommentar bis zum Zeileende

wert2 = "Nochmal Hallo Welt"

-- Diese ganze Zeile ist auch ein Kommentar!
```

Kommentarblöcke: Um einen Textblock (auch über mehrere Zeilen) als Kommentar zu markieren, muss er in spezielle Klammern gesetzt werden. Die öffnende Klammer besteht aus den beiden Symbolen `{-` und die schließende Klammer besteht aus `-}`. Zum Beispiel ist im folgenden Programm nur `wert2 = "Hallo Welt"` Programmcode, der Rest ist ein Kommentar:

4. Programmieren und Programmiersprachen

```
{- Hier steht noch gar keine Funktion,  
   da auch die naechste Zeile noch im  
   Kommentar ist  
  
wert = "Hallo Welt"  
  
   gleich endet der Kommentar -}  
  
wert2 = "Hallo Welt"
```

4.2.6. Fehler

Syntaxfehler Jede:r Programmierer:in erstellt Programme, die fehlerhaft sind. Es gibt jedoch verschiedene Arten von Fehlern, die wir kurz erläutern. *Syntaxfehler* entstehen, wenn der Quelltext ein Programm enthält, das syntaktisch nicht korrekt ist. Z.B. kann das Programm Symbole enthalten, die die Programmiersprache nicht erlaubt (z.B. $5!$ für die Fakultät von 5, aber die Sprache verfügt nicht über den Operator $!$). Ein anderer syntaktischer Fehler ist das Fehlen von Klammern, oder allgemein die nicht korrekte Klammerung (z.B. $(5+3)$ oder auch $(4*2))$).
Logischer Fehler Eine andere Klasse von Fehlern sind sogenannte *logische* oder *semantische* Fehler. Ein solcher Fehler tritt auf, wenn das Programm nicht die gewünschte Funktionalität, aber irgendeine andere Funktionalität, implementiert.

Typfehler Syntaxfehler sind eher leicht zu entdecken (auch automatisch), während logische Fehler eher schwer zu erkennen sind. In die Klasse der semantischen Fehler fallen auch sogenannte *Typfehler*. Ein Typfehler tritt auf, wenn Konstrukte der Sprache miteinander verwendet werden, obwohl sie nicht zueinander passen. Ein Beispiel ist $1+'A'$, da eine Zahl nicht mit einem Buchstaben addiert werden kann. Wir werden später Typfehler genauer behandeln.

Compilezeitfehler Programmierfehler können auch danach unterschieden werden, *wann* sie auftreten. Hierbei wird in *Compilezeitfehler* und *Laufzeitfehler* unterschieden. Wenn ein Fehler bereits beim Übersetzen des Programms in Maschinensprache entdeckt wird, dann liegt ein Compilezeitfehler vor. In diesem Fall bricht der Compiler die Übersetzung ab, und meldet den Fehler. Auch der GHCi liefert solche Fehlermeldungen. Betrachte beispielsweise den folgenden Quellcode in der Datei `fehler.hs`

```
-- 1 und 2 addieren  
eineAddition = (1+2)  
  
-- 2 und 3 multiplizieren  
eineMultiplikation = (2*3))
```

Laden wir diese Datei im GHCi , so erhalten wir eine Fehlermeldung:

```
Prelude> :load fehler.hs
[1 of 1] Compiling Main           ( fehler.hs, interpreted )

fehler.hs:5:27: parse error on input ‘)’
Failed, modules loaded: none.
```

Es empfiehlt sich, die ausgegebene Fehlermeldung genau zu lesen, denn sie verrät oft, wo sich der Fehler versteckt (in diesem Fall in Zeile 5 und Spalte 27), um welche Art von Fehler es sich handelt (in diesem Fall ein „parse error“, was einem Syntaxfehler entspricht), und welches Symbol zum Fehler geführt hat (in diesem Fall die schließende Klammer).

Ein *Laufzeitfehler* ist ein Fehler, der nicht vom Compiler entdeckt wird, und daher erst beim *Ausführen* des Programms auftritt. Das Programm bricht dann normalerweise ab. Gute Programme führen eine Fehlerbehandlung durch und vermeiden daher das plötzliche Abbrechen des Programms zur Laufzeit. Ein Beispiel für einen Laufzeitfehler ist die Division durch 0.

Laufzeit-
fehler

```
Prelude> div 10 0
*** Exception: divide by zero
```

Andere Beispiele sind das Lesen von Dateien, die gar nicht existieren, etc.

Stark und statisch getypte Programmiersprache wie Haskell haben den Vorteil, dass viele Fehler bereits vom Compiler entdeckt werden, und daher Laufzeitfehler vermieden werden.

5. Grundlagen der Programmierung in Haskell

In diesem Kapitel werden wir die Programmierung in Haskell genauer kennen lernen. Es sei vorab erwähnt, dass wir im Rahmen dieses Vorkurses nicht den gesamten Umfang von Haskell behandeln können. Wir werden viele wichtige Konzepte von Haskell in diesem Rahmen überhaupt nicht betrachten: Eine genaue Betrachtung des Typenkonzepts fehlt ebenso wie wichtige Datenstrukturen (z.B. Arrays, selbstdefinierte Datentypen, unendliche Datenstrukturen). Auch auf die Behandlung von Ein- und Ausgabe unter Verwendung der `do`-Notation werden wir nicht eingehen.

Ziel des Vorkurses ist vielmehr das Kennenlernen und der Umgang mit der Programmiersprache Haskell, da diese im ersten Teil der Veranstaltung „Programmierparadigmen und Compilerbau“ weiter eingeführt und verwendet wird.

5.1. Ausdrücke und Typen

Das Programmieren in Haskell ist im Wesentlichen ein Programmieren mit *Ausdrücken*. Ausdrücke werden aufgebaut aus kleineren Unterausdrücken. Wenn wir als Beispiel einfache arithmetische Ausdrücke betrachten, dann sind deren kleinsten Bestandteile Zahlen $1, 2, \dots$. Diese können durch die Anwendung von arithmetischen Operationen $+, -, *$ zu größeren Ausdrücken zusammengesetzt werden, beispielsweise $17*2+5*3$. Fast jeder Ausdruck besitzt einen *Wert*, im Fall von elementaren Ausdrücken, wie 1 oder 2 , kann der Wert direkt abgelesen werden. Der Wert eines zusammengesetzten Ausdrucks muss berechnet werden. In Haskell stehen, neben arithmetischen, eine ganze Reihe andere Arten von Ausdrücken bereit, um Programme zu formulieren. Die wichtigste Methode, um in Haskell Ausdrücke zu konstruieren, ist die Anwendung von Funktionen auf Argumente. In obigem Beispiel können die arithmetischen Operatoren $+, -, *$ als Funktionen aufgefasst werden, die infix ($17*2$) anstelle von präfix ($* 17 2$) notiert werden.

Ausdruck

Wert

Haskell ist eine *streng getypte* Programmiersprache, d.h. jeder Ausdruck und jeder Unterausdruck hat einen *Typ*. Werden Ausdrücke aus kleineren Ausdrücken zusammengesetzt, so müssen die Typen stets zueinander passen, z.B. dürfen Funktionen, die auf Zahlen operieren, nicht auf Strings angewendet werden, da die Typen nicht zueinander passen.

Typ

Mit dem Kommando `:type Ausdruck` kann man sich im GHCi, den Typ eines Ausdrucks anzeigen lassen, z.B.:

```
Prelude> :type 'C'   
'C' :: Char
```

5. Grundlagen der Programmierung in Haskell

Char

Es heißt auch der Ausdruck (bzw. der Wert) `'C'` hat den Typ `Char`. Hierbei steht `Char` für *Character*, d.h. welches das englische Wort für Buchstabe ist. Typnamen wie `Char` beginnen in Haskell stets mit einem Großbuchstaben. Mit dem Kommando `:set +t` wird der GHCi in einen Modus versetzt, so dass er stets zu jedem Ergebnis auch den Typ anzeigt, das letzte berechnete Ergebnis ist im GHCi immer über den Namen `it` (für „es“) ansprechbar, deshalb wird der Typ des Ergebnisses in der Form `it :: Typ` angezeigt. Wir probieren dies aus:

```
> ghci
GHCi, version 8.10.7: http://www.haskell.org/ghc/  :? for help
Prelude> :load einfacheAusdruecke.hs
[1 of 1] Compiling Main ( einfacheAusdruecke.hs, interpreted )
Ok, modules loaded: Main.
*Main> :set +t
*Main> zwei_mal_Zwei
4
it :: Integer
*Main> oft_fuenf_addieren
55
it :: Integer
*Main> beides_zusammenzaehlen
59
it :: Integer
```

Integer

Die Ergebnisse aller drei Berechnungen sind vom Typ `Integer`, der beliebig große ganze Zahlen darstellt. In Haskell ist es auch möglich, den Typ eines Ausdrucks selbst anzugeben (die Schreibweise ist *Ausdruck :: Typ*), der GHCi überprüft dann, ob der Typ richtig ist. Ein Beispiel hierzu ist:

```
*Main> 'C'::Char
'C'
it :: Char
*Main> 'C'::Integer

<interactive>:2:1:
  Couldn't match expected type 'Integer' with actual type 'Char'
  In the expression: 'C' :: Integer
  In an equation for 'it': it = 'C' :: Integer
```

Da `'C'` ein Zeichen und daher vom Typ `Char` ist, schlägt die Typisierung als Zahl vom Typ `Integer` fehl.

Typinferenz

In den allermeisten Fällen muss Typ *nicht* selbst angegeben werden, da der GHCi den Typ selbstständig herleiten kann. Dieses Feature wird auch *Typinferenz* genannt. Manchmal ist es jedoch hilfreich, sich selbst die Typen zu überlegen, sie anzugeben, und anschließend den GHCi zum Überprüfen zu verwenden.

Im folgenden Abschnitt stellen wir einige vordefinierte und eingebaute Datentypen vor, die Haskell zur Verfügung stellt.

5.2. Basistypen

5.2.1. Wahrheitswerte: Der Datentyp Bool

Die Boolesche Logik (benannt nach dem englischen Mathematiker George Boole) ist eine sehr einfache Logik (siehe auch Kap: 2). Sie wird in vielen Bereichen insbesondere in wohl jeder Programmiersprache verwendet, sie ist zudem *die* Grundlage der Hardware von Rechnern. Wir führen sie hier nur sehr oberflächlich ein. Die Boolesche Logik baut auf sogenannten atomaren Aussagen auf. Für eine solche Aussage kann nur gelten: Die Aussage ist entweder *wahr* oder die Aussage ist *falsch*. Beispiele für solche atomaren Aussagen aus dem natürlichen Sprachgebrauch sind:

- Heute regnet es.
- Mein Auto hat die Farbe blau.
- Fritz ist noch nicht erwachsen.

In Programmiersprachen haben Aussagen häufig eine Form wie $x < 5$, die entweder wahr oder falsch sind.

In Haskell gibt es den Datentyp `Bool`, der die beiden Wahrheitswerte „wahr“ und „falsch“ durch die *Datenkonstruktoren* `True` und `False` darstellt, d.h. wahre Aussagen werden zu `True`, falsche Aussagen werden zu `False` ausgewertet. Datenkonstruktoren (wie `True` und `False`) beginnen in Haskell nie mit einem Kleinbuchstaben, daher fast immer mit einem Großbuchstaben.

Bool
Daten-
konstruktor

Die Boolesche Logik stellt sogenannte *Junktoren* zur Verfügung, um aus atomaren Aussagen und Wahrheitswerten größere Aussagen (auch aussagenlogische *Formeln* genannt) zu erstellen, d.h. Formeln werden gebildet, indem kleinere Formeln (die auch nur Variablen oder Wahrheitswerte sein können) mithilfe von *Junktoren* verknüpft werden.

Junktor

Die drei wichtigsten Junktoren sind die Negation, die Und-Verknüpfung und die Oder-Verknüpfung. Hier stimmt der natürlichsprachliche Gebrauch von „nicht“, „und“ und „oder“ mit der Bedeutung der Junktoren überein.

Um z.B. die atomare Aussage A_1 : „Heute regnet es.“ zu negieren, würde man sagen A_2 : „Heute regnet es *nicht*“. Dabei gilt offensichtlich: Die Aussage A_1 ist genau dann wahr, wenn die Aussage A_2 falsch ist, und umgekehrt. In mathematischer Notation wird die Negation als \neg geschrieben. Anstelle von A_2 könnte man also auch $\neg A_1$ schreiben.

\neg

Die Bedeutung (Auswertung) von Booleschen Junktoren wird oft durch eine Wahrheitstabelle repräsentiert. In dieser ist für alle möglichen Belegungen der atomaren Aussagen, der Wert der Verknüpfung angegeben. Für \neg sieht diese Wahrheitstabelle so aus:

A	$\neg A$
wahr	falsch
falsch	wahr

Die Und-Verknüpfung (mathematisch durch das Symbol \wedge repräsentiert) verknüpft zwei Aussagen durch ein „Und“. Z.B. würde die Verundung der Aussagen A_1 : „Heute regnet es“ und A_2 : „Ich esse heute Pommes“ natürlichsprachlich durch „Heute regnet es *und* ich esse heute

\wedge

5. Grundlagen der Programmierung in Haskell

Pommes“ ausgedrückt werden. In der Booleschen Logik schreibt man $A_1 \wedge A_2$. Die so zusammengesetzte Aussage ist nur dann wahr, wenn beide Operanden des \wedge wahr sind, im Beispiel ist die Aussage also nur wahr, wenn es heute regnet und ich heute Pommes esse.

Die Wahrheitstabelle zum logischen Und ist daher:

A_1	A_2	$A_1 \wedge A_2$
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

∨

Die Oder-Verknüpfung (mathematisch durch das Symbol \vee repräsentiert) verknüpft analog zwei Aussagen durch ein „Oder“, d.h. $A_1 \vee A_2$ entspricht der Aussage „Es regnet heute *oder* ich esse heute Pommes“. Die so gebildete Aussage ist wahr, sobald einer der Operanden wahr ist (aber auch dann, wenn beide wahr sind). D.h.: Wenn es heute regnet, ist die Aussage wahr, wenn ich heute Pommes esse, ist die Aussage wahr, und auch wenn beide Ereignisse eintreten, ist die Aussage wahr.

Die Wahrheitstabelle zum logischen Oder ist:

A_1	A_2	$A_1 \vee A_2$
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

In Haskell gibt es vordefinierte Operatoren für die Junktoren: `not`, `&&` und `||`, wobei die folgende Tabelle deren Entsprechung zu den mathematischen Junktoren zeigt:

Bezeichnung	Mathematische Notation	Haskell-Notation
logische Negation	$\neg F$	<code>not F</code>
logisches Und	$F_1 \wedge F_2$	<code>F1 && F2</code>
logisches Oder	$F_1 \vee F_2$	<code>F1 F2</code>

Wir probieren die Wahrheitswerte und die Junktoren gleich einmal im GHCi mit einigen Beispielen aus:

```

*Main> not True ↵
False
it :: Bool
*Main> True && True ↵
True
it :: Bool
*Main> False && True ↵
False
it :: Bool
*Main> False || False ↵
False
it :: Bool
*Main> True || False ↵
True
it :: Bool

```

5.2.2. Ganze Zahlen: Int und Integer

Für ganze Zahlen stellt Haskell zwei verschiedene Typen zur Verfügung: Der Typ `Int` hat als Werte die ganzen Zahlen im Bereich zwischen -2^{31} bis $2^{31} - 1$. Der zweite Typ `Integer` umfasst die gesamten ganzen Zahlen. Die Darstellung der Werte vom Typ `Int` und der Werte vom Typ `Integer` (im entsprechenden Bereich) ist identisch, d.h. wenn z.B. 1000 eingegeben wird, weiß der Interpreter eigentlich nicht, welcher Typ gemeint ist. Eine Möglichkeit ist, den Typ durch Angabe des Typs festzulegen, also `1000::Int` bzw. `1000::Integer` einzugeben. Wir kein Typ angegeben, so führt der Interpreter manchmal sogenanntes *Defaulting* durch (wenn es nötig ist) und nimmt automatisch den Typ `Integer` an. Fragen wir den Interpreter doch mal nach dem Typ von 1000:

`Int`,
`Integer`

```

Prelude> :type 1000 ↵
1000 :: (Num t) => t

```

Der ausgegebene Typ ist weder `Integer` noch `Int`, sondern `(Num t) => t`. Den Teil vor dem `=>` heißt *Typklassenbeschränkung*. Wir erklären Typklassen hier nicht genau, aber die Typangabe lässt sich wie folgt interpretieren: 1000 hat den Typ `t`, wenn `t` ein Typ der Typklasse `Num` ist¹. Sowohl `Int` als auch `Integer` sind Typen der Typklasse `Num`, d.h. der Compiler kann für die *Typvariable* `t` sowohl `Integer` als auch `Int` einsetzen. Die vordefinierten Operatoren für ganze Zahlen erläutern wir später.

5.2.3. Gleitkommazahlen: Float und Double

Haskell stellt die Typen `Float` und `Double` (mit doppelter Genauigkeit) für Gleitkomma-

`Float`,
`Double`

¹Typklassen bündeln verschiedene Typen und stellen gemeinsame Operationen auf ihnen bereit. Dadurch, dass die Typen `Int` und `Integer` in der `Num` Typklasse sind (der Typklasse für Zahlen), wird sichergestellt, dass typische Operationen auf Zahlen (Addition, Multiplikation, usw.) für Daten beider Typen bereitstehen.

5. Grundlagen der Programmierung in Haskell

zahlen (auch *Fließkommazahlen* genannt) zur Verfügung. Die Kommastelle wird dabei wie im Englischen üblich mit einem Punkt vom ganzzahligen Teil getrennt, insbesondere ist die Darstellung für Zahlen vom Typ `Float` und vom Typ `Double` identisch. Auch für Gleitkommazahlen gibt es eine Typklasse, die als Typklassenbeschränkung verwendet wird (die Klasse heißt `Fractional`). Für das Defaulting nimmt der GHCi den Typ `Double` an. Man kann analog wie bei ganzen Zahlen experimentieren:

```
Prelude> :type 10.5
10.5 :: (Fractional t) => t
```

Der Punkt muss nicht immer zwingend angegeben werden, denn auch `Float` und `Double` gehören zur Typklasse `Num`. Beachte, dass das Rechnen mit Gleitkommazahlen *ungenau* werden kann, da diese nur eine bestimmte Anzahl von Nachkommastellen berücksichtigen, wobei `Double` doppelt so viele Nachkommastellen berücksichtigt wie `Float`. Einige Aufrufe im GHCi, welche die Ungenauigkeit zeigen, sind:

```
Prelude> :set +t
Prelude> (1.0000001)::Float
1.0000001
it :: Float
Prelude> (1.00000001)::Float
1.0
it :: Float
Prelude> 1.0000000000000001
1.0000000000000001
it :: Fractional a => a
Prelude> (1.0000000000000001)
1.0
it :: Fractional a => a
```

5.2.4. Zeichen und Zeichenketten

Zeichen sind in Haskell eingebaut durch den Typ `Char`. Ein Zeichen wird eingerahmt durch einzelne Anführungszeichen, z.B. `'A'` oder `'&'`. Es gibt einige spezielle Zeichen, die alle mit einem `\` („Backslash“) eingeleitet werden. Dies sind zum einen sogenannte Steuersymbole zum anderen die Anführungszeichen und der Backslash selbst. Hier eine kleine Auflistung:

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
<code>'\\'</code>	Backslash <code>\</code>
<code>'\''</code>	einfaches Anführungszeichen <code>'</code>
<code>'\"'</code>	doppeltes Anführungszeichen <code>"</code>
<code>'\n'</code>	Zeilenumbruch
<code>'\t'</code>	Tabulator

Zeichenketten bestehen aus einer Folge von Zeichen. Sie sind in Haskell durch den Typ `String` implementiert und werden durch doppelte Anführungszeichen umschlossen, z.B. ist die Zeichenkette "Hallo" ein Wert vom Typ `String`. Beachte, dass der Typ `String` eigentlich nur eine Abkürzung für `[Char]` ist, d.h. Strings sind gar nicht primitiv eingebaut (daher eigentlich auch keine Basistypen), sondern nichts anderes als Listen von Zeichen. Die eckigen Klammern um `Char` im Typ `[Char]` besagt, dass der Typ eine Liste von Zeichen ist. Wir werden Listen in einem späteren Abschnitt genauer erörtern.

String

5.3. Funktionen und Funktionstypen

In Haskell sind einige Funktionen bzw. Operatoren für Zahlen bereits vordefiniert, die arithmetischen Operationen Addition, Subtraktion, Multiplikation und Division sind über die Operatoren `+`, `-`, `*` und `/` verfügbar:

Bezeichnung	mathematische Notation(en)	Haskell-Notation
Addition	$a + b$	$a + b$
Subtraktion	$a - b$	$a - b$
Multiplikation	$a \cdot b$, oft auch ab	$a * b$
Division	$a : b$, a/b , $a \div b$	a / b

Die Operatoren werden (wie in der Mathematik) als infix-Operationen verwendet, z.B. `3 * 6`, `10.0 / 2.5`, `4 + 5 * 4`. Der GHCi beachtet dabei die Punkt-vor-Strich-Regel. Z.B. ist der Wert von `4 + 5 * 4` die Zahl 24 und *nicht* 36 (was dem Ausdruck `(4 + 5) * 4` entspräche). Das Minuszeichen wird nicht nur für die Subtraktion, sondern auch zur Darstellung negativer Zahlen verwendet (in diesem Fall präfix, d.h. vor der Zahl). Manchmal muss im Interpreter durch zusätzlich geklammert werden, damit eindeutig ist, ob es sich um eine negative Zahl oder um die Subtraktion handelt. Ein Beispiel hierfür ist der Ausdruck `2 * -2`. Wird dieser so im Interpreter eingegeben, erhält man eine Fehlermeldung:

```
Prelude> 2 * -2 
<interactive>:2:1:
Precedence parsing error
  cannot mix '*' [infixl 7] and
  prefix '-' [infixl 6] in the
  same infix expression
```

Werden jedoch Klammern um den Ausdruck gesetzt (`-2`), so kann der Interpreter den Ausdruck erkennen:

```
Prelude> 2 * (-2) 
-4
Prelude>
```

5. Grundlagen der Programmierung in Haskell

Zum Vergleich von Werten stellt Haskell die folgenden Vergleichsoperatoren zur Verfügung:

Bezeichnung	mathematische Notation(en)	Haskell-Notation
Gleichheit	$a = b$	$a == b$
Ungleichheit	$a \neq b$	$a /= b$
echt kleiner	$a < b$	$a < b$
echt größer	$a > b$	$a > b$
kleiner oder gleich	$a \leq b$	$a <= b$
größer oder gleich	$a \geq b$	$a >= b$

Die Vergleichsoperatoren nehmen zwei Argumente und liefern einen Wahrheitswert, d.h. sie werten zu `True` oder `False` aus. Beachte, dass der Gleichheitstest `==` und der Ungleichheitstest `/=` nicht nur für Zahlen definiert ist, sondern für viele Datentypen verwendet werden kann. Z.B. können damit auch Boolesche Werte verglichen werden.

Wir testen einige Beispiele:

```
Prelude> 1 == 3 
False
Prelude> 3*10 == 6*5 
True
Prelude> True == False 
False
Prelude> False == False 
True
Prelude> 2*8 /= 64 
True
Prelude> 2+8 /= 10 
False
Prelude> True /= False 
True
```

Einige Beispiele für die Vergleiche sind:

```
Prelude> 5 >= 5 
True
Prelude> 5 > 5 
False
Prelude> 6 > 5 
True
Prelude> 4 < 5 
True
Prelude> 4 < 4 
False
Prelude> 4 <= 4 
True
```

5.3. Funktionen und Funktionstypen

Die arithmetischen Operationen und die Vergleichsoperationen sind auch Funktionen, sie besitzen jedoch die Spezialität, dass sie infix verwendet werden. Die meisten Funktionen werden präfix verwendet, wobei die Notation in Haskell leicht verschieden ist, von der mathematischen Notation. Betrachten wir die Funktion, die zwei Zahlen erhält und den Rest einer Division mit Rest berechnet. Sei f diese Funktion. In der Mathematik würde zur Anwendung der Funktion f auf die zwei Argumente 10 und 3 der Ausdruck $f(10, 3)$ verwendet, um anschließend den Wert 1 zu berechnen (denn $10 \div 3 = 3$ Rest 1). In Haskell ist dies ganz ähnlich, jedoch werden die Argumente der Funktion *nicht* geklammert, sondern jeweils durch ein Leerzeichen getrennt. D.h. in Haskell würde man `f 10 3` oder auch `(f 10 3)` schreiben. Der Grund für diese Schreibweise liegt vor allem darin, dass man in Haskell auch *partiell anwenden* darf, d.h. bei der Anwendung von Funktionen auf Argumente müssen nicht immer genügend viele Argumente vorhanden sein. Für unser Beispiel bedeutet dies: Man darf in Haskell auch schreiben `f 10`. Mit der geklammerten Syntax wäre dies nur schwer möglich. `f 10` ist in diesem Fall immer noch eine Funktion, die *ein* weiteres Argument erwartet. In Haskell ist die Funktion zur Berechnung des Restes tatsächlich schon vordefiniert sie heißt dort `mod`. Analog gibt es die Funktion `div`, die den ganzzahligen Anteil der Division mit Rest berechnet. Wir probieren dies gleich einmal im Interpreter aus:

partielle
Anwen-
dung

`mod`, `div`

```
Prelude> mod 10 3 ↵
1
Prelude> div 10 3 ↵
3
Prelude> mod 15 5 ↵
0
Prelude> div 15 5 ↵
3
Prelude> (div 15 5) + (mod 8 6) ↵
5
```

Tatsächlich können Infix-Operatoren wie `+` und `==` auch in Präfix-Schreibweise verwendet werden, indem man sie in Klammern setzt. Z.B. kann `5 + 6` auch als `(+) 5 6` oder `True == False` auch als `(==) True False` geschrieben werden. Umgekehrt können zweistellige Funktionen wie `mod` und `div` auch in Infix-Schreibweise verwendet werden, indem der Funktionsname in Hochkommata (diese werden durch die Tastenkombination `↑` + `↓` eingegeben) umschlossen wird, d.h. `mod 5 3` ist äquivalent zu `5 'mod' 3`.

In Haskell haben nicht nur Basiswerte einen Typ, sondern jeder Ausdruck hat einen Typ. Daher haben auch Funktionen einen Typ. Als einfaches Beispiel betrachten wir erneut die Booleschen Funktionen `not`, `(&&)` und `(||)`. Die Funktion `not` erwartet einen booleschen Ausdruck (d.h. einen Ausdruck vom Typ `Bool`) und liefert einen Wahrheitswert vom Typ `Bool`. In Haskell wird dies wie folgt notiert: Die einzelnen Argumenttypen und der Ergebnistyp werden durch `->` voneinander getrennt, d.h. `not :: Bool -> Bool`. Der GHCi verrät uns dies auch:

Funktions-
typ, `->`

```
Prelude> :type not ↵
not :: Bool -> Bool
```

5. Grundlagen der Programmierung in Haskell

Die Operatoren `(&&)` und `(||)` erwarten jeweils zwei boolesche Ausdrücke und liefern als Ergebnis wiederum einen booleschen Wert. Daher ist ihr Typ `Bool -> Bool -> Bool`. Wir überprüfen dies im GHCi:

```
Prelude> :type (&&)
(&&) :: Bool -> Bool -> Bool
Prelude> :type (||)
(||) :: Bool -> Bool -> Bool
```

Allgemein hat eine Haskell-Funktion f , die n Eingaben (d.h. Argumente) erwartet, einen Typ der Form

$$f :: \underbrace{\text{Typ}_1}_{\substack{\text{Typ des} \\ \text{1. Arguments}}} \rightarrow \underbrace{\text{Typ}_2}_{\substack{\text{Typ des} \\ \text{2. Arguments}}} \rightarrow \dots \rightarrow \underbrace{\text{Typ}_n}_{\substack{\text{Typ des} \\ \text{n. Arguments}}} \rightarrow \underbrace{\text{Typ}_{n+1}}_{\substack{\text{Typ des} \\ \text{Ergebnisses}}}$$

Der Pfeil `->` in Funktionstypen ist rechts-geklammert, d.h. z.B. ist `(&&) :: Bool -> Bool -> Bool` äquivalent zu `(&&) :: Bool -> (Bool -> Bool)`. Das passt nämlich gerade zur partiellen Anwendung: Der Ausdruck `(&&) True` ist vom Typ her eine Funktion, die noch einen Booleschen Wert als Eingabe nimmt und als Ausgabe einen Booleschen Wert liefert. Daher kann `(&&)` auch als Funktion interpretiert werden, die *eine* Eingabe vom Typ `Bool` erwartet und als Ausgabe eine *Funktion* vom Typ `Bool -> Bool` liefert. Auch dies lässt sich im GHCi testen:

```
Prelude> :type ((&&) True)
((&&) True) :: Bool -> Bool
Prelude> :type ((&&) True False)
((&&) True False) :: Bool
```

Kehren wir zurück zu den Funktionen und Operatoren auf Zahlen. Wie muss der Typ von `mod` und `div` aussehen? Beide Funktionen erwarten zwei ganze Zahlen und liefern als Ergebnis eine ganze Zahl. Wenn wir vom Typ `Integer` als Eingaben und Ausgaben ausgehen, bedeutet dies: `mod` erwartet als erste Eingabe eine Zahl vom Typ `Integer`, als zweite Eingabe eine Zahl vom Typ `Integer` und liefert als Ausgabe eine Zahl vom Typ `Integer`. Daher ist der Typ von `mod` (wie auch von `div`) der Typ `Integer -> Integer -> Integer` (wenn wir uns auf den `Integer`-Typ beschränken). Daher können wir schreiben:

```
mod :: Integer -> Integer -> Integer
div :: Integer -> Integer -> Integer
```

Fragen wir den Interpreter nach den Typen von `mod` und `div` so erhalten wir etwas allgemeinere Typen:

```
Prelude> :type mod
mod :: (Integral a) => a -> a -> a
Prelude> :type div
div :: (Integral a) => a -> a -> a
```

5.3. Funktionen und Funktionstypen

Dies liegt daran, dass `mod` und `div` auch für andere ganze Zahlen verwendet werden können. Links vom `=>` steht hier wieder eine sogenannte Typklassenbeschränkung. Rechts vom `=>` steht der eigentliche Typ, der jedoch die Typklassenbeschränkung einhalten muss. Zu verstehen ist dies folgendermaßen: `mod` hat den Typ `a -> a -> a` für alle Typen `a` die `Integral`-Typen sind (dazu gehören u.a. `Integer` und `Int`).

Ähnliches gilt für den Gleichheitstest (`==`): Er kann für jeden Typ verwendet werden, der zur Typklasse `Eq` gehört:

```
Prelude> :type (==)   
(==) :: (Eq a) => a -> a -> Bool
```

Wir fragen die Typen einiger weiterer bereits eingeführter Funktionen und Operatoren im GHCi ab:

```
Prelude> :type (==)   
(==) :: (Eq a) => a -> a -> Bool  
Prelude> :type (<)   
(<) :: (Ord a) => a -> a -> Bool  
Prelude> :type (<=)   
(<=) :: (Ord a) => a -> a -> Bool  
Prelude> :type (+)   
(+) :: (Num a) => a -> a -> a  
Prelude> :type (-)   
(-) :: (Num a) => a -> a -> a
```

Bei einer Anwendung einer Funktion auf Argumente müssen die Typen der Funktion stets zu den Typen der Argumente passen. Wendet man z.B. die Funktion `not` auf ein Zeichen (vom Typ `Char`) an, so passt der Typ nicht: `not` hat den Typ `Bool -> Bool` und erwartet daher einen booleschen Ausdruck. Der GHCi bemerkt dies sofort und produziert einen *Typfehler*:

```
Prelude> not 'C'   
  
<interactive>:2:5:  
  Couldn't match expected type 'Bool' with actual type 'Char'  
  In the first argument of 'not', namely 'C'  
  In the expression: not 'C'  
  In an equation for 'it': it = not 'C'
```

Sind Typklassen im Spiel (z.B. wenn Zahlen verwendet werden deren Typ noch nicht sicher ist), so sind die Fehlermeldungen manchmal etwas merkwürdig. Betrachte das folgende Beispiel:

```
Prelude> not 5 

<interactive>:2:5:
  No instance for (Num Bool) arising from the literal '5'
  In the first argument of 'not', namely '5'
  In the expression: not 5
  In an equation for 'it': it = not 5
```

In diesem Fall sagt der Compiler nicht direkt, dass die Zahl nicht zum Typ `Bool` passt, sondern er bemängelt, dass Boolesche Werte nicht der Klasse `Num` angehören, d.h. er versucht nachzuschauen, ob man die Zahl `5` nicht als Booleschen Wert interpretieren kann. Da das nicht gelingt (`Bool` gehört nicht zur Klasse `Num`, da Boolesche Werte keine Zahlen sind), erscheint die Fehlermeldung.

5.4. Einfache Funktionen definieren

Bisher haben wir vordefinierte Funktionen betrachtet, nun werden wir Funktionen selbst definieren. Z.B. kann man eine Funktion definieren, die jede Zahl verdoppelt als:

```
verdopple x = x + x
```

Funktions-
definition

Die Syntax für Funktionsdefinitionen in Haskell sieht folgendermaßen aus:

$$\text{funktion_Name } par_1 \dots par_n = \text{Haskell_Ausdruck}$$

Wobei *funktion_Name* eine Zeichenkette ist, die mit einem Kleinbuchstaben oder einem Unterstrich beginnt und den Namen der Funktion darstellt, unter dem sie aufgerufen werden kann. *par₁ ... par_n* sind die *formalen Parameter* der Funktion, in der Regel stehen hier verschiedene Variablen, beispielsweise *x, y, z*. Rechts vom Gleichheitszeichen folgt ein beliebiger Haskell-Ausdruck, der die Funktion definiert und bestimmt welcher Wert berechnet wird, hier dürfen die Parameter *par₁ ... par_n* verwendet werden.

Man darf dem Quelltext auch den Typ der Funktion hinzufügen². Beschränken wir uns auf `Integer`-Zahlen, so nimmt `verdopple` als Eingabe eine `Integer`-Zahl und liefert als Ausgabe ebenfalls eine `Integer`-Zahl. Daher ist der Typ von `verdopple` der Typ `Integer -> Integer`. Mit Typangabe erhält man den Quelltext:

```
verdopple :: Integer -> Integer
verdopple x = x + x
```

Schreibt man die Funktion in eine Datei und lädt sie anschließend in den `GHCi`, so kann man sie ausgiebig ausprobieren:

²Man muss dies i.A. jedoch nicht tun, da der `GHCi` die Typen auch selbst berechnen kann!

```

Prelude> :load programme/einfacheFunktionen.hs
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs)
Ok, modules loaded: Main.
*Main> verdopple 5
10
*Main> verdopple 100
200
*Main> verdopple (verdopple (2*3) + verdopple (6+9))
84

```

Haskell stellt *if-then-else*-Ausdrücke zur Verfügung. Diese werden zur *Fallunterscheidung* bzw. *Verzweigung* verwendet: Anhand des Wahrheitswerts einer Bedingung wird bestimmt, welchen Wert ein Ausdruck haben soll. Die Syntax ist

$$\text{if } b \text{ then } e_1 \text{ else } e_2.$$

Hierbei muss b ein Ausdruck vom Typ `Bool` sein, und die Typen der Ausdrücke e_1 und e_2 müssen identisch sein. Die Bedeutung eines solchen *if-then-else*-Ausdrucks ist: *Wenn b wahr ist (zu `True` ausgewertet), dann ist der Wert des gesamten Ausdrucks e_1 , anderenfalls (b wertet zu `False` aus) ist der Wert des gesamten Ausdrucks e_2 .* D.h. je nach Wahrheitswert von b „verzweigt“ die Funktion zu e_1 bzw. zu e_2 ³.

Wir betrachten einige Beispiele zu *if-then-else*:

- `if 4+5 > 7 then 100 else 1000` ist gleich zu 100, da $4+5 = 9$ und 9 größer als 7 ist.
- `if False then "Hallo" else "Hallihallo"` ist gleich zum String "Hallihallo", da die Bedingung falsch ist.
- `if (if 2 == 1 then False else True) then 0 else 1` ist gleich zu 0, da der innere *if-then-else*-Ausdruck gleich zu `True` ist.

Man kann mit *if-then-else*-Ausdrücken z.B. auch eine Funktion definieren, die nur gerade Zahlen verdoppelt:

```

verdoppleGerade :: Integer -> Integer
verdoppleGerade x = if even x then verdopple x else x

```

Die dabei benutzte Funktion `even` ist in Haskell bereits vordefiniert, sie testet, ob eine Zahl gerade ist. Die Funktion `verdoppleGerade` testet nun mit `even`, ob das Argument `x` gerade ist, und mithilfe einer Fallunterscheidung (*if-then-else*) wird entweder die Funktion `verdopple` mit `x` aufgerufen, oder (im *else*-Zweig) einfach `x` selbst zurück gegeben. Machen wir die Probe:

³Solche Fallunterscheidungen kennt man auch von imperativen Programmiersprachen, wobei sie dort jedoch eine etwas andere Bedeutung haben, da in Haskell e_1 und e_2 Ausdrücke sind und keine Befehle. Insgesamt ist in Haskell `if b then e1 else e2` wieder ein Ausdruck. Aus diesem Grund gibt es in Haskell (im Gegensatz zu vielen imperativen Programmiersprachen) kein *if-then*-Konstrukt, denn dann wäre der Wert von `if b then e` für falsches b nicht definiert!

5. Grundlagen der Programmierung in Haskell

```
*Main> :reload ↵
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )
Ok, modules loaded: Main.

*Main> verdoppleGerade 50 ↵
100
*Main> verdoppleGerade 17 ↵
17
```

Man kann problemlos mehrere `if-then-else`-Ausdrücke verschachteln und damit komplexere Funktionen implementieren. Z.B. kann man eine Funktion implementieren, die alle Zahlen kleiner als 100 verdoppelt, die Zahlen zwischen 100 und 1000 verdreifacht und andere Zahlen unverändert zurück gibt, als:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
              (if x <= 1000 then 3*x else x)
```

Die Haskell-Syntax beachtet die Einrückung, daher kann man die Klammern um das zweite `if-then-else` auch weglassen:

```
jenachdem :: Integer -> Integer
jenachdem x = if x < 100 then 2*x else
              if x <= 1000 then 3*x else x
```

Man muss jedoch darauf achten, dass z.B. die rechte Seite der Funktionsdefinition um mindestens ein Zeichen gegenüber dem Funktionsnamen eingerückt ist. Z.B. ist

```
jenachdem :: Integer -> Integer
jenachdem x =
if x < 100 then 2*x else
  if x <= 1000 then 3*x else x
```

falsch, da das erste `if` nicht eingerückt ist. Der GHCi meldet in diesem Fall einen Fehler:

```
Prelude> :reload ↵
[1 of 1] Compiling Main ( programme/einfacheFunktionen.hs )

programme/einfacheFunktionen.hs:9:1:
  parse error (possibly incorrect indentation)
Failed, modules loaded: none.
Prelude>
```

Wir betrachten eine weitere Funktion, die zwei Eingaben erhält, zum Einen einen Booleschen Wert b und zum Anderen eine Zahl x . Die Funktion soll die Zahl x verdoppeln, wenn b wahr ist, und die Zahl x verdreifachen, wenn b falsch ist:

```
verdoppeln_oder_verdreifachen :: Bool -> Integer -> Integer
verdoppeln_oder_verdreifachen b x =
  if b then 2*x else 3*x
```

Wir testen die Funktion im GHCi:

```
*Main> verdoppeln_oder_verdreifachen True 10 ↵
20
*Main> verdoppeln_oder_verdreifachen False 10 ↵
30
```

Wir können die Funktion `verdoppeln` von vorher nun auch unter Verwendung von `verdoppeln_oder_verdreifachen` definieren:

```
verdoppeln2 :: Integer -> Integer
verdoppeln2 x = verdoppeln_oder_verdreifachen True x
```

Man kann sogar das Argument x in diesem Fall weglassen, da wir ja partiell anwenden dürfen, d.h. eine noch elegantere Definition ist:

```
verdoppeln3 :: Integer -> Integer
verdoppeln3 = verdoppeln_oder_verdreifachen True
```

Hier sei nochmals auf die Typen hingewiesen: Den Typ `Bool -> Integer -> Integer` von `verdoppeln_oder_verdreifachen` kann man auch geklammert schreiben als `Bool -> (Integer -> Integer)`. Genau das haben wir bei `verdoppeln3` ausgenutzt, denn das Ergebnis von `verdoppeln3` ist der Rückgabewert der partiellen Anwendung von `verdoppeln_oder_verdreifachen` auf `True` und daher eine *Funktion* vom Typ `Integer -> Integer`. D.h. in Haskell gibt es keine Bedingung, dass eine Funktion als Ergebnistyp einen Basistyp haben muss, es ist (wie z.B. bei `verdoppeln3`) durchaus erlaubt auch Funktionen als Ergebnis zurück zu geben.

Etwas analoges gilt für die Argumente einer Funktion: Bisher waren dies stets Basistypen, es ist aber auch an dieser Stelle erlaubt, *Funktionen* selbst zu übergeben. Betrachte zum Beispiel die folgende Funktion:

```
wende_an_und_addiere f x y = (f x) + (f y)
```

Funktionen
als Para-
meter

5. Grundlagen der Programmierung in Haskell

Die Funktion `wende_an_und_addiere` erhält drei Eingaben: Eine Funktion `f` und zwei Zahlen `x` und `y`. Die Ausgabe berechnet sie wie folgt: Sie wendet die übergebene Funktion `f` sowohl einmal auf `x` als auch einmal auf `y` an und addiert schließlich die Ergebnisse. Z.B. kann man für `f` die Funktion `verdopple` oder die Funktion `jenachdem` übergeben:

```
*Main> wende_an_und_addiere verdopple 10 20 ↵
60
*Main> wende_an_und_addiere jenachdem 150 3000 ↵
3450
```

Wir können allerdings nicht ohne Weiteres jede beliebige Funktion an `wende_an_und_addiere` übergeben, denn die Typen müssen passen. Sehen wir uns den Typ von `wende_an_und_addiere` an:

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
```

Wenn man an den äußeren `->`-Pfeilen zerlegt, kann man die einzelnen Typen den Argumenten und dem Ergebnis zuordnen:

```
wende_an_und_addiere :: (Integer -> Integer) -> Integer -> Integer -> Integer
                        Typ von f      Typ von x      Typ von y      Typ des
                                                Ergebnisses
```

D.h. nur Funktionen, die einen Ausdruck vom Typ `Integer` erwarten und eine Zahl liefern können an `wende_an_und_addiere` übergeben werden. Beachte auch, dass man die Klammern im Typ

```
(Integer -> Integer) -> Integer -> Integer -> Integer
```

nicht weglassen darf, denn der Typ

```
Integer -> Integer -> Integer -> Integer -> Integer
```

ist implizit rechtsgeklammert, und entspricht daher dem Typ

```
Integer -> (Integer -> (Integer -> (Integer -> Integer))).
```

Funktionen, die andere Funktionen als Argumente akzeptieren oder zurückgeben, bezeichnet man als *Funktionen höherer Ordnung*.

Wir betrachten noch eine Funktion, die zwei Eingaben erhält und den String "Die Eingaben sind gleich!" als Ergebnis liefert, wenn die Eingaben gleich sind. Man könnte versucht sein, die Funktion so zu implementieren:

```
sonicht x x = "Die Eingaben sind gleich!"
```

Dies ist allerdings keine gültige Definition in Haskell, der GHCi beanstandet dies auch sofort:

Funktion
höherer
Ordnung

```
einfacheFunktionen.hs:31:9:
  Conflicting definitions for 'x'
  Bound at: einfacheFunktionen.hs:31:9
            einfacheFunktionen.hs:31:11
  In an equation for 'sonicht'
Failed, modules loaded: none.
```

Die Parameter einer Funktion müssen nämlich alle verschieden sein, d.h. für `sonicht` darf man nicht zweimal das gleiche `x` für die Definition verwenden. Richtig ist

```
vergleiche x y = if x == y then "Die Eingaben sind gleich!" else ""
```

Die Funktion `vergleiche` gibt bei Ungleichheit den leeren String `""` zurück. Der Typ von `vergleiche` ist `vergleiche :: (Eq a) => a -> a -> String`, d.h. er beinhaltet eine Typklassenbeschränkung (aufgrund der Verwendung von `==`): Für jeden Typ `a`, der zur Klasse `Eq` gehört, hat `vergleiche` den Typ `a -> a -> String`.

Wir betrachten als weiteres Beispiel die Funktion `zweimal_anwenden`:

```
zweimal_anwenden :: (a -> a) -> a -> a
zweimal_anwenden f x = f (f x)
```

Die Funktion erwartet eine Funktion und ein Argument und wendet die übergebene Funktion zweimal auf das Argument an. Da hier eine fast beliebige Funktion verwendet werden kann, enthält der Typ von `zweimal_anwenden` Typvariablen (das `a`). Für diese kann man einen beliebigen Typen einsetzen, allerdings muss für jedes `a` der selbe Typ eingesetzt werden. Ist der Typ einer Funktion (wie der Typ von `f` in `zweimal_anwenden`) von der Form `a -> a`, so handelt es sich um eine Funktion, deren Ergebnistyp und deren Argumenttyp identisch sein müssen, aber ansonsten nicht weiter beschränkt sind. Wir führen einige Tests durch:

5. Grundlagen der Programmierung in Haskell

```
*Main> zweimal_anwenden verdopple 10
40
*Main> zweimal_anwenden (wende_an_und_addiere verdopple 100) 10
640
*Main> zweimal_anwenden (vergleiche True) True

<interactive>:2:30:
  Couldn't match type 'Bool' with '[Char]'
  Expected type: String
  Actual type: Bool
  In the first argument of 'vergleiche', namely 'True'
  In the first argument of 'zweimal_anwenden', namely
    '(vergleiche True)'

<interactive>:2:36:
  Couldn't match type 'Bool' with '[Char]'
  Expected type: String
  Actual type: Bool
  In the second argument of 'zweimal_anwenden', namely 'True'
  In the expression: zweimal_anwenden (vergleiche True) True
```

Der letzte Test `zweimal_anwenden (vergleiche True) True` geht schief (hat einen Typfehler), da `(vergleiche True)` den Typ `Bool -> String` hat, und daher nicht für den Typ `(a -> a)` eingesetzt werden kann.

Da in den Typen von Ausdrücken und Funktion Typvariablen erlaubt sind, sagt man Haskell hat ein *polymorphes* Typsystem.

Polymorpher
Typ

5.5. Rekursion

Der Begriff Rekursion stammt vom lateinischen Wort „*recurrere*“ was „zurücklaufen“ bedeutet. Die Technik, die sich hinter der Rekursion versteckt, besteht darin, dass eine Funktion definiert wird, indem sie sich in der Definition selbst wieder aufruft. Ein altbekannter Spruch (Witz) zur Rekursion lautet:

„*Wer Rekursion verstehen will, muss Rekursion verstehen.*“

rekursiv Man nennt eine Funktion (direkt) *rekursiv*, wenn sie sich selbst wieder aufruft, d.h. wenn die Definition von f von der Form

$$f \dots = \dots f \dots$$

ist. Man kann diesen Begriff noch verallgemeinern, da f sich nicht unbedingt direkt selbst aufrufen muss, es geht auch indirekt über andere Funktionen, also in der Art:

```
f ... = ... g ...
g ... = ... f ...
```

In diesem Fall ruft `f` die Funktion `g` auf, und `g` ruft wieder `f` auf. Daher sind `f` und `g` *verschränkt rekursiv*.

Nicht jede rekursive Funktion ist sinnvoll, betrachte z.B. die Funktion

```
endlos_eins_addieren x = endlos_eins_addieren (x+1)
```

Die Auswertung der Funktion wird (egal für welchen Wert von `x`) nicht aufhören, da sich `endlos_eins_addieren` stets erneut aufrufen wird. Etwa in der Form:

```
endlos_eins_addieren 1
→ endlos_eins_addieren (1+1)
→ endlos_eins_addieren ((1+1)+1)
→ endlos_eins_addieren (((1+1)+1)+1)
→ ...
```

D.h. man sollte im Allgemeinen rekursive Funktionen so implementieren, dass irgendwann *sicher*⁴ ein Ende erreicht wird und kein rekursiver Aufruf mehr erfolgt. Dieses Ende nennt man auch *Rekursionsanfang*, den rekursiven Aufruf nennt man auch *Rekursionsschritt*. Wir betrachten als erste sinnvolle rekursive Funktion die Funktion `erste_rekursive_Funktion`:

Rekursions-
anfang und
Rekursions-
schritt

```
erste_rekursive_Funktion x = if x <= 0 then 0 else
                             x+(erste_rekursive_Funktion (x-1))
```

Der Rekursionsanfang wird gerade durch den `then`-Zweig des `if-then-else`-Ausdrucks abgedeckt: Für alle Zahlen die kleiner oder gleich 0 sind, findet kein rekursiver Aufruf statt, sondern es wird direkt 0 als Ergebnis geliefert. Der Rekursionsschritt besteht darin `erste_rekursive_Funktion` mit `x` erniedrigt um 1 aufzurufen und zu diesem Ergebnis den Wert von `x` hinzuzählen. Man kann leicht überprüfen, dass für jede ganze Zahl `x` der Aufruf `erste_rekursive_Funktion x` irgendwann beim Rekursionsanfang landen muss, daher *terminiert* jede Anwendung von `erste_rekursive_Funktion` auf eine ganze Zahl.

Es bleibt zu klären, was `erste_rekursive_Funktion` berechnet. Wir können zunächst einmal im GHCi testen:

⁴Tatsächlich ist es manchmal sehr schwer, diese Sicherheit zu garantieren. Z.B. ist bis heute unbekannt, ob die Funktion `f x = if n == 1 then 1 else (if even x then f (x 'div' 2) else f (3*x+1))` für jede natürliche Zahl terminiert, d.h. den Rekursionsanfang findet. Dies ist das sogenannte Collatz-Problem.

5. Grundlagen der Programmierung in Haskell

```
*Main> erste_rekursive_Funktion 5
15
*Main> erste_rekursive_Funktion 10
55
*Main> erste_rekursive_Funktion 11
66
*Main> erste_rekursive_Funktion 12
78
*Main> erste_rekursive_Funktion 100
5050
*Main> erste_rekursive_Funktion 1
1
*Main> erste_rekursive_Funktion (-30)
0
```

Man stellt schnell fest, dass für negative Zahlen stets der Wert 0 als Ergebnis herauskommt. Wir betrachten den Aufruf von `erste_rekursive_Funktion 5`. Das Ergebnis 15 kann man per Hand ausrechnen:

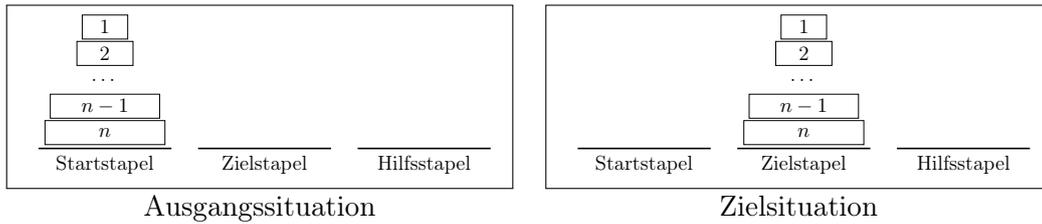
```
erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15
```

Durch längeres Anschauen der Definition erkennt man, dass für jede nicht-negative Zahl x gerade $x + (x - 1) + (x - 2) + \dots + 0$ (oder als mathematische Summenformel $\sum_{i=0}^x i$) berechnet wird. Man kann dies auch beweisen, was wir allerdings an dieser Stelle nicht tun wollen. Es ist erwähnenswert, dass obige Beispielauswertung durch Hinschauen geschehen ist, in der Veranstaltung „Grundlagen der Programmierung 2“ wird dieses Berechnen viel genauer durchgeführt und erklärt.

Der Trick beim Entwurf einer rekursiven Funktion besteht darin, dass man eine Problemstellung *zerlegt*: Der Rekursionsanfang ist der einfache Fall, für den man das Problem direkt lösen kann. Für den Rekursionsschritt löst man nur einen (meist ganz kleinen) Teil des Problems (im obigen Beispiel war das gerade das Hinzuaddieren von x) und überlässt den Rest der Problemlösung der Rekursion.

Das sogenannte „Türme von Hanoi“-Spiel lässt sich sehr einfach durch Rekursion lösen. Die Anfangssituation besteht aus einem Turm von n immer kleiner werdenden Scheiben, der auf dem Startfeld steht. Es gibt zwei weitere Felder: Das Zielfeld und das Hilfsfeld. Ein gültiger Zug besteht darin, die oberste Scheibe eines Turmes auf einen anderen Stapel zu legen, wobei stets nur kleinere Scheiben auf größere Scheiben gelegt werden dürfen.

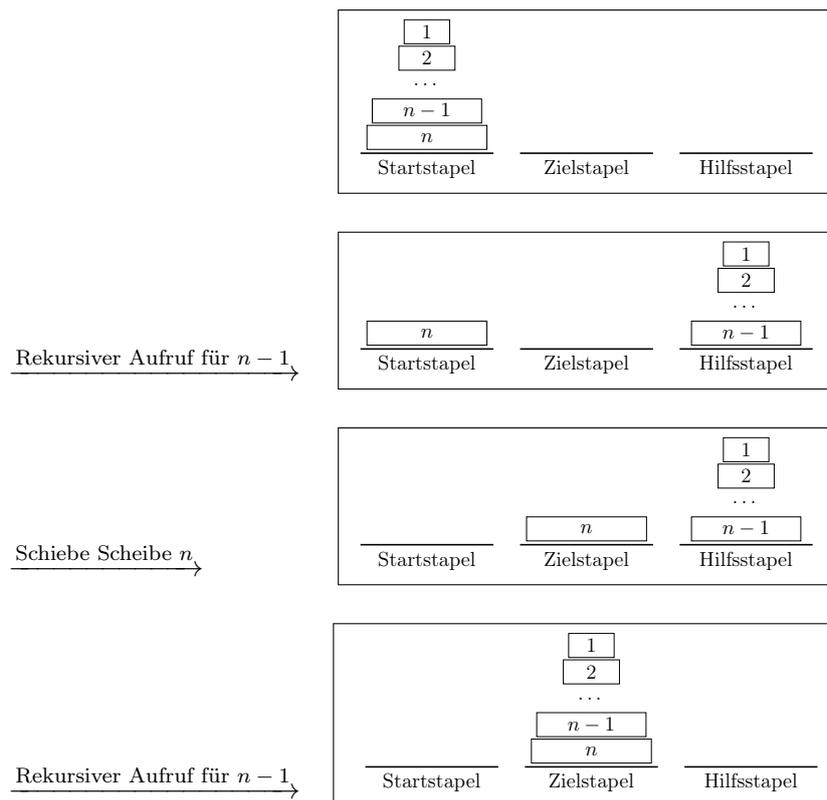
Anschaulich können die Start- und die Zielsituation dargestellt werden durch die folgenden Abbildungen:



Anstatt nun durch probieren alle möglichen Züge usw. zu berechnen, kann das Problem mithilfe von Rekursion sehr einfach gelöst werden:

1. Schiebe mittels Rekursion den Turm der ersten $n - 1$ oberen Scheiben vom Startstapel auf den Hilfsstapel
2. Schiebe die n . Scheibe vom Startstapel auf den Zielstapel
3. Schiebe mittels Rekursion den Turm der $n - 1$ Scheiben vom Hilfsstapel auf den Zielstapel.

Oder mit Bildern ausgedrückt:



Beachte, dass der Rekursionsanfang gerade der Fall $n = 1$ ist, da dann nur eine Scheibe verschoben wird. Die Rekursion terminiert, da bei jedem Aufruf die Problemistanz (der Turm) um 1 verringert wird. Wir betrachten an dieser Stelle nicht die Implementierung in Haskell und verschieben diese auf einen späteren Abschnitt.

Als weiteres Beispiel betrachten wir die Funktion `n_mal_verdoppeln`. Die Funktion soll eine Zahl `x` genau `n` mal verdoppeln. Da wir die Funktion `verdoppeln` bereits definiert haben,

5. Grundlagen der Programmierung in Haskell

genügt es diese Funktion n -mal aufzurufen. Da wir den Wert von n jedoch nicht kennen, verwenden wir Rekursion: Wenn n die Zahl 0 ist, dann verdoppeln wir gar nicht, sondern geben x direkt zurück. Das ist der Rekursionsanfang. Wenn n größer als 0 ist, dann verdoppeln wir x einmal und müssen anschließend das Ergebnis noch $n-1$ -mal verdoppeln, das erledigt der Rekursionsschritt. In Haskell programmiert, ergibt dies die folgende Implementierung:

```
n_mal_verdoppeln :: Integer -> Integer -> Integer
n_mal_verdoppeln x n = if n == 0 then x
                       else n_mal_verdoppeln (verdopple x) (n-1)
```

In Haskell kann man auch mehrere Definitionsgleichungen für eine Funktion angeben, diese werden von oben nach unten abgearbeitet, so dass die erste „passende“ Gleichung gewählt wird: Man darf anstelle einer Variablen nämlich auch ein sogenanntes Pattern (Muster) verwenden. Bei Zahlentypen kann man einfach eine Zahl hinschreiben. Damit können wir uns das `if-then-else` sparen:

```
n_mal_verdoppeln2 :: Integer -> Integer -> Integer
n_mal_verdoppeln2 x 0 = x
n_mal_verdoppeln2 x n = n_mal_verdoppeln2 (verdopple x) (n-1)
```

Der Effekt ist der gleiche: Ist $n = 0$, so passt die erste Zeile und wird daher verwendet, ist n ungleich 0, so passt die erste Zeile nicht, und die zweite Zeile der Definition wird verwendet. Wir testen beide Funktion im Interpreter:

```
*Main> n_mal_verdoppeln2 10 2
40
*Main> n_mal_verdoppeln2 10 3
80
*Main> n_mal_verdoppeln2 10 10
10240
*Main> n_mal_verdoppeln 10 2
40
*Main> n_mal_verdoppeln 2 8
512
```

Gibt man für n jedoch eine negative Zahl ein, so hält der Interpreter nicht an. Den Fall hatten wir übersehen. Wir könnten einfach x zurückgeben, besser ist es jedoch, eine *Fehlermeldung* zu erzeugen. Hierfür gibt es in Haskell die eingebaute Funktion `error`. Sie erwartet als Argument einen String, der beim Auftreten des Fehlers ausgegeben wird. Die Implementierung mit Fehlerabfang ist:

```
n_mal_verdoppeln3 :: Integer -> Integer -> Integer
n_mal_verdoppeln3 x 0 = x
n_mal_verdoppeln3 x n =
  if n < 0 then
    error "in n_mal_verdoppeln3: negatives Verdoppeln ist verboten"
  else
    n_mal_verdoppeln3 (verdopple x) (n-1)
```

Ein Testaufruf im GHCi ist:

```
*Main> n_mal_verdoppeln3 10 (-10)
*** Exception: in n_mal_verdoppeln3: negatives Verdoppeln ist verboten
```

Eine weitere Möglichkeit, die Funktion `n_mal_verdoppeln` zu definieren, ergibt sich durch die Verwendung von sogenannten *Guards* (Wächtern zu deutsch), die ebenfalls helfen ein `if-then-else` zu sparen: Guard

```
n_mal_verdoppeln4 x 0 = x
n_mal_verdoppeln4 x n
  | n < 0      = error "negatives Verdoppeln ist verboten"
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

Hinter dem Guard, der syntaktisch durch `|` repräsentiert wird, steht ein Ausdruck vom Typ `Bool` (beispielsweise `n < 0`). Beim Aufruf der Funktion werden diese Ausdrücke von oben nach unten ausgewertet, liefert einer den Wert `True`, dann wird die entsprechende rechte Seite als Funktionsdefinition verwendet. `otherwise` ist ein speziell vordefinierter Ausdruck

```
otherwise = True
```

der immer zu `True` auswertet. D.h. falls keines der vorhergehenden Prädikate `True` liefert, wird die auf `otherwise` folgende rechte Seite ausgewertet. Die Fallunterscheidung über `n` kann auch komplett über Guards geschehen:

```
n_mal_verdoppeln4 x n
  | n < 0      = error "negatives Verdoppeln ist verboten"
  | n == 0     = x
  | otherwise = n_mal_verdoppeln4 (verdopple x) (n-1)
```

Dabei unterscheidet sich die Funktionalität von `n_mal_verdoppeln4` nicht von `n_mal_verdoppeln3`, lediglich die Schreibweise ist verschieden.

Mit Rekursion kann man auch Alltagsfragen lösen. Wir betrachten dazu die folgende Aufgabe:

5. Grundlagen der Programmierung in Haskell

In einem Wald werden am 1.1. des ersten Jahres 10 Rehe gezählt. Der erfahrene Förster weiß, dass sich im Laufe eines Jahres, die Anzahl an Rehen durch Paarung verdreifacht. In jedem Jahr schießt der Förster 17 Rehe. In jedem 2. Jahr gibt der Förster die Hälfte der verbleibenden Rehe am 31.12. an einen anderen Wald ab. Wieviel Rehe gibt es im Wald am 1.1. des Jahres n ?

Wir müssen zur Lösung eine rekursive Funktion aufstellen, welche die Anzahl an Rehen nach n Jahren berechnet, d.h. die Funktion erhält n als Eingabe und berechnet die Anzahl. Der Rekursionsanfang ist einfach: Im ersten Jahr gibt es 10 Rehe. Für den Rekursionsschritt denken wir uns das Jahr n und müssen die Anzahl an Rehen aufgrund der Anzahl im Jahr $n - 1$ berechnen. Sei k die Anzahl der Rehe am 1.1. im Jahr $n - 1$. Dann gibt es am 1.1. im Jahr n genau $3 * k - 17$ Rehe, wenn $n - 1$ kein zweites Jahr war, und $\frac{3*k-17}{2}$ Rehe, wenn $n - 1$ ein zweites Jahr war. Feststellen, ob $n - 1$ ein zweites Jahr war, können wir, indem wir prüfen, ob $n - 1$ eine gerade Zahl ist.

Mit diesen Überlegungen können wir die Funktion implementieren:

```
anzahlRehe 1 = 10
anzahlRehe n = if even (n-1) then ((3*anzahlRehe (n-1))-17) 'div' 2
                else 3*(anzahlRehe (n-1))-17
```

Testen zeigt, dass die Anzahl an Rehen sehr schnell wächst:

```
*Main> anzahlRehe 1
10
*Main> anzahlRehe 2
13
*Main> anzahlRehe 3
11
*Main> anzahlRehe 4
16
*Main> anzahlRehe 5
15
*Main> anzahlRehe 6
28
*Main> anzahlRehe 7
33
*Main> anzahlRehe 8
82
*Main> anzahlRehe 9
114
*Main> anzahlRehe 10
325
*Main> anzahlRehe 50
3626347914090925
```

let-
Ausdruck

In Haskell gibt es `let`-Ausdrücke, mit diesen kann man lokal (im Rumpf einer Funktion)

den Wert eines Ausdrucks an einen Variablennamen binden (der Wert ist unveränderlich!). Dadurch kann man es sich z.B. ersparen gleiche Ausdrücke immer wieder hinzuschreiben. Die Syntax ist

```
let  Variable1  =  Ausdruck1
     Variable2  =  Ausdruck2
     ...
     VariableN  =  AusdruckN
in   Ausdruck
```

Hierbei müssen Variablennamen mit einem Kleinbuchstaben oder mit einem Unterstrich beginnen, und auf die gleiche Einrückung aller Definitionen ist zu achten.

Z.B. kann man `anzahlRehe` dadurch eleganter formulieren:

```
anzahlRehe2 1 = 10
anzahlRehe2 n = let k = (3*anzahlRehe2 (n-1))-17
                  in  if even (n-1) then k `div` 2
                     else k
```

5.6. Listen

Eine Liste ist eine Folge von Elementen, z.B. ist `[True, False, False, True, True]` eine Liste von Wahrheitswerten und `[1,2,3,4,5,6]` eine Liste von Zahlen. In Haskell sind nur *homogene* Listen erlaubt, d.h. die Listenelemente ein und derselben Liste müssen alle den gleichen Typ besitzen. Z.B. ist die Liste `[True, 'a', False, 2]` in Haskell nicht erlaubt, da in ihr Elemente vom Typ `Bool`, vom Typ `Char`, und Zahlen vorkommen. Der GHCi bemängelt dies dementsprechend sofort:

```
Prelude> [True, 'a', False, 2]

<interactive>:1:6:
  Couldn't match expected type 'Bool' against inferred type 'Char'
  In the expression: 'a'
  In the expression: [True, 'a', False, 2]
  In the definition of 'it': it = [True, 'a', False, ....]
```

Der Typ einer Liste ist von der Form `[a]`, wobei `a` der Typ der *Listenelemente* ist. Einige Beispiele im Interpreter:

5. Grundlagen der Programmierung in Haskell

```
Prelude> :type [True,False]
[True,False] :: [Bool]
Prelude> :type ['A','B']
['A','B'] :: [Char]
Prelude> :type [1,2,3]
[1,2,3] :: (Num t) => [t]
```

Für die Liste `[1,2,3]` ist der Typ der Zahlen noch nicht festgelegt, daher die Typklassenbeschränkung für `Num`. Man kann in Haskell beliebige Dinge (gleichen Typs) in eine Liste stecken, nicht nur Basistypen. Für den Typ bedeutet dies: In `[a]` kann man für `a` einen beliebigen Typ einsetzen. Z.B. kann man eine Liste von Funktionen (alle vom Typ `Integer -> Integer`) erstellen:

```
*Main> :type [verdopple, verdoppleGerade, jenachdem]
[verdopple, verdoppleGerade, jenachdem] :: [Integer -> Integer]
```

Man kann sich diese Liste allerdings nicht anzeigen lassen, da der GHCi nicht weiß, wie er Funktionen anzeigen soll. Man erhält dann eine Fehlermeldung der Form:

```
*Main> [verdopple, verdoppleGerade, jenachdem]

<interactive>:1:0:
  No instance for (Show (Integer -> Integer))
    arising from a use of ‘print’ at <interactive>:1:0-38
  Possible fix:
    add an instance declaration for (Show (Integer -> Integer))
  In a stmt of a ‘do’ expression: print it
```

Eine andere komplizierte Liste, ist eine Liste, die als Elemente wiederum Listen erhält, z.B. `[[True,False], [False,True,True], [True,True]]`. Der Typ dieser Liste ist `[[Bool]]`: Für `a` in `[a]` wurde einfach `[Bool]` eingesetzt.

5.6.1. Listen konstruieren

Wir haben bereits eine Möglichkeit gesehen, Listen zu erstellen: Man trennt die Elemente mit Kommas ab und verpackt die Elemente durch eckige Klammern. Dies ist jedoch nur sogenannter syntaktischer Zucker. In Wahrheit sind Listen *rekursiv* definiert, und man kann sie auch rekursiv erstellen. Der „Rekursionsanfang“ ist die leere Liste, die keine Elemente enthält. Diese wird in Haskell durch `[]` dargestellt und als „Nil“⁵ ausgesprochen. Der „Rekursionsschritt“ besteht darin aus einer Liste mit $n - 1$ Elementen eine Liste mit n Elementen zu konstruieren, indem ein neues Element vorne an die Liste der $n - 1$ Elemente angehängt wird. Hierzu dient der *Listenkonstruktor* `:` (ausgesprochen „Cons“⁶). Wenn `xs` eine Liste mit $n - 1$ Elementen ist

⁵Diese Bezeichnung stammt vom lateinischen Wort „nihil“ für „Nichts“.

⁶Kurzform von „Constructor“, also einem Konstruktor.

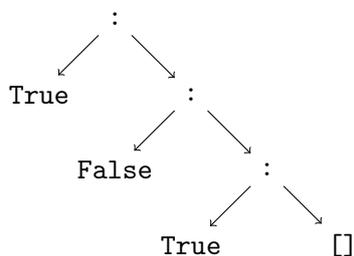
und x ein neues Element ist, dann ist $x:xs$ die Liste mit n Elementen, die entsteht, indem man x vorne an xs anhängt. In $x:xs$ sagt man auch, dass x der *Kopf* (engl. head) und xs der *Schwanz* (engl. tail) der Liste ist. Wir konstruieren die Liste `[True,False,True]` rekursiv:

- `[]` ist die leere Liste
- `True:[]` ist die Liste, die `True` enthält
- `False:(True:[])` ist die Liste, die erst `False` und dann `True` enthält
- Daher lässt sich die gesuchte Liste als `True:(False:(True:[]))` konstruieren.

Tatsächlich kann man diese Liste so im GHCi eingeben:

```
*Main> True:(False:(True:[]))
[True,False,True]
```

Man kann sich eine Liste daher auch als Baum vorstellen, dessen innere Knoten mit `:` markiert sind, wobei das linke Kind das Listenelement ist und das rechte Kind die Restliste ist. Z.B. für `True:(False:(True:[]))`:



Natürlich kann man `:` und `[]` auch in Funktionsdefinitionen verwenden. Daher kann man relativ einfach eine rekursive Funktion implementieren, die eine Zahl n als Eingabe erhält und eine Liste erstellt, welche die Zahlen von n bis 1 in dieser Reihenfolge enthält:

```
nbis1 :: Integer -> [Integer]
nbis1 0 = []
nbis1 n = n:(nbis1 (n-1))
```

Der Rekursionsanfang ist für $n = 0$ definiert: In diesem Fall wird die leere Liste konstruiert. Für den Rekursionsschritt wird die Liste ab $(n - 1)$ rekursiv konstruiert und anschließend die Zahl n mittels `:` vorne an die Liste angehängt. Wir testen `nbis1` im GHCi:

5. Grundlagen der Programmierung in Haskell

```
*Main> nbis1 0
[]
*Main> nbis1 1
[1]
*Main> nbis1 10
[10,9,8,7,6,5,4,3,2,1]
*Main> nbis1 100
[100,99,98,97,96,95,94,93,92,91,90,89,88,87,86,85,84,83,82,81,80,79,
 78,77,76,75,74,73,72,71,70,69,68,67,66,65,64,63,62,61,60,59,58,57,
 56,55,54,53,52,51,50,49,48,47,46,45,44,43,42,41,40,39,38,37,36,35,
 34,33,32, 31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16,15,14,13,
 12,11,10,9,8,7,6,5,4,3,2,1]
```

5.6.2. Listen zerlegen

Oft will man auf die Elemente einer Liste zugreifen. In Haskell sind die Funktion `head` und `tail` dafür bereits vordefiniert:

- `head :: [a] -> a` liefert das erste Element einer nicht-leeren Liste
- `tail :: [a] -> [a]` liefert den Schwanz einer nicht-leeren Liste.

Beachte, dass `head` und `tail` für beliebige Listen verwendet werden können, da ihr Typ *polymorph* ist: Er enthält Typvariablen an der Position für den Elementtyp. Einige Aufrufe von `head` und `tail` im GHCi:

```
*Main> head [1,2]
1
*Main> tail [1,2,3]
[2,3]
*Main> head []
*** Exception: Prelude.head: empty list
```

Die in Haskell vordefinierte Funktion `null :: [a] -> Bool` testet, ob eine Liste leer ist. Mit `head`, `tail` und `null` kann man beispielsweise eine Funktion definieren, die das letzte Element einer Liste extrahiert:

```
letztesElement :: [a] -> a
letztesElement xs = if null xs then
    error "Liste ist leer"
else
    if null (tail xs) then head xs
    else letztesElement (tail xs)
```

Für eine Liste mit mehr als einem Element ruft sich `letztesElement` rekursiv mit dem Schwanz der Liste auf. Enthält die Liste nur ein Element (der Test hierfür ist `null (tail xs)`),

so wird das erste Element dieser Liste als Ergebnis zurück geliefert (dies ist der Rekursionsanfang). Der Fehlerfall, dass die Liste gar keine Elemente enthält, wird direkt am Anfang abgefangen, und eine Fehlermeldung wird generiert. Der Typ der Funktion `letztesElement` ist `[a] -> a`, da sie eine Liste erhält (hierbei ist der konkrete Typ der Elemente egal) und ein Listenelement liefert. Wir testen `letztesElement`:

```
Main> letztesElement [True,False,True]
True
*Main> letztesElement [1,2,3]
3
*Main> letztesElement (nbis1 1000)
1
*Main> letztesElement [[1,2,3], [4,5,6], [7,8,9]]
[7,8,9]
```

Die Programmierung mit `head`, `tail` und `null` ist allerdings nicht wirklich elegant. Haskell bietet hierfür wesentlich schönere Möglichkeiten durch Verwendung sogenannter *Pattern*. Man kann bei einer Funktionsdefinition (ähnlich wie bei den Funktionen auf Zahlen) ein Muster angeben, anhand dessen die Listen zerlegt werden sollen. Hierfür verwendet man die Konstruktoren `[]` und `:` innerhalb der Parameter der Funktion. Wir betrachten als Beispiel die Implementierung von `head`:

Pattern

```
eigenesHead []      = error "empty list"
eigenesHead (x:xs) = x
```

Die Definition von `eigenesHead` ist hierbei so zu interpretieren: Wenn die Eingabe eine Liste ist, die zu dem Muster `(x:xs)` passt (die Liste daher mindestens ein Element hat), dann gebe den zu `x` passenden Teil zurück und wenn die Liste zum Muster `[]` passt (die Liste daher leer ist) dann generiere eine Fehlermeldung. Die Fälle werden bei der Auswertung von oben nach unten geprüft. Analog ist `tail` definiert als.

```
eigenesTail []      = error "empty list"
eigenesTail (x:xs) = xs
```

Eine mögliche Definition für die Funktion `null` ist:

```
eigenesNull []      = True
eigenesNull (x:xs) = False
```

Da die Muster von oben nach unten abgearbeitet werden, kann man alternativ auch definieren

5. Grundlagen der Programmierung in Haskell

```
eigenesNull2 []      = True
eigenesNull2 xs     = False
```

In diesem Fall passt das zweite Muster (es besteht nur aus der Variablen `xs`) für jede Liste. Trotzdem wird für den Fall der leeren Liste `True` zurückgeliefert, da die Muster von oben nach unten geprüft werden. Falsch wird die Definition, wenn man die beiden Fälle in falscher Reihenfolge definiert:

```
falschesNull xs = False
falschesNull [] = True
```

Da das erste Muster immer passt, wird die Funktion `falschesNull` für jede Liste `False` liefern. Der GHCi ist so schlau, dies zu bemerken und liefert beim Laden der Datei eine Warnung:

```
Warning: Pattern match(es) are overlapped
       In the definition of ‘falschesNull’: falschesNull [] = ...
Ok, modules loaded: Main.
*Main> falschesNull [1]
False
*Main> falschesNull []
False
*Main> eigenesNull []
True
*Main>
```

Kehren wir zurück zur Definition von `letztesElement`: Durch Pattern können wir eine elegantere Definition angeben:

```
letztesElement2 []      = error "leere Liste"
letztesElement2 (x:[]) = x
letztesElement2 (x:xs) = letztesElement2 xs
```

Beachte, dass das Muster `(x:[])` in der zweiten Zeile ausschließlich für einelementige Listen passt.

5.6.3. Einige vordefinierte Listenfunktionen

In diesem Abschnitt führen wir einige Listenfunktionen auf, die in Haskell bereits vordefiniert sind, und verwendet werden können:

- `length :: [a] -> Int` berechnet die Länge einer Liste (d.h. die Anzahl ihrer Elemente).

- `take :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert die Liste der ersten k Elemente von xs
- `drop :: Int -> [a] -> [a]` erwartet eine Zahl k und eine Liste xs und liefert xs ohne die der ersten k Elemente.
- `(++) :: [a] -> [a] -> [a]` erwartet zwei Listen und hängt diese aneinander zu einer Liste, kann infix in der Form `xs ++ ys` verwendet werden. Man nennt diese Funktion auch „append“.
- `concat :: [[a]] -> [a]` erwartet eine Liste von Listen und hängt die inneren Listen alle zusammen. Z.B. gilt `concat [xs,ys]` ist gleich zu `xs ++ ys`.
- `reverse :: [a] -> [a]` dreht die Reihenfolge der Elemente einer Liste um.

5.6.4. Nochmal Strings

Wir haben Zeichenketten (Strings) vorher schon eingeführt. Tatsächlich ist die Syntax "Hallo Welt" nur syntaktischer Zucker: Zeichenketten werden in Haskell als Listen von Zeichen intern dargestellt, d.h. unser Beispielstring ist äquivalent zur Liste `['H','a','l','l','o',' ',' ','W','e','l','t']`, die man auch mit `[]` und `:` darstellen kann als `'H':('a':('l':('l':('o':(' ':(' '(' (W':('e':(l':('t':[]))))))))))`. Alle Funktionen die auf Listen arbeiten, kann man daher auch für Strings verwenden, z.B.

```
*Main> head "Hallo Welt"
'H'
*Main> tail "Hallo Welt"
"allo Welt"
*Main> null "Hallo Welt"
False
*Main> null ""
True
*Main> letztesElement "Hallo Welt"
't'
```

Es gibt allerdings spezielle Funktionen, die nur für Strings funktionieren, da sie die einzelnen Zeichen der Strings „anfassen“. Z.B.

- `words :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Worten*
- `unwords :: [String] -> String`: Macht aus einer Liste von Worten einen einzelnen String.
- `lines :: String -> [String]`: Zerlegt eine Zeichenkette in eine *Liste von Zeilen*

Z.B. kann man unter Verwendung von `words` und `length` die Anzahl der Worte eines Texts zählen:

```
anzahlWorte :: String -> Int
anzahlWorte text = length (words text)
```

5.7. Paare und Tupel

Paare stellen in Haskell – neben Listen – eine weitere Möglichkeit dar, um Daten zu strukturieren. Ein Paar wird aus zwei Ausdrücken e_1, e_2 konstruiert, indem sie geschrieben werden als (e_1, e_2) . Hat e_1 den Typ T_1 und e_2 den Typ T_2 , dann ist der Typ des Paares (T_1, T_2) . Der Unterschied zu Listen besteht darin, dass Paare eine feste Stelligkeit haben (nämlich 2) und dass die Typen der beiden Ausdrücke nicht gleich sein müssen. Einige Beispiele für Paare in Haskell sind:

```
Main> :type ("Hallo",True)
("Hallo",True) :: ([Char], Bool)
Main> :type ([1,2,3], 'A')
([1,2,3],False) :: (Num t) => ([t], Char)
*Main> :type (letztesElement, "Hallo" ++ "Welt")
(letztesElement, "Hallo" ++ "Welt") :: ([a] -> a, [Char])
```

Der Zugriff auf Elemente eines Paares kann über zwei vordefinierte Funktionen erfolgen:

- `fst :: (a,b) -> a` nimmt ein Paar und liefert das linke Element.
- `snd :: (a,b) -> b` liefert das rechte Element eines Paares.

Wie man am Typ der beiden Funktionen sieht (der Typvariablen `a, b` enthält), sind sie polymorph, d.h. sie können für Paare beliebigen Typs verwendet werden. Beispiele für die Anwendung sind:

```
*Main> fst (1, 'A')
1
*Main> snd (1, 'A')
'A'
*Main>
```

Bei der Definition von `fst` und `snd` kann man wieder Pattern (Muster) verwenden, um die entsprechenden Elemente auszuwählen:

```
eigenesFst (x,y) = x
eigenesSnd (x,y) = y
```

Hier wird das Pattern (x, y) zur Selektion des gewünschten Elements verwendet, wobei $(,)$ der Paar-Konstruktor ist, x ist eine Variable, die das linke Element eines Paares bezeichnet und y eine Variable, die das rechte Element bezeichnet.

Tupel stellen eine Verallgemeinerung von Paaren dar: Ein n -Tupel kann n Elemente verschiedenen Typs aufnehmen. Auch hier ist die Stelligkeit fest: Ein n -Tupel hat Stelligkeit n . Das Erzeugen von n -Tupeln verläuft analog zum Erzeugen von Paaren, einige Beispiele sind:

```
*Main> :set +t
*Main> ('A',True,'B')
('A',True,'B')
it :: (Char, Bool, Char)
*Main> ([1,2,3],(True,'A',False,'B'),'B')
([1,2,3],(True,'A',False,'B'),'B')
it :: ([Integer], (Bool, Char, Bool, Char), Char)
```

Der GHCi kann Tupel mit bis zu 15 Elementen drucken, intern können noch größere Tupel verwendet werden. Zur Auswahl bestimmter Elemente von Tupeln sind keine Funktionen vordefiniert, man kann sie aber leicht mit Hilfe von Pattern selbst implementieren:

```
-----
erstes_aus_vier_tupel (w,x,y,z) = w
-- usw.
viertes_aus_vier_tupel (w,x,y,z) = z
-----
```

5.7.1. Die Türme von Hanoi in Haskell

Wir betrachten nun die Implementierung des „Türme von Hanoi“-Spiels in Haskell. Wir implementieren hierfür eine Funktion, die als Ergebnis die *Liste der Züge* berechnet. Der Einfachheit halber nehmen wir an, dass die drei Stapel (Startstapel, Zielstapel, Hilfsstapel) (am Anfang) den Zahlen 1, 2 und 3 versehen sind. Ein Zug ist dann ein Paar (a, b) von zwei (unterschiedlichen) Zahlen a und b aus der Menge $\{1,2,3\}$ und bedeute: ziehe die oberste Scheibe vom Stapel a auf den Stapel b .

Wir implementieren die Funktion `hanoi` mit 4 Parametern: Der erste Parameter ist die Höhe des Turms n , die nächsten drei Parameter geben die Nummern für die drei Stapel an.

Die Hanoi-Funktion ist nun wie folgt implementiert:

```
-----
-- Basisfall: 1 Scheibe verschieben
hanoi 1 start ziel hilf = [(start,ziel)]

-- Allgemeiner Fall:
hanoi n start ziel hilf =
  -- Schiebe den Turm der Hoehe n-1 von start zu hilf:
  (hanoi (n-1) start hilf ziel)
  ++
  -- Schiebe n. Scheibe von start auf ziel:
  [(start,ziel)]
  ++
  -- Schiebe Turm der Hoehe n-1 von hilf auf ziel:
  ++ (hanoi (n-1) hilf ziel start)
-----
```

5. Grundlagen der Programmierung in Haskell

Der Basisfall ist der Fall $n = 1$: In diesem Fall ist ein Zug nötig, der die Scheibe vom Startstapel auf den Zielstapel bewegt. Im allgemeinen Fall werden zunächst $n - 1$ Scheiben vom Startstapel auf den Hilfsstapel durch den rekursiven Aufruf verschoben (beachte, dass dabei der Hilfsstapel zum Zielstapel (und umgekehrt) wird). Anschließend wird die letzte Scheibe vom Startstapel zum Zielstapel verschoben, und schließlich wird der Turm der Höhe $n - 1$ vom Hilfsstapel auf den Zielstapel verschoben.

Gestartet wird die Funktion mit 1, 2 und 3 als Nummern für die 3 Stapel:

```
start_hanoi n = hanoi n 1 2 3
```

Z.B. kann man nun mit einem Stapel der Höhe 4 testen:

```
*Main> start_hanoi 4  
[(1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,3), (1,2),  
 (3,2), (3,1), (2,1), (3,2), (1,3), (1,2), (3,2)]
```

Wer möchte kann nachprüfen, dass diese Folge von Zügen tatsächlich zum Ziel führt.

A. Anmeldung an RBI-Rechner

A.1. Voraussetzung

Zum Login bei den RBI-Rechnern benötigen Sie einen **RBI-Account**. Wenn Sie sich für den Vorkurs angemeldet haben, müsste bereits ein RBI-Account für Sie existieren. Wenden Sie sich hierfür an Ihre Tutoren.

Wenn nicht, kommen Sie bitte direkt zu den Räumlichkeiten der RBI im Kellergeschoss der Robert-Mayer-Str. 11-15. Dort erhalten Sie dann ein Antragsformular, welches Sie ausgefüllt und unterschrieben in der RBI abgeben.

Für die aktuellsten Informationen und weitere Hinweise zur Nutzung des RBI-Accounts, können Sie unsere Website nutzen: <https://www.rbi.informatik.uni-frankfurt.de/RBI/de>

A.2. Vom PC in den Poolräumen

Die RBI besitzt zwei verschiedene Räumlichkeiten mit Poolrechnern: **Fischerräume** und **Raum 026**.

Die **Fischerräume** befinden sich im Institut für Informatik in der Robert-Mayer-Straße 11-15 im Erdgeschoss (hinter dem Magnushörsaal) und sind nur durch einen separaten Eingang zur Emil-Sulzbach-Straße zugänglich.

A. Anmeldung an RBI-Rechner

Tabelle A.1.: Namen der PC-Rechner in den Fischerräumen

R21 kl.E-Raum	R 22 E-Raum	R 23 K-Raum	R24 kl.K-Raum	R25 P-Raum
eurylos eridyke epikles	erebos eumolos eos euos elektra eleusius eryx	kretheus kepheus kirke kratos kreon kalliste kroton klytios kadmos kalypso	kamilla kerkyon klaros	phylla phobos pandion pythia proteus priamos phoenix phoebe paris pan pallas polyphem polydora pegasos penelope

Das Computerlab im **Raum 026** befindet sich im Kellergeschoss des Instituts für Informatik.

Tabelle A.2.: Namen der PC-Rechner im Computerlab 026

adastos	admeta	aither
alkmene	amata	apate
ajax	agylla	arges
anaxo	ares	atropos
achilles	adamas	aletheia
axylos	arabia	acheloos
aktor	adonis	anemoi

Einloggen: Zunächst im Feld 'Benutzername' den RBI-Loginnamen eingeben, dann mit der [Enter]-Taste in das Feld 'Passwort' wechseln, das Passwort eingeben und mit [Enter] bestätigen.

Ausloggen: Im K-Menü Log Out (bzw. Abmelden) anklicken, ausloggen mit End Current Session (Aktuelle Sitzung beenden) bestätigen.

A.3. Vom eigenen PC aus

Um sich von einem eigenen PC in einen Rechner der RBI einzuloggen, öffnen Sie bitte ein Terminal bzw. Eingabeaufforderung (in Windows) und geben dort den nachfolgenden Befehl ein. Für [loginname RBI] geben Sie bitte ihren Nutzernamen ihres RBI accounts an und für

A.4. Vom Vorkurs-Laptop aus

[rechnername] einen der Namen unserer PC-Rechner (siehe Tabellen oben):

```
ssh [loginname RBI]@[rechnername].rbi.informatik.uni-frankfurt.de
```

Bsp für das Login des User *mustermann* am Rechner *eos*:

```
ssh mustermann@eos.rbi.informatik.uni-frankfurt.de
```

Im Anschluss daran werden Sie gebeten das Passwort Ihres RBI-Accounts einzugeben. Wenn alles richtig eingegeben wurde, sind Sie dann an dem entsprechenden PC der RBI eingeloggt und können dort Programme ausführen, Dateien speichern etc.

Zum Ausloggen können sie einfach `logout` im Terminal eingeben.

A.4. Vom Vorkurs-Laptop aus

Um sich von den Vorkurs-Laptops in einen Rechner der RBI einzuloggen, gehen Sie bitte genauso vor wie zum Login vom eigenen PC aus. Bitte speichern Sie ihre Daten und Programme ausschließlich auf dem remote PC, mit dem Sie sich verbunden haben, da auf den Vorkurs Laptops nichts dauerhaft gespeichert werden kann. Wenn der Laptop ausgeschaltet und wieder eingeschaltet wird, sind alle dort gespeicherten Dateien gelöscht.