Skript

Vorkurs Informatik

Sommersemester 2020

Prof. Dr. David Sabel
Conrad Rau
Ronja Düffel
Stand: 26. März 2020

Inhaltsverzeichnis

Teil I.

Digitaltechnik

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*

positive und 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.

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:

$$Z_2 = z_{n-1}z_{n-2}z_{n-3}\dots z_1 z_0$$

Die Position zählt also von rechts nach links hoch. Da das erste (rechteste!) 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 mathmatischen 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 $2\cdot 10^2 + 4\cdot 10^1 + 3\cdot 10^0 = 2\cdot 100 + 4\cdot 10 + 3\cdot 3 = 243$. Die Dezimalzahl 324_{10} besteht zwar aus den gleichen Zeichen, repräsentiert aber einen anderen Wert, nämlich $3\cdot 10^2 + 2\cdot 10^1 + 4\cdot 10^0 = 300 + 20 + 4 = 324$.

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.:

$324 : 2 =$		162																	Rest 0
	$162 : 2 =$		81																Rest 0
		$81 : 2 =$		40															Rest 1
			$40 : 2 =$		20														Rest 0
				$20 : 2 =$		10													Rest 0
					$10 : 2 =$		5												Rest 0
						$5 : 2 =$		2											Rest 1
							$2 : 2 =$		1										Rest 0
								$1 : 2 =$		0									Rest 1

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 ($a_6a_5a_4a_3a_2a_1a_0$) 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. Die ASCII-Tabelle hat $2^7 = 128$ Zeichen. Im Folgenden interessieren uns aber erstmal nur Zahlen.

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 $^-$, für die entsprechende Schaltzeichen (Logik-Gatter) eingeführt werden (siehe Tab. 2.1).

Operatoren

Logik-Gatter

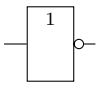
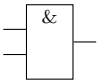
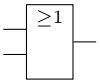
Operator	Schaltzeichen	Benennung
$-$		NICHT - Gatter
$\wedge$		UND - Gatter
$\vee$		ODER - Gatter

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 $\wedge$, $\vee$, $^-$ 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 $a, b, c, \ldots$.

Schaltvariable

Beispiel 2.1.
Funktion der allgemeinen Algebra:

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

Funktion der Schaltalgebra:

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

Definition 2.2 (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 $x = (a, b, c, \ldots)$, für mehrere Ausgangsvariablen $y = (y_1, y_2, y_3, \ldots)$.

Schaltfunktion

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 $f(a) = \overline{a}$ 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 $^-$ gibt es drei gleichwertige Darstellungen.

- Wahrheitstabelle,
- Schaltzeichen und
- Funktion

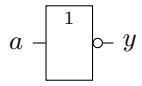
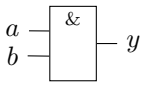
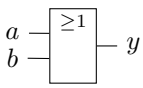
Name	Wahrheitstabelle	Schaltzeichen	Funktion
Nicht, Not, Komplement, Negation	$a \mid \overline{a}$ 0 | 1 1 | 0		$y = f(a) = \overline{a}$
Und, And, Konjunktion	$a \quad b \mid a \wedge b$ 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1		$y = f(a, b) = a \wedge b$
Oder, Or, Disjunktion	$a \quad b \mid a \vee b$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1		$y = f(a, b) = a \vee b$

Tabelle 2.2.: Grundverknüfungen 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 $g(a,b) = \overline{\overline{a} \vee \overline{b}}$ sind gleich. Das lässt sich leicht mit einer Wahrheitstabelle überprüfen.

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

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 genutzt, um Schaltfunktionen darzustellen, hier führen wir sie ausführlicher ein. Die Wahrheitstabelle ist eine Tabelle, die alle möglichen Kombinationen 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 $^-$ dargestellt werden.

Beispiel 2.9.

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

a	b	c	$\overline{c}$	$a \vee b$	$a \wedge \overline{c}$	$\overline{a \wedge \overline{c}}$	$(a \vee b) \wedge (\overline{a \wedge \overline{c}})$	$b \wedge c$	$g(a, b, c)$
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 $^-$, 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 $a \wedge \overline{b} \wedge c$ schreibt man $a\overline{b}c$.

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

$x = 3 + (5 \cdot 2) = 3 + 10 = 13$ und
$y = (3 + 5) \cdot 2 = 8 \cdot 2 = 16.$

2.2.3. Darstellung mit Schaltsymbolen

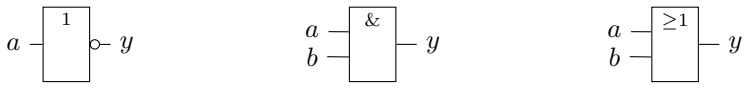
Alle Verknüpfungen können auch durch Schaltsymbole dargestellt werden (vergl. Tab. 2.1). Also lassen sich auch alle Schaltfunktion mit Hilfe von Schaltsymbolen als Schaltung darstellen. Die technischen Bauteile, die die jeweiligen Verknüpfungen darstellen, nennt man *Gatter*. Sie können technisch auf 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 60617-12 geregelt. Logische Gatter werden durch ein hochkant stehendes Rechteck dargestellt. Die Funktion des Gatters wird durch ein Symbol innerhalb des Rechtecks angeben. & steht für UND (Konjunktion, $\wedge$), ≥ 1 für ODER (Disjunktion,$\vee$). Das Zeichen für ODER behält man besser, wenn man sich klar macht, dass eine Disjunktion immer dann zu 1 auswertet, wenn mindestens eine der Eingangsvariablen den Wert 1 hat.

Das Gatter für die Negation ($^-$) hat als Symbol im Recheck eine 1 und am Ausgang einen kleinen Kreis.

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



NICHT-Gatter UND-Gatter ODER-Gatter

Abbildung 2.1.: Schaltsymbole der Grundverknüpfungen $^{-}, \wedge$ 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.[1]

Durch Kombination der Symbole lassen sich nun Schaltfunktionen darstellen

Beispiel 2.11.
Die Schaltfunktion $f(a, b) = (\overline{a} \vee b)$ kann durch Kombination eines NICHT-Gatters und eines ODER-Gatters dargestellt werden (Abb.: 2.2).

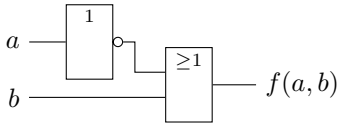


Abbildung 2.2.: Schaltung für $f(a, b) = (\overline{a} \vee b)$

Beispiel 2.12.
Die Schaltfunktion $g(a, b, c) = (a \wedge \overline{c}) \vee (\overline{b} \wedge c)$ kann durch Kombination zweier NICHT-Gatter, zweier UND-Gatter und eines ODER-Gatters dargestellt werden (siehe Abb.: 2.3).

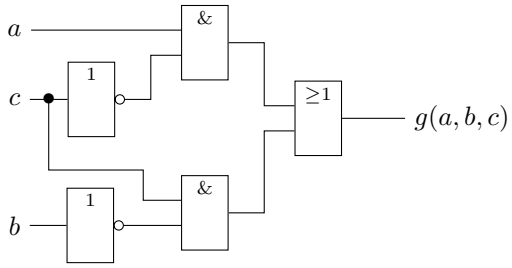


Abbildung 2.3.: Schaltung für $g(a, b, c) = ((a \wedge \overline{c}) \vee (\overline{b} \wedge c)$

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).

[1] Einen guten Überblick liefert die Wikipediaseite: https://de.wikipedia.org/wiki/Logikgatter

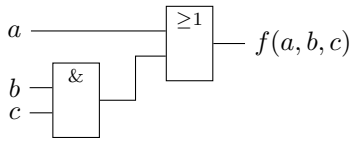


Abbildung 2.4.: Schaltung für $g(a, b, c) = a \vee (b \wedge c)$

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

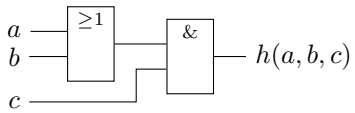


Abbildung 2.5.: Schaltung für $h(a, b, c) = (a \vee b) \wedge c$

2.3. Gesetze der Schaltalgebra

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

Tipp: Wenn man sich $\wedge$ als Multiplikation und $\vee$ als Addition vorstellt, kommen einem viele der Gesetze gar nicht neu vor.

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:

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

5. Neutrale Elemente:

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

6. Inverse Elemente:

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

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:

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

10. De Morgansche Regeln:

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

Hier beginnt das Skript für den 2. Tag

2.4. Weitere Verknüpfungen

Außer der bereits genannten Grundverknüpfungen AND, OR und NOT, kommen in der Digitaltechnik auch die Verknüfungen 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 genau eine der Eingangsvariablen 1 ist.
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.

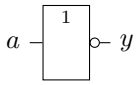
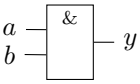
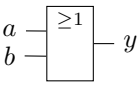
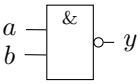
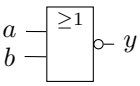
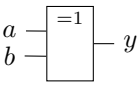
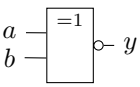
Name	Wahrheitstabelle	Schaltzeichen	Funktion
Nicht, Not, Komplement, Negation	$a \mid \overline{a}$ 0 | 1 1 | 0		$y = f(a) = \overline{a}$
Und, And, Konjunktion	$a \quad b \mid a \wedge b$ 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1		$y = f(a,b) = a \wedge b$
Oder, Or, Disjunktion	$a \quad b \mid a \vee b$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1		$y = f(a,b) = a \vee b$
Nand	$a \quad b \mid \overline{a \wedge b}$ 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0		$y = f(a,b) = \overline{a \wedge b}$
Nor	$a \quad b \mid \overline{a \vee b}$ 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0		$y = f(a,b) = \overline{a \vee b}$
Xor	$a \quad b \mid a \oplus b$ 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0		$y = f(a,b) = a \oplus b$
Xnor	$a \quad b \mid \overline{a \oplus b}$ 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 1		$y = f(a,b) = \overline{a \oplus b}$

Tabelle 2.5.: Wichtige Verknüpfungen in der Digitaltechnik

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 $^-$ bilden ein vollständiges Operatorensystem. Ferner bilden sowohl die Verknüpfung NAND, als auch die Verknüpfung NOR ein vollständiges Operatorensystem.

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

Beweis 2.14.
Negation

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

Konjunktion

a	b	$a \wedge b$	$x = a \overline{\wedge} b$	$x \overline{\wedge} x$	$y = a \overline{\vee} a$	$z = b \overline{\vee} b$	$y \overline{\vee} z$
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	$a \vee b$	$x = a \overline{\wedge} a$	$y = b \overline{\wedge} b$	$x \overline{\wedge} y$	$z = (a \overline{\vee} b)$	$z \overline{\vee} z$
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

2.5. Normalformen

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

Definition 2.15 (Disjunktive und Konjunktive Normalform)**.**

DNF Die *disjunktive Normalform* (DNF) einer Schaltfunktion ist eine VerODERung von UND-Termen, bzw. eine Disjunktion von Konjunktionstermen (vergl. Bsp. 2.16, Gleichung 2.1). Enthalten die Konjunktionsterme *jede* Eingangsvariable in negierter oder nicht negierter Form, spricht man von
KDNF einer *kanonischen disjunktiven Normalform* (KDNF)(siehe Bsp. 2.16,Gl. 2.2).

KNF Die *konjunktivie Normalform* (KNF) einer Schaltfunktion ist eine VerUNDung von ODER-Termen, bzw. eine Konjunktion von Disjunktionstermen (Bsp. 2.16, Gl.2.3). Kommen in allen Disjunktionstermen alle Eingangsvariablen in negierter oder nicht negierter Form vor, so handelt es sich
KKNF um eine *kanonische konjunktive Normalform* (KKNF) (Bsp. 2.16, Gl.2.4).

Beispiel 2.16.
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:

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

Die KDNF ist:

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

Eine KNF der Funktion wäre:

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

Die KKNF lautet:

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

2.5.1. Min- und Maxterme

Definition 2.17 (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 2.6).

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
0	0	0	0	1	0
0	1	0	0	0	1

Tabelle 2.6.: 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 2.7).

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

Tabelle 2.7.: 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. 0 0 $\rightarrow$ Index 0 und somit $m_0 = \overline{a} \wedge \overline{b}$ 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 2.18.
Betrachten wir die Funktion $f_1(a,b) = m_0 \vee m_2$.
Minterm m_0 ist in diesem Fall $(\overline{a} \wedge \overline{b})$, Minterm m_2 ist $(a \wedge \overline{b})$ also gilt:

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

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 $(\overline{b} \wedge \overline{a})$, allerding nur, weil beide Eingangsvariablen negiert sind und in der bool'schen Algebra das Kommutativgesetz gilt. Der Minterm m_2 ist hier $(b \wedge \overline{a})$, 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:

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

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:

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

Beispiel 2.19.
Die Funktion aus Beispiel 2.8 kann folgendermaßen in eine DNF umgeformt werden.

$$\begin{aligned}
g(a,b,c) &= (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c) \\
&\overset{9}{=} (a \vee b) \wedge (\overline{a} \vee \overline{\overline{c}}) \vee (b \wedge c) \\
&= (a \vee b) \wedge (\overline{a} \vee c) \vee (b \wedge c) \\
&\overset{4}{=} ((a \vee b) \wedge \overline{a}) \vee (((a \vee b) \wedge c) \vee (b \wedge c) \\
&\overset{4}{=} ((\overline{a} \wedge a) \vee (\overline{a} \wedge b)) \vee ((a \wedge c) \vee (b \wedge c)) \vee (b \wedge c)
\end{aligned}$$

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

$$\begin{aligned}
&((\overline{a} \wedge a) \vee (\overline{a} \wedge b)) \vee ((a \wedge c) \vee (b \wedge c)) \vee (b \wedge c) \\
&\overset{6}{=} 0 \vee (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c) \vee (b \wedge c) \\
&\overset{7}{=} (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c) \vee (b \wedge c) \\
&\overset{8}{=} (\overline{a} \wedge b) \vee (a \wedge c) \vee (b \wedge c)
\end{aligned}$$

Diese DNF sieht zwar anders aus als die in Gleichung 2.1 angegebene, aber beide stellen die gleiche Schaltfunktion dar. Das wird anhand einer Wahrheitstabelle deutlich.

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

$$
\begin{aligned}
g(a,b,c) &= (a \vee b) \wedge (\overline{a \wedge \overline{c}}) \vee (b \wedge c) \\
&\overset{9}{=} (a \vee b) \wedge (\overline{a} \vee \overline{\overline{c}}) \vee (b \wedge c) \\
&= (a \vee b) \wedge (\overline{a} \vee c) \vee (b \wedge c) \\
&\overset{4}{=} (a \vee b) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee c \vee c) \\
&= (a \vee b) \wedge (\overline{a} \vee b \vee c) \wedge (\overline{a} \vee c)
\end{aligned}
$$

Dies ist bereits eine KNF. Sie sieht zwar anders aus, als die in Beispiel 2.16, Gleichung 2.3 angegebene, stellt aber die gleiche Schaltfunktion dar, wie mit einer Wahrheitstabelle schnell zu überprüfen ist.

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

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. 2.15). Gleiches gilt für die KKNF.

Beispiel 2.20.
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.

Natürlich kann man für eine gegebene Funktionsgleichung auch zunächst die Wahrheittabelle aufstellen und DNF, bzw. KNF dann daraus ablesen, wenn die Schaltfunktion nicht zu viele Eingangsvariablen hat, denn Wahrheittabellen werden sehr schnell sehr groß.

2.6. KV-Diagramme

Das **Karnaugh-Veitch-Digramm** 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 Variablen 1 ist. So ist dem Feld mit Index 0 im KV-Diagramm für eine Eingangsvariable (a) (Abb.: 2.7) die Belegung 0, bzw. $\overline{a}$ zugeordnet und dem Feld mit Index 1 die Belegung 1 bwz. a. Im KV-Diagramm für zwei Eingangsvariablen (a,b) ist Feld 0 die Belegung 0 0, bzw. $\overline{a}, \overline{b}$, Feld 1 die Belegung 0 1, bzw. $\overline{a}\, b$, Feld 2 die Belegung 1 0, bzw. $a\, \overline{b}$ und Feld 3 die Belegung 1 1, bzw. $a\, b$. Die Indizes der Felder im KV-Diagramm sind also identisch mit den Indizes der Min- und Maxterme, die ihnen zugeordnet sind (vergl. Tab. 2.8).

Bemerkung: Im Prinzip braucht man die Indizes nicht, wenn man verstanden hat, wie das mit den Balken an der Seite des KV-Diagramms geht. Und in den Übungen werden Sie keine Indizes in den KV-Diagrammen finden. Wir haben ein Video dazu gemacht, wie man KV-Diagramme ohne Indizes nutzt, da ich es sehr schwierig fand, das im Text zu erklären.

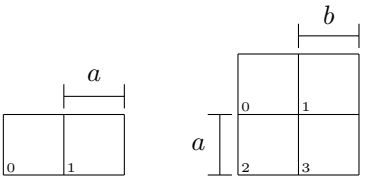


Abbildung 2.6.: KV-Diagramme für eine (a) und für zwei (a,b) Eingangsvariablen

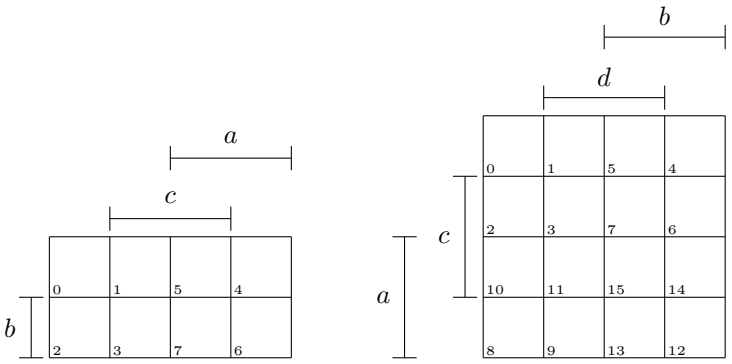


Abbildung 2.7.: KV-Diagramme für drei (a, b, c) und für vier (a, b, c, d) Eingangsvariablen

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

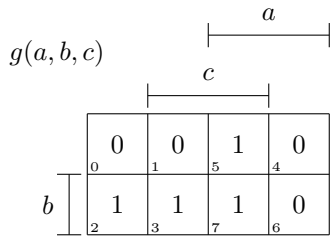
Tabelle 2.8.: Feldindizes und zugeordnete Belegung der Eingangsvariablen sowie Min- und Maxterme bei KV-Diagrammen für vier Eingangsvariablen.

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 2.21.
Betrachten wir wieder die Funktion aus Beispiel 2.8 in KDNF (vergl. Bsp. 2.20),

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

Die 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.

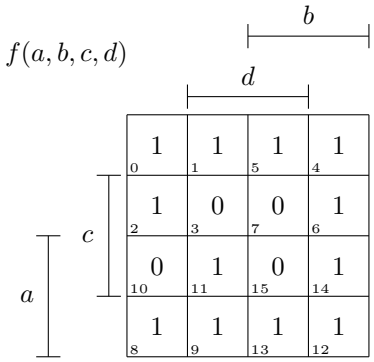


Liegt eine gegebene Schaltfunktion in KKNF vor, so muss in alle Felder, deren zugeordnete Maxterme in der KKNF auftauchen, eine 0 eingetragen werden. In alle anderen Felder muss eine 1 eingetragen werden.

Beispiel 2.22.
Betrachten wir die vierstellige Funktion $f(a, b, c, d)$ in KKNF:

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

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.



Hier beginnt das Skript für den 3. Tag

2.7. Minimierung von Schaltfunktionen

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

Beispiel 2.23.
Die Funktionsgleichung

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

Kann vereinfacht werden zu

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

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 2.24.
Betrachten wir die Funktion $f(a, b, c)$ aus Beispiel 2.23. Abbildung 2.8 zeigt die KDNF der Funktion als Schaltung realisiert.

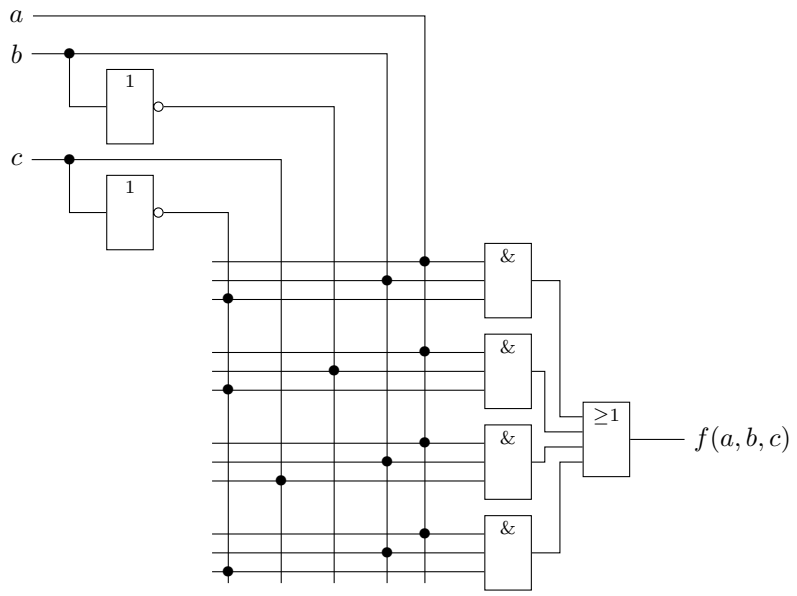


Abbildung 2.8.: Schaltung für $f(a, b, c) = (a \wedge b \wedge \overline{c}) \vee (a \wedge \overline{b} \wedge \overline{c}) \vee (a \wedge b \wedge c) \vee (a \wedge b \wedge \overline{c})$.

Abbildung 2.9 zeigt die Schaltung der minimierten Funktion.

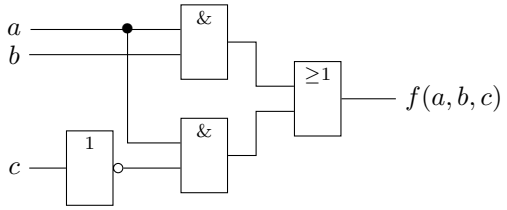


Abbildung 2.9.: Schaltung für $f(a, b, c) = (a \wedge \overline{c}) \vee (a \wedge b)$

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

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

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

Das heißt, unterscheiden sich Und- bzw. Oder-Terme nur in der Negation einer einzigen Variablen, so können sie zu einem Term verschnmolzen werden, bei dem diese Variable wegelassen werden kann.

2.7.1. Minimierung mit den Gesetzen der Schaltalgebra

Beispiel 2.25.
Bei der KDNF der Funktion

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

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. 2.6). Daher können Felder mit benachbarten 1-en bzw. 0-en zusammengefasst werden.

Beispiel 2.26.
Betrachten wir die Funktion aus Beispiel 2.25. In ein KV-Diagramm eingetragen, sieht die Funktion wie folgt aus:

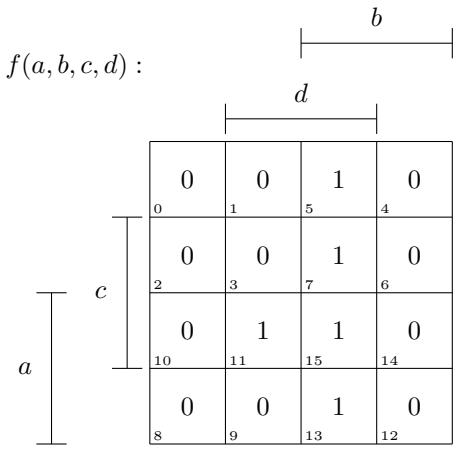
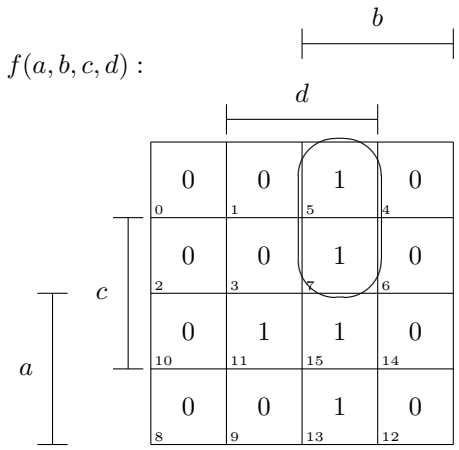


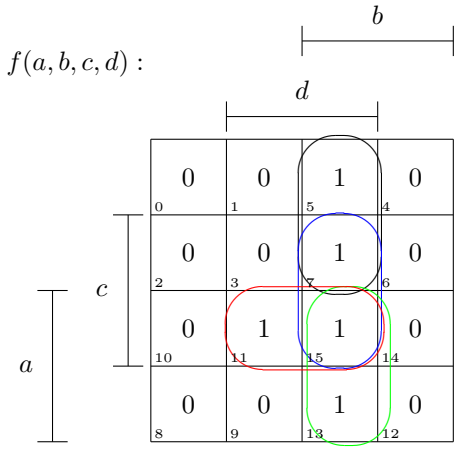
Abbildung 2.10.: KV-Diagramm für Funktion $f(a,b,c,d) = (a\bar{b}cd) \vee (abcd) \vee (ab\bar{c}d) \vee (\bar{a}bcd) \vee (\bar{a}b\bar{c}d)$

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



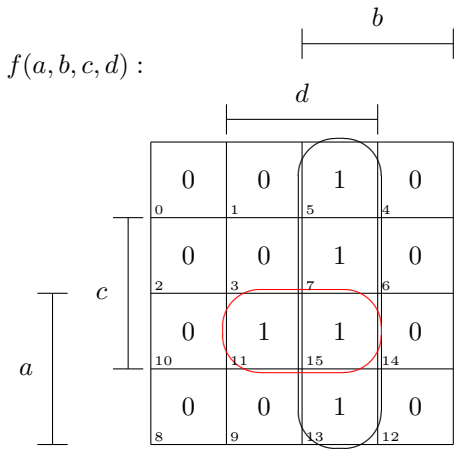
Dem zusammengefassten Block ist der Term $(\overline{a} \wedge b \wedge d)$ oder $(\overline{a}bd)$ zugeordnet. Wenn wir uns zurück erinnern an Beispiel 2.25, entsprechen die Felder mit Indizes 5 und 7 den Mintermen $(\overline{a}b\overline{c}d)$ und $(\overline{a}bcd)$, welche zu $(\overline{a}bd)$ 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.



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 enspricht 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 2.27 (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

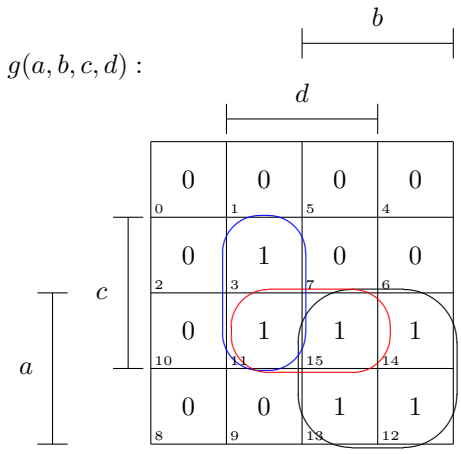
Ein *Primimplikant* von f ist ein Implikant m von f, der nicht weiter verkürzt werden kann. D.h. es ist ein Implikant minimaler Länger. Im KV-Diagramm bedeutet das, dass keinen größern Block von 1-en gibt, der m enthält.

In Beispiel 2.26 sind $(\overline{a}bd)$, (cbd),(abd), (acd) und (bd) Implikanten, (acd) und (bd) sind Primiplikanten. $(\overline{a}bd)$, (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 2.28 (Kernprimimplikant)**.**
Ein *Kernprimimplikant* ist ein Primimplikant, der eine 1 enthält, die von keinem andern Primimplikanten abgedeckt wird.

Beispiel 2.29.
Im KV-Diagramm der Funktion



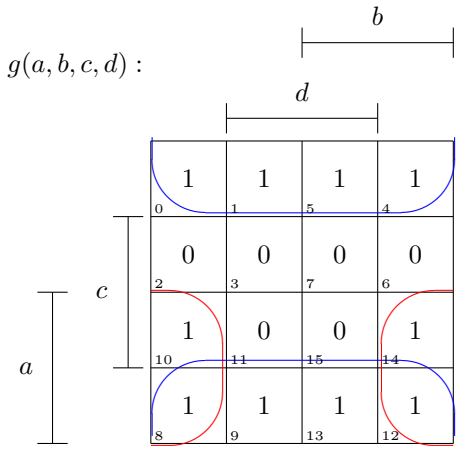
Sind (ab), (acd) und $(\overline{b}cd)$ Primimplikanten. $(\overline{b}cd)$ ist Kernprimimplikant, da er als einziger Primimplikant die 1 in dem Feld mit Index 3 abdeckt. (bd) ist Kernimplikant, da er als einziger Primimplikant die 1-en in den Feldern mit Indizes 12, 13 und 14 abdeckt. (acd) ist *kein*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

Definition 2.30 (relativ und absolut eliminierbar)**.**
Ein Primimplikant heißt *absolut eliminierbar*, wenn alle 1-en, die er abdeckt, auch durch Kernprimimplikanten abgedeckt werden. (acd) aus Beispiel 2.29 ist also ein absolut eliminierbarer Primimplikant. absolut eliminierbar

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

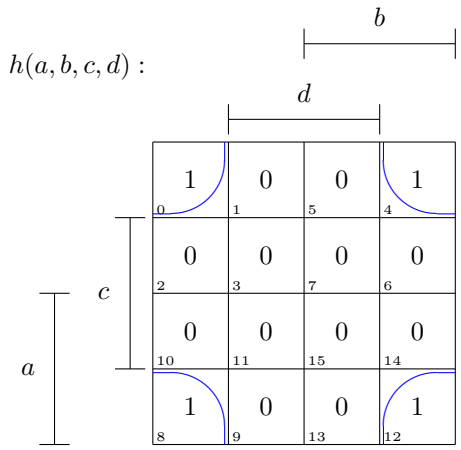
Achtung! Die Ränder der KV-Diagramme sind auch benachbart. D.h. Felder am linken und rechten Rand, sowie oben und unten können auch in einem Block zusammengefasst werden.

Beispiel 2.31.
Hier bilden die Felder $0, 1, 4, 5, 8, 9, 12$ und 13 den Primimplikanten $\overline{c}$. Die Felder $8, 10, 12$ und 14 werden zum Primimplikanten $(a\overline{d})$



Auch die Ecken eines KV-Diagramms sind benachbart!

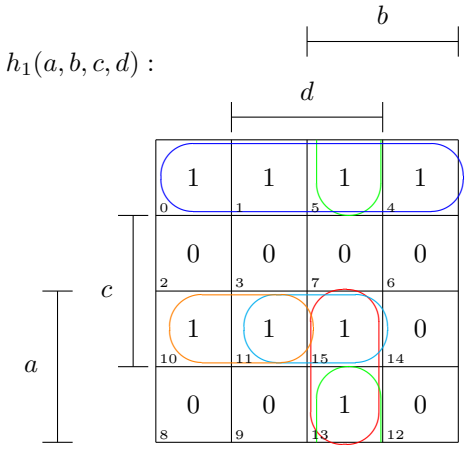
Die Felder mit Indizes $0, 4, 8$ und 12 werden zum Primimplikanten $(\overline{c}\overline{d})$ zusammengefasst.



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 Kernprimiplikanten in der minimalen DNF vertreten. Die absolut eleminierbaren Primimplikanten benötigt man nicht, denn ihre 1-en sind bereits von den Kernprimimplikanten überdeckt. Aus den relativ eleminierbaren Primimplikanten wählt man dann die aus, die die restlichen 1-en überdecken.

Beispiel 2.32.
Im KV-Diagramm der Funktion $h_1(a, b, c, d)$ sind bereits alle Primimplikanten eingezeichent:



Kernprimimplikanten sind $(\overline{a}\overline{c})$, und $(a\overline{b}c)$. Alle anderen sind relativ eleminierbare 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 Primiplikanten $(b\overline{c}d)$ und (acd) abzudecken.

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

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

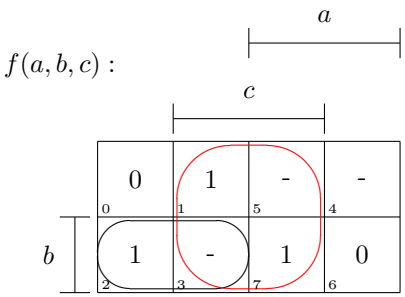
Primimplikate

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

2.8. Unvollständig spezifizierte Funktionen

Es kommt vor, dass Schaltfunktionen nicht vollständig spezifizert 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 zu an der Stelle zu 1 oder zu 0 auswerten soll.

Beispiel 2.33.
Funktion $f(a, b, c)$ ist durch folgendes KV-Diagramm gegeben.



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

So kommt man auf die minimierte Form

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

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

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

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. Implementation der Schaltung

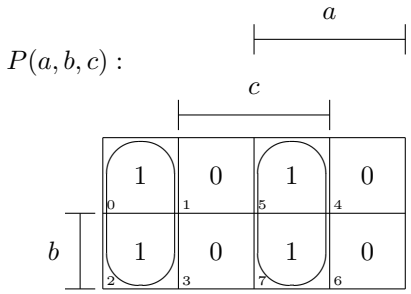
   Beispiel 2.34.
   Wenn wir z.B. eine Schaltung $P(a, b, c)$ entwerfen wollen, die immer dann eine 1 ausgibt wenn die Eingabebelegung ein Palindrom[2] ist.

   Wie sieht die Wahrheitstabelle für solch eine Funktion aus?

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

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

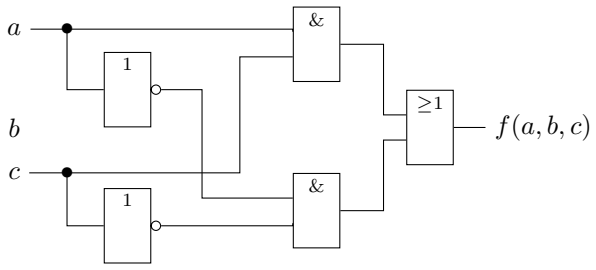
Das zugehörige KV-Diagramm sieht so aus:



Die minimierte DNF lautet dann:

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

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

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

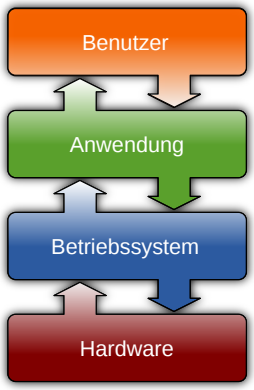


Abbildung 3.1.:

Unix

Unix ist ein Betriebssystem und wurde 1969 in den *Bell Laboratories* (später *AT&T*) entwickelt. Als Betriebssytem 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 (**G**NU's **N**ot **U**nix) 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.

GNU Linux

Da der Sourcecode frei zugänglich ist, kann jeder seine eigenen Anwendung und Erweiterungen programmieren und diese veröffentlichen. Es gibt vielzählige Linux-Distributionen (Red Hat, SuSe, Ubuntu,...), welche unterschiedliche Software Pakete zur Verfügung stellen. Auf den Rechnern der **R**echner**b**etriebsgruppe **I**nformatik der Goethe Universität (RBI) ist Red Hat Linux installiert.

Terminal

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

textbasiertes Terminal
CLI

Textbasierte Terminals stellen dem Benutzer eine Kommandozeile zur Verfügung. Über diese kann der Benutzer, durch Eingabe von Befehlen, mithilfe der Computertastatur, mit Programmen, die über ein CLI (**c**ommand **l**ine **i**nterface) verfügen, interagieren. 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 meist, dass die Kommandozeile auf eine Eingabe wartet. Man muss den Befehl kennen, den 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. Der Benuter bedient die Programme durch Klicken auf bestimmte Teile des Bildschirms, welche durch Icons

(kleine Bilder) oder Schriftzüge gekennzeichnet sind. Damit das funktionert, benötigen die Programme eine graphische Benutzeroberfläche, auch GUI (**g**raphical **u**ser **i**nterface) genannt. Auf den Rechnern der RBI findet man, 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. Tastatureigaben werden immer an das angezeigte Terminal weitergeleitet. In der Regel ist lediglich das durch die Tastenkombination Strg + Alt + F7 erreichbare Terminal graphisch. Alle andern Terminals sind textbasiert. Auf den RBI-Rechnern ist das graphische Terminal als das aktive Terminal eingestellt, sodass ein Benutzer der nicht Strg + Alt + F1 , . . . , Strg + Alt + F6 drückt, die textbasierten Terminals nicht zu Gesicht bekommt.

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 erreicht man beispielsweise mit `/home/hans/protokoll.pdf` eine persönliche Datei des Benutzers „hans“, ein Verzeichnis auf einem Netzwerklaufwerk mit `/usr`, eine Festplattenpartition mit `/dev/sda1/` und sogar die Maus mit `/dev/mouse`.

Dateibaum
Verzeichnis

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 Verzeichnis, dem *übergeordneten* Verzeichnis (engl.: *parent directory*). Nur das *Wurzelverzeichnis* (engl.:*root directory*) ist in sich selbst enthalten. Es trägt den Namen „/“.

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. Die Bäume mit denen man es meist in der Informatik zu tun hat, stehen auf dem Kopf. Die Wurzel befindet sich oben, die Blätter unten.

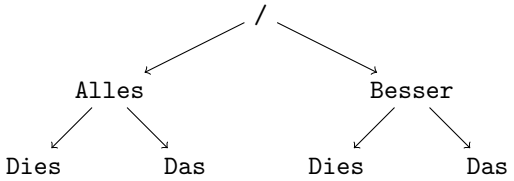


Abbildung 3.2.: Ein Dateibaum

Die beiden Verzeichnisse mit Namen `Dies` lassen sich anhand ihrer Position im Dateibaum leicht auseinanderhalten. Den Weg durch den Dateibaum zu dieser Position nennt man *Pfad* (engl::*path*). Gibt man den Weg von der Wurzel aus an, so spricht man vom *absoluten* Pfad. Gibt man den Weg vom Verzeichnis aus an, in dem man sich gerade befindet, so spricht man vom *relativen* Pfad. 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 Verzeich-

Pfad
absolut
relativ
slash

nissen. 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 aus dem 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 auf bestimmte Dateien verwaltet werden. Dabei wird unterschieden zwischen Lese-(*r*ead), Schreib-(*w*rite) und Ausführrechten (*x* ecute). Für die Vergabe dieser Rechte, wird zwischen dem Besitzer (*o*wner) der Datei, einer Gruppe (*g*roup) von Benutzern und allen Nutzern, die nicht zu der Gruppe gehören (*o*ther), unterschieden. Um bestimmten Nutzern Zugriffsrechte für bestimmte Dateien zu erteilen, können diese Nutzer 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, muss sich der Benutzer zunächst anmelden. Dies geschieht durch Eingabe eines Benutzernamens und des zugehörigen Passwortes. Diesen Vorgang nennt man „sich *Einloggen*“. Loggt man sich an einem textbasierten Terminal ein, so startet nach dem Einloggen automatisch eine (Unix)-*Shell*. Dies ist die traditionelle Benutzerschnittstelle unter UNIX/Linux. Der Benutzer kann nun über die Kommandozeile Befehle eintippen, 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.

Einloggen
Shell
Eingabe-aufforderung

Benutzer, die sich an einem graphischen Terminal einloggen, müssen zunächst ein virtuelles textbasiertes Terminal starten, um eine Shell zu Gesicht zu bekommen. Ein virtuelles textbasiertes Terminal kann man in der Regel über das Startmenü, Unterpunkt „Konsole“ oder „Terminal“, gestartet werden. Unter der graphischen Benutzeroberfläche KDE kann man solch ein Terminal auch starten, indem man mit der rechten Maustaste auf den Desktop klickt und im erscheinenden Menü den Eintrag „Konsole“ auswählt (Abb.: **??**).

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 Bildschirmein- 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 zunächt einmal das alte Passwort eingegeben werden. Dannach 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.

Netzwerkpasswort

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].
```

Arbeitsverzeichnis
Homeverzeichnis

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 *„Arbeitsverzeichnis“* 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/Lernzentrum/Vorkurs/WS1314/Skript/
```

whoami : gibt den Benutzernamen aus.

```
> whoami ↩
ronja
```

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

```
> hostname ↩
nash
```

Verzeichnis erstellen
Argument

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.

```
> mkdir Zeug ↩
```

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

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

```
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
> mkdir Wichtiges ↩
> cd Wichtiges ↩
> pwd ↩
 /home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/Wichtiges/
> cd .. ↩
> pwd ↩
/home/ronja/Lernzentrum/Vorkurs/WS1314/Skript/
```

`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 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.

Option

```
> 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  Jul 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. Juli diesen 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 linkesten 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 Besitzers (engl.: *owner*) der Datei. In diesem Fall darf die Besitzerin alice die Datei lesen (*r*ead) und verändern (*w*rite). 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 letzen drei Positionen sind für die Zugriffrechte aller andern Personen (engl.: *other*). Auch diesen ist gestattet die Datei zu lesen, sie dürfen sie aber nicht verändern.

Zugriffsrechte

`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 (*u*ser,*g*roup,*o*ther, oder *a*ll) Rechte man wie ändern (`+` hinzufügen, `-` wegnehmen) will.

```
> chmod go +w protokoll.pdf ↵
```

Dieser Befehl gibt der Gruppe `g` und allen anderen Nutzern `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 ↵
```

nimmt allen anderen Nutzern `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 Erkärung 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.

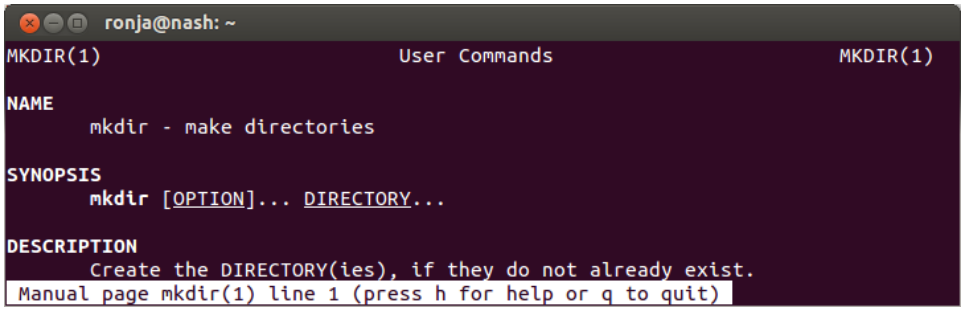


Abbildung 3.3.: man page des Befehls `mkdir`

3.1.4. History und Autovervollständigung

History

Ein sehr nützliches Hilfmittel beim Arbeiten mit der Shell ist die *history*. Alle Befehle, die man in der Shell eingibt, 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 Drüchen von ↑ den vorletzen, usw. ↓ arbeitet in die andere Richtung, also z.B. vom vorletzen Befehl zum zuletzt eingegebenen Befehl. Mit den Cursortasten ← und →, kann man innerhalb des Befehls navigieren, um Änderungen vorzunehmen.

Autovervollständigung

Ein weiteres nützliches Hilfmittel ist die *Autovervollständigung*. Hat man den Anfang eines Befehls, oder eines Datei- (oder Verzeichnis-) Namens eingegeben, so kann man den Namen durch Betätigen der *Tab*-Taste ⇆ automatisch vervollständigen lassen, solange der angegebene Anfang eindeutig ist. Ist dies nicht der Fall, so kann man sich mit nochmaliges Betätigen der Tab-Taste ⇆, eine Liste aller in Frage kommenden Vervollständigungen anzeigen lassen (Abb. 3.4).

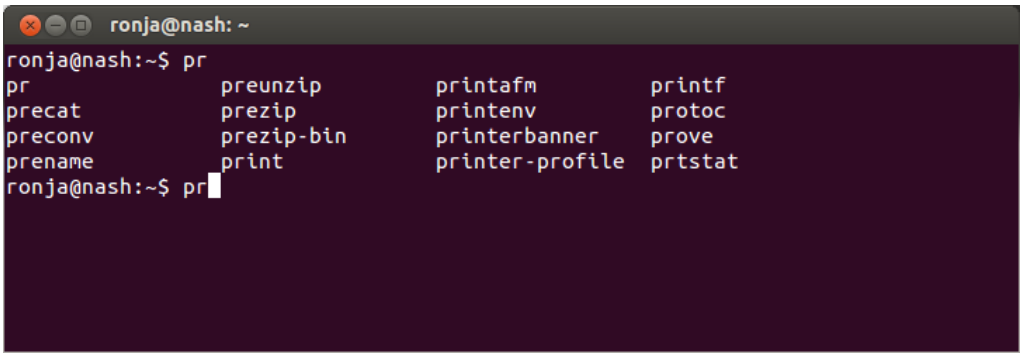


Abbildung 3.4.: Autovervollständigung für die Eingabe `pr`

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. Texteditor

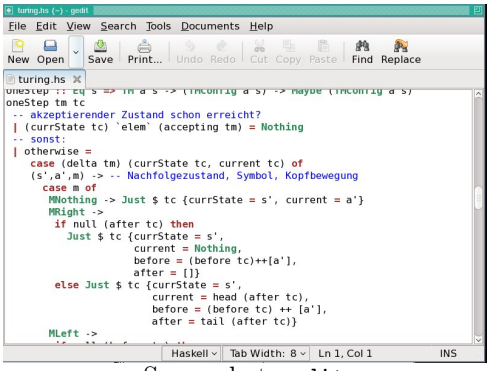
Es gibt unzählige Texteditoren. Unter KDE ist z.B. der Editor `kate` (`http://kate-editor.org/`), unter Gnome der Editor `gedit` (`http://projects.gnome.org/gedit/`) sehr empfehlenswert. Diese können allerdings nur im graphischen Modus ausgeführt werden, dafür ist ihre Bedienung dank vorhandener Mausbedienung 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.

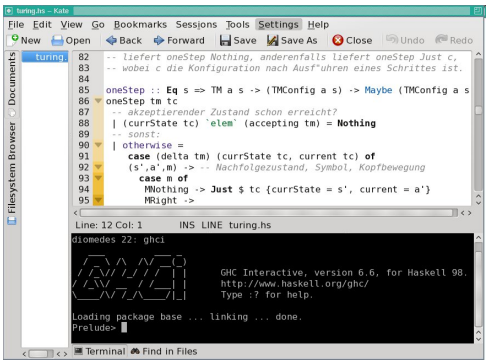
Hat man kein graphisches Interface zur Verfügung, so kann man einen Text-basierten Editor verwenden. Ein solcher weit verbreiteter Texteditor heißt `vi` (sprich: [vi: ai]). Dieser steht nicht nur in der RBI zur Verfügung, er ist auf fast jedem Unix-System vorhanden. Wir werden kurz auf seine Bedienung eingehen. Mit dem Befehl `vi` wird der Editor[1] 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 `vi` unterscheidet einige Betriebsmodi, die sich wie folgt beschreiben lassen. Wichtig dabei ist die Position des *Cursors*, auf die sich die meisten Aktionen beziehen. `vi`

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!“ (d , d) oder „Tausche den Buchstaben unterm Cursor durch den folgenden aus!“ (r). Natürlich ist es auch möglich, den Cursor zu

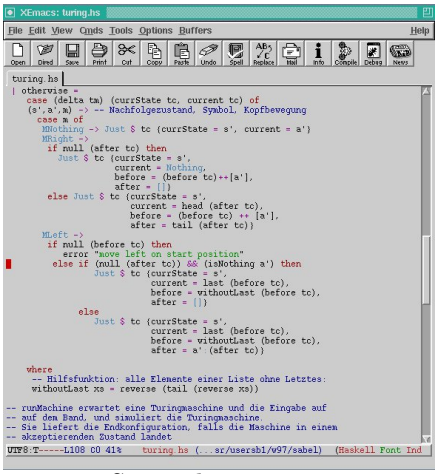
[1] Oftmals handelt es sich schon um eine verbesserte Variante `vim` (für `vi` *improved*). Diese Unterscheidung soll uns hier nicht kümmern.



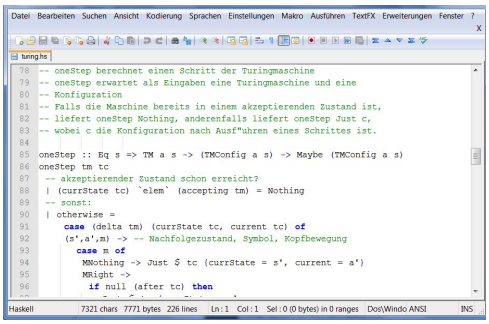
Screenshot `gedit`



Screenshot `kate`



Screenshot `xemacs`



Screenshot Notepad++ (MS Windows)

Abbildung 3.5.: Screenshots verschiedener Editoren

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 zweiteren 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 hergewechselt werden. Durch Betätigen der [Esc]-Taste gelangt man wieder zurück in den Befehlsmodus.
3. Die `VISUELL`en 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 [v]-Taste in den gewöhnlichen visuellen Modus. Dieser ermöglicht das Markieren eines Textbereiches durch Bewegen des Cursors. Nach Eingabe und Ausführung eines Befehls – etwa [x], „markierten Bereich löschen“ – findet man sich im Befehlsmodus wieder. Auch die [Esc]-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 man die verschiedenen Programmiersprachen grob einteilen kann. Anschließend wird kurz erläutert, wie man den Interpreter GHCi für die funktionale Programmiersprache Haskell (insbesondere auf den Rechnern der RBI) verwendet. Genauere Details zum Programmieren in Haskell werden erst im nächsten Kapitel erläutert.

4.1. Programme und Programmiersprachen

Rechner

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).

Maschinenprogramm

Ein *ausführbares* Computerprogramm ist eine Folge *Maschinencodebefehlen*, die man auch als *Maschinenprogramm* bezeichnet. 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 diese 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 spricht man auch vom „Zustand“ des Rechners und meint damit die gesamte Speicherbelegung).

höhere Programmiersprache

Quellcode

Compiler

Interpreter

Allerdings sind Maschinenprogramme eher schwierig zu erstellen und für den menschlichen Programmierer schwer zu verstehen. Deshalb gibt es sogenannte *höhere Programmiersprachen*, die es dem Programmierer erlauben, besser verständliche Programme zu erstellen. Diese Programme versteht der Computer allerdings nicht direkt, d.h. sie sind nicht ausführbar. Deshalb spricht man oft auch von sogenanntem *Quelltext* oder *Quellcode*. 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 guter Compiler/Interpreter eine Fehlermeldung aus, die dem Programmierer mitteilt, an welcher Stelle der Fehler steckt. Je nach Programmiersprache und je nach Compiler/Interpreter kann man hier schon Programmierfehler erkennen und mithilfe der Fehlermeldung korrigieren.

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

Im Allgemeinen unterscheidet man Programmiersprachen in *imperative* und in *deklarative* Programmiersprachen.

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 verständlicher sind, und dass meistens vom tatsächlichen Speicher abstrahiert wird, indem sogenannte Programmvariablen verwendet werden. Diese sind im Grunde Namen für Speicherbereiche, wobei der Programmierer im Programm nur die Namen verwendet, und die Abbildung der Namen auf den tatsächlichen Speicher durch den Compiler oder den Interpreter geschieht (der Programmierer braucht sich hierum nicht zu kümmern). Z.B. kann X für eine Variable stehen. Verwendet man in einem imperativen Programm die Variable X beispielsweise im *Ausdruck* $X + 5$, 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. Als Kontrollstrukturen bezeichnet man sowohl *Verzweigungen* als auch *Schleifen*. 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.

prozedurale Programmiersprache

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. Hierdurch kann man Programmcode sparen, da ein immer wieder verwendetes Programmstück nur einmal programmiert werden muss. Bekannte prozedurale Programmiersprachen sind z.B. *C*, *Fortran* und *Pascal*.

objektorientierte Programmiersprache

Eine weitere Unterklasse der imperativen Programmiersprachen, sind die sogenannten *Objektorientierten Programiersprachen*. In objektorientierten Programmiersprachen werden Programme durch sogenannte *Klassen* dargestellt. Klassen geben das Muster vor, wie Instanzen dieser Klasse (Instanzen nennt man auch *Objekte*) aussehen. Klassen bestehen im Wesentlichen aus *Attributen* und *Methoden*. Attribute legen die Eigenschaften fest, die ein Objekt haben muss (z.B. könnte man sich 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 kann man Unterklassen erzeugen, 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). Daher zählt man objektorientierte Sprachen zu den imperativen Programmiersprachen. Bekannte objektorientierte Programmierspra-

che 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 dadurch das gewünschte Ergebnis zu erhalten. Programme deklarativer Programmiersprachen beschreiben im Allgemeinen nicht die Speichermanipulationen, sondern bestehen aus (oft mathematischen) *Ausdrücken*. Zur Ausführung des Programms werden diese Ausdrücke *ausgewertet*. In der Schule führt man eine solche Auswertung oft per Hand für arithmetische Ausdrücke durch. Will man z.B. den Wert des Ausdrucks $(5 \cdot 10 + 3 \cdot 8)$ ermitteln, so wertet man den Ausdruck aus (was man in der Schule auch oft 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, so 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 deklarative Programmiersprachen i.A. keine Speichermanipulationen direkt durchführen, sind meist weder eine Zuweisung noch Schleifen zur Programmierung vorhanden (diese manipulieren nämlich den Speicher). Um jedoch Ausdrücke wiederholt auszuwerten, wird *Rekursion* bzw. werden *rekursive Funktionen* verwendet. Diese werden wir später genauer betrachten. An dieser Stelle 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 Mengen 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 braucht man zur Auswertung eigentlich gar keinen Rechner, man könnte das Ergebnis auch stets per Hand mit Zettel und Stift berechnen (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.

referentielle Transparenz

Aus dieser Sicht klingen funktionale Programmiersprachen sehr mathematisch und es ist zunächst nicht klar, was man außer arithmetischen Berechnungen damit anfangen kann. 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.)

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

operieren dürfen. So kann man z.B. Funktionen definieren, die als Eingabe einen Text erhalten und Schreibfehler im Text erkennen und den verbesserten Text als Ausgabe zurückliefern, oder man kann Funktionen schreiben, 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 „Grundlagen der Programmierung 2“ 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 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).

Das aktuelle Haskell-Logo

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.

GHC,GHCi

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 „Grundlagen der Programmierung 2“ ist die Benutzung des Interpreters ausreichend. Die Homepage des GHC ist `http://www.haskell.org/ghc`.

4.2.1. GHCi auf den Rechnern der RBI

Auf den RBI-Rechnern ist der Compiler GHC und der Interpreter GHCi bereits installiert. Den Haskell-Interpreter GHCi findet man unter `/opt/rbi/bin/ghci`, d.h. er kann mit dem Kommando `/opt/rbi/bin/ghci` (ausgeführt in einer Shell) gestartet werden[2].

4.2.2. GHCi auf dem eigenen Rechner installieren

Für die Installation des GHC und GHCi bietet es sich an, die *Haskell Platform* zu installieren, diese beinhaltet neben GHC und GHCi einige nützliche Werkzeuge und Programmbibliotheken für Haskell. Unter

`http://hackage.haskell.org/platform/`

[2] wenn die Umgebungsvariable `PATH` richtig gestetzt ist, genügt auch das Kommando `ghci`

stehen installierbare Versionen für verschiedene Betriebssysteme (MS Windows, Mac OS) zum Download zur Verfügung. Für Linux-basierte Systeme kann für einige Distributionen (Ubuntu, Debian, Fedora, Arch Linux, Gentoo, NixOS) die Haskell Platform über den Paketmanager installiert werden. Die Seite `http://hackage.haskell.org/platform/linux.html` enthält dazu Links (und auch Informationen für weitere Distributionen).

Alternativ (nicht empfohlen) kann der GHC/GHCi alleine installiert werden, er kann von der Homepage `http://www.haskell.org/ghc/` heruntergeladen werden.

Nach der Installation ist es i.A. möglich den GHCi zu starten, indem man das Kommando `ghci` in eine Shell eintippt (unter MS Windows ist dies auch über das Startmenü möglich).

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 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

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
```

Gibt man ungültige Ausdrücke ein (also Ausdrücke, die keine Haskell-Ausdrücke sind), so erhält man im Interpreter eine Fehlermeldung, z.B.

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

Hierbei sollte man 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 `1:8` verraten, dass der Fehler in der 2. Zeile und der 9. Spalte (im interaktiven Modus werden die Zeilen stets weitergezählt, daher ist Zeile 1 für den Text `GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help` verbraucht) ist. Tatsächlich ist dort das `+`-Zeichen, dem keine Zahl folgt. 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:

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

4.2.4. Quelltexte erstellen und im GHCi laden

Normalerweise erstellt man den Quelltext eines Haskell-Programms in einem Editor (siehe Kapitel 3), und speichert das Haskell-Programm in einer Datei. 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 bezeichnet man auch als *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 7.10.3: 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`, so müssen wir dem `:load`-Kommando nicht nur den Dateinamen (`hallowelt.hs`), sondern den Verzeichnispfad mit übergeben (also `:load programme/hallowelt.hs`). Wir zeigen was passiert, wenn man den Verzeichnispfad vergisst, und wie es richtig ist:

```
> ghci ↩
GHCi, version 7.10.3: 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:

```
*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, aber rechts vom `=` stehen *Ausdrücke*,

```
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`

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*.

Funktionsaufruf

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

```
> ghci ↩
GHCi, version 7.10.3: 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
```

Beachte, dass Funktionsnamen in einer Quelltextdatei mit einem Kleinbuchstaben oder dem Unterstrich _ beginnen *müssen*, anschließend können Großbuchstaben und verschiedene Sonderzeichen folgen. Hält man sich nicht an diese Regel, so erhält man beim Laden des Quelltexts eine Fehlermeldung. Ersetzt man z.B. `zwei_mal_Zwei` durch `Zwei_mal_Zwei` so erhält man:

Funktionsnamen

```
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? Man sagt auch: Der Quelltext soll *dokumentiert* sein. Der Grund hierfür ist, dass man selbst nach einiger Zeit den Quelltext wieder verstehen, verbessern und ändern kann, oder auch andere Programmierer 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: Fügt man im Quelltext in einer Zeile zwei Minuszeichen gefolgt von einem Leerzeichen ein, d.h. „`-- `“, so werden alle Zeichen danach bis zum Zeilenende als Kommentar erkannt und dementsprechend vom GHCi ignoriert. 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: Man kann in Haskell einen ganzen Textblock (auch über mehrere Zeilen) als Kommentar markieren, indem man ihn in durch spezielle Klammern einklammert. 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:

```
{- 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

Jeder Programmierer 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))`). Ein 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.

Syntaxfehler

Logischer Fehler

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 man eine Zahl nicht mit einem Buchstaben addieren kann. Wir werden später Typfehler genauer behandeln.

Typfehler

Man kann Programmierfehler auch danach unterscheiden, *wann* sie auftreten. Hierbei unterscheidet man in *Compilezeitfehler* und *Laufzeitfehler*. Wenn ein Fehler bereits beim Übersetzen des Programms in Maschinensprache entdeckt wird, dann spricht man von einem Compilezeitfehler. In diesem Fall bricht der Compiler die Übersetzung ab, und meldet dem Programmierer den Fehler. Auch der GHCi liefert solche Fehlermeldungen. Betrachte beispielsweise den folgenden Quellcode in der Datei `fehler.hs`

Compilezeitfehler

```
-- 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.

Laufzeitfehler

```
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, das 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 „Grundlagen der Programmierung 2“ 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,...`. 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 Falle 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*. Setzt man Ausdrücke aus kleineren Ausdrücken zusammen, so müssen die Typen stets zueinander passen, z.B. darf man Funktionen, die auf Zahlen operieren, nicht auf Strings anwenden, da die Typen nicht zueinander passen.

Typ

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

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

Man sagt der Ausdruck (bzw. der Wert) `'C'` *hat den Typ* `Char`. Hierbei steht `Char` für *Character*, d.h. dem englischen Wort für Buchstabe. Typnamen wie `Char` beginnen in Haskell stets mit einem Großbuchstaben. Mit dem Kommando `:set +t` kann man den GHCi in einen Modus versetzen, 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:

`Char`

```
> ghci ↩
GHCi, version 7.10.3: 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. Man kann in Haskell den Typ eines Ausdrucks auch selbst angeben (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 der Programmierer den Typ *nicht* selbst angegeben, da der GHCi den Typ selbstständig herleiten kann. Dieses Feature nennt man auch *Typinferenz*. 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. 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 findet man häufig Aussagen der 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`

Datenkonstruktor

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.

Will man z.B. die atomare Aussage A_1: „Heute regnet es.“ negieren, würde man sagen A_2: „Heute regnet es *nicht*“, d.h. man negiert die Aussage, dabei gilt offensichtlich: Die Aussage A_1 ist genau dann wahr, wenn die Aussage A_2 falsch ist, und umgekehrt. In mathematischer Notation schreibt man die Negation als $\neg$ und könnte daher anstelle von A_2 auch $\neg A_1$ schreiben.

$\neg$

Die Bedeutung (Auswertung) von Booleschen Junktoren wird oft durch eine Wahrheitstabelle repräsentiert, dabei gibt man für alle möglichen Belegungen der atomaren Aussagen, den Wert der Verknüpfung an. 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 man 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 Pommes“ ausdrücken. 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.

$\wedge$

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.

$\vee$

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:

Bezeichnnung	Mathematische Notation	Haskell-Notation
logische Negation	$\neg$ F	`not` F
logisches Und	$F_1 \wedge F_2$	F_1 `&&` F_2
logisches Oder	$F_1 \vee F_2$	F_1 `||` F_2

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`

`Int`, `Integer`

Für ganze Zahlen sind in Haskell zwei verschiedene Typen eingebaut: 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. z.B. wenn man `1000` eingibt, so weiß der Interpreter eigentlich nicht, welchen Typ man meint, man kann dies durch Angabe des Typs festlegen, indem man `1000::Int` bzw. `1000::Integer` eingibt. Lässt man den Typ weg, so führt der Interpreter manchmal sogenanntes *Defaulting* durch (wenn es nötig ist) und nimmt automatisch den Typ `Integer` an. Fragen wir den Interpeter doch mal nach dem Typ von `1000`:

```
Prelude> :type 1000 ↩
1000 :: (Num t) => t
```

Der ausgegebene Typ ist weder `Integer` noch `Int`, sondern `(Num t) => t`. Den Teil vor dem `=>` nennt man *Typklassenbeschränkung*. Wir erklären Typklassen hier nicht genau, aber man kann den

Typ wie folgt interpretieren: `1000` hat den Typ `t`, *wenn* `t` *ein Typ der Typklasse* `Num` *ist*[1]. Sowohl `Int` als auch `Integer` sind Typen der Typklasse `Num`, d.h. der Compiler kann für die *Typvariable* `t` sowohl `Integer` als `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 Gleitkommazahlen (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:

`Float`, `Double`

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

Man muss den Punkt nicht stets angeben, 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 als `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> 1.000000000000001 ↩
1.000000000000001
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. Ein kleine Auflistung ist

[1] Typklassen 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 beiden Typs bereitstehen.

Darstellung in Haskell	Zeichen, das sich dahinter verbirgt
`'\\'`	Backslash \
`'\''`	einfaches Anführungszeichen '
`'\"'`	doppeltes Anführungszeichen "
`'\n'`	Zeilenumbruch
`'\t'`	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, Substraktion, 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 man dem Interpreter durch zusätzliche Klammern helfen, damit er „weiß“, ob es sich um eine negative Zahl oder um die Subtraktion handelt. Ein Beispiel hierfür ist der Ausdruck `2 * -2`, gibt man diesen im Interpreter ein, so 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
```

Klammert man jedoch `(-2)`, so kann der Interpreter den Ausdruck erkennen:

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

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. kann man auch Boolesche Werte damit vergleichen.

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
```

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 man zur Anwendung der Funktion f auf die

zwei Argumente 10 und 3 schreiben $f(10, 3)$, 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 schreiben `f 10 3` oder auch `(f 10 3)`. 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 ganzahligen Anteil der Division mit Rest berechnet. Wir probieren dies gleich einmal im Interpreter aus:

partielle Anwendung

`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 kann man die Infix-Operatoren wie `+` und `==` auch in Präfix-Schreibweise verwenden, 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 kann man zweistellige Funktionen wie `mod` und `div` auch in Infix-Schreibweise verwenden, indem man den Funktionsnamen in Hochkommata (diese werden durch die Tastenkombination ⇧ + ` eingegeben) umschließt, 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:

Funktionstyp, `->`

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

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

$$f :: \underbrace{Typ_1}_{\substack{\text{Typ des}\\ \text{1. Arguments}}} \texttt{ -> } \underbrace{Typ_2}_{\substack{\text{Typ des}\\ \text{2. Arguments}}} \texttt{ -> } \ldots \texttt{ -> } \underbrace{Typ_n}_{\substack{\text{Typ des}\\ n\text{. Arguments}}} \texttt{ -> } \underbrace{Typ_{n+1}}_{\substack{\text{Typ des}\\ \text{Ergebnisses}}}$$

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 man `(&&)` auch als Funktion interpretieren, die *eine* Eingabe vom Typ `Bool` erwartet und als Ausgabe eine *Funktion* vom Typ `Bool -> Bool` liefert. Auch dies kann man im GHCi testen:

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

Kehren wir zurück zu den Funktion 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
```

Fragt man den Interpreter nach den Typen von `mod` und `div` so erhält man etwas allgemeinere Typen:

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

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. Man kann dies so verstehen: `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 man Zahlen verwendet 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
```

Funktionsdefinition

Die Syntax für Funktionsdefinitionen in Haskell sieht folgendermaßen aus:

$$\textit{funktion\_Name} \quad \textit{par}_1 \quad \ldots \quad \textit{par}_n \quad \texttt{=} \quad \textit{Haskell\_Ausdruck}$$

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. $par_1 \ldots par_n$ 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 $par_1 \ldots par_n$ verwendet werden.

Man darf dem Quelltext auch den Typ der Funktion hinzufügen[2]. 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:

```
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

`if-then-else`

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

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` auswertet), *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.

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

[3] Solche 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` e_1 `else` e_2 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!

- `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:

```
*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 `verdoppele` von vorher nun auch unter Verwendendung 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.

Funktionen als Parameter

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)
```

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:

$$\texttt{wende\_an\_und\_addiere} :: \underbrace{\texttt{(Integer -> Integer)}}_{\text{Typ von f}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von x}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ von y}} \texttt{ -> } \underbrace{\texttt{Integer}}_{\text{Typ des Ergebnisses}}$$

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))).
```

Funktion höherer Ordnung

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:

```
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 Verwendendung 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:

```
 *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 ... =  ...  f  ...
```

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)
→   ...
```

Rekursionsanfang und Rekursionsschritt

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`:

```
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

[4] Tatsächlich 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.

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:

```
*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:

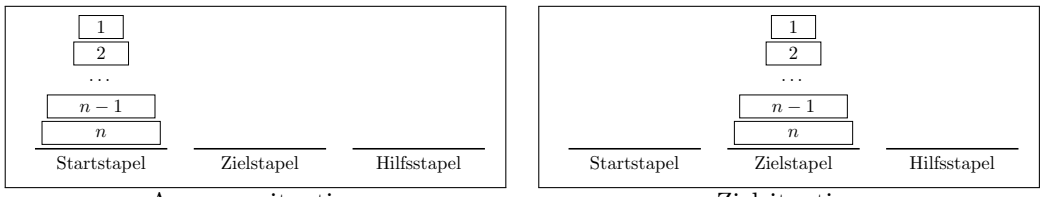
```
  erste_rekursive_Funktion 5
= 5 + erste_rekursive_Funktion 4
= 5 + (4 + erste_rekursive_Funktion 3)
= 5 + (4 + (3 + erste_rekursive_Funktion 2))
= 5 + (4 + (3 + (2 + erste_rekursive_Funktion 1)))
= 5 + (4 + (3 + (2 + (1 + erste_rekursive_Funktion 0))))
= 5 + (4 + (3 + (2 + (1 + 0))))
= 15
```

Durch längeres Anschauen der Definition erkennt man, dass für jede nicht-negative Zahl x gerade $x+(x-1)+(x-2)+\ldots+0$ (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 erwähnswert, 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:

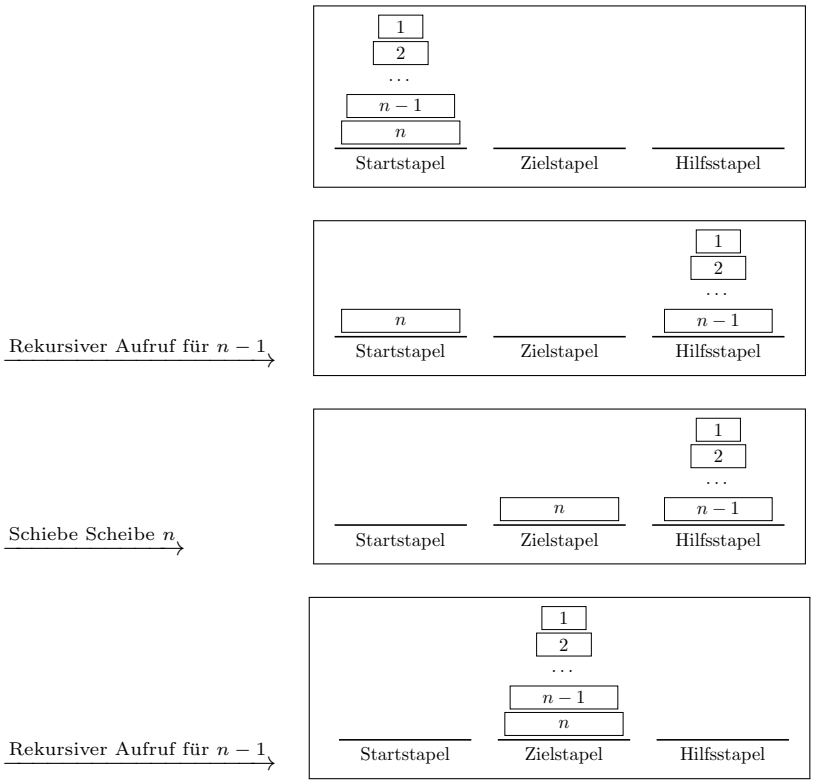


Ausgangssituation Zielsituation

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 Probleminstanz (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, 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:

> 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. Festellen, 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
```

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`-Ausdruck

`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:

```
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 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:

Nil

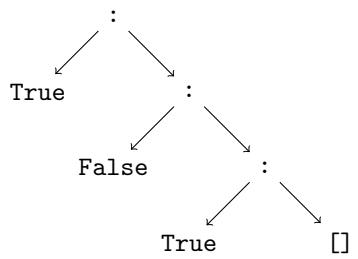
Cons

- `[]` 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] Diese Bezeichnung stamm vom lateinischen Wort „nihil“ für „Nichts“.
[6] Kurzform von „Constructor“, also einem Konstruktor.

```
*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

```
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)
```

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 schließich 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.