Informatik A

Skript zur Vorlesung
Wintersemester 2001/2002 - Freie Universität Berlin

Dozent : ***Dr. Frank Hoffmann***
Autor : ***Marco Block***

7. Februar 2002

Inhaltsverzeichnis

Vorwort

Es gibt viele Gründe, weshalb Studenten Vorlesungen nicht besuchen können. Sei es nun eine Überschneidung der Vorlesungen oder ein studentenunwürdiger Beginn. Dieses Skript zielt in erster Linie nicht auf solche Probleme, sondern soll dem Leser als Lernhilfe und Nachschlagewerk dienen. Sicherlich ersetzt kein Skript eine Vorlesung!

Anregungen und Kritik an:

Marco Block - block@inf.fu-berlin.de oder

Dr. Frank Hoffmann - hoffmann@inf.fu-berlin.de

Um dieses Skript frei von Unklarheiten, Fehlern und Versäumnissen zu machen, bitten wir um kollektive Fehleranalyse.

Besonderer Dank gilt Frank Hoffmann, der die Korrektur höchst selbst in die Hand genommen und alle Abbildungen erstellt hat. Ebenso Tomasz Naumowicz, Philipp Wilimzig und anderen.

Einleitung

Informatik A und B sind Vorlesungen, die Studenten anderer Fachrichtungen die Möglichkeit bietet, Informatik als Nebenfach zu studieren. Ebenso sollen diese beiden Vorlesungen den Studenten mit dem Wissen ausstatten, welches ihm ermöglicht, Hauptstudiumkurse zu absolvieren und sich dem *Informatiker* gleichwertig zu fühlen. Damit das gelingt, müssen sehr viele Aspekte der Informatik behandelt werden. Sicherlich ist es nicht möglich jede Problematik bis ins Detail zu besprechen, aber man sollte Zusammenhänge erkennen und selber in der Lage sein, sich gezielt Informationen, die das jeweilige Thema betreffen, zu beschaffen. Das Informatikstudium umfasst die Bereiche Mathematik, Technische, Anwendungsorientierte , Theoretische , sowie Praktische Informatik.

Information A betrachtet nun Teildisziplinen aus Technischer und Theoretischer Informatik. Es werden aber auch relevante mathematische Kenntnisse gefestigt und neu vermittelt. Ebenso wird Haskell als eine funktionale Programmiersprache eingeführt, die das logische Verständnis für komplexe Vorgehensweisen der Algorithmentechnik unterstützt und fördert.

Informatik B übernimmt den Part der Anwendungsorientierten und Praktischen Informatik. Hier wird Java als Programmiersprache gelehrt und die Kenntnisse in Algorithmik und Datenstrukturen vertieft.

Kapitel 1

Informationssysteme

1.1 Informatik, Information, Informationssystem

Als Informatiker müssen wir zunächst einmal wissen, welche Definition den Begriff Informatik eigentlich prägt.

Informatik(nach Broy) Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung, Speicherung, Übertragung und Darstellung von Information.

Aber was verstehen wir unter „Information“? Dazu gibt es mehrere Faktoren.

Die **äußere Form** (Darstellung), **Bedeutung** („abstrakte“ Information) und der **Bezug zur realen Welt** (Gültigkeit).

Information Information ist der abstrakte Gehalt eines Dokuments, einer Aussage oder Beschreibung. Ihre äußere Form heißt Repräsentation. In der Informatik führt die Repräsentation zu "Datenstrukturen" bzw. "Objektstrukturen" und die Verarbeitung führt zu "Algorithmen" bzw. "Prozessen".

Informationssystem Ein Informationssystem ist ein Tupel (A,R,I) bestehend aus

R : Menge von Repräsentationen

A : Menge von abstrakten Informationen (sematisches Modell)

I : R $\rightarrow$ A Interpretation

Die gängigste und einfachste Form der Repräsentation (in der Informatik) sind Sequenzen ("Wörter") von Symbolen aus einem zu Grunde liegenden Alphabet Σ. Dazu müssen wir uns aber mit den grundlegenden Begriffen und Definitionen in Zusammenhang mit Wörtern auseinandersetzen.

1.2 Grundlegende Begriffe im Zusammenhang mit Wörtern

Σ sei eine endliche nichtleere Menge von Symbolen (Alphabet)

Beispiele:

1. $\Sigma = \{0, 1\}$
2. $\Sigma = \{a, b, ..., z\}$

Ein **Wort** (String) über Σ ist eine endliche, möglicherweise leere Folge von Symbolen aus dem Alphabet Σ. Die leere Folge (leeres Wort) bezeichnen wir mit ε.

1. $w = x_1 x_2 ... x_n$, alle $x_i \in \Sigma$
2. $v = \varepsilon$,das leere Wort über Σ

Die **Länge** eines Wortes ist die Anzahl der vorkommenden Symbole.

1. $|\varepsilon| = 0$
2. $|x_1 x_2 ... x_n| = n$

Wir bezeichnen mit

Σ^* Menge aller Wörter über Σ

Σ^+ Menge aller nichtleeren Wörter über Σ, also $\Sigma^* \setminus \{\varepsilon\}$

Σ^k Menge aller Wörter der Länge k

Damit ist also
$\Sigma^* = \bigcup_{k \geq 0} \Sigma^k$
Um zwei Worte zu verbinden bedient man sich der **Konkatenation** $\_\circ\_ : \Sigma^* \text{x } \Sigma^* \to \Sigma^*$

Beispiele:

1. hal $\circ$ lo = hallo
2. $\varepsilon \circ w = w \circ \varepsilon = w$, für beliebiges $w \in \Sigma^*$

Hier nun eine alternative (und zwar rekursive) Definition von Σ^*:

Σ^* ist die Menge von Wörtern über Σ, die durch die Gleichung

$$\Sigma^* = \{\varepsilon\} \cup (\Sigma \circ \Sigma^*)$$

gegeben ist.
Von rechts nach links gelesen ist das eine Vorschrift zur Bildung aller Wörter, d.h.

1. $\varepsilon \in \Sigma^*$ (per Axiom)
2. Wenn $x \in \Sigma$ und $w \in \Sigma^*$, so ist auch $w' = x \circ w \in \Sigma^*$

Beispiel:

Für $\Sigma = \{0, 1\}$ entstehen so nacheinander als Elemente von Σ^*: ϵ, 0, 1, 00, 10, 11, 000, ...

Auch die Längenfunktion kann man rekursiv schreiben.

$|\;| : \Sigma^* \mapsto \mathbb{N}$
$|\ \varepsilon\ | = 0$
für $x \in \Sigma$, $w \in \Sigma^*$: $|\ x \circ w\ | = 1 + |\ w\ |$

Eine Menge $L \subseteq \Sigma^*$ heißt **formale Sprache** über Σ.

Beispiele:

1. $\Sigma^* = \{0, 1\}$; $\Sigma^* \supset Prim = \{Primzahlen\ in\ Binaerdarstellung\}$
2. $\Sigma^* = \{0, 1, ..., 9\}$; $\Sigma^* \supset Even = \{alle\ geraden\ Dezimalzahlen\}$

Eine zentrale Fragestellung der Informatik ist die Suche nach Algorithmen die entscheiden, ob ein Wort aus Σ^* zu einer Sprache $L \subseteq \Sigma^*$ gehört oder nicht. Für die Sprache *Even* ist das leicht zu bewerkstelligen, für *Prim* kennt man allerdings keine effizienten Verfahren!

1.3 Beispiele für Informationssysteme

(1) Gesucht ist eine formale Sprache, die die Dezimalzahlen repräsentiert. Dezimalzahlen sollen sich dadurch auszeichnen, daß sie keine führenden 0en haben. Ausgenommen der 0 selber.

R = Dezimalzahlen, A = Natürliche Zahlen

$\Sigma = \{0, 1, ..., 9\}$

R $= \{0\} \cup (\{1, 2, ..., 9\} \circ \Sigma^*)$

I : R $\rightarrow$ A

I(0) = „Null“

I(1) = „Eins“

...

I(11) = „Elf“

Anmerkung: „Null“, „Eins“,... selbst, sind eigentlich wieder nur sprachliche Darstellungen der abstrakten, konkreten Zahl.

(2) Gesucht ist eine Sprache, die natürliche Zahlen in Unärdarstellung repräsentieren. Dieses geschieht als Abstraktion von „natürlicher Zahl“ als Anzahl von Objekten, etwa von Elementen aus $\Sigma = \{\ |\ \}$.

$\Sigma = \{\ |\ \}$

R = „Dezimalzahlen“

A = „natürliche Zahlen in unärer Darstellung“

I : R $\rightarrow$ A

I(0) = ε

I(1) = |

I(2) = ||

...

(3) Gesucht ist Informationssystem, dem die Binärdarstellung der natürlichen Zahlen zu Grunde liegt.

$\Sigma = \{\ 0, 1\ \}$

R = Binärdarstellung einer natürlichen Zahl

A = Dezimalzahldarstellung

I : $a_n ... a_1 a_0 \mapsto \sum_{i=0}^{n} a_i 2^i$, mit $0 \leq i \leq n$; $a_i \in \{0, 1\}$

I(0) = 0

I(1) = 1

I(01) =2

...

I(1001) =9

(4) Geordnete Paare als Darstellung rationaler Zahlen

R = $\mathbb{Z}$ x $\mathbb{N}$, geordnete Paare als Darstellung rationaler Zahlen

A = Q

I : $(p, q) \mapsto \frac{p}{q}$, mit $q \neq 0$.

Nun kann man sich die Frage stellen, wie findet man eine Standardform (Normalform) einer Darstellung. Bei unserem Beispiel (4) ist das einfach, hier könnte man als Normalform den gekürzten Bruch wählen. Oft aber ist das Finden einer Standardform algorithmisch schwer.

Als *Zwischenfazit* stellen wir fest, dass Information in ihrer Abstraktion sich nicht aufschreiben, sondern nur repräsentieren lässt. Verschiedene Darstellungen sind offentsichtlich für verschiedene algorithmische Zwecke verschieden gut geeignet! (Man betrachte etwa die Multiplikation natürlicher Zahlen in römischer Darstellung...!?)

Semantische Äquivalenz Sei I : R → A ein Informationsystem. Zwei Repräsentationen r_1, $r_2 \in$ R heißen **semantisch äquivalent**, falls I(r_1) = I(r_2).

Kapitel 2

Aussagenlogik, Boolesche Terme, Boolesche Algebra

2.1 Definition Boolescher Terme

(G. Boole, enlg. Mathematiker, 1815-1864)

Ziel Informationssystem, welches Boolesche Terme als Darstellung von Aussagen in Wahrheitswerte ($true$ oder $false$) abbildet.

Zitat ***(Aristoteles (384-322 v. Chr.))*: „Eine Aussage ist ein sprachliches Gebilde, von dem es sinnvoll ist zu sagen, es sei wahr oder falsch.“** (Dieser Ausgangspunkt führt aber auch schnell zu Paradoxien, wie : "Die Aussage dieses Satzes ist falsch.")

- brauchen formales System zur Darstellung von Aussagen, es werden nur Aussagen zugelassen, die nach bestimmten Mustern zusammengesetzt sind (von elementaren Bestandteilen zu komplexen Aussagen)
- Aussagen erfordern Bezugssystem, dadurch erhalten wir die Möglichkeit der Verifikation

Jedes *Bezugssystem* hat :

1. Menge E von elementaren Aussagen mit festgelegtem Wahrheitswert immer true, false $\in E$

2. Aussagenschemata : Diese enthalten Variable aus Variablenmenge V, die als Platzhalter für andere Aussagen dienen. Der Wahrheitswert ist dann abhängig von konkreter Belegung der Variablen.

Boolesche Terme (Induktive Definition Boolescher Terme über einer abzählbaren Menge V von Variablen und einer Menge E von elementaren Aussagen.)

1. true, false sind Boolesche Terme;
2. Jedes $a \in E$ und jedes $x \in V$ ist Boolescher Term;
3. Ist t ein Boolescher Term, so auch $(\neg t)$;
4. $t_1, t_2 \in$Boolesche Terme, dann auch $(t_1 \vee t_2)$;
5. Minimalitätsprinzip : Es gibt keine Booleschen Terme, die sich nicht durch eine endliche Sequenz von Operationsanwendungen (0)...(3) erzeugen lassen.

Als Abkürzungen gebrauchen wir:

$$(t_1 \wedge t_2) =_{def} (\neg((\neg t_1) \vee (\neg t_2)))$$
$$(t_1 \Rightarrow t_2) =_{def} ((\neg t_1) \vee t_2)$$
$$(t_1 \Leftrightarrow t_2) =_{def} ((t_1 \wedge t_2) \vee ((\neg t_1) \wedge (\neg t_2)))$$

Boolesche Operatoren :

Dies sind $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Deren Bindungsstärke ist in dieser Reihenfolge von links nach rechts abnehmend, daher kann oft auch auf vollständige Klammerung verzichtet werden. Bei Operatoren gleicher Stärke, die von links nach rechts geklammert sind, können Klammern weggelassen werden. Auf eine äußere Klammerung kann ebenfalls verzichtet werden.

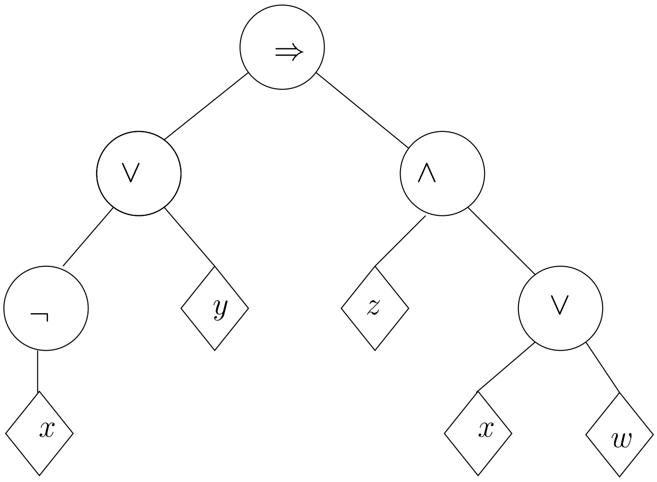
Beispiel :

(1) $x \wedge (\neg y \vee (z \Leftrightarrow w))$ *lässt sich nicht vereinfachen!*
(2) $(((x \wedge y) \vee (z \wedge (\neg v))) \Leftrightarrow (w \vee x))$
lässt sich vereinfachen zu : $x \wedge y \vee z \wedge \neg v \Leftrightarrow w \vee x$

An folgendem Beispiel wollen wir die einfache Konstruktion einer sogenannten Baumdarstellung eines Booleschen Terms zeigen

Beispiel :

$(((\neg x \vee y) \Rightarrow (z \wedge (x \vee w)))$
Hinweis : Substitution von Variablen durch andere Boolesche Terme (sehen wir aber später...)



Baumdarstellung für Term $t = (\neg x \vee y) \Rightarrow (z \wedge (x \vee w))$

Es ist wichtig zu verstehen, dass Boolesche Terme bis jetzt für uns nur abstrakte Konstrukte sind, für ihre Interpretation brauchen wir das Rechnen mit Wahrheitswerten.

2.2 Boolesche Algebra, Rechnen mit Wahrheitswerten

Boolesche Funktionen

Folgende Darstellungen sind für die beiden Wahrheitswerte in der Informatikliteratur gebräuchlich :

{falsch, wahr} = {false, true} = {L,O} = {0,1}

Wir benutzen $\mathbb{B}$ ={0,1} als Menge der beiden Booleschen Wahrheitswerte und können somit Boolesche Funktionen definieren.

Funktionstabelle mit $f : \mathbb{B} \text{ x } \mathbb{B} \to \mathbb{B}$ (und $f : \mathbb{B} \to \mathbb{B}$)

a	b	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Tabelle 2.1: 2-stellige Funktionen

Diese Funktionstabelle wird uns sehr behilflich sein. Sie beinhaltet alle Booleschen Funktionen, von bis zu zwei Booleschen Argumenten. Nun schauen wir uns mal an, welche Funktionen von f_0 ... f_{15} wir bezeichnen können.

f_0	„Kontradiktion„
f_1	and
f_2	not (a ⇒ b)
f_3	a
f_4	not (b⇒ a)
f_5	b
f_6	Antivalenz, xor
f_7	or
f_8	nor
f_9	Äquivalenz
f_{10}	not (b)
f_{11}	b ⇒ a
f_{12}	not (a)
f_{13}	a ⇒ b
f_{14}	nand
f_{15}	„Tautologie„

Tabelle 2.2: Nomenklatur 2-stelliger Boolescher Funktionen

Nun wollen wir zeigen, dass für alle b_1, $b_2 \in \mathbb{B}$ gilt :

$$\text{impl}(b_1, b_2) = \text{or }(\text{not}(b_1),\, b_2)$$

Den Beweis können wir anhand einer Wertetabelle führen.

b_1	b_2	impl (b_1 ,b_2)	not (b_1)	or(not (b_1), b_2)
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	0	1

Wir sehen Spalte drei und fünf tragen die gleichen Einträge, folgerichtig ist die obige Identität bewiesen.

Boolesche Terme und Wahrheitswerte

Jetzt können wir die Frage beantworten, „*Wie gehören die Definition Boolescher Terme und das Rechnen mit Wahrheitswerten eigentlich zusammen?*“.

Idee:

> Jede Belegung $\beta : V \rightarrow \{0,\, 1\}$ der in einem Booleschen Term vorkommenden Variablen mit Wahrheitswerten induziert eine Interpretation I_β des Booleschen Terms, das heißt die Zuordnung eines Wahrheitswertes. Die Interpretation folgt der induktiven Definition der Booleschen Terme.

I_β : Boolescher Term $\mapsto$ Wahrheitswert

$I_\beta\ (a)$ ist bereits definiert, $\forall$ Elementaraussagen $a \in E$

I_β (true) $=_{def}$ 1

I_β (false) $=_{def}$ 0

$I_\beta\ (x) =_{def} \beta(x)\ ,\ \forall\ x \in V$

$I_\beta\ (\neg t) =_{def}$ not $(I_\beta(t))$, $\forall\ t \in$Boolesche Terme

$I_\beta\ (t_1 \vee t_2) =_{def}$ or $(I_\beta(t_1),\ I_\beta(t_2))$

$I_\beta\ (t_1 \wedge t_2) =_{def}$ and $(I_\beta(t_1),\ I_\beta(t_2))$

$I_\beta\ (t_1 \Rightarrow t_2) =_{def}$ impl $(I_\beta(t_1),\ I_\beta(t_2))$

$I_\beta\ (t_1 \Leftrightarrow t_2) =_{def}$ equiv $(I_\beta(t_1),\ I_\beta(t_2))$

Semantische Äquivalenz Boolescher Terme Boolesche Terme t_1, t_2 heißen semantisch äquivalent($t_1 \equiv t_2$), falls für alle Belegungen $\beta : V \to \mathbb{B}$ gilt : $I_\beta(t_1) = I_\beta(t_2)$.

Jetzt kommt die nachträgliche Rechtfertigung, für die Definition der Operatoren $\Rightarrow$, $\wedge$, $\Leftrightarrow$ auf Ebene der Booleschen Terme:

Satz 2.2.1 *$I_\beta\ (t_1 \Rightarrow t_2) = I_\beta\ (\neg t_1 \vee t_2)$*

Beweis

$$\begin{aligned} I_\beta(t_1 \Rightarrow t_2) &=_{def} \text{impl}(I_\beta(t_1), I_\beta(t_2)) \\ &= \text{or}(\text{not}(I_\beta(t_1)), I_\beta(t_2)) \\ &= \text{or}(I_\beta(\neg t_1), I_\beta(t_2)) \\ &= I_\beta(\neg t_1 \wedge t_2) \end{aligned}$$

Also haben wir gezeigt : Jeder Boolesche Term ist zu einem anderen semantisch äquivalent, in dem nur die Operatoren true, false, $\neg$, $\vee$ vorkommen.

Einige semantische Äquivalenzen haben den Rang eines Gesetzes. Wir haben hier die wichtigsten zusammengestellt.

Um einen formalen Beweis für die Richtigkeit dieser Äquivalezen zu führen benutzt man eine Wertetabelle, bei kleineren Ausdrücken könnte das z.B. so aussehen.

Beispiel: $\neg(x \vee y) \equiv (\neg x \wedge \neg y)$

Ausdruck	$\equiv$	Ausdruck	Gesetz
$\neg\neg x$	$\equiv$	x	Involution
$x \wedge y$	$\equiv$	$y \wedge x$	Kommutativität
$x \vee y$	$\equiv$	$y \vee x$	Kommutativität
$x \wedge (y \vee z)$	$\equiv$	$(x \wedge y) \vee (x \wedge z)$	Distributivität
$x \vee (y \wedge z)$	$\equiv$	$(x \vee y) \wedge (x \vee z)$	Distributivität
$x \wedge x$	$\equiv$	x	Idempotenz
$x \vee x$	$\equiv$	x	Idempotenz
$x \wedge (x \vee y)$	$\equiv$	x	Absorption
$x \vee (x \wedge y)$	$\equiv$	x	Absorption
$\neg(x \wedge y)$	$\equiv$	$\neg x \vee \neg y$	de Morgan'sche Regel
$\neg(x \vee y)$	$\equiv$	$\neg x \wedge \neg y$	de Morgan'sche Regel
$x \vee (y \wedge \neg y)$	$\equiv$	x	Neutralität
$x \wedge (y \vee \neg y)$	$\equiv$	x	Neutralität

Tabelle 2.3: Boolesche Gesetze

$\beta(x)$	$\beta(y)$	$I_\beta(x \vee y)$	*not* $I_\beta(x \vee y)$	$I_\beta(\neg x)$	$I_\beta(\neg y)$	*or* $I_\beta(\neg x,\ \neg y)$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Die Spalten 4 und 7 tragen die gleichen Werte für gleiche Eingaben. Die entsprechenden Terme sind damit als semantisch äquivalent zu bezeichnen.

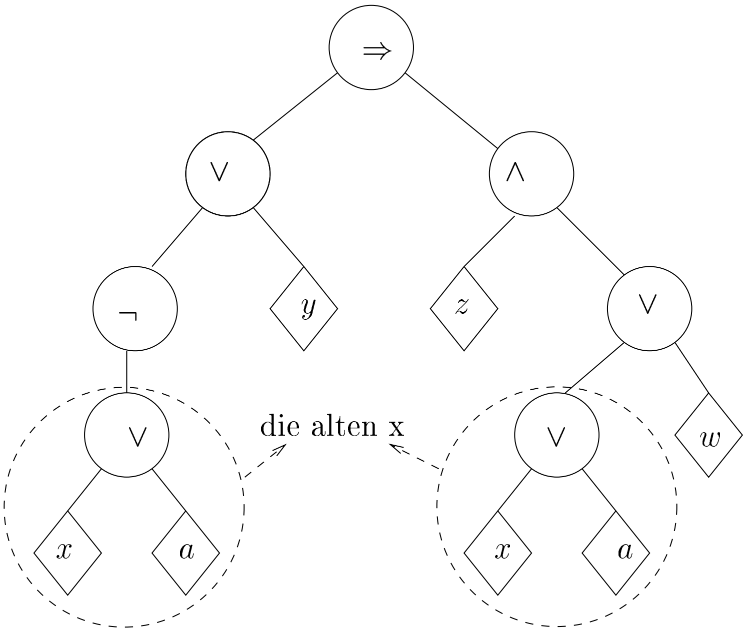
Nun kann man sich aber leicht vorstellen, dass das bei großen Termen sehr aufwendig ist (siehe Gesetzes-Tabelle). Durch das Konzept der **Substitution** kann man aber die Booleschen Gesetze auch in viel allgemeineren Situationen verwenden.

Dazu bezeichne $t[t_1/x]$ den Booleschen Term, der entsteht, wenn in einem Term t jedes Vorkommen der Variablen x durch den Term t_1 ersetzt wird.

Beispiel: $t = (\neg x \vee y) \Rightarrow ((z \wedge (x \vee w)$

nun ist $t[(x \vee a)/x] = (\neg(x \vee a) \vee y \Rightarrow (z \wedge ((x \vee a)) \vee w))$

Die Substitution kann man sich auch sehr gut in der Baumdarstellung des Ausdrucks verdeutlichen.



Die Substitution $t[x \vee a/x]$

Folgender Satz sollte einsichtig sein.

Satz 2.2.2 *Kongruenzeigenschaft der semantischen Äquivalenz bezüglich Substitution.*

Seien t_1, t_2 semantisch äquivalente Boolesche Terme und x sei eine Variable. Für beliebige Boolesche Terme t gilt

$$t[t_1/x] \equiv t[t_2/x]$$
$$t_1[t/x] \equiv t_2[t/x]$$

Erfüllbarkeit Boolescher Terme

Erfüllbarkeit Ein Boolescher Term t heißt *erfüllbar*, wenn es eine Belegung $\beta : V \to \mathbb{B}$ der Variablen gibt, mit $I_\beta(t) = 1$.

Tautologie t heißt *Tautologie* (oder auch *gültig*), wenn jede Belegung erfüllend ist.

Folgende Regeln gelten für eine gegebene Tautologie t und einen Booleschen Term x.

$$\begin{aligned} t \vee x &\equiv t \\ t \wedge x &\equiv x \end{aligned}$$

Kontradiktion t heißt *Kontradiktion(unerfüllbar)*, wenn keine Belegung erfüllend ist.

Folgende Regeln gelten für eine gegebene Kontradiktion k und einen Booleschen Term x.

$$\begin{aligned} k \vee x &\equiv x \\ k \wedge x &\equiv k \end{aligned}$$

Die Menge aller Booleschen Terme zeigt bezüglich der Eigenschaft *Tautologie/Kontradiktion* eine „Spiegeleigenschaft“ dargestellt in folgender Grafik :

Tautologien	erfüllbare aber nicht gültige Terme		Kontradiktionen
G	F	$\neg F$	$\neg G$

Die Menge der Booleschen Terme als Spiegelbild

Wir stellen uns nun die Frage : „*Wie überprüft man die Erfüllbarkeit eines Booleschen Terms?*“. Eine naive Lösung benutzt „**brute force**“, d.h. alle 2^n möglichen Belegungen der n Variablen werden einfach durchgetestet. Sehr realistisch scheint das nicht zu sein, was uns die folgende Überlegung für $n = 64$ zeigt.

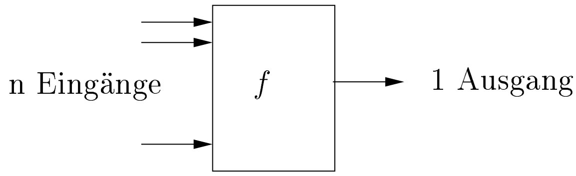
Beispiel: Wir nehmen an, dass das Testen einer Belegung 10^{-6} Sekunden in Anspruch nimmt. Da nun $2^{64} \approx 1.8 * 10^{19}$ ist, würde dieses Verfahren $\sim 5 * 10^5$ Jahre dauern, da ein Jahr $31.5 * 10^6$ Sekunden hat.

Ausblick Es ist sogar so, dass die *Erfüllbarkeit Boolescher Terme* ein Urtyp der sogenannten NP-schweren Probleme ist, bei denen es vermutlich keine effizienten algorithmischen Lösungen gibt, die wesentlich besser als „**brute force**“ sind.

2.3 Boolesche Funktionen

Der Begriff Boolesche Funktion

Wir führen den Begriff der n-stelligen Boolesche Funktion ein. Eine Boolesche Funktion f ist die konkrete Beschreibung eines Input-Output-Verhaltens bei n 0-1-Eingängen und einem 0-1-Ausgang.



Blackboxdarstellung einer n-stelligen Booleschen Funktion

Boolesche Funktion Eine Abbildung $f : \mathbb{B}^n \to \mathbb{B}$ heißt n-stellige Boolesche Funtion.

Im Folgenden wollen wir uns den Zusammenhang zwischen Booleschen Termen und Booleschen Funktion anschauen.

Sei t sei ein Boolescher Term mit Variablen $V = \{x_1, \dots, x_n\}$. t repräsentiert eine Boolesche Funktion f_t durch

$$(\beta : V \to \mathbb{B}) \mapsto (\beta(x_1), \beta(x_2), \dots, \beta(x_n)) \in \mathbb{B}^n$$
$$f_t(b_1, \dots, b_n) = I_\beta(t) \in \mathbb{B}$$

Also : Wir werten t bezüglich der Belegung der Variablen mit konkreten Wahrheitswerten $b_1, \dots, b_n$ aus.

Beispiel: $t = \neg x_1 \vee (x \wedge x_2)$

$$f_t : \begin{array}{ccc} (0,0) & \mapsto & 1 \\ (0,1) & \mapsto & 1 \\ (1,0) & \mapsto & 0 \\ (1,1) & \mapsto & 1 \end{array}$$

Wenn wir uns die Frage stellen, ob t der einzige Boolesche Term ist, der dieses konkreten Input-Output-Verhalten realisiert, dann sollten wir die Antwort (siehe vorhergehenden Abschnitt) kennen. **Nein, es gibt unendlich viele!**
Auf der einen Seite haben wir die (grau dargestellte unendliche) Menge der semantisch äquivalenten Terme, die alle ein und dasselbe konkrete f repräsentieren. Auf der anderen Seite die Menge der Booleschen Funktionen, bei denen es aber nur endlich viele verschiedene n-stellige gibt.

Satz 2.3.1 *Es gibt* 2^{2^n} *Boolesche Funktionen* $f : \mathbb{B}^n \mapsto \mathbb{B}$.

Beweis Dieser Beweis hat 2 Teile

1. $|\mathbb{B}^n| = 2^n$ wobei, allgemein X^n die Menge aller Tupel $(x_1, ..., x_n)$ mit $x_i \in X$ bezeichnet.

2. Wie viele Funktionen $g : X \to \{0, 1\}$ gibt es? Funktionen sind dadurch spezifiziert, welcher Wert auf den einzelnen $x \in X$ angenommen wird.

   Lemma 1 *Es gibt* $2^{|X|}$ *verschiedene Funktionen mit* $x \to \{0, 1\}$.

   (Beweis : Induktion über $|X|$*)*

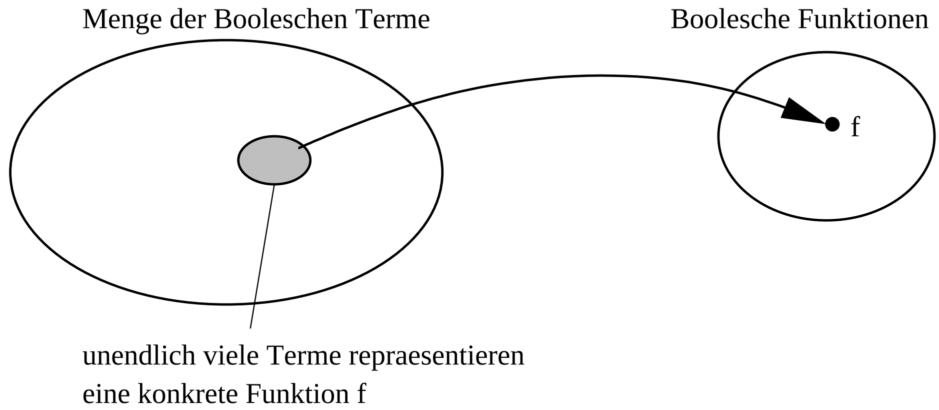


Abbildung 2.1: Mengen Boolescher Terme und Boolescher Funktionen

Vollständige Basen

Es gibt Teilmengen der Boolesche Operatoren, mit deren Hilfe man alle anderen Operatoren semantisch äquivalent darstellen kann. Man nennt solche Mengen **vollständige Basen**.

Beispiele:

1. Nach unserer Definition der Booleschen Terme sind $\{\neg, \vee\}$ vollständig, ebenso $\{\neg, \wedge\}$. Dies lässt sich leicht durch die „de Morgan"-Regel zeigen.

2. Der NAND-Operator, definiert durch $NAND(x, y) = \neg(x \wedge y)$ ist allein schon vollständige Basis.

   Beweis *:*

   (a) $\neg x \equiv NAND(x, x)$

   (b) z.z., dass $NAND(NAND(x, y),\ NAND(x, y)) \equiv x \wedge y$

x	y	$NAND(x,y)$	$NAND(NAND(x,y),\ NAND(x,y))$	$x \wedge y$
0	0	1	0	0
0	1	1	0	0
1	0	1	0	0
1	1	0	1	1

   Wir haben gezeigt, dass sich $\{\neg, \wedge\}$ durch Kombinationen von $\{NAND\}$ ausdrücken lässt. Weil sich jede Boolesche Funktion durch Anwendungen der Operatoren aus $\{\neg, \wedge\}$ realisieren läßt, gilt das damit auch für $\{NAND\}$.

3. Der NOR-Operator, definiert durch $NOR(x, y) = \neg(x \vee y)$ ist vollständig. Der Beweis lässt sich analog zu (2) führen.

Wenn man NAND-Bauteile (bzw. NOR-Bauteile) hat, weil sie vielleicht auf dem Computerteilemarkt besonders günstig sind, dann lassen sich daraus **alle** anderen Booleschen Funktionen realisieren!

2.4 Disjunktive (DNF) und Konjunktive (KNF) Normalform

Kann man jede Boolesche Funktion durch Boolesche Terme repräsentieren? Ja! Und oft wird für diese Repräsentationen die DNF oder KNF benutzt.

Literal Ein Literal ist eine Variable oder deren Negation.

Disjunktive Normalform Eine Disjunktion $t_1 \vee ... \vee t_n$ von Booleschen Termen heißt Disjunktive Normalform (DNF), wenn jeder Term t_i eine Konjunktion von Literalen ist (oder auch ein einzelnes Literal).

Konjunktive Normalform Eine Konjunktion $t_1 \wedge ... \wedge t_n$ von Booleschen Termen heißt Konjunktive Normalform (KNF), wenn jeder Term t_i eine Disjunktion von Literalen ist (oder auch ein einzelnes Literal).

Beispiele:

1. $(x \wedge y \wedge \neg x) \vee (y \wedge z) \vee (z \wedge x)$ ist DNF
2. $(x \vee z) \wedge \neg y \wedge (y \vee x) \wedge w$ ist KNF
3. $(\neg x \wedge z \wedge y)$ ist sowohl DNF, als auch KNF!

Satz 2.4.1 *Jede Boolesche Funktion $f : \mathbb{B}^n \to \mathbb{B}$ ist durch folgende kanonische Normalform repräsentiert.*

(1) Kanonische Disjunktive Normalform :

$$dnf(f) := \bigvee_{(b_1,\ldots,b_n)\in f^{-1}(1)} (x_1^{b_1} \wedge x_2^{b_2} \wedge ... \wedge x_n^{b_n}) \quad , \; mit\; x_i^{b_i} = \begin{cases} x_i \,, wenn\; b_i=1 \\ \neg x_i \,, sonst \end{cases}$$

$$\text{für } f^{-1}(1) = \emptyset \;,\; sei\; dnf(f) = false$$

(2) Kanonische Konjunktive Normalform :

$$knf(f) := \bigwedge_{(b_1,\ldots,b_n)\in f^{-1}(0)} (x_1^{\neg b_1} \vee x_2^{\neg b_2} \vee ... \vee x_n^{\neg b_n}) \quad , \; mit\; x_i^{b_i} = \begin{cases} x_i \,, wenn\; b_i=1 \\ \neg x_i \,, sonst \end{cases}$$

$$\text{für } f^{-1}(0) = \emptyset \;,\; sei\; knf(f) = true$$

Beweis (für dnf(f)):

Wenn $(b_1, ..., b_n) \in f^{-1}(1)$, dann ist

$$\mathrm{dnf}(f) = ... \vee (x_1^{b_1} \wedge x_2^{b_2} \wedge ... \wedge x_n^{b_n}) \vee ... .$$

Dies wird zu 1 ausgewertet, also auch dnf(f). Wenn dnf(f) zu 1 ausgewertet wird, dann muss wenigstens eine Konjunktion (man sagt auch Minterm) zu 1 ausgewertet werden.

Beweis (für knf(f)):

Wenn $(b_1, ..., b_n) \in f^{-1}(0)$, dann enthält

$$\mathrm{knf}(f) = ... \wedge x_1^{\neg b_1} \vee x_2^{\neg b_2} \vee ... \vee x_n^{\neg b_n} \wedge ... .$$

Diese Disjunktion wird zu 0 ausgewertet, also auch die gesamte knf(f).

Ein alternativer Beweis der Aussage:

„Jeder Boolesche Term t ist semantisch äquivalent zu einem Term in DNF.“

Beweis: durch strukturelle Induktion der induktiven Definition der Booleschen Terme folgend

Zu zeigen ist :

$t \equiv (x_{11} \wedge x_{12} \wedge ... \wedge x_{1m_1}) \vee (x_{21} \wedge x_{22} \wedge ... \wedge x_{2m_2}) \vee ... \vee (x_{k1} \wedge x_{k2} \wedge ... \wedge x_{km_k})$

für ein $k \geq 1$ und $m_1, ..., m_k \geq 1$

oder kürzer geschrieben :

$$t \equiv \bigvee_{i=1}^{k} (\bigwedge_{j=1}^{m_i} x_{ij})$$

wobei alle x_{ij} Literale sind.

1) Falls $t = x$ oder $t = \neg x$ für eine Variable x, so ist t in DNF.

2) $t = \neg t'$ und t' schon in DNF $t' \equiv \bigvee(\bigwedge x_{ij})$

Durch Anwendung der Booleschen Gesetze erhält man :

$$t \equiv \neg(\bigvee(\bigwedge x_{ij})) \equiv \bigwedge(\neg(\bigwedge x_{ij})) \equiv \bigwedge(\bigvee(\neg x_{ij})) \equiv \bigwedge(\bigvee x'_{ij})$$

$x'_{ij} \equiv \neg x_{ij}$ (Literale)

$+$ Anwendung des Distributivgesetzes

3) $t = t_1 \vee t_2$, t_1 und t_2 in DNF, klar ...

Vereinfachung von DNF

Regeln:

(1) Treten in DNF Teilterme der Form

$$... \vee (y_1 \wedge ... \wedge y_{i-1} \wedge x_i \wedge y_{i+1} \wedge ... \wedge y_l) \vee ...$$
$$... \vee (y_1 \wedge ... \wedge y_{i-1} \wedge \neg x_i \wedge y_{i+1} \wedge ... \wedge y_l) \vee ...$$

auf, so werden diese zusammengefasst zu

$$... \vee (y_1 \wedge ... \wedge y_{i-1} \wedge y_{i+1} \wedge ... \wedge y_l) \vee ...$$

(2) Für folgenden Term

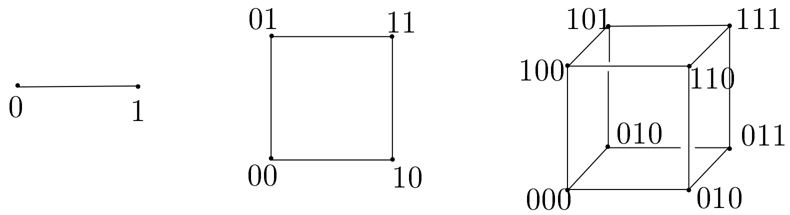
$$... \vee (y_1 \wedge y_2) \vee ... \vee (y_2)$$

schreiben wir vereinfacht

$$... \vee (y_2) \vee ...$$

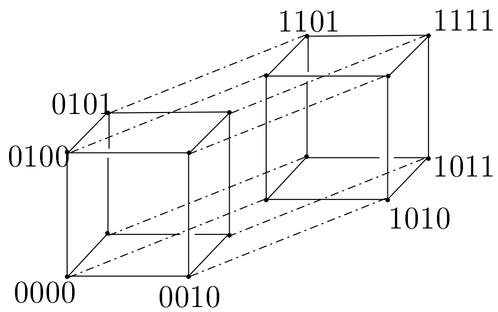
Geometrische / Grafische Interpretation der DNF

Wir interpretieren n-Tupel über $\mathbb{B}$, also $\mathbb{B}^n$, als Ecken des n-dimensionalen Hyperwürfels.



$n = 1$ $n = 2$ $n = 3$

Merke:der n–dimensionale Würfel entsteht
aus zwei Kopien $n-1$–dimensionaler Würfel,
indem sich entsprechende Knoten jeweils durch eine
neue Kante verbunden werden

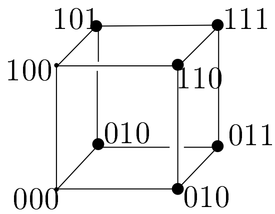


$n = 4$

Interpretation von 0/1–Tupeln als Knoten in n–dimensionalen Würfeln

Beispiel:

geg.: $f : \mathbb{B}^n \mapsto \mathbb{B}$, n=3



Tupel im Urbild $f^{-1}(1)$ sind markiert

kanonische DNF mittels Wertetabelle :

$$\text{dnf}(f) = (x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee ...$$

6 Terme, jeder Knoten (der einer zu 1 ausgewerteten Belegung der Variablen entspricht) wird einzeln aufgeführt!

Wie kann ich diese Menge erfüllbarer Belegungen alternativ beschreiben?

$x_1 \vee (\neg x_1 \wedge x_2) \equiv x_2 \vee (x_1 \wedge \neg x_2) \equiv x_1 \vee x_2$

Wichtig : *Größen der Darstellung und Anzahl der verschieden verwendeten Booleschen Operatoren sind entscheidende Faktoren für Effizienz der physikalischen Realisierung der Booleschen Funktion.*

2.5 Rechenstrukturen

Algorithmen arbeiten auf Daten, die aus bestimmten „Trägermengen“ stammen. Für die Formulierung der Algorithmen sind die verfügbaren Funktionen entscheidend.

Rechenstruktur = Trägermengen + Funktionen

Problem :

Oft sind Funktionen nicht vollständig definiert, sondern nur partiell.

Sei $f : M_1 \times M_2 \times ... \times M_n \to M_{n+1}$ n-stellige Funktion

Trick :

Erweiterung zu totaler Funktion mittels $M^\perp = M \bigcup \{\perp\}$, wobei $\perp$(„bottom“) nichtdefinierte Ereignisse repräsentiert.

$$f : M_1^\perp \times M_2^\perp \times ... \times M_n^\perp \mapsto M_{n+1}^\perp$$

Falls eines der Argumente „bottom“ ist, dann auch das Ergebnis.

Beispiel :

Rechenstruktur $\mathbb{N}$ der natürlichen Zahlen.

Trägermengen („Sorten") = { $\mathbb{N}^{\perp}$, $\mathbb{B}^{\perp}$ }

Funktionssymbole : $F = \{true, false, \neg, \vee, \wedge, zero, pred, add, mult, sub, div, \leq, =_?\}$

Symbol		Spezifikation
$true$, $false$	:	$\to \mathbb{B}^{\perp}$
$\neg$	:	$\mathbb{B}^{\perp} \to \mathbb{B}^{\perp}$
$\vee, \wedge$	:	$\mathbb{B}^{\perp} \times \mathbb{B}^{\perp} \to \mathbb{B}^{\perp}$
$zero$	:	$\to \mathbb{N}^{\perp}$
$succ, pred$	:	$\mathbb{N}^{\perp} \to \mathbb{N}^{\perp}$
$add, mult, sub, div$	:	$\mathbb{N}^{\perp} \times \mathbb{N}^{\perp} \to \mathbb{N}^{\perp}$
$\leq, =_?$	:	$\mathbb{N}^{\perp} \times \mathbb{N}^{\perp} \to \mathbb{B}^{\perp}$

Tabelle 2.4: Beispiel, Rechenstruktur : natürliche Zahlen

Insbesondere ist aber :

$pred(0) = \perp$ und $div(x, 0) = \perp$

Wie bei Booleschen Termen gibt es, auch hier Variablen und Termbildung ist möglich (Baumdarstellung). Die Interpretation der Terme durch eine Belegung der Variablen ist ebenfalls gegeben. Wir können auch hier Termersetzungen machen.

Beispiel:

$succ(zero) \leq (add(succ(succ(zero)), pred(succ(zero)))$

Wie funktioniert die Termauswertung bei $\perp$?

Beispiel:

$add(succ(zero), succ(pred(zero)))$, wenn $pred(zero) = \perp$, dann auch $succ(pred(zero)) = \perp$ und damit der gesamte Ausdruck.

Wir betrachten nun Terme über $\mathbb{N}$, die zu Werten in $\mathbb{B}$ ausgewertet werden und Variable für Parameter aus $\mathbb{N}$ enthält.

Beispiel:

$((x+2) \leq 5)$ ist ein Prädikat $\mathrm{P}(x)$ für x die Variable

$((x*y) \leq (z+10)$ ist Prädikat $\mathrm{Q}(x,y,z)$ für Variablen x,y,z

Durch eine Belegung ergibt sich für konkrete Parameter ein Wahrheitswert

$\mathrm{P}(3) = 1$ (wahre Aussage)

$\mathrm{Q}(3,4,1) = 0$ (falsche Aussage)

Wir können Prädikate danach unterscheiden, ob sie für alle Belegungen wahre Aussagen ergeben, oder ob es mindestens eine Belegung gibt, die zu 1 ausgewertet wird.

Die Aussageformen:

- Für alle $x \in X$ gilt $\mathrm{P}(x)$. („P ist allgemeingültig über der Grundmenge X“)
- Es gibt mindestens ein $x \in X$, für das $\mathrm{P}(x)$ gilt. („P ist erfüllbar über der Grundmenge X“)

Schreibweise:

$\forall x \in X : \mathrm{P}(x)$, dabei ist $\forall$ der sogenannte Allquantor

$\exists x \in X : \mathrm{P}(x)$, dabei ist $\exists$ der sogenannte Existenzquantor

Beispiel:

$(\forall x, y \in \mathbb{N} : x * y \leq 100) = 0$

$(\forall x, y \in \{1,2,3\} : x * y \leq 10) = 1$

$(\forall x \in \mathbb{N}\ \exists y \in \mathbb{N} : x + y \leq 100) = 0$

...

Man baut sich induktiv die Menge der syntaktisch korrekten Formeln der „Prädikatenlogik erster Stufe“, diese haben : Variable, Funktionssymbole, Prädikatssymbole, Quantoren für Variable.

In Tabelle 2.5 sind wichtige semantischen Äquivalenzen aufgeführt.

$\neg\forall x \in X : P(x)$	$\equiv$	$\exists x \in X : \neg P(x)$
$\neg\exists x \in X : P(x)$	$\equiv$	$\forall x \in X : \neg P(x)$
$(\forall x \in X : P(x)) \wedge (\forall x \in X : Q(x))$	$\equiv$	$\forall x \in X : (P(x) \wedge Q(x))$
$(\exists x \in X : P(x)) \vee (\exists x \in X : Q(x))$	$\equiv$	$\exists x \in X : (P(x) \vee Q(x))$
$\forall x \in X, \forall y \in Y : P(x,y)$	$\equiv$	$\forall y \in Y, \forall x \in X : P(x,y)$
$\exists x \in X, \forall y \in Y : P(x,y)$	$\equiv$	$\exists y \in Y, \exists x \in X : P(x,y)$

Tabelle 2.5: Rechnen mit Quantoren

ABER: Im allgemeinen nicht äquivalent

1. $(\forall x \in X : P(x)) \vee (\forall x \in X : Q(x)) \not\equiv \forall x \in X : (P(x) \vee Q(x))$

   Beispiel:

   $X = \mathbb{N}$
   P(x) „ist gerade natürliche Zahl“
   Q(x) „ist ungerade natürliche Zahl“

   Ebenso sind im allgemeinen die folgenden Aussagen nicht äquivalent :

2. $\exists x \in X, \forall y \in Y : P(x, y) \not\equiv \forall x \in X, \exists y \in Y : P(x, y)$

   Beispiel:

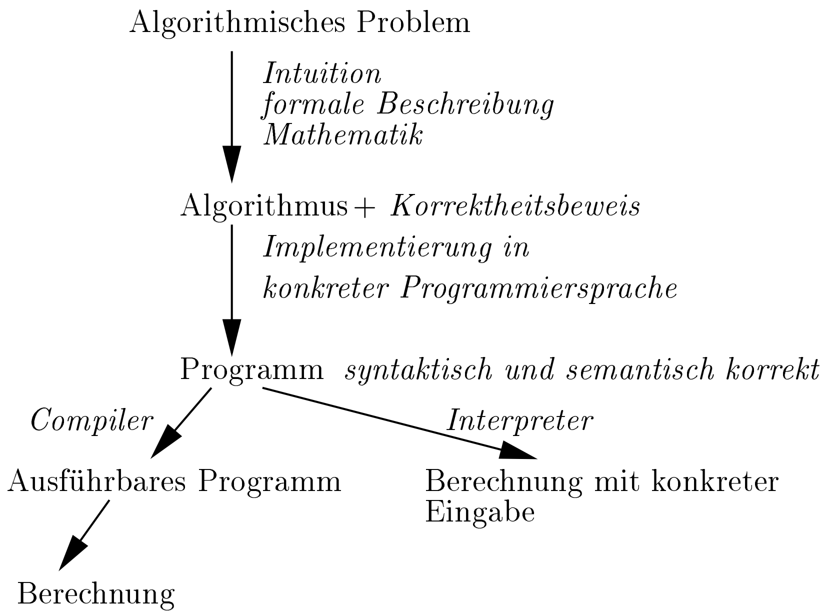
   $X = Y = \mathbb{Z}$
   $P(x, y) = (x + y = 1)$

Kapitel 3

Funktionale Programmierung in Haskell

3.1 Grundbegriffe zu Algorithmik und Programmierung

Wie geht man an algorithmische Probleme heran?



Algorithmisches Problem Für jede mögliche Eingabeinstanz aus einem vorgebenem Bereich, soll eine bestimmte Ausgabe (Antwort, Ergebnis) berechnet werden, die bestimmte, festgelegte Eigenschaften haben soll.

Beispiele:

1. Folge von Elementen aus Grundmenge mit Ordnungsrelation $\mapsto$ geordnete Folge
2. Text T und Wort P über Alphabet $\sum$ $\mapsto$ alle Vorkommen von P in T auflisten

3. Start s , Ziel t im Eisenbahnnetz $\mapsto$ kürzester (billigster) Weg von s nach t

Algorithmus Ist ein Verfahren mit einer präzisen (d.h. in einer eindeutigen Sprache abgefassten) endlichen Beschreibung unter Verwendung tatsächlich ausführbarer Operationen in einer Rechenstruktur zur Lösung eines algorithmischen Problems (es kann sehr viele Algorithmen geben, die dasselbe Problem lösen).

Historie (Auszug):

300 v. Chr. Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers, zweier natürlicher Zahlen.

800 n. Chr. Persisch-arabischer Mathematiker, Muhammed ibu Musa abu Djafar alchoresmi, veröffentlicht eine Aufgabensammlung für Kaufleute und Testamentsvollstrecker. Diese wurde als „Liber Algorithmi“ ins Lateinische übersetzt (Algorithmus ist Kunstwort aus diesem Namen und arithmos (griechisch: Zahl)).

1574 Adam Rieses Rechenbuch

1614 erste Logarithmentafel (30 Jahre lang wurde daran gerechnet)

1703 Binärzahlen von Leibniz eingeführt

1822-1833 Charles Babbage, Difference Engine

1931 Gödels Unvollständigkeitssatz, beendet Traum vieler Mathematiker, die Beweisführung aller Sätze in der Mathematik mit algorithmisch, konstruierten Beweisen zu bewerkstelligen

1936 Churche These, vereinheitlicht Welt der verschiedenen Sprachen der Notation von Algorithmen, indem für viele der damaligen Sprachen die gleiche Ausdrucksfähigkeit nachgewiesen wurde, dies als „Berechenbarkeit“ postuliert wird.

ab 1940 erster programmierbarer Computer (von Zuse, „Z1“)

Ein intuitiver Algorithmenbegriff begegnet uns auch im täglichen Leben in der Form von Bedienungsanleitungen, Bauanleitungen, Kochrezepten, ...
Es gibt noch weitere wichtige Begriffe, die in Zusammenhang mit Algorithmen verwendet werden.

terminierend : Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten ein Ergebnis liefert.

determinierter Ablauf, determiniertes Ergebnis

randomisierter Ablauf : Das Ergebnis ist determiniert, aber der Algorithmus wählt (einige) Einzelschritte während der Abarbeitung zufällig aus.

Bausteine für Algorithmen:

- Elementaroperationen
- sequentielle / parallele Ausführung
- bedingte Ausführung
- Schleifen / Iterationen
- Aufruf von Unterprogrammen
- Rekursion
- (Darstellung in PseudoCode)

Programm:

Darstellung eines Algorithmus entsprechend den *syntaktischen* Regeln einer Programmiersprache. (Hinweis : formale Sprachen)

Compilierung:

Umwandlung des Programms in ein Programm in der entsprechenden Maschinensprache.

Berechnung / Ausführung:

Compiliertes Programm wird mit konkreter Eingabe gestartet und abgearbeitet.

Interpreter:

Übersetzt Programm Schritt für Schritt und führt dieses für konkrete Eingabe sofort aus.

Ein Interpreter für die funktionale Programmiersprache *Haskell* ist *Hugs*. Der Name Haskell erinnert an den Mitbegründer des λ - Kalküls (Lambda-...) Haskell B. Curry (1900-1982).

3.2 Einführung : Programmieren mit Haskell

—- hier Online Einführung von Haskell

Wir haben nun einige Beispiele für *primitive Rekursion* kennengelernt (fac(n), sumInt, ...)

Die Auswertung einer Funktion für Paramter n wird auf Auswertung eines kleineren Parameters zurückgeführt. Was passiert intern?

Beispiel:

Die folgende Funktion realisiert die Fakultätsberechnung:

```
fac :: Int -> Int
fac n
   | n==0      = 1
   | otherwise = n * fac (n-1)
```

Nun schauen wir uns an, wie Hugs intern z.B. **fac 4** berechnet :

```
fac 4
-> 4 * fac 3
-> 4 * 3 * fac 2
-> 4 * 3 * 2 * fac 1
-> 4 * 3 * 2 * 1 * fac 0
-> 4 * 3 * 2 * 1 * 1
```

nun berechnet Hugs die Multiplikationen von rechts nach links

```
-> 24
```

... und liefert diesen Wert zurück.

Geometrisches Funktionsbeispiel:

Problem: In wieviele Regionen maximal, kann die Ebene durch n Geraden zerteilt werden?

Vorüberlegung: Um einen Maximalwert zu erhalten, sollten keine 3 Geraden durch einen Punkt verlaufen. Es sollte auch keine Parallelen geben.

```
region :: Int -> Int
region n
| n==0      = 1
| n>0       = region (n-1) + n
| otherwise = error ,, ... ‘‘
```

Warum liefert diese Funktion eine Lösung für unsere Problemstellung?

Die n-te Gerade wird als „Strahl“ durch die $n-1$ Geraden „geschossen“; immer, wenn der Strahl eine Gerade trifft, werden aus einer existierenden Region 2 (eine neue ...); die Region „nach“ der letzten Geraden wird in 2 geteilt.

```
region (n-1) + (n-1) + 1
```

als geschlossener Ausdruck :

$$\frac{n*(n+1)}{2} + 1$$

Den Beweis dazu kann man mittels vollständiger Induktion führen.

3.3 Grundlegende Haskellsyntax und Standardtypen in Hakell

Funktionen und **Variablen** beginnen mit einem Kleinbuchstaben, **Module** und **Typnamen** mitgegen mit Großbuchstaben. Es gibt reservierte Wörter :

> case, class, data, deriving, do, else, if, import, in, infix, instance, module, newtype, of, then, type, where

Es gibt eine Einrückregel, die zur Abgrenzung der Reichweite einer Funktionsdefinition dient.

```
funktion ....
.alles was
.hier steht
.gehört zur
.funktion
```

3.3.1 Einfache Datentypen

Boolesche Werte **Bool**

Werte: True, False

Operationen: not (Negation), && (Konjunktion), || (Disjunktion)

Beispiel (/=) :

```
exOr :: Bool -> Bool -> Bool
exOr x y
| (x&&y) || (not x && not y) = False
| otherwise                  = True
```

Ganze Zahlen **Int**

Werte: $-(2^{31}-1), ..., 0, ..., 2^{31}-1$ ($2^{31}-1$ entspricht 2.147.483.647)

Operatoren (Infixschreibweise): $+$, $*$, $-$, ^

Operatoren (Präfixschreibweise): div, mod, abs, negate

(die zweistelligen Operatoren in Infix : a 'div' b)

Vergleichsoperatoren: $>$, $>=$, $==$, $/=$, $<=$, $<$ :: Int -> Int -> Bool

Hinweis : Beim Rechnen mit **Int** wird intern nicht nachgeprüft, ob Bereich verlassen wird (Fehlerquelle!).

Gleitkommazahlen **Float**

Operatoren (Infixschreibweise): + , − , ∗ , /, ^ , ∗∗

Operatoren (Präfixschreibweise): abs, sin, cos, exp, log, sqrt, negate

(ceiling, floor :: Float -> Int)

Vergleichsoperatoren: > , >= , == , / = , <= , < :: Float -> Float -> Bool

Beispiel:

```
signum :: Float -> Float
signum x
| x<0       = -1.0
| x=0       =  0.0
| otherwise =  1.0
```

3.3.2 Rekursionsformen

Wir unterscheiden folgende allgemeine Formen der Rekursion

Rekursion: Funktion, die sich in ihrer Definition selbst aufruft

Primitive Rekursion: $f(0)$ ist definiert und $f(n)$ ruft (nur!) $f(n-1)$ auf

Lineare Rekursion: $f(0)$ ist definiert und $f(n)$ ruft nur ein $f(k)$ für ein $k < n$ auf

Nichtlineare (kaskadenartige) Rekursion: Anfangswert definiert, $f(n)$ kann mehrere $f(k)$ mit $k < n$ aufrufen

Beispiel (Fibonacci-Zahlen):

Fibonacci (Leonardo von Pisa) und seine Hasen ...

- ein Hasenpaar wird auf einer Insel ausgesetzt
- ein Jahr bis zur Geschlechtsreife
- zeugt am Ende des zweiten und jeden weiteren Lebensjahres Nachwuchs (ein Paar)

die Entwicklung der Größe der Hasenpopulation lässt sich wie folgt beschreiben :

$$f_0 = 0 \ , \ f_1 = 1 \ , \ f_2 = 1 \ , \ f_3 = 2 \ , \ f_4 = 3, \ f_5 = 5, \ ...$$

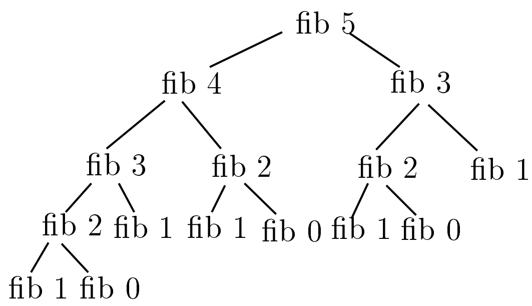
,daraus läßt sich als allgemein Rekursionsvorschrift für $n \geq 2$ ableiten :

$$f_n = f_{n-1} + f_{n-2}$$

Das wollen wir nun in Haskell formulieren :

```
fib :: Int -> Int
fib n
| n==0      = 0
| n==1       = 1
| otherwise = fib (n-1) + fib (n-2)
```

Um die Laufzeit abzuschätzen, betrachten wir den entstehenden Rekursionsbaum, der die Anzahl der Aufrufe darstellt.



Rekursionsbaum bei der Berechnung von fib 5

k_n sei die Anzahl der Knoten im Baum von fib n. Es gilt :

$$k_0 = k_1 = 1;\ k_2 = 3\ ;\ k_n = 1 + k_{n-1} + k_{n-2}$$

Behauptung: $k_n \geq f_{n+1}$

Beweis mit vollständiger Induktion.

Anker: richtig für k_0 , k_1

$$k_n = 1 + k_{n-1} + k_{n-2} \geq 1 + f_n + f_{n-1} \geq f_{n+1}$$

Behauptung: $k_n \leq f_{n+3}$ - 1

Wieder mit vollst. Induktion.

Anker: richtig für k_0 , k_1

$$k_n = 1 + k_{n-1} + k_{n-2} \leq 1 + (f_{n+2} - 1) + (f_{n+1} - 1) \leq f_{n+3} - 1$$

Damit haben wir den folgenden Satz bewiesen.

Satz 3.3.1 *Die Rekursionskosten für die Berechnung der Fibonacci-Zahlen mittels der Funktion fib, kann man durch*

$$f_{n+1} \leq k_n \leq f_{n+3} - 1$$

abschätzen.

Nun wollen wir aber noch eine effiziente Lösung zur Berechnung der Fibonacci-Zahlen kennenlernen. Die Laufzeit dieser Berechnungsvariante ist linear.

```
fibPair n = (fib n, fib (n+1))
```

Nun „schieben“ wir dieses Paar über die Folge der Fibonacci-Zahlen

```
fibStep :: (Int, Int) -> (Int, Int)
fibStep (u,v) = (v, u+v)
```

Damit können wir fibPair definieren und zwar rekursiv :

```
fibPair :: Int -> (Int, Int)
fibPair n
| n==0    = (0,1)
| otherwise = fibStep (fibPair (n-1))
```

nun beschreiben wir noch die eigentliche Fibonacci-Zahl

```
fastFib = fst.fibPair
```

Abschliessend wollen wir noch die Funktionsaufrufe betrachten und die Laufzeit abschätzen.

```
fastFib 4
-> fst.fibStep (fibPair 3)
-> fst.fibStep (fibStep (fibPair 2))
-> fst.fibStep (fibStep (fibStep (fibPair 1)))
-> fst.fibStep (fibStep (fibStep (fibStep (fibPair 0))))
-> fst.fibStep (fibStep (fibStep (fibStep (0,1))))
-> fst.fibStep (fibStep (fibStep (1,1)))
-> fst.fibStep (fibStep (1,2))
-> fst.fibStep (2,3)
-> fst.(3,5)
```

und wir erhalten als Ergebnis

```
-> 3
```

Da wir nun eine Rekursionstiefe von n haben und pro Tiefenrückschritt jeweils eine Aktion ausführen, benötigt diese Funktion $\approx 2n$ Funktionsaufrufe. Da das Wachstum der Funktion $2n$ sich nur um einen konstanten Faktor von n unterscheidet, spricht man von *linearer* Laufzeit.

3.3.3 Tupel und Listen

Beispiel (Rauminventarliste):

ersteinmal werden wir uns geeignete Typen definieren

```
type Object   = String
type Index    = Int
type Room     = Int
type Entry    = (Index, Room, Object)
type Database = [Entry]
```

Die Funktion equipment, liefert uns die Objekte zu den jeweiligen Räumen

```
equipment :: Database -> Room -> [Object]
equipment db room = [o|(i,r,o)<-db, r==room]
```

(i,r,o)<-db entspricht : gehe elementenweise durch die Liste db (von vorn nach hinten) und teste, ob das Prädikat r==room erfüllt ist. Sollte es erfüllt sein, so füge das Object o in die entstehende Liste ein.

(Das ist kein vollständiges Programm, aber verdeutlicht das Prinzip Tupel und Liste)

3.3.4 Monomorphe / Bimorphe Listenfunktionen

Beispiele:

```
and, or :: [Bool] -> Bool
sum     :: [Int] -> Int
product :: [Float] -> Float
```

3.3.5 Polymorphe Funktionen

Im Gegensatz zu Funktionen, die sich auf einen Datentyp spezialisieren, wie z.B.

```
negate :: Bool -> Bool
```

können wir polymorphe Funktionen definieren, die uns ein größeres Abstraktionsniveau gestatten. Wir müssen also beispielsweise die length-Funktion für Listen nicht für jeden Datenyp schreiben

```
length :: [Bool] -> Int
length :: [Int]  -> Int
...
```

es reicht aus, einen Platzhalter für diese Datentypen zu definieren :

```
length :: [a] -> Int
```

Man sollte beachten, dass die Typvariable a mit einem kleinen Buchstaben beginnen sollte. Ein weiteres Beispiel (a und b können beliebige Typen sein) :

```
fst :: (a, b) -> a
```

***Tipp*:** Falls der allgemeinste Typ einer Funktion gesucht ist, kann man in Hugs „:type“ eingeben.

Die Typangabe in Haskell nicht nicht unbedingt notwendig, da die Typauswertung nach dem Aufruf zur Laufzeit statt findet.

Vorteile: kompakte Programme, wiederverwendbare Funktionen

Nachteile: Verzicht auf Kontrollmechanismen, Fehler bei Laufzeit an unerwarteter Stelle

Beispiel (fib ohne Typvereinbarung)**:**

```
fib 4.0
-> 3
fib (4.2-2.2)
-> 1
fib (4.1-2.2)
-> error ...
```

3.3.6 Konzept des Overloading

Es gibt Operatoren, die auf verschiedene Typen anwendbar sind.

Beispiele:

```
<= gibt es für Int, Float, Char, ...
+  für Int, Float
```

Die Typauswertung findet zur Laufzeit statt, im Unterschied zur Polymorphie können überladene Operatoren verschiedene Algorithmen aufrufen.

Beispiel:

```
== bei Int wird direkt auf Darstellung ausgewertet
== bei Float vorher Umwandlung in Standardform
```

Solche Operatoren können für eigene Typen auch neu definiert werden.

3.4 Realisierung rekursiver Listenfunktionen

Um Listen rekursiv verarbeiten (oder aufbauen) zu können, müssen wir zunächst den kleinsten Fall, die **leere Liste**, betrachten (leere Liste = [])

Eine **nichtleere Liste** aus [t] („gefüllt“ mit Objekten des Typs t) zerfällt in *head* vom Typ t und *tail* vom Typ [t]. Um die ganze Liste als Muster zu beschreiben, verwenden wir z.B. **x:xs**, wobei x das erste Element der Liste und xs den Rest (ohne das erste Element) der Liste repräsentiert.

Jede Liste kann eindeutig mittels Operators „:“ gewonnen werden (dieser ist rechtsassoziativ).

Beispiel:

```
[3,1,4] = 3:(1:(4:[]))
```

Rekursive Funktionen, die Listen verarbeiten, werden mittels dieser Muster beschrieben. Dabei rufen sie sich selbst auf, aber mit kürzerem

Beispiele:

(1) Testen, ob eine Zahl x in der Liste list vorkommt

```
bisher :
   isElem :: Int -> [Int] -> Bool
   isElem x list   = ([y | y<-list, y==x]/=[])
rekursiv :
   isElem :: Int -> [Int] -> Bool
   isElem x []     = False
   isElem x (y:ys) = (x==y) || isElem x ys
```

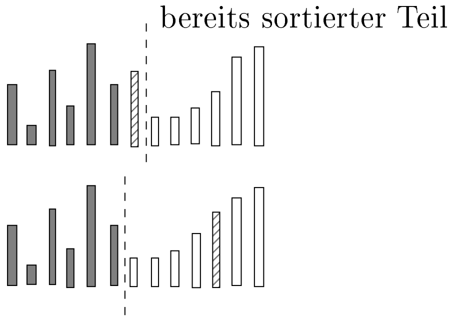
(2) Insertionsort, „Einfügen in den sortierten Rest“

Kurzbeschreibung :

Nimm head der Liste heraus und und füge es in den Rest der Liste ein, welcher zuvor rekursiv sortiert wird. Die leere Liste ist schon sortiert.

```
   insertSort :: [Int] -> [Int]
   insertSort []     = []
   insertSort (x:xs) = insert x (insertSort xs)
      where
         insert :: Int -> [Int] -> [Int]
         insert x []     = [x]
         insert x (y:ys)
            | x<=y      = x:(y:ys)
            | otherwise = y:(insert x ys)
```

Visualisierung:



Insert fügt nächstes Element in bereits sortierte Teilliste ein

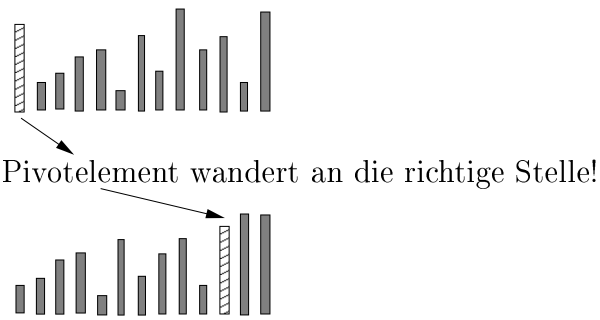
(3)Quicksort

Kurzbeschreibung :

Nimm Kopf x der Liste(nennt man Pivotelement), bilde 2 Teillisten mit Elementen <=x bzw. >x. Diese beiden Teillisten werden dann rekursiv weiterverarbeitet. Zwischen die beiden Ergebnisse wird x eingefügt.

```
qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:xs) = qsort[y|y<-ys,y<=x]++[x]++qsort[y|y<-ys,y>x]
```

Visualisierung:



Aufteilung der Liste in Teillisten durch Vergleich mit Pivotelement

(4) Mergesort

Kurzbeschreibung :

Zunächst teilen wir die Liste in möglichst gleichgrosse Teillisten (einelementig oder leer), dann mischen wir die Teillisten paarweise und wenden dieses Verfahren rekursiv an.

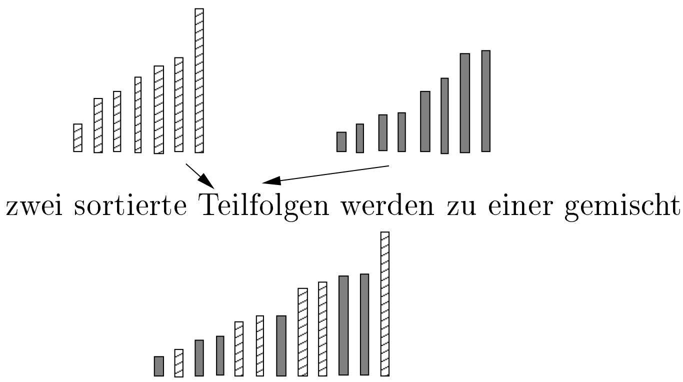
Die Funktion merge „mischt“ die beiden sortierten Teilliste zu einer sortierten Liste:

```
merge :: [Int] -> [Int] -> [Int]
merge [] []     = []
merge [] (x:xs) = x:(merge [] xs)
merge (y:ys) [] = y:(merge ys [])
merge (x:xs) (y:ys)
   | x<=y      = x:(merge xs (y:ys))
   | otherwise = y:(merge (x:xs) ys)
```

nun die Hauptfunktion, die merge benutzt :

```
msort :: [Int] -> [Int]
msort xs
   | length xs<2 = xs
   | otherwise   = merge (msort first) (msort second)
   where
      first  = take half xs
      second = drop half xs
      half   = div (length xs) 2
```

Visualisierung:



Die Merge–Operation bei Mergesort

3.5 Aufwandanalyse von Algorithmen

Um Algorithmen zu analysieren, stellen wir uns die Frage, welchen Aufwand (bzw. Komplexität) diese haben. Aber welches allgemeine Maß können wir dazu verwenden?

Eigentlich müßten wir, um die Komplexität von Algorithmen berechnen (vergleichen) zu können, die Rechenzeit und den Speicherbedarf eines konkreten Rechners benutzen. Um aber ein größeres Abstraktionsniveau zu erhalten, müssen wir eine Abschätzung finden, die keine Abhängigkeiten, zur gewählten Sprache, Compiler bzw. Rechner haben. Wir schätzen also die Anzahl der notwendigen Elementarschritte auf einem „abstrakten“ Rechenmodell (welche dabei zugelassen sind, muss definiert werden!).

worst-case-Analysen Es wird der Aufwand eines Algorithmus im schlechtesten Fall bei der Eingabe einer bestimmten Größe betrachtet und abgeschätzt.

average-case-Analysen Der Aufwand über alle möglichen Eingaben wird gemittelt („Erwartungswert“).

best-case-Analysen Hier analysiert man die Situation, die den Algorithmus am schnellsten zum Terminieren bringt und beschreibt dann deren Laufzeit. (Diese Analyse wird sehr selten gefordert)

Bei allen drei Analysen unterscheiden wir jeweils noch zwei Komplexitätsangaben, **obere Schranke** d.h. der betrachtete Algorithmus braucht zur Lösung eines algorithmischen Problems nur höchstens so viele Elementaroperationen und **untere Schranke** d.h. der Algorithmus braucht mindestens so viele Elementaroperationen.

Beispiele für Aufwandsanalysen:

(1) Analyse der benötigten Vergleiche bei Insertionsort

worst case :

Eingabefolge der Länge n ist monoton fallend.
Vergleiche :
$1 + 2 + 3 + ... + (n-1) = \frac{n*(n-1)}{2}$ Vergleiche
nun benutzen wir noch folgende Umformung :
$\frac{n*(n-1)}{2} = \frac{1}{2}n^2 - \frac{n}{2}$

Das Wachstum dieser Funktion unterscheidet sich nur um einen konstanten Faktor von n^2.Bei diesem Analyseergebnis sprechen wir von *quadratischem* Aufwand.

average case :

Liefert auch ein quadratisches Ergebnis (n^2), da im Erwartungswert das in eine sortierte Liste einzufügende Element grösser als $\sim$ die Hälfte der Listenelemente ist.

(2) Analyse von Quicksort

worstcase :

Bei jedem Rekursionsschritt ist das gewählte Pivotelement ein extremes.
D.h., eine der beiden Teillisten ist leer.
$\frac{n*(n-1)}{2}$ Vergleiche werden benötigt.

Wir haben hier ebenfalls einen quadratischen Aufwand.

average case :

$c * n * log_2\, n$ Vergleiche benötigt der Algorithmus.
c ist dabei eine Konstante.

(3) Analyse von Mergesort

worstcase :

$c * n * log_2\, n$ Vergleiche .
c ist dabei eine Konstante.

Da mergesort mit dem quicksort-Verfahren sehr verwandt ist, erstaunt die Laufzeit nicht gerade. Doch wie wir im worstcase-Verhalten von quicksort gesehen haben, kann der entstehende Baum entarten in quadratische Laufzeit verursachen. Anders geschieht es da bei mergesort, hier konstruieren wir den Baum von unten nach oben. Der entstehende Baum kann nicht entartet sein, folglich verhält sich der Algorithmus im worstcase wie im averagecase, also mit einer Komplexität von $n * log\, n$.

Ausblick :

Eine offentsichtliche untere Schranke für vergleichsbasiertes Sortieren ist $n-1$, da man jedes Element wenigstens einmal mit einem anderen vergleichen muss.

Man kann zeigen, dass jedes vergleichsbasierte Sortierungsverfahren sogar mindestens $n * log_2\, n$ Vergleiche benötigt, um n Objekte zu sortieren.

Es gibt Situationen, wo man schneller sortieren kann! Beispielsweise könnte man Listen sortieren, die nur aus Elementen eines festen Bereichs bestehen. (siehe Countingsort, Radixsort, Bucketsort)

3.6 Funktionen höherer Ordnung

Dies sind Funktionen, die Funktionen als Argumente erhalten können. Beispiele dafür sind in Haskell die vordefinierten Funktionen: map, filter, foldr, foldr1, foldl, foldl1.

Mapping:

Mit Mapping sind wir in der Lage eine bestimmte Funktion auf jedes Element einer Liste anzuwenden und erhalten eine Liste mit den entsprechenden Ergebnissen. Dabei sei f eine Funktion, die a nach b abbildet. Nun starten wir mit der Liste, bestehend aus Elementen vom Typ a und erhalten nach jeweiliger Anwendung die Liste der Ergebnisse vom Typ b.

```
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x<-xs]
```

oder rekursiv definiert :

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x):(map f xs)
```

Beispiel :

Berechnung der ersten 10 Fibonacci-Zahlen.

```
map fib [1..10]
(Aber : map length ['a'..'z'] liefert einen error!)
```

Filtering:

Das Filtern ermöglicht es uns, ganz bestimmte Elemente, nämlich genau die, die das Prädikat erfüllen, in die neu entstehende Liste zu übernehmen. Wir übergeben an die Funktion also ein Prädikat und eine Liste und erhalten eine Liste zurück (wir verändern den Typ a dabei nicht!)

```
filter :: (a -> Bool) -> [a] -> [a]
filter xs = [x | x<-xs, p x]
```

oder rekursiv definiert :

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs     = []
filter p (x:xs)
   | p x       = x:(filter p xs)
   | otherwise = filter p xs
```

Beispiel :

```
isSorted :: [Int] -> Bool
isSorted xs = (quicksort xs == xs)
```

Folding:

Es gibt mehrere Funktionen, die das Falten einer Liste ermöglichen. Ziel dieser Faltung ist dann ein Ergebniselement.

Wir schauen uns zuersteinmal die Faltung einer Liste von rechts nach links an.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f y []     = y
foldr f y (x:xs) = f y (foldr f b xs)
```

Dabei ist (a -> b -> b) eine Funktion die zwei Variablen verarbeitet und in den entsprechenden Typ ueberfuehrt. Der Startwert ist vom Typ b und gibt den Wert der Faltung auf leerer Liste an. Hier waehlen wir meistens das neutrale Element zur Funktion (bei (+) die 0 ...). [a] ist die von rechts nach links zu faltende Liste. b liefert uns dann das Ergebnis.

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f xs = foldr f (xs!!(length xs)) (take((length xs)-1)xs)
```

oder rekursiv definiert :

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]    = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Bei dieser Funktion koennen wir auf den Startwert verzichten, dieser ist einfach das letzte Element der Liste

So faltet foldr die Liste [1..10] bezüglich Addition und Startwert 0.

```
foldr (+) 0 [1..10] 1+(...(8+(9+(10+0)))...)
```

Wir können nun ebenfalls die Liste von links nach rechts falten. Für dieses Vorgehen existiert die foldl-Funktion.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Links- und Rechtsfaltung muessen nicht zum gleichen Ergebnis fuehren!!!

So faltet foldl die Liste : Beispiel : foldl (+) 0 [1..10] (...(((0+1)+2)+3)+...)+10

Weitere Beispiele:

(1) Summenberechnung

```
summe' :: [Int] -> Int
summe' xs = foldr (+) 0 xs
summe'' :: [Int] -> Int
summe'' xs = foldr1 (+) xs
```

(2) Listenlänge

```
laenge :: [a] -> Int
laenge = foldr einsplus 0
   where
      einsplus :: a -> Int -> Int
      einsplus x n = n+1
```

(3) Reversefunktion

```
reverse :: [a] -> [a]
reverse = foldr snoc []
   where
      snoc :: a -> [a] -> [a]
      snoc x xs = xs++[x]
```

(4) Map-Funktion

```
map :: (a -> b) -> [a] -> [b]
map f = foldr ((:).f) []
```

(5) Insertionsort

```
insertionsort :: Ord a => [a] -> [a]
insertionsort xs = foldr insert [] xs
   where
      insert :: Int -> [Int] -> [Int]
      insert x []     = [x]
      insert x (y:ys)
         | x<=y      = x:(y:ys)
         | otherwise = y:(insert x ys)
```

(6) Incrementierung

```
increment :: a -> Int -> Int
increment x y = y+1
```

3.7 Funktionskomposition

Wir können die Hintereinanderausführung von Funktionen, durch den Operator „ . “ darstellen.

$$f(g(x)) = (f.g)\ x$$

Allgemein hat Funktionskomposition den Typ:

```
(.) :: (b->c)->(a->b)->(a->c)
```

wobei (b->c) die Funktion f, (a->b) die Funktion g und (a->c) die entstandene Komposition f.g repräsentiert. Funktionskompositionen mehrere Funktionen sind assoziativ, d.h.

$$(f.g).h = f.(g.h)$$

Aber : $f.g\ x$ liefert nicht das gewünschte, da die Funktion g eine stärkere Bindung besitzt und demnach zuerst g x ausgewertet wird und dann der Punktoperator, also $f.(g\ x)$. An einem Beispiel läßt sich leicht sehen, dass das nicht stimmen kann :

```
not.not True
-> not.False
-> Typfehler ...
```

Beispiel zur Funktionskomposition :

Die Iteration einer Funktion ist wie folgt beschreibbar :

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f
   | n>0       = f.iter (n-1) f
   | otherwise = id
```

id ist die Identitätsfunktion, die mit f x = x definiert ist. Man könnte id abstrakt als das neutrale Element der Funktionen beschreiben (dabei muss man aber vorsichtig sein).

Nun schauen wir uns anhand eines Beispiels an, wie die Funktion iter arbeitet :

```
iter 3 (*2)
-> (*2).iter 2 (*2)
-> ((*2).(*2)).iter 1 (*2)
-> (((*2).(*2)).(*2)).iter 0 (*2)
-> (((*2).(*2)).(*2)).(id)
```

Haskell würde < <function> > als Ergebnis liefern. Das bedeutet, wir erhalten eine Funktionskomposition, die hier einen zusätzlichen Parameter erwartet :

Beispiel :

```
iter 3 (*2) 5
...
-> 40
```

Nun könnte man also auf der Konsole äquivalent dazu folgende Eingabe tätigen :

```
(((*2).(*2)).(*2)).(id) 5
```

Wir erwarten als Ergebnis 40? Nein! Haskell liefert zurecht **error ...** , weil

```
(((*2).(*2)).(*2)).(id) 5
-> (((*2).(*2)).(*2)).5
```

Was soll Haskell hier machen?

Eine richtige Eingabe sollte so aussehen :

```
((((*2).(*2)).(*2)).(id)) 5
-> 40
```

nun Definition der iter-Funktion mit Faltung :

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f = foldr (.) id (replicate n f)
```

replicate liefert eine n-elementige Liste der Funktion f

Beispiel :

```
replicate 3 f
-> [f,f,f]
```

nun arbeitet foldr diese Liste ab, startet dann mit der Identitätsfunktion und liefert eine Funktionskomposition.

Kapitel 4

Codierungstheorie

4.1 Grundbegriffe der Codierungstheorie

Codierung Darstellung von Information durch Zeichenfolgen

Forderungen :

1. Eindeutige Decodierbarkeit
2. Kompakte (kurze) Codierungen (wichtig für *Datenkompression*)
3. Fehler bei Verarbeitung/Übertragung nach Möglichkeit erkennen/korregieren

(dagegen : *Kryptographie* - codierte Information soll möglichst nur durch Berechtigte decodierbar sein)

Grundschema der Informationsübertragung

$$\begin{tikzcd}
\text{Informationsquelle} \arrow[d, "\textit{Datenkompression}"] & & \text{Informationssenke} \\
\text{Quellencode} \arrow[d, "\textit{Kanalcodierung}"] & & \text{Zielcode} \arrow[u, "\textit{Quellendecodierung}"'] \\
\text{Sender} \arrow[rr, "+\ \text{Rauschen}", "\text{Übertragungskanal}"'] & & \text{Empfänger} \arrow[u, "\textit{Kanaldecodierung}"']
\end{tikzcd}$$

Achtung :

> Forderung nach kurzer Codierung und Forderung nach Fehlererkennung widersprechen sich! Fehlererkennung erfordert *Redundanz*!

Definition:

Information Wort über Alphabet A

codierte Information Wort über Alphabet B (meistens B=$\mathbb{B}$=$\{0, 1\}$)

$$c : A \to B^+ = B^* \setminus \{\varepsilon\}$$

Es muss eine injektive Abbildung sein. Diese wird erweitert zu

$$c : A^+ \to B^+$$

durch $c(a_1a_2...a_k) = c(a_1) \circ c(a_2) \circ ... \circ c(a_k)$ | $\circ$ ist Konkatenation

Menge der Codewörter:

$$C(A) = \{c(a) \mid a \in A\}$$

Forderung:

Existenz einer Umkehrfunktion

$d : \{c(w) \mid w \in A^+\} \to A^+$mit $d(c(w)) = w$

(eindeutig decodierbar)

Definition : **Blockcode**

$c : A \to \mathbb{B}^n$, bei festem n

Jedes Zeichen wird durch $0-1$-Wort der gleichen Länge n codiert (2^n Zeichen codierbar!)

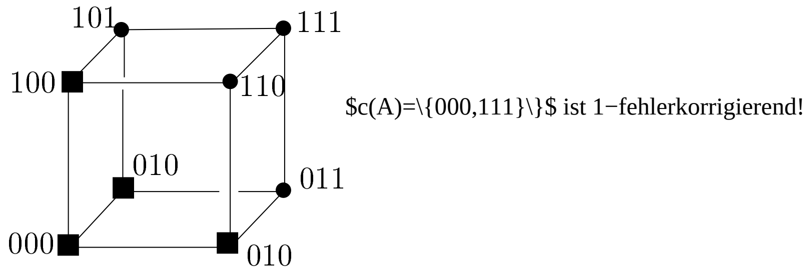
Definition : **Hamming-Abstand**

Sei b, $b' \in \mathbb{B}$ mit $b = (b_1, ..., b_n)$ und $b' = (b'_1, ..., b'_n)$, dann sei der Hamming-Abstand wie folgt definiert :

$d_H\ (b, b') = |\{i|b_i \neq b'_i\}|$

Für einen Blockcode c sei

$d_H(c) = min\ \{d_H(c(a), c(a'))|a \neq a'\ \wedge\ a, a' \in A\}$



Definition : Decodierung eines Blockcodes mittels Maximum-Likelihood-Methode

> Nachricht $b \in \mathbb{B}^n$ wird decodiert als Zeichen $a \in A$ mit $d_H(c(a), b)$ ist minimal für alle a.

d_H ist Metrik (=Abstandsfunktion) für Menge $\mathbb{B}^n$.

Einschub:

allgemein: $d : X * X \to \mathbb{R}$ ist Metrik für Menge X falls

(1) $d(x, y) \geq 0$, $\forall x, y \in X$

(2) $d(x, y) = 0 \Leftrightarrow x = y$

(3) $d(x, y) = d(y, x)$, $\forall x, y$

(4) $d(x, z) \leq d(x, y) + d(y, z)$, $\forall x, y, z \in X$ (Δ-Ungleichung)

Weitere Beispiele:

(a) Euklidischer Abstand d_2

$r = (x, y)$, $r' = (x', y') \in \mathbb{R}^2$

$d_2(r, r') = \sqrt{(x - x')^2 + (y - y')^2}$ ist der übliche Abstand in der Ebene.

(b) Manhattan-Abstand d_1

$r = (x, y)$, $r'(x', y') \in \mathbb{R}^2$

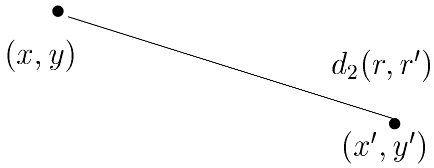
$d_1(r, r') = |x - x'| + |y - y'|$

(c) $0 - 1-$Metrik für beliebige Grundmenge X

wenn $x \neq x'$ so $d(x, x') = 1$

sonst $d(x, x') = 0$

Euklidischer Abstand $r := (x, y)$, $r' = (x', y')$



$$d_2(r, r') = \sqrt{(x - x')^2 + (y - y')^2}$$

Manhattan Abstand

$$d_1(r, r') = |x - x'| + |y - y'|$$

Definition: sei $c : A \to \mathbb{B}^n$ ein Blockcode.

1) c heißt k-fehlererkennend, wenn $\forall\ a \in A$, $b \in \mathbb{B}^n$ gilt, dass
$1 \leq d_H(b, c(A)) \leq k \Rightarrow b \notin c(A)$
2) c heißt k-fehlererkennend, wenn $\forall\ a \in A$, $b \in \mathbb{B}^n$ gilt, dass
$d_H(b, c(a)) \leq k \Rightarrow \forall a' \neq a : d_H(b, c(a')) > k$
(Wenn b eine fehlerhafte Nachricht ist, s kann $c(a)$ als eindeutiges Codewort mit $\leq k$ Fehlern zugeordnet werden!)
Achtung : Immer die Maximum-Likelihood-Methode voraussetzend!

Fakt: $d_H(c) \geq 2k + 1 \Longrightarrow k$-fehlerkorregierend

geometrisch :

4.2 Einfache Methoden zur Fehlererkennung

Im Folgenden ist $c : A \to \mathbb{B}^n$ Blockcode. c ist injektiv, also $d_H(c) \geq 1$.
Wie kann man daraus durch das Anfügen von redundanten Bits Fehlererkennungscodes ableiten?

1. **Paritätsbit**

$c_p : A \to \mathbb{B}^{n+1}$
$c_p(a) =_{def} c(a) \circ b_p(a)$
wobei $b_p(a) = \begin{cases} 0\, gerade\ Anzahl\ von\ Einsen \\ 1\, sonst \end{cases}$
$\Rightarrow d_H(c_p) \geq 2$, weil alle Codewörter $c_p(a)$gerade Anzahl von Einsen haben

2. **Doppelcodierung**

   $c^2 : A \rightarrow \mathbb{B}^{2n}$
   $c^2(a) =_{def} c(a) \circ c(a)$
   $\Rightarrow d_H(c^2) \geq 2$

3. **Doppelcodierung mit Paritätsbit**

   $c^2 : A \rightarrow \mathbb{B}^{2n+1}$
   $c_p^2(a) =_{def} c(a) \circ c(a) \circ b_p(a)$
   Beobachtung : $d_H(c_p^2) \geq 3$ und damit ist c_p^2 1-fehlerkorregierend.
   Beweis : $d_H(c(a), c(a')) \geq 2 \Longrightarrow d_H(c_p^2(a), c_p^2(a)) \geq 4$
   also nur interessant :
   Wenn $d_H(c(a), c(a')) = 1$
   so ist $d_H(c_p^2(a), c_p^2(a')) = 3$
   ,weil der Abstand verdoppelt wird und das Paritätsbit kippt.

Wir „erkaufen" die Möglichkeit zur Fehlererkennung/-korrektur durch zusätzliche Bits, die für Informationsgehalt redundant sind!

Beispiel :

z	$c(z)$	$c_p(z)$	$c^2(z)$	$c_p^2(z)$
a	00	000	0000	00000
b	10	101	1010	10101
c	01	011	0101	01011
d	11	110	1111	11110
Zeichen	Blockcode	+Paritätsbit	+Nachricht doppelt	+Nachricht doppelt, Paritätsbit

Tabelle 4.1: verschiedene Fehlererkennungsverfahren

$c^2(z)$ aus Spalte 4 ist trotz doppelter Datenmenge nicht 1−fehlerkorregierend, da beispielsweise a mit einem Fehlerbit an der ersten Position, beispielsweise den gleichen Hammingabstand hätte wie b an der dritten Position :

$$d_H(c^2(a), 1000) = d_H(c^2(b), 1000) = 1$$

Die Spalte 4 hingegen ist 1−fehlerkorregierend.

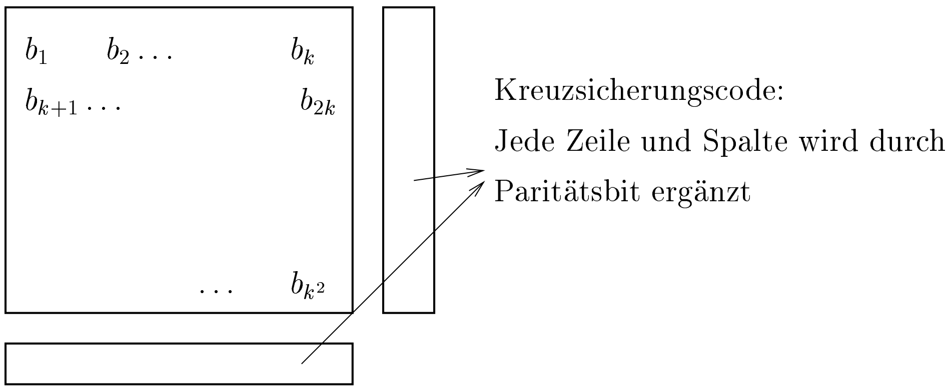
Definition : **Informationsrate** (IR)

IR $= \frac{\#Informationsbit}{Codewortlnge}$

Wir haben nun also eine mögliche Codierung gefunden (c_p^2), die 1−fehlerkorregierend ist. Welche Informationsrate hat unser Verfahren?

$\mathrm{IR}(c_p^2) = \frac{n}{2n+1} < \frac{1}{2}$

4. **Kreuzsicherungscode**



Sei $n = k^2$ und $c : A \longrightarrow \mathbb{B}^{\ltimes}$ mit $c(a) = b_1 ... b_k \circ b_{k+1} ... b_{2k} \circ ...$

Wenn wir uns jetzt die Informationsrate dieses Verfahrens anschauen, dann sehen wir eine Verbesserung.

$\mathrm{IR}(c_x) = \frac{n}{n+2\sqrt{n}} \sim 1 - \frac{2}{\sqrt{n}}$

Satz 4.2.1 $d_H(c_x) \geq 3$ *(d.h 1−fehlerkorregierend)*

Beweis (durch Fallunterscheidung)

1. $d_H(c(a), c(a')) \geq 3$ (Fakt)
2. $d_H(c(a), c(a')) = 2$

   $\Rightarrow$ Fehler in verschiedenen Spalten/Zeilen
   $\Rightarrow d_H(c_x(a), c_x(a')) = 4 \; oder \; 6$

3. $d_H(c(a), c(a')) = 1$

   $\Rightarrow$ 1 Fehler und 2 kippende Paritätsbits
   $\Rightarrow d_H(c_x(a), c_x(a')) = 3$

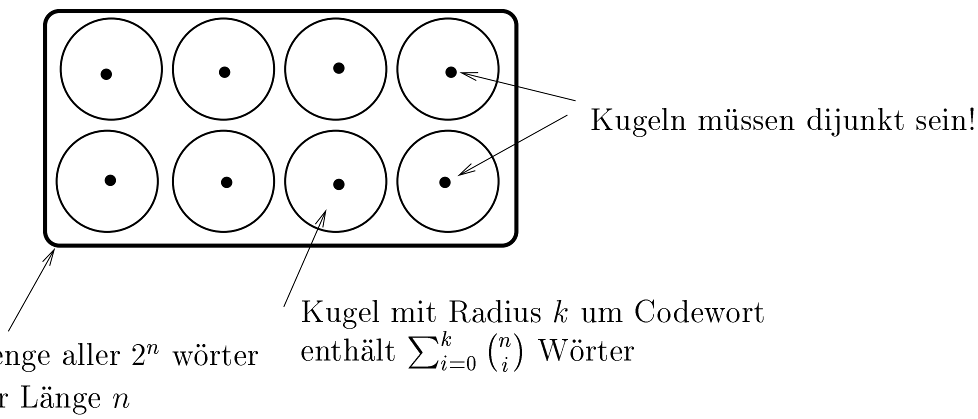
Beobachtung :

c sei k-fehlerkorregierender Code

$\Rightarrow \sum_{i=0}^{k} \binom{n}{i} * |A| \leq 2^n$

Beweis :

$\binom{n}{i}$ =Anzahl der Wörter in Hamming-Abstand i (=i gekippte Bits) von gegebenem Codewort $c(a)$ für $a \in A$



Folgerung :

Es könnte also 1−fehlerkorregierenden Code mit 4 Informationsbits und 3 Kontrollbits geben. Und den gibt es auch :

Hammingcode Ham(3,2)

$b_1, ..., b_4$ Informationsbits

$b_5 = b_2 \oplus b_3 \oplus b_4$

$b_6 = b_1 \oplus b_3 \oplus b_4$

$b_7 = b_1 \oplus b_2 \oplus b_4$

4.3 Codierung mit variabler Codewortlänge

Wir hatten bisher (bei der Blockcodierung $c : A \rightarrow \mathbb{B}^n$)eine feste Länge des Codewortes gefordert. Dadurch war es uns besser möglich Fehler zu erkennen und diese zu korregieren. Nun wollen wir aber die Codierung mit variablen Codewortlängen fomulieren. Dadurch erhalten wir bei geschickter Codewahl gute oder bessere Komprimierungseffekte.

Dazu nehmen wir uns den zu codierenden Text und bestimmen die Anzahl der auftretenden Zeichen und deren relative Häufigkeiten (Anzahl des Auftretens eines Zeichens in Bezug auf ein anderes). Man sieht auch leicht ein, dass ein sehr oft auftretenden Zeichen ein kürzeres Codewort erhalten sollte.

Unser Ziel könnte im Folgenden formuliert werden, als die Suche nach der minimal codierten Nachricht (Gesamtlänge) . Es gibt viele Verfahren, die ein solches Problem lösen, aber folgende wollen wir nur erwähnen : Null-Unterdrückung, Lauflängen-Codierung, Paarkodierung, Mustерersetzung, ...

Wir werden nun ein Verfahren kennenlernen (Präfixcode), das auf statistischer Codierung beruht. Wir gehen davon aus, dass die Häufigkeiten bekannt sind.

Präfixcodes Ein Code, bei dem kein Codewort Präfix eines anderen Codewortes ist, heißt Präfixcode.

Wir können einen Codewortbaum so darstellen :

codierte Zeichen $\longleftrightarrow$ Blätter

Codewort $\longleftrightarrow$ Kantenmarkierungen auf Weg von Wurzel zu Blatt

Beispiel :

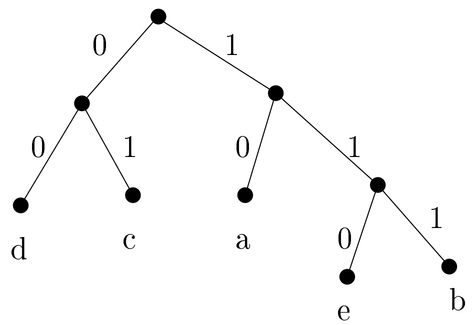
$c(a) = 10$

$c(b) = 111$

$c(c) = 01$

$c(d) = 00$

$c(e) = 110$



Nun haben wir folgende codierte Nachricht : 11001111

Wir suchen den ersten auftretenden Präfix in unserer Codierung.

110 ist der Präfix, der durch $c(e)$ repräsentiert wird, dann suchen wir ab dieser Stelle den nächsten Präfix, usw. Man kann aber die Decodierung auch als Traversierung des Codebaumes auffassen. Demnach starten wir in der Wurzel

und gehen für 1 nach rechts, 1 wieder nach rechts und 0 nach links. Da wir ein Blatt erreicht haben, ist gerade dieses das codierte Zeichen, also e. Das nächste Zeichen finden wir wieder beginnend an der Wurzel ...

Wir erhalten also für 11001111 die Nachricht „*ecb*“.

Lemma 2 *Eine mit einem Präfixcode codierte Nachricht ist eindeutig decodierbar. (Achtung! Umkehrung gilt nicht.)*

Beweis (indirekt, d.h. $p \Longrightarrow q$ ist semantisch äquivalent zu $\neg p \Longrightarrow \neg q$)

Angenommen $\exists\ a_1...a_m \neq a'_1...a'_k$ mit $c(a_1...a_m) = c(a'_1...a'_k)$

sei i kleinster Index mit $a_{i+1} \neq a'_{i+1}$ und $a_i = a'_i$

$\Longrightarrow c(a_1...a_i) = c(a'_1...a'_i)$

Fall 1 : $|c(a_{i+1})| = |c(a'_{i+1})|$

$\Longrightarrow c(a_{i+1}) \neq c(a'_{i+1})$

hier Widerspruch zur Annahme, dass die Gesamtcodierungen gleich sind!

Fall 2 : $|c(a_{i+1})| < |c(a'_{i+1})|$

$\Longrightarrow$ aus Annahme folgt, $c(a_{i+1})$ ist Präfix von $c(a'_{i+1})$

das ist widerum ein Widerspruch zum Präfixcode!

Fall 3 : $|c(a_{i+1})| > |c(a'_{i+1})|$

$\Longrightarrow$ aus Annahme folgt, $c(a'_{i+1})$ ist Präfix von $c(a_{i+1})$

das ist widerum ein Widerspruch zum Präfixcode!

Satz 4.3.1 *Ungleichung von Kraft*

(1) Sei c ein Präfixcode für$A = \{a_1, ..., a_m\}$,

es gilt :

$$\sum_{i=1}^{m} \frac{1}{2^{|c(a_i)|}} \leq 1$$

(2) Seien $n_1, ...n_m$ *natürliche Zahlen und* $\sum_{i=1}^{m} \frac{1}{2^{n_m}} \leq 1$. *Dann gibt es einen Präfixcode für m Zeichen mit den Codewortlängen* $n_1, ..., n_m$.

Beweis (1) Induktion nach Anzahl der Blätter im Codewortbaum. O.B.d.A. alle inneren Knoten haben Verzweigungsgrad 2

Induktionsanker :

$n = 1$, $n = 2$ ...erfüllt

Induktionsvoraussetzung :

richtig für Blätter $\leq n$

Induktionsschritt :

- wir betrachten Zwillingsblätter der Tiefe m und ihren Vaterknoten
- entfernen beide Blätter v_1, v_2 und wenden Induktionsvoraussetzung auf den entstehenden Baum mit n Blättern an

$$\begin{aligned}\sum_{i=1}^{n+1} 2^{-m_i} &= 2^{-m} + 2^{-m} + \sum_{i=3}^{n+1} 2^{-m_i} \\ &= 2^{-(m-1)} + \sum_{i=3}^{n+1} 2^{-m_i} \\ &\ \text{Baum mit Blättern } v, v_3, v_4, ..., v_{n+1} \\ &\leq 1\end{aligned}$$

(2) Wir ordnen die Längen $m_1 \leq m_2 \leq ... \leq m_m$

Setzen $c(a_1) = 0...0$ $(=n_1)$

Annahme $c(a_1), ..., c(a_i)$ mit $i < m$ ist schon bestimmt, nun wählen wir $c(a_{i+1})$ als lexikographisch kleinstes Wort, das die $c(a_1), ..., c(a_i)$ nicht als Präfix enthält. Gibt es ein solches?

Ja, - insgesamt $2^{n_{i+1}}$ Wörter der Länge n_{i+1}.

$c(a_1)$ verbietet $2^{n_{i+1}-n_1}$
$c(a_2)$ verbietet $2^{n_{i+1}-n_2}$
$\vdots$
$c(a_i)$ verbietet $2^{n_{i+1}-n_i}$

demnach verboten : $\sum_{j=1}^{i} 2^{n_{i+1}-n_j} = 2^{n_{i+1}} \sum_{j=1}^{i} 2^{-n_j} < 2^{n_{i+1}}$ q.e.d.

Beispiel :

Codewortlängen $1, 3, 3, 3, 4, 4$

Test : $1 * \frac{1}{2} + 3 * \frac{1}{8} + 2 * \frac{1}{16} = 1$

Codewörter : $\{0, 100, 101, 110, 1110, 1111\}$

4.4 Codierung bei gegebener Wahrscheinlichekeitsverteilung der Zeichen

$A = \{a_1, ..., a_n\}$
$p : A \to (0,1)$
$p_i = p(a_i)$ ist Wahrscheinlichkeit des Auftretens von Zeichen a_i soll gelten $\sum_{i=1}^{n} p_i = 1$
Sei $c : A \to \mathbb{B}^+$ Präfixcode
Definition :

$$\begin{aligned} n(c) &= \sum_{i=1}^{n} |c(a_i)| * p_i \\ & \quad \text{Erwartungswert für die Länge eines Codewortes} \\ &= \sum_{i=1}^{n} d_T(a_i) * p_i \\ & \quad T \text{ Codebaum } d_T(a_i) \text{ Tiefe des Blattes } a_i \end{aligned}$$

c ist optimal, wenn für alle c' gilt, dass $n(c) \leq n(c')$.

Der Huffman-Algorithmus liefert optimale Präfixcodes!

Huffman-Algorithmus

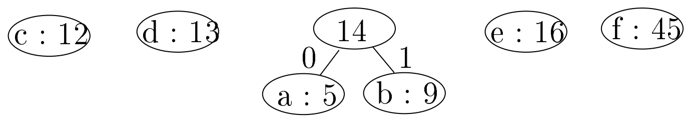
Der Huffman-Algorithmus arbeitet *bottom-up*, d.h. er bearbeitet kleinere Teilprobleme zuerst und „hangelt“ sich so zu grösseren weiter (beziehend auf Teillösungen). Dabei verwendet er die Greedy-Strategie.

```
Initialisierung :
   Zeichen bezüglich Häufigkeit aufsteigend sortieren
   und als Liste von n Bäumen (je einem Knoten) mit
   Markierung a_i|p_i ablegen.
Rekursiv (n-1 mal):
   Streiche die zwei Bäume T_1,T_2 mit minimalem Gewicht
   aus der Liste und füge einen neuen Baum ein, dessen
   Wurzel die Bäume T_1,T_2 als Teilbäume hat und mit der
   Summe der Gewichte markiert ist.
```

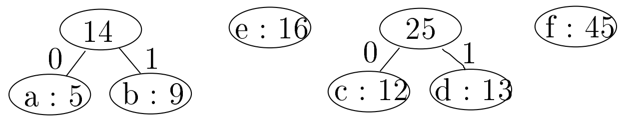
Beispiel für Huffman–Codierung

1.Schritt

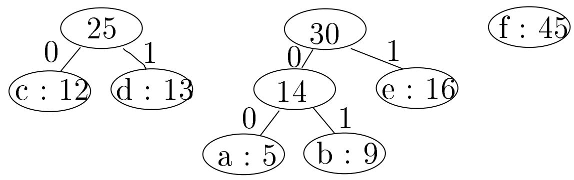
2.Schritt



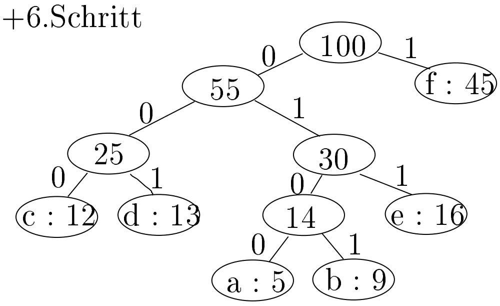
3.Schritt



4.Schritt



5.+6.Schritt



Die Codewörter entsprechen den Wegmarkierungen von der Wurzel zu den Blätter. Für d ist es 001, für f das Wort 1.

Die erwartete Codewortlänge für diesen Text ist 2,24.

Lemma 3 *Seien $x, y \in A$ Zeichen mit kleinster Wahrscheinlichkeit in einer zu codierenden Nachricht und sei $p(x) \leq p(y)$. Dann gibt es einen optimalen Präfixcode $c : A \to \{0,1\}^+$bei dem im zugehörigen Baum x, y als Zwillinge autreten, sich also nur im letzten Bit unterscheiden.*

Beweis Sei c ein optimaler Präfixcode, T der zugehörige Baum. Betrachten

Zwillinge a, b in T auf maximaler Tiefe. O.b.d.A. $p(a) \leq p(b)$.

Wir definieren c' aus c, indem wir Codewort $c(a)$ mit $c(x)$ und $c(b)$ mit $c(y)$ austauschen.

SKIZZE

Zu zeigen : $n(c') \leq n(c)$

$$\begin{aligned} n(c') &= \sum_{i=1}^{m} p_i \, d_{T'}(a_i) \\ &= n(c') - p(a)d_T(a) - p(b)d_T(b) - p(x)d_T(x) - p(y)d_T(y) \\ &\quad + p(x)d_T(a) + p(y)d_T(b) + p(a)d_T(x) + p(b)d_T(y) \\ &= n(c) + (p(x) - p(a))\,(d_T(a) - d_T(x)) \\ &\quad + (p(y) - p(b))\,(d_{T'}(b) - d_{T'}(y)) \end{aligned}$$

Also : $n(c') \leq n(c)$

Lemma 4 *Sei T ein optimaler Präfixbaum für den Code c von (A, p). Wenn $x, y \in A$ Zwillinge sind mit Vaterknoten z, dann ist :*

$T' = T \setminus \{x, y\}$ ein optimaler Präfixbaum

für $A' = (A \setminus \{x, y\} \bigcup \{z\})$ und Wahrscheinlichkeit p' mit

$$p'(a) = \begin{cases} p(a) & a \neq z \\ p(x) + p(y) & a = z \end{cases}$$

Beweis $n(c) = n(c') + p(x) + p(y)$

Angenommen, c' ist nicht optimal. Sei $n(c'') < n(c')$. Im Baum T'' ist z ein Blatt. Durch Verzweigen zu neuen Blättern x, y erhalten wir den Code c^*für A mit $n(c^*) = n(c'') + p(x) + p(y) < n(c)$.

Das ist aber ein Widerspruch.

Kapitel 5

Haskell weiterführende Konzepte

5.1 Typklassen

Beispiel: Definition einer Funktion, die testes, ob ein Element in einer Liste ist.

```
elem :: a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = (x==y) || elem x ys
```

Dabei setzt (x==y) voraus, dass a den Operator == hat. Das ist aber nicht immer gegeben.

Fassen alle Typen mit == zu Klasse **Eq** zusammen. Dazu definieren wir

```
class Eq a where
   (==) :: a -> a -> Bool
```

Die Typen einer Klasse heißen *Instanzen*.

Beispiel: kennen bereits schon

- Int, Float, Bool, Char, diese sind Instanzen der Klasse Eq
- Haskell macht standardmäßig Tupel und Listen über Typen aus Eq zu Instanzen von Eq

Beispiel: (Int, Char) ist Instanz von Eq

Achtung: Die Standard == Definition muß nicht das sein, was man haben möchte. Man muß diese gegebenfalls selbst definieren.

Wichtig: Funktionen gehören nicht zu Eq.

Nun können wir zur vorher definierten elem-Funktion den genauen Typ angeben.

```
elem :: (Eq a) => a -> [a] -> Bool
```

(Eq a) heißt Kontext, indem elem definiert ist, dieser kann auch mehreren Typvariablen entsprechen.
Tip: Haskell bietet die Möglichkeit mit :type, den allgemeinsten Typ einer konkreten Funktion abzufragen.

5.1.1 Definition einer Klasse

Um eine Klasse zu definieren, bedarf es einer Signatur.

Beispiel:

```
class Name a where
  ...
  Funktionen, die Typvariable a betreffen
  ...
```

Wie macht man einen konkreten Typ zur Instanz einer Klasse?

```
instance Name Type where
  ...
  konkrete Definition der Signaturfunktion (muss nicht
  die kanonische sein) von Name für Type
  ...
```

Beispiel:

```
class Visible a where
   toString :: a -> String
   size     :: a -> Int
```

(1) eine mögliche Instanzierung

```
instance Visible Char where
   toString ch = [ch]
   size     _  = 1
```

(2) weitere mögliche Instanzierung

```
instance Visible a => Visible [a] where
   toString    = concat.map toString
   size        = foldr (+) 1.map size
```

5.1.2 Abgeleitete Klassen

Definition: Falls {Signatur Klasse **A**} $\subset$ {Signatur Klasse **B**}, dann „erbt **B** von **A**", bzw. „**B** kann aus **A** abgeleitet werden".

Beispiel:

```
class Eq a => Ord a where
   compare           :: a -> a -> Ordering
   (<),(<=),(>=),(>) :: a -> a -> Bool
   max, min          :: a -> a -> a
```

Wir haben hier die Signatur von Ord formuliert, gleichermaßen wurde aber der Operator (==) von Eq geerbt.
(Dabei enthält Ordering die Objekt LT, EQ, GT)

Man muss nicht alle Funktionen explizit angeben, es gibt auch default-Definitionen.

Beispiel:

(1) Default Definition von compare, falls (<=) und (==) bekannt

```
compare x y
   | x==y      = EQ
   | x<=y      = LT
   | otherwise = GT
```

(2) Falls compare gegeben

```
x<=y = (compare x y /= GT)
x<y  = (compare x y == LT)
x>=y = (compare x y /= LT)
x>y  = (compare x y == GT)
```

5.2 Algebraische Typen

5.2.1 Aufzählende Typen

Beispiele:

```
data Bool     = False | True
data Ordering = LT | EG | GT
```

oder auch

```
data Season   = Spring | Summer | Fall | Winter
data Temp     = Cold | Hot
```

Funktionen können mit Pattern Matching definiert werden.

```
wheather :: Season -> Temp
wheather Summer = Hot
wheather _      = Cold
```

5.2.2 Produkttypen

Produkttypen entsprechen dem kartesischen Produkt (entspricht Tupelbildung)

Beispiel:

(1) Datentyp People enthält 2-stelligen Konstrukto Person, der auf Name und Age zugreift.

```
data People = Person Name Age
type Name = String
type Age  = Int
```

(2) alternativ dazu

```
type People = (Name, Age)
```

Vorteile des algebraischen Typs

- Konstruktor wird immer mit angegeben, => bessere Lesbarkeit des Programms
- bei Fehlermeldung wird Typ angegeben

Vorteile von Tupeln

- kürzer
- polymorphe Funktion verwendbar

5.2.3 Alternativen

Beispiel: (entspricht mengentheoretischer Vereinigung)

```
data Shape = Circle Float | Rectangle Float Float
```

Funktion wieder mit Pattern Matching

```
isRound :: Shape -> Bool
isRound (Circle _)      = True
isRound (Rectangle _ _) = False
```

Allgemein können wir so Typen der Form definieren :

```
data Typename =
   Const1 t11 ... t1k1 |
   Const2 t21 ... t2k2 |
   ...
   Constl tl1 ... tlkl
```

ki sind nichtnegative ganze Zahlen, die die Stelligkeit von Const i angeben.

5.2.4 Rekursive/polymorphe algebraische Typen

Binäre Bäume mit ganzahligen Knoten

- möglicherweise leer
- in Knoten gespeicherte ganze Zahlen
- Ausgrad jedes inneren Knoten ≤ 2

```
data NTree = NilT | Node Int NTree NTree
```

Skizze
Wir könnten den Baum auch polymorph machen, indem wir Int durch a ersetzen.

Beispiel: Knotenanzahl eines Baumes ermitteln

```
size :: NTree -> Int
size NilT = 0
size (Node n t1 t2) = 1 + size t1 + size t2
```

Binäre Bäume mit polymorphen Knoten

```
data Tree a = Nil | Node a (Tree a) (Tree a)
              deriving (Eq, Ord, Show, Read)
```

Funktionen werden wie gehabt definiert.

Beispiel: Tiefe eines Baumes ermitteln

```
depth :: Tree a -> Int
depth Nil          = 0
depth (Node t1 t2) = 1 + max (depth t1) (depth t2)
```

Anmerkung: Um den Nachweis bestimmter Eigenschaften für spezielle Listenfunktionen zu führen, wählt man meisst die Induktion über Listen. Bei Baumstrukturen verwendet man die strukturelle Induktion über Tiefe/Größe.

Um nun das Prädikat **P(tr)** für alle Bäume **tr** vom Typ **t** nachzuweisen

(1) *Induktionsanfang:* Beweise **P(Nil)**

(2) *Induktionsschritt:* Beweise **P(Node x tr1 tr2)** für alle **x** vom Typ **t**, vorausgesetzt **P(tr1)** und **P(tr2)** gelten.

Beispiel:

Zeige: $size\,tr < 2^{(depth\,tr)}$

Beweis (durch strukturelle Induktion)

Induktionsanfang:

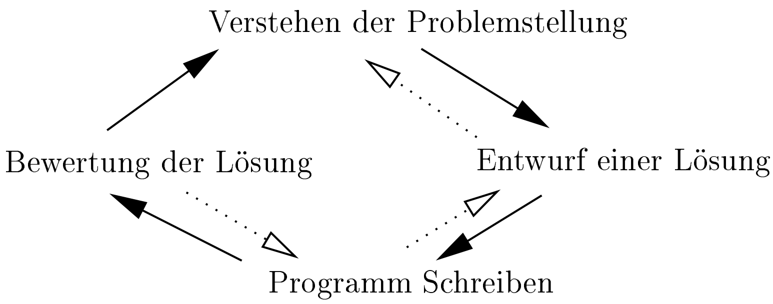
$$\begin{aligned} size\,NilT &= 0 \\ &< 2^0\,(=1) \end{aligned}$$

Induktionsschritt:

$$\begin{aligned} size\,(Node\,x\,\mathrm{t1}\,t2) &= 1 + size\,t1 + size\,t2 \\ &< 1 + 2^{(depth\,t1)} + 2^{(depth\,t2)} \quad , nach\,I.V. \\ &\leq 1 + 2 * 2^{(max\,(depth\,t1)(depth\,t2))} \\ &\leq 1 + 2^{(max\,(depth\,t1)(depth\,t2)\,+1)} \\ &= 1 + 2^{(depth(Node\,x\,t1\,t2))} \end{aligned}$$

5.3 Allgemeine Prinzipien der Programmentwicklung

Programmentwicklung als Zyklus:



Verstehen des Problems:

- Eingaben, Ausgaben des Problems, welche speziellen Eigenschaften
- Beispiele grafisch veranschaulichen
- Ist Spezifikation vollständig?
- mathematische Anforderungsdefinition

Lösungsentwurf:

- verwandte Probleme, Einschränkungen, Verallgemeinerungen anschauen
- Zerlegung in Teilaufgaben, gibt es dafür Standardlösungen?
- Welche Rechenstrukturtypen werden benötigt
- Algorithmen und Datenstrukturen festlegen

Implementierung:

- syntaktisch korrekt, welche Möglichkeiten bietet die Programmiersprache?
- (in Haskell sinnvoll: allgemeinere Funktionen für spezielle Anwendung schreiben als notwendig und diese dann einschränken (Bsp: map, foldr, ...))
- gut lesbar und wiederverwendbar schreiben
- Nutzen von Programmbibliotheken

Evaluierung:

- Test an typischen, zufälligen, kritischen Instanzen, benchmarks
- Fehleranalyse (*Debugging*)
- Effiziensanalyse (Theorie und Praxis), Vergleich zu anderen Lösungen
- Verifikation, dass Anforderung erfüllt werden (kann nur so gut sein, wie die Spezifikation, benötigt vollständige Beschreibung der Semantik der Sprache, sehr formal, für große Projekte schwer machbar)

Am Wichtigsten: Kritischer Blick auf die eigene Arbeit und Versuch, aus Fehlern zu lernen :).

5.4 Modularisierung in Haskell

- komplexe Algorithmen/Programme werden (bzw. müssen) in Teilaufgaben zerlegt, hierarchisch
- diese sollten voneinander unabhängig realisierbar sein und in sich abgeschlossen
- die einzelnen Bausteine heißen *Module* und sind selbst wieder Haskell-Skripts
- der Algorithmus, der ein Modul aufruft, braucht nur die Funktionalität zu kennen, nicht wie diese im Modul realisiert wird (Information hiding), Modul und aufrufender Algorithmus können getrennt entworfen werden, Reimplementierung einzelner Module möglich
- Modularisierung erleichtert auch Verständnis der Algorithmen
- man kann Bibliotheken von Modulen aufbauen und diese dann bei Bedarf in jeden beliebigen Algorithmus einfügen

Syntax: Modulnamen (großer Anfangsbuchstabe), als *Modulname.hs* bzw. *Modulname.lhs* abspeichern

```
module Modulname where
 function1 x = ...
 function2 y = ...
 ...
```

Importieren eines Moduls:

```
module Modul1 where
 import Modul2
 function x = ...
 ...
```

Import/Export-Kontrollen

Man kann explizit angeben, was importiert/exportiert wird.

Beispiel:

```
module Modul1 (function1, function2) where
```

Dabei ist auch klar, dass

```
module Modul1 where
```

dasselbe ist, wie

```
module Modul1 (module Modul1) where
```

Einschränkung des Imports:

```
import Modul1 hiding (function)
```

Standardmäßig wird Prelude immer als Modul importiert.

5.4.1 Implemetierung der Huffman-Codierung in Haskell

Präfixcodes $\longleftrightarrow$ Codewörterbäume
Codewort für c $\longleftrightarrow$ Markierung auf Weg von Wurzel zu Blatt mit Label c

Beim Huffman-Code: Knoten im Baum zusätzlich mit Häufigkeit (aller Blätter darunter zusammen) markiert

Hauptschritte die realisiert werden müssen:

- Häufigkeiten der Zeichen im Text feststellen (module MakeCode)
- Verschmelzen und Sortieren von Bäumen (module MakeCode)
- Codebaum in Codiertabelle übersetzen (module MakeCode)
- Codieren/Decodieren (module Coding)

Welche Typen brauchen wir?

Codewörter sind Bitfolgen.

```
data Bit   = L | R
             deriving (Eq,Show)
type HCode = [Bit]
data Tree  = Leaf Char Int | Node Int Tree Tree
type Table = [(Char,HCode)]
```

Diese werden zu einem Modul (module Types) zusammengefasst.

(Eine Skizze dazu befindet sich im Thompson.)

Der Coding-Modul

```
> module Coding (codeMessage, decodeMessage)
> import Types (Tree(Leaf, Node), Bit(L,R),HCode,Table)
```

Wir importieren alles, es ist aber lesbarer, dies nocheinmal hinzuschreiben.

```
> codeMessage :: Table -> [Char] -> HCode
> codeMessage tbl = concat.map (lookupTable tbl)
```

lookupTable ist eine Funktion, die nach einem Codewort für einzelne Zeichen nachschaut (Definition auf Funktionsebene für codeMessage! Diese wird nicht exportiert)

```
> lookupTable :: Tale -> Char -> HCode
> lookupTable [] c = error „lookupTable“
> lookupTable ((ch,n):tb) c
>     | ch==c     = n
>     | otherwise = lookupTable tb c
```

Diese Funktion wird nicht exportiert!

```
> decodeMessage :: Tree -> HCode -> [Char]
> decodeMessage tr = decodeByt tr
>     where
>        decodeByt (Node n t1 t2) (L:rest)
>                       = decodeByt t1 rest
>        decodeByt (Node n t1 t2) (R:rest)
>                       = decodeByt t2 rest
>        decodeByt (Leaf c n) rest
>                       = c : decodeByt tr rest
>        decodeByt t [] = []
```

5.5 Lazy Programming

Grundprinzip in Haskell: lazy evaluation „Argumente einer Funktion werden erst dann evaluiert, wenn sie tatsächlich gebraucht werden“

- Das Erzeugen von Zwischenergebnissen muss nicht „teuer“ sein
- Haskell kann dadurch auch mit unendlichen Objekten, wie z.B. einer Liste aller Primzahlen umgehen

Lazy Evaluation

Beispiel: $f\,x\,y = x + y$

Auswertung von $f\,(6-5)\,(f\,3\,2)$

```
f(6-5)(f 3 2)
-> (6-5)+(f 3 2)
-> (6-5)+(3+2)
-> 1+5
-> 6
```

Beispiel: Duplizierte Argumente werden nur einmal ausgewertet!

Auswertung von $h\,x\,y = x + x$

```
h (3-2) 5
-> (3-2)+(3-2)
-> 1+1
-> 2
```

Beispiel: Die Implementierung ersetzt Ausdrücke durch Graphen nicht notwendigerweise Bäume!

Auswertung von $g\,(x,\,y) = x + 1$

```
g (2+3,4+5)
-> (2+3)+1
```

(4+5) wird nicht ausgewertet!

```
-> 5+1
-> 6
```

Beispiel: Pattern Matching

```
f :: [Int] -> [Int] -> Int
f [] ys          = 0
f (x:xs) []      = 0
f (x:xs) (y:ys) = x+y
```

mögliche Auswertung

```
f [1..3] [1..3]
-> f (1:[2..3]) [1..3]      -- kann nicht Fall 1 sein
-> f (1:[2..3]) (1:[2..3]) -- kann nicht Fall 2 sein (also 3)
-> 1+1
-> 2
```

- Auswertung bis man sieht welcher Fall eintritt
- analog bei guards und lokalen Definitionen

Reihenfolge der Evaluation:

von aussen nach innen

$$f\, e_1\, (g\, e_2)$$

erst f dann g

$$f\, e_1 + g\, e_2$$

von links nach rechts

List Comprehension:

allgemeine Form $[e \mid q_1, ..., q_k]$
Qualifier q_i

- Generator $p \leftarrow LExpr$ (p ist Pattern, $LExpr$ ist Ausdruck mit Listentyp)
- Test

In q_i kann man auf Variable in q_1 bis q_{i-1} verweisen.

Beispiel: Pythagoräische Tripel

```
pyTriple n = [(x,y,z)|x<-[2..n],y<-[x+1..n],
                      z<-[y+1..n],x*x+y*y==z*z]
pyTriple 10 = [(3,4,5),(6,8,10)]
```

Auswertung intern

```
[e|v<-[a1,..,an],q2,...,qk]
-> [e{a1/v}|q2{a1/v},...,qk{a1/v}] ++ ...
   ... ++ [e{an/v}|q2{an/v},...,qk{an/v}]
```

bei Tests

```
[e|True,q2,...,qk]
->[e|q2,...,qk]
[e|False,q2,...,qk]
->[]
```

Beispiel:

```
[n*n|n<-[1..7],n*n<30]
-> [1*1|1*1<30]++[2*2|2*2<30]++..++[7*7|7*7<30]
-> [1|True]++[4|True]++..++[49|False]
-> [1,4,9,16,25]
```

Lazy Evaluation fördert den sogenannten data-directed Programmierstil, Datenstrukturen werden nur bei Bedarf erzeugt.

Beispiel:

```
sumFP n = sum (map (^4) [1..n])
-> sum (map (^4) (1:[2..n]))
-> sum ((^4) 1:map (^4) [2..n])
-> (1^4)+sum (map(^4) [2..n])
-> 1 + ...
...
-> 1+(16+(81+(625+...+n^4)))
```

Die Zwischenlisten werden dabei garnicht erzeugt.

Minimum einer Liste berechnen:

Mit insSort sortieren und Kopf nehmen, die Liste wird gar nicht vollständig sortiert.

```
insSort [8,6,1,7,5]
-> ins 8 (ins 6 (ins 1 (ins 7 (ins 5 []))))
-> ins 8 (ins 6 (ins 1 (ins 7 [5])))
-> ins 8 (ins 6 (ins 1 (5:ins 7 [])))
-> ins 8 (ins 6 (1:(5:ins 7 [])))
-> ins 8 (1:ins 6 (5:ins 7 []))
-> 1:ins 8 (ins 6 (5:ins 7 []))
```

Und bei head.insSort wird jetzt das Minimum ausgegeben.

Unendliche Listen:

Was passiert bei Evaluation von

```
ones = 1:ones
-> [1,1,... Interrupted!
```

aber

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x:y:zs) = x+y
```

gibt bei

```
addFirstTwo ones
-> addFirstTwo (1:ones)
-> addFirstTwo (1:1:ones)
-> 1+1
-> 2
```

Standardmäßig gibt es Listen [n..] und [n,m..]

Beispiel: Prelude-Funktion iterate

```
iterate :: (a->a)->a->[a]
iterate f x = x:iterate f (f x)
```

liefert also

```
[x,f x, f(f x), ...]
```

Ein Primzahltest (Sieb des Eratosthenes)

Wir streiche aus der Liste aller ganzen Zahlen, die Vielfachen von bereits erkannten Primzahlen, übrig bleiben die Primzahlen

```
2 3 4 5 6 7 8 9 10 11 12 13 14
2 3   5   7   9    11    13
2 3   5   7        11    13
usw.
primes :: [Int]
primes = sieve [2..]
sieve (x:xs) = x:sieve [y|y<-xs, mod y x>0]
```

Auswertung

```
primes
-> 2:sieve[y|y<-[3..],mod y 2>0]
-> 2:sieve(3:[y|y<-[4..],mod y 2>0])
-> 2:3:sieve[z|z<-[y|y<-[4..],mod y 2>0],mod z 3>0]
...
```

Primzahltest mittels memberOrd-Funktion

```
memberOrd :: Ord a => [a] -> a -> Bool
memberOrd (x:xs) n
   | x<n       = memberOrd xs n
   | x==n      = True
   | otherwise = False
```

Achtung: Normale member-Funktion geht nicht!

Wie kann man Eigenschaften für Programme mit Listen beweisen?

- nichtterminierende Programme haben den Wert *undef*
- *undef* gibt es über jedem Typ a

```
undef :: a
undef = undef
```

Wir haben damit partielle Listen, die auch *undef* enthalten. Induktionsbeweis, - die Eigenschaft P(xs) für unendliche, partielle Listen soll gezeigt werden, dazu brauchen wir:

Induktionsanfang: Beweise P([]) und P(undef)

Induktionsschritt: Beweise P(x:xs) aus P(xs)

Eine unendliche Liste von undefinierten Werten wird approximiert durch die Folge von Listen

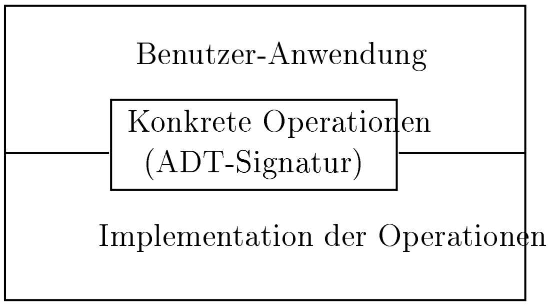
undef, a_0:undef, a_0:a_1:undef, ...

Eigenschaften gelten für unendliche Listen, wenn sie für die Approximation gelten.

5.6 Abstrakte Datentypen

Neben Modulen sind ADT's (Abstrakte Datentypen) eine Möglichkeit zur Realisierung einer *divide&conquer*-Taktik und insbesondere des *information hiding.*

Sie stellen die Schnittstelle zu einem konkreten Typ her, durch Angabe einer Menge von konkreten Operationen mit bestimmter Funktionalität. Der konkrete Typ kann dann nur über diese Operationen angesprochen werden.



Vorteile:

- der Benutzer braucht sich nicht um die Implementierung kümmern, verschiedene Implementierungen möglich
- man muss sich nur auf eine Signatur einigen
- information hiding

Beispiel: Queues (Warteschlangen)

Warteschlangen arbeiten nach dem first-in-first-out Prinzip (FIFO). Wir wollen uns zu dieser Datenstruktur noch Funktionen wie Einfügen/Entfernen von Objekten und eine leere Queue realisieren.

```
module Queue (
   Queue,
   emptyQ,    -- Queue a
   isemptyQ,  -- Queue a -> Bool
   addQ,      -- a -> Queue a -> Queue a
   remQ,      -- Queue a -> (a, Queue a)
) where
```

Jetzt folgt die konkrete Implementierung, z.B. mit Listen

```
newtype Queue a = Qu [a]
emptyQ = Qu []
isemptyQ (Qu []) = True
isemptyQ _       = False
addQ x (Qu xs)   = Qu (xs ++ [x])
remQ (Qu xs)
   | not (isemptyQ (Qu xs)) = (head xs, Qu (tail xs))
   | otherwise              = error „remQ“
```

Beobachtung: Diese Implementierung macht remQ besonders effizient, aber addQ ist „teuer“. Wenn man das Verfahren umdreht und in den Kopf der Liste einfügt, dann ist es umgekehrt.

Ausweg: Implementierung mittels zweier Listen.

```
data Queue a = Qu [a] [a]
```

... (wie gehabt)

```
addQ x (Qu xs ys)    = Qu xs (x:ys)
remQ (Qu (x:xs) ys) = (x, Qu xs ys)
remQ (Qu [] [])     = error ,,remQ"
remQ (Qu [] ys)     = remQ (Qu (reverse ys) [])
```

Fazit:

- nach aussen hin ist nicht sichtbar, welche Implementierung gewählt wurde
- Effizienz aber unterschiedlich!

5.6.1 Entwurf von ADT's

1. Was soll die Funktionalität sein
2. insbesondere „natürliche" Operationen ermöglichen wie
   (a) leeres Objekt
   (b) transformieren/kombinieren/kollabieren von Objekten
   (c) Auswertung von Attributen

Beispiel:

```
module Tree (
   Tree,
   nil,         -- Tree a
   isNil,       -- Tree a -> Bool
   isNode,      -- Tree a -> Bool
   leftSub,     -- Tree a -> Tree a
   rightSub,    -- Tree a -> Tree a
   treeVal,     -- Tree a -> a
   insTree,     -- Ord a => a -> Tree a -> Tree a
   delete,      -- Ord a => a -> Tree a -> Tree a
   minTree,     -- Ord a => Tree a -> Maybe a
) where
data Tree a = Nil | Node a (Tree a) (Tree a)
```

Interessant sind nur insTree, delete und minTree. Diese kann man so implementieren, dass binäre Suchbäume realisiert werden.

Suchbaum:

Ein Baum Node val t1 t2 ist ein Suchbaum, wenn

(1) alle Werte in t1 <= Wert val

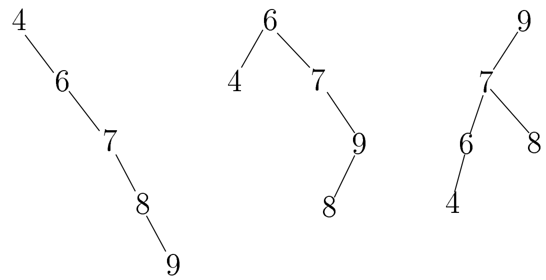
(2) val <= alle Werte in t2

(3) t1, t2 sind Suchbäume

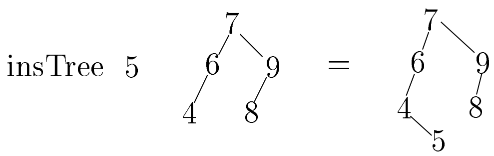
(4) Nil ist Suchbaum

Beispiel:

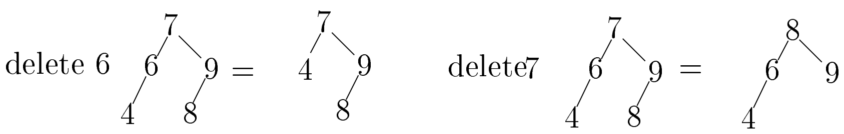
Einige mögliche Suchbäume für Menge $\{4, 6, 7, 8, 9\}$:



Einfüge-Operation:



Streich-Operation:




```
insTree val Nil = (Node val Nil Nil)
insTree val (Node v t1 t2)
   | v==val = Node v t1 t2  -- Wert ist schon im Suchbaum
   | val>v  = Node v t1 (insTree val t2)
   | val<v  = Node v (insTree val t1) t2
delete val (Node v t1 t2)
   | val<v     = Node v (delete val t1) t2
   | val>v     = Node v t1 (delete val t2)
   | isNil t2  = t1
```

```
  | isNil t1  = t2
  | otherwise = join t1 t2
```

join t1 t2 ist Hilfsfunktion und wird nicht exportiert

setzt minTree t2 als Wurzel über t1 (delete minTree t2)

Fazit:

- die Operationen müssen so geschrieben sein, dass die Invariante „bin Suchbaum" erhalten bleibt!
- können Suchbaumfunktion auch über sortierte Listen realisieren (minTree „billig", aber insTree und delete?)
- balancierte Bäume als Ziel (zusätzliche Struktur aufsetzen z.B. Rot-Schwarz-Bäume)
- amortisierte Analyse statt worst-case-Analyse der einzelnen Operationen

Anhang A

Anhang

A.1 Wichtiges zur Booleschen Algebra

Boolesche Funktionen

f_0	„Kontradiktion„
f_1	and
f_2	not (a $\Rightarrow$ b)
f_3	a
f_4	not (b$\Rightarrow$ a)
f_5	b
f_6	Antivalenz, xor
f_7	or
f_8	nor
f_9	Äquivalenz
f_{10}	not b
f_{11}	b $\Rightarrow$ a
f_{12}	not a
f_{13}	a $\Rightarrow$ b
f_{14}	nand
f_{15}	„Tautologie„

Tabelle A.1: Nomenklatur 2-stelliger Funktionen

Gesetze

Ausdruck	$\equiv$	Ausdruck	Gesetz
$\neg\neg x$	$\equiv$	x	Involution
$x \wedge y$	$\equiv$	$y \wedge x$	Kommutativität
$x \vee y$	$\equiv$	$x \vee y$	Kommutativität
$x \wedge (y \vee z)$	$\equiv$	$(x \wedge y) \vee (x \wedge z)$	Distribution
$x \vee (y \wedge z)$	$\equiv$	$(x \vee y) \wedge (x \vee z)$	Distribution
$x \wedge x$	$\equiv$	x	Idempotenz
$x \vee x$	$\equiv$	x	Idempotenz
$x \wedge (x \vee y)$	$\equiv$	x	Absorption
$x \vee (x \wedge y)$	$\equiv$	x	Absorption
$\neg(x \wedge y)$	$\equiv$	$\neg x \vee \neg y$	de Morgan'sche Regel
$\neg(x \vee y)$	$\equiv$	$\neg x \wedge \neg y$	de Morgan'sche Regel
$x \vee (y \wedge \neg y)$	$\equiv$	x	Neutralität
$x \wedge (y \vee \neg y)$	$\equiv$	x	Neutralität

Tabelle A.2: Boolesche Gesetze

A.2 Wichtige Haskellfunktionen

```
splitAt :: Integral t => t -> [u] -> ([u],[u])
```

zerlegt Listen in Anfang(Präfix) und Rest, wobei die Indizes bis zum eingegebenen Element zur ersten Liste im Tupel und der Rest zur zweiten gehört.

```
Beispiel :
   splitAt 6 "pferdewagen"
   -> ("pferde","wagen")
reverse :: [t] -> [t]
```

dreht die Eingabeliste um

```
Beispiel 1:
   reverse "einnegermitgazellezagtimregennie"
   -> "einnegermitgazellezagtimregennie"
Beispiel 2:
   reverse "12345"
   -> "54321"
zip :: [t] -> [u] -> [(t, u)]
```

macht Tupel, mit dem ersten Element aus t und dem zweiten aus u; sollte eine Liste länger sein als die andere, so fallen diese Werte weg

```
Beispiel :
   zip [1,2,3] ["Mike","Paul","Nina"]
   -> [(1,"Mike"), (2,"Paul"), (3,"Nina")]
unzip :: [(t,u)] -> ([t],[u])
```

liefert zwei Listen als Tupel; unzip ist nicht die Umkehrfunktion zu zip! (Daten können verloren gehen ...)

Literaturverzeichnis

[Pat] Patterson, David A. und John Hennessy : Computer Organisation and Design (Second Edition). The Hardware/Software Interface. San Francisco 1997

[Has] Thompson, Simon : Haskell-The Craft of Functional Programming, AW 1999 (2.Auflage)

weiterführende Literatur : ***Technische Informatik***

Hagmann, Gert : Grundlagen der Elektrotechnik. Wiesbaden 1997

Hagmann, Gert : Aufgabensammlung zu den Grundlagen der Elektrotechnik. Wiesbaden 1999

Schiffmann, Wolfram und Schmitz, Robert : Technische Informatik 1. Grundlagen der digitalen Elektronik. Heidelberg 1999

Hering, Ekbert - Bressler, Klaus und Gutekunst, Jürgen : Elektronik für Ingenieure. Düsseldorf 1994

weiterführende Literatur : ***Theoretische Informatik***

Schöning, Uwe : Theoretische Informatik - kurzgefaßt. Heidelberg 1997

weiterführende Literatur : ***Funktionale Programmierung***

Pepper, Peter : Funktionale Programmierung in OPAL, ML, HASKELL und GOFER. Berlin 1999

Index