Informatik A: Inoffizielles Skript

David Lichteblau
`lichtebl@math.fu-berlin.de`

3. Februar 2000

Vorwort

Zur Vorlesung Informatik A im WS 99/00 gibt es kein Skript. Das ist zwar bedauerlich, läßt sich aber nicht ändern, also müssen alle, die verrückt genug sind, so früh morgens nicht zu schlafen, selbst mitschreiben. Leider kann ich meine eigene Schrift kaum lesen und bin deshalb gezwungen, meine Mitschrift abzutippen, wenn ich sie jemals wieder verwenden will. Daher gibt es sie hiermit gleich in halbwegs ordentlicher Form.

Kapitel 1

Boolesche Algebra und Boolesche Funktionen

1.1 Aspekte von Information

- Äußere Form (Darstellung, Repräsentation) 22. Okt.
  Beispiel: 10110 (Binärzahl)

- Bedeutung (Abstrakte Information)
  Beispiel: 41 (Zahl in einer Tabelle)

- Bezug zur realen Welt (Worauf wird Bezug genommen?)

Beispiel: Entfernung zwischen A und B ist 41

- Gültigkeit (Wahrheitsgehalt)
  Beispiel: Angabe richtig? Welche Einheit? Luftlinie?

1.2 Informationssysteme

Man verwendet:

Σ Alphabet, Menge der Symbole.

Σ^* Menge möglicher *Worte/Strings*

w Einzelner String aus den Symbolen $a_i \in \Sigma$ (Länge k):
$w = a_1 a_2 a_3 \ldots a_{k-1} a_k$

$|w|$ Die Länge eines Strings. $|w| = k$

ϵ Der leere String: $|\epsilon| = 0$

Definition 1 (Informationssystem). Ein Informationssystem besteht aus einer Menge R von Repräsentationen
Zusatz #1 ($R \subseteq \Sigma^*$), einer Menge A $\boxed{\text{von abstrakten Informationen}}$ und einer Interpretation $I\colon R \longrightarrow A$.

Beispiel. Strings von Ziffern sollen als natürliche Zahlen interpretiert werden (Basis k):

- $\Sigma = \{0, 1, 2, \ldots, k\}$
- $A = \mathbb{N}_0$

- $R = \Sigma^*$
- Interpretation $I\colon R \longrightarrow A$ mit
  $I[0] =$ "Null"
  $I[1] =$ "Eins"
  $I[2] =$ "Zwei"
  $\vdots$

Beispiel. Alternative Darstellung natürlicher Zahlen

- $\mathbb{N}_0 = \{\epsilon, |, ||, |||, \ldots\}$
- $\Sigma = \{0, 1, 2, \ldots, k\}$
- $A = \mathbb{N}_0$
- $R = \Sigma^*$
- $$\begin{aligned} I[0] &= \epsilon \\ I[1] &= | \\ I[2] &= || \\ &\vdots \end{aligned}$$

Beispiel. Natürliche Zahlen im Binärsystem

- $\Sigma = \{0, 1\}$
- $A = \mathbb{N}_0$
- $R = \Sigma^*$

- $w = a_k a_{k-1} \dots a_2 a_1 a_0$
- $I[w] = \sum_{i=0}^{k} 2^i a_i$

Siehe auch unten.

Beispiel. Brüche

- $A = \mathbb{Q}$
- $R = \mathbb{Z} \times \mathbb{N}$
- Interpretation also:

$$\begin{aligned} I : \mathbb{Z} \times \mathbb{N} &\longrightarrow \mathbb{Q} \\ (z, n) &\longmapsto \tfrac{z}{n} \end{aligned}$$

Anmerkung zur Umkehrbarkeit. Die Interpretation muß nicht umkehrbar sein. Die oben beschriebene Darstellung der rationalen Zahlen als Brüche ist nicht umkehrbar, da etwa $\frac{6}{9} = \frac{2}{3}$. Im ersten Beispiel ist eine Umkehrung jedoch möglich. [Anmerkung: Jedenfalls, wenn man keine mehrsymboligen Repräsentationen zuläßt, die mit einer 0 beginnen.]

Zusatz #2

Darstellung im Binärsystem Diese Darstellung kann durch wiederholtes Teilen durch die Basis 2 gewonnen werden, wie in Tabelle 1.1 am Beispiel der 75 gezeigt. Bei anderen Zahlensystemen geht man entsprechend vor.

Definition 2 (Semantische Äquivalenz). (A, R, I) sei ein Informationssystem. Dann heißen $r_1 \in R$ und $r_2 \in R$ semantisch äquivalent, falls $I[r_1] = I[r_2]$, und man schreibt $r_1 \equiv r_2$.

75	ungerade	$a_0 = 1$
$\lfloor \frac{75}{2} \rfloor = 37$	ungerade	$a_1 = 1$
$\lfloor \frac{37}{2} \rfloor = 18$	gerade	$a_2 = 0$
$\lfloor \frac{18}{2} \rfloor = 9$	ungerade	$a_3 = 1$
$\lfloor \frac{9}{2} \rfloor = 4$	gerade	$a_4 = 0$
$\lfloor \frac{4}{2} \rfloor = 2$	gerade	$a_5 = 0$
$\lfloor \frac{2}{2} \rfloor = 1$	ungerade	$a_6 = 1$

Tabelle 1.1: Darstellung von $(75)_{10}$ im Binärsystem

1.3 Boolesche Terme

Der Begriff der Aussage wurde von ARISTOTELES beschrieben:

> Eine Aussage ist ein sprachliches Gebilde, von dem es sinnvoll ist zu sagen, es sei wahr oder falsch.

Es werden unterschieden:

1. - Elementare Aussagen. Diese enthalten keine Variablen, d.h. $\boxed{\text{haben festen Wahrheitswert.}}$ Zusatz #3
   - Aussagen mit Variablen, d.h. Identifikatoren für frei wählbare Aussagen
2. - atomare Aussagen
   - zusammengesetzte Aussagen

Man verwendet:

E Menge von elementaren Aussagen ("7 ist prim", "9 ist eine gerade Zahl")

ID unendliche, abzählbare Menge von Variablen

Definition 3 (Boolesche Terme). Induktive Definition.

1. *true* und *false* sind Boolesche Terme.

2. Alle atomaren, elementaren Aussagen sind Boolesche Terme. Alle Variablen sind Boolesche Terme.

3. Ist t ein Boolescher Term, dann ist auch $(\neg t)$ ein Boolescher Term.

4. Sind t_1 und t_2 Boolesche Terme, dann sind auch $(t_1 \wedge t_2)$, $(t_1 \vee t_2)$, $(t_1 \Longrightarrow t_2)$ und $(t_1 \Longleftrightarrow t_2)$ Boolesche Terme.

27. Okt. **Axiom 4 (Minimalitätsprinzip).** *Alle Booleschen Terme werden durch obige Definition erfüllt.*

Aus dem Minimalitätsprinzip ergibt sich, daß etwa $((t_1 \wedge t_2) \Longrightarrow (\neg a \vee b))$ kein Boolescher Term ist. Dennoch vereinbart man die folgenden Regeln zur Vereinfachung Boolescher Terme:

Einsparen von Klammern. Regeln:

1. $\neg$ bindet stärker als $\wedge$, $\vee$, $\Longrightarrow$ und $\Longleftrightarrow$.

2. $\wedge$ bindet stärker als $\vee$.
   $\wedge$ und $\vee$ binden stärker als $\Longrightarrow$ und $\Longleftrightarrow$.

3. Operatoren gleicher Bindungsstärke werden von links nach rechts geklammert.

Äußere Klammern brauchen nicht gesetzt zu werden; sie werden zur Abgrenzung nicht benötigt.

Beispiel. Vereinfachung von $((t_1 \wedge t_2) \Longrightarrow ((\neg a) \vee b))$.

$$\begin{array}{cl}
((t_1 \wedge t_2) \Longrightarrow ((\neg a) \vee b)) & \\
((t_1 \wedge t_2) \Longrightarrow (\neg a \vee b)) & \text{(Nach 1.)} \\
(t_1 \wedge t_2 \Longrightarrow \neg a \vee b) & \text{(Nach 2.)} \\
t_1 \wedge t_2 \Longrightarrow \neg a \vee b &
\end{array}$$

Beispiel. Umschreiben eines vereinfachten Terms als Booleschen Term durch Einklammern der Terme links und rechts der jeweils schwächsten Bindung.

$$\begin{array}{cl}
a \wedge b \wedge \neg c \vee d \Longrightarrow \neg a \wedge b & \\
((a \wedge b \wedge \neg c \vee d) \Longrightarrow (\neg a \wedge b)) & \text{(Nach 2.)} \\
(((a \wedge b \wedge \neg c) \vee d) \Longrightarrow ((\neg a) \wedge b)) & \text{(Nach 2. und 1.)} \\
((((a \wedge b) \wedge \neg c) \vee d) \Longrightarrow ((\neg a) \wedge b)) & \text{(Nach 3.)} \\
((((a \wedge b) \wedge (\neg c)) \vee d) \Longrightarrow ((\neg a) \wedge b)) & \text{(Nach 1.)}
\end{array}$$

Beispiel. Umschreiben eines vereinfachten Terms als Booleschen Term durch Einklammern der Terme neben der jeweils stärksten Bindung:

$$
\begin{array}{rl}
a \Longrightarrow b \wedge \neg c \vee d & \\
a \Longrightarrow b \wedge (\neg c) \vee d & \text{(nach 1.)} \\
a \Longrightarrow (b \wedge (\neg c)) \vee d & \text{(nach 2.)} \\
a \Longrightarrow ((b \wedge (\neg c)) \vee d) & \text{(nach 2.)} \\
(a \Longrightarrow ((b \wedge (\neg c)) \vee d)) &
\end{array}
$$

Beispiel. In vielen Termen können nicht alle Klammern durch die Vereinfachungsregeln entfernt werden:

$$
\begin{array}{rl}
(x \wedge (y \vee (z \Longrightarrow a))) & \\
x \wedge (y \vee (z \Longrightarrow a)) & \text{(Weiter geht's nicht.)}
\end{array}
$$

Beispiel. Obwohl (wie sich zeigen wird) folgendes sinnvoll ist, erlauben die Vereinfachungsregeln es zunächst nicht:

$$
\begin{array}{rl}
(a \wedge (b \wedge c)) & \\
(a \wedge b \wedge c) & \text{(falsche Umformung)} \\
((a \wedge b) \wedge c) &
\end{array}
$$

Definition 5. Man sagt, ein Term t enthält einen freien Identifikator x, falls gilt:

1. $t = x$
2. $t = \neg t'$ und t'enthält x.
3. $t = t_1 \,\mathrm{op}\, t_2$ mit $\mathrm{op} \in \{\wedge, \vee, \Longrightarrow, \Longleftrightarrow\}$ und t_1 oder t_2 enthalten x.

Definition 6. Ist $\{x_1, x_2, x_3, \ldots, x_k\}$ die Menge der freien Identifikatoren, die in einem Term t vorkommen, so nennt man t Aussage in $x_1, x_2, x_3, \ldots, x_k$.

1.4 Boolesche Funktionen I

Wir verwenden die Zeichen 29. Okt.

1 – wahr

0 – falsch

und die Menge $\mathbb{B} = \{0, 1\}$.

Die Booleschen Funktionen not, and, or, impl und equiv sind anhand der Wahrheitstafeln in Tabelle 1.2 definiert.

Lemma 7. *Es gelten:*

- $\mathrm{and}(b_1, b_2) = \mathrm{not}(\mathrm{or}(\mathrm{not}(b_1), \mathrm{not}(b_2)))$
- $\mathrm{impl}(b_1, b_2) = \mathrm{or}(\mathrm{not}(b_1), b_2)$
- $\mathrm{equiv}(b_1, b_2) = \mathrm{or}(\mathrm{and}(b_1, b_2), \mathrm{and}(\mathrm{not}(b_1), \mathrm{not}(b_2)))$

Beweis. Der Beweis erfolgt mit Hilfe von Wertetafeln. (Aus Gründen der Übersichtlichkeit wurden einige Terme unter Ver-

$\text{not}\colon \mathbb{B} \longrightarrow \mathbb{B}$

b	$\text{not}(b)$
1	0
0	1

$\text{and}\colon \mathbb{B}^2 \longrightarrow \mathbb{B}$

$b_1 \diagdown b_2$	1	0
1	1	0
0	0	0

$\text{or}\colon \mathbb{B}^2 \longrightarrow \mathbb{B}$

$b_1 \diagdown b_2$	1	0
1	1	1
0	1	0

$\text{impl}\colon \mathbb{B}^2 \longrightarrow \mathbb{B}$

$b_1 \diagdown b_2$	1	0
1	1	0
0	1	1

$\text{equiv}\colon \mathbb{B}^2 \longrightarrow \mathbb{B}$

$b_1 \diagdown b_2$	1	0
1	1	0
0	0	1

Tabelle 1.2: Definition einiger Boolescher Funktionen

wendung der Numerierung vereinfacht.)

1	2	3(1,2)	4(1)	5(2)	6(4,5)	7(6)
b_1	b_2	and(b_1, b_2)	not(b_1)	not(b_2)	or$(4,5)$	not(6)
0	0	0	1	1	1	0
0	1	0	1	0	1	0
1	0	0	0	1	1	0
1	1	1	0	0	0	1

Die Wahrheit der Behauptung folgt aus der Gleichheit von Spalte 3 und Spalte 7. □

1.5 Interpretation Boolescher Terme

Idee. (1) Atomare, elementare Aussagen (E) haben bereits einen Wahrheitswert. (2) Grundlage der Interpretation ist eine Belegung aller freien Variablen $\beta\colon ID \longrightarrow \mathbb{B}$. (3) Zusammengesetzte Terme werden duch die entsprechende Operation auf den Wahrheitswerten interpretiert.

Definition 8 (Boolesche Belegung). Eine Abbildung

$$\beta\colon ID \longrightarrow \mathbb{B}$$

heißt Boolesche Belegung.

Definition 9. Jede Boolesche Belegung $\beta\colon ID \longrightarrow \mathbb{B}$ definiert eine Interpretation der Booleschen Terme in der folgenden Art und Weise.

- $I_\beta[a]$ ist bestimmt für alle $a \in E$

- $I_\beta[true] = 1$
- $I_\beta[false] = 0$
- $I_\beta[x] = \beta(x)$ für alle $x \in ID$
- $I_\beta[\neg t] = \mathrm{not}(I_\beta[t])$ für einen Term t
- $I_\beta[t_1 \wedge t_2] = \mathrm{and}(I_\beta[t_1], I_\beta[t_2])$ für Terme t_1 und t_2
- $I_\beta[t_1 \vee t_2] = \mathrm{or}(I_\beta[t_1], I_\beta[t_2])$ für Terme t_1 und t_2
- $I_\beta[t_1 \Longrightarrow t_2] = \mathrm{impl}(I_\beta[t_1], I_\beta[t_2])$ für Terme t_1 und t_2
- $I_\beta[t_1 \Longleftrightarrow t_2] = \mathrm{equiv}(I_\beta[t_1], I_\beta[t_2])$ für Terme t_1 und t_2

Definition (Semantische Äquivalenz). Zwei Terme t_1 und t_2 heißen semantisch äquivalent, falls sie für jede Belegung

$$\beta \colon ID \longrightarrow \mathbb{B}$$

die gleiche Interpretation haben, d.h.

$$I_\beta[t_1] = I_\beta[t_2]$$

Satz 10. *Für beliebige Boolesche Terme t_1, t_2 und für jede Belegung $\beta \colon ID \longrightarrow \mathbb{B}$ gelten:*

$$\begin{aligned}
&I_\beta[t_1 \wedge t_2] = I_\beta[\neg(\neg t_1 \vee \neg t_2)] \\
&I_\beta[t_1 \Longrightarrow t_2] = I_\beta[\neg t_1 \vee t_2] \\
&I_\beta[t_1 \Longleftrightarrow t_2] = I_\beta[(t_1 \wedge t_2) \vee (\neg t_1 \wedge \neg t_2)]
\end{aligned}$$

Beweis. Umformung nach Definition der Interpretation:

$$\begin{aligned}
I_\beta[t_1 \wedge t_2] &= \mathrm{and}(I_\beta[t_1], I_\beta[t_2]) \\
&= \mathrm{not}(\mathrm{or}(\mathrm{not}(I_\beta[t_1]), \mathrm{not}(I_\beta[t_2]))) \\
&= \mathrm{not}(\mathrm{or}(I_\beta[\neg t_1], I_\beta[\neg t_2])) \\
&= \mathrm{not}(I_\beta[\neg t_1 \vee \neg t_2]) \\
&= I_\beta[\neg(t_1 \vee \neg t_2)]
\end{aligned}$$

□

Gesetze der Booleschen Algebra Einige grundlegende Gesetze, die zur Umformung Boolescher Terme nützlich sind, sind in Tabelle 1.3 zusammengestellt.

Beweis (Involutionsgesetz). In präziser Notation: 3. Nov.

$\beta(x)$	$I_\beta[\neg x]$	$I_\beta[\neg\neg x]$	$I_\beta[x]$
0	1	0	0
1	0	1	1

□

Beweis (Absorptionsgesetz). In vereinfachter Notation:

x	y	$x \vee y$	$x \wedge (x \vee y)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

□

$\neg\neg x \equiv x$	Involutionsgesetz
$\left.\begin{array}{c} x \wedge y \equiv y \wedge x \\ x \vee y \equiv y \vee x \end{array}\right\}$	Kommutativgesetze
$\left.\begin{array}{c} (x \vee y) \vee z \equiv x \vee (y \vee z) \\ (x \wedge y) \wedge z \equiv x \wedge (y \wedge z) \end{array}\right\}$	Assoziativgesetze
$\left.\begin{array}{c} x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z) \\ x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z) \end{array}\right\}$	Distributivgesetze
$\left.\begin{array}{c} x \wedge x \equiv x \\ x \vee x \equiv x \end{array}\right\}$	Idempotenzgesetze
$\left.\begin{array}{c} x \wedge (x \vee y) \equiv x \\ x \vee (x \wedge y) \equiv x \end{array}\right\}$	Absorptionsgesetze
$\left.\begin{array}{c} \neg(x \wedge y) \equiv \neg x \vee \neg y \\ \neg(x \vee y) \equiv \neg x \wedge \neg y \end{array}\right\}$	Gesetz von de Morgan
$\left.\begin{array}{c} x \vee (y \wedge \neg y) \equiv x \\ x \wedge (y \vee \neg y) \equiv x \end{array}\right\}$	Neutralitätsgesetz

Tabelle 1.3: Gesetze der Booleschen Algebra

1.6 Substitution

Definition 11 (Substitution). Seien t_1 und t_2 Boolesche Terme; $x \in ID$. Mit $t_1[t_2/x]$ bezeichnet man den Term, der entsteht, wenn jedes Auftreten von x in t_1 durch den Term t_2 ersetzt wird.

Sind $x1, x2, x3, \ldots, x_n$ paarweise verschiedene Variablen, so schreibt man auch $t[t_1/x_1, t_2/x_2, t_3/x_3, \ldots, t_n/x_n]$.

Beispiele. Einfache Beispiele sind:

$$\begin{aligned}
&y \wedge \neg z[\neg y/x] \equiv y \wedge \neg z \\
&x \vee \neg z[\neg y/x] \equiv \neg y \vee \neg z \\
&x \vee (y \wedge \neg x)[y \vee x/x] \equiv (y \vee z) \vee (y \wedge \neg(y \vee z))
\end{aligned}$$

Theorem 12 (Kongruenzeigenschaft). Sind die Bool. Terme t_1 und t_2 semantisch äquivalent, dann gilt für beliebige Terme und $x \in ID$ (Kongruenzeigenschaft der semantischen Äquivalenz für die Substitution):

1. $t_1[t/x] \equiv t_2[t/x]$
2. $t[t_1/x] \equiv t[t_2/x]$

Beispiel. Gesetze zur Vereinfachung des folgenden Terms sind bereits bekannt. Mit Hilfe der Kongruenzeigenschaft läßt sich nun beweisen, daß die Umformung tatsächlich einen semantisch äquivalenten Term ergibt:

$$\neg x_1 \vee \underbrace{x_2 \wedge x_3 \vee x_2 \wedge x_3 \wedge \neg x_2)}_{t_1} \equiv \neg x_1 \vee \underbrace{x_2 \wedge x_3}_{t_2}$$

$s_1 = x \vee x \wedge y \equiv x = s_2$

- Nach Regel 1:
  $t_1 = s_1[x_2 \wedge x_3/x, \neg x/y]$
  $\equiv s_2[x_2 \wedge x_3/x, \neg x_1/y] = x_2 \wedge x_3 = t_2$
- Nach Regel 2:
  $t = \neg x_1 \vee z$, $t[t_1/z] \equiv t[t_2/z]$

1.7 Erfüllbarkeit von Termen

Definition 13 (Erfüllbarkeit). Ein Term t wird erfüllbar genannt, wenn eine Belegung β existiert mit $I_\beta[t] = 1$.

Definition (Tautologie). Ein Term t wird allgemeingültig (Tautologie) genannt, wenn für jede Belegung β gilt: $I_\beta[t] = 1$.

Definition (Kontradiktion). Ein Term t wird Kontradiktion genannt, wenn für jede Belegung β gilt: $I_\beta[t] = 0$.

Bemerkung. Die drei Entscheidungsprobleme sind algorithmisch "gleich schwer":

t ist erfüllbar
$\Longleftrightarrow$ $\neg t$ ist keine Tautologie
$\Longleftrightarrow$ t ist keine Kontradiktion

Ein naiver Ansatz zur Lösung dieses Problems ist das Durchprobieren aller Belegungen; leider mit exponentieller Laufzeit (zur Anzahl der Variablen). (Es gibt Lösungen, die für

Zusatz #4 sehr viele, aber nicht alle Terme schneller sind.)

Bemerkung. Das Erfüllbarkeitsproblem ist Prototyp für NP-vollständige Probleme.

1.8 Boolesche Funktionen II

Definition 14 (Boolesche Funktion). Eine Funktion

$$f\colon \mathbb{B}^n \longrightarrow \mathbb{B}$$

($n \in \mathbb{N}$) wird Boolesche Funktion genannt.

Lemma 15. *Sei t ein Boolescher Term mit den freien Variablen $x_1, \ldots, x_n$. Jede Belegung $\beta\colon \{x_1, \ldots, x_n\} \longrightarrow \mathbb{B}^n$ entspricht dann eieneindeutig einem $(x_1, \ldots, x_n) \in \mathbb{B}^n$.*

Bemerkung. t definiert eine Boolesche Funktion f_t durch

$$f_t(b1, \ldots, b_n) = I_{\beta(b1,\ldots,b_n)}[t] = I_{(b1,\ldots,b_n)}[t]$$

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

$$\begin{aligned} f_t(0,0) &= 1 \\ f_t(0,1) &= 0 \\ f_t(1,0) &= 1 \\ f_t(1,1) &= 1 \end{aligned}$$

Bemerkung. $\boxed{t_1 \equiv t_2 \Longleftrightarrow f_{t_1} = f_{t_2}}$ Fehler #4

Beispiel. Boolesche Funktionen werden von Prozessoren ausgeführt:

- Addition – jedes Bit in der Summe zweier Zahlen ist Boolesche Funktion der Bits der Summanden.

- Speicherzugriff – Sind x_0 und x_1 der Inhalt zweier Speicherplätze und gibt y den Speicherplatz an, so läßt sich ein Zugriff auf die Speicherplätze realisieren mit Hilfe einer Booleschen Funktion:

$$f_t(y, x_0, x_1)$$
$$t = (\neg y \wedge x_0) \vee (y \wedge x_1)$$

  Dann gelten nämlich:

$$f_t(0, x_0, x_1) = x_0$$
$$f_t(1, x_0, x_1) = x_1$$

5. Nov. **Definition 16 (Literal).** Unter einem Literal versteht man eine Variable (Beispiele: $x_1, x_2, \ldots, x_n$) oder ihre Negation (Beispiele: $\neg x_1, \neg x_2, \ldots, \neg x_n$).

Definition 17 (DNF). Eine Disjunktion $t_1 \vee t_2 \vee \cdots \vee t_n$ heißt disjunktive Normalform (DNF), falls jeder Term t_i eine Konjunktion von Literalen ist.

Definition 18 (KNF). Eine Konjunktion $t_1 \wedge t_2 \wedge \cdots \wedge t_n$ heißt konjunktive Normalform (KNF), falls jeder Term t_i eine Disjunktion von Literalen ist.

Beispiele. Man bemerke, daß die beiden unteren Terme semantisch äquivalent sind.

$(x_1 \wedge \neg x_2) \vee x_1 \vee (\neg x_1 \wedge x_3 \wedge x_2)$	DNF
$(x_1 \vee x_3) \wedge (x_1 \vee \neg x_2)$	KNF
$(\neg x_1) \wedge (x_2) \wedge (\neg x_4)$	KNF
$(\neg x_1 \wedge x_2 \wedge \neg x_4)$	und auch DNF

Vereinbarung. Es sei $x_i^0 = \neg x_i$ und $x_i^1 = x_i$.

Definition 19 (Kanonische DNF). Sei $f\colon \mathbb{B}^n \longrightarrow \mathbb{B}$. Dann heißt

$$\mathrm{dnf}_f = \bigvee_{(b_1,\ldots,b_n)\in f^{-1}[\{1\}]} (x_1^{b_1} \wedge \cdots \wedge x_n^{b_n})$$

die kanonische DNF der Funktion. Fehler #5

Definition 20 (Kanonische KNF). Sei $f\colon \mathbb{B}^n \longrightarrow \mathbb{B}$. Dann heißt

$$\mathrm{knf}_f = \bigwedge_{(b_1,\ldots,b_n)\in f^{-1}[\{0\}]} (x_1^{\neg b_1} \vee \cdots \vee x_n^{\neg b_n})$$

die kanonische KNF der Funktion. Fehler #6

[Anmerkung: Achtung! In der Vorlesung wurde $f^{-1}(a)$ anstelle von $f^{-1}[\{a\}]$ geschrieben.]

Satz 21. *Jede Boolesche Funktion ist durch ihre kanonische KNF und durch ihre kanonische DNF darstellbar.*

Erläuterung. Beispiel: dnf_f einer Booleschen Funktion f. Wir betrachten die Konjunktion $K = x_1^{b_1} \wedge \cdots \wedge x_n^{b_n}$. Diese wird interpretiert für eine Belegung β, die den $(x_1, \ldots, x_n)$ bestimmte $(b_1', \ldots, b_n')$ zuordnet. Die Interpretation hat die Eigenschaft, daß sie genau dann 1 ist, wenn die $(b_1', \ldots b_n') = (b_1, \ldots, b_n)$, d.h.

$$I_\beta[K] = \begin{cases} 1 & \text{falls } (b_1', \ldots, b_n') = (b_1, \ldots, b_n) \\ 0 & \text{sonst} \end{cases}$$

Jede der Konjunktionen, die durch die Disjunktion in der kanonischen DNF "ausprobiert" werden, hat als $(b_1, \ldots, b_n)$ aber gerade einen der Werte, für die f wahr ist, denn sie werden ja aus $f^{-1}[\{1\}]$ gewählt. Für jedes Argumentstupel, für das f wahr ist, ist also auch genau eine der Disjunktionen in dnf_f wahr; für Argumentstupel, für die f nicht wahr ist, ist auch keine der Disjunktionen in dnf_f wahr. Also ist dnf_f für eine bestimmte Belegung genau dann wahr, wenn auch f wahr wäre. Die Funktion läßt sich durch ihre kanonische DNF darstellen.

[Anmerkung: Falls der letzte Absatz vollkommen unverständlich war: Bitte keine Sorgen machen. Der stammt auch gar nicht aus der Vorlesung, sondern aus meiner Feder. Jetzt geht es aber wieder im Originalton weiter.]

Sei beispielweise

$$(b_1, b_2, b_3, b_4) = (0, 1, 1, 0) \in f^{-1}[\{1\}]$$

Dann ist

$$\begin{aligned} K &= x_1^{b_1} \wedge x_2^{b_2} \wedge x_3^{b_3} \wedge x_4^{b_4} \\ &\equiv \neg x_1 \wedge x_2 \wedge x_3 \wedge \neg x_4 \end{aligned}$$

Ist nun $(b'_1, \ldots, b'_n)$ gerade $(0, 1, 1, 0)$, dann ist K wahr:

$$K \equiv \neg 0 \wedge 1 \wedge 1 \wedge \neg 0 \equiv 1 \wedge 1 \wedge 1 \wedge 1 \equiv 1$$

(In vereinfachter Schreibweise mit 1 für *true* und 0 für *false*.) Ändert man dagegen die Belegung, so ist diese Konjunktion
Fehler #7 immer falsch. Beispiel: $(b'_1, \ldots, b'_n) = (0, 1, 0, 0)$:

$$K \equiv \neg 0 \wedge 1 \wedge 0 \wedge \neg 0 \equiv 1 \wedge 1 \wedge 0 \wedge 1 \equiv 0$$

Die kanonische KNF funktioniert analog; während aber die DNF überprüft, ob das Argumentstupel in der Urbildmenge der 1 liegt, stellt die KNF sicher, daß das Argumentstupel nicht in der Urbildmenge der 0 liegt.

Beispiel. Betrachtet wird die Funktion $f\colon \mathbb{B}^2 \longrightarrow \mathbb{B}$ mit

$$\begin{array}{ll} f(0,0)=0 & f(1,1)=0 \\ f(0,1)=1 & f(1,0)=1 \end{array}$$

Dann ist $f^{-1}[\{1\}] = \{(0,1),(1,0)\}$ und die kanonische DNF lautet:

$$\begin{aligned} \mathrm{dnf}_f &= \bigvee\nolimits_{(b_1,b_2)\in f^{-1}[\{1\}]}(x_1^{b_1} \wedge x_2^{b_2}) \\ &= (x_1^0 \wedge x_2^1) \vee (x_1^1 \wedge x_2^0) \\ &= (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2^0) \end{aligned}$$

Beispiel. Keineswegs allerdings ist die kanonische Darstellung notwendigerweise besonders kurz. Man betrachte etwa die

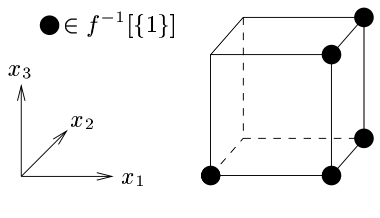


Abbildung 1.1: Darstellung des Graphen von f

Funktion, deren Graph in Abbildung 1.1 veranschaulicht ist. Die kanonische DNF hat fünf Konjunktionen zu je drei Literalen, also insgesamt 15 Literale. Dabei läßt sie sich kurz darstellen als $x_1 \vee (\neg x_2 \wedge \neg x_3)$!

1.9 Rechenstrukturen

Man betrachtet verschiedene Rechenstrukturen, die für unterschiedliche Aufgaben benötigt werden:

- Flächenberechnung – reelle Zahlen
- Sortieren – ganze/rationale/reelle Zahlen
- Anzahlbestimmungen – natürliche Zahlen
- Suchen eines Wortes in einem Text – Zeichenfolgen

Beispiel. Die Rechenstruktur NAT.

- Trägenmengen: $\mathbb{N}$, $\mathbb{B}$
- Funktionssymbole:
  - $\text{zero}: \longrightarrow \mathbb{N}$
  - $\text{succ}, \text{pred}: \mathbb{N} \longrightarrow \mathbb{N}$
  - $\text{add}, \text{sub}, \text{mult}, \text{div}: \mathbb{N}^2 \longrightarrow \mathbb{N}$
  - $\stackrel{?}{=}, \leq: \mathbb{N}^2 \longrightarrow \mathbb{B}$
  - not: $\mathbb{B} \longrightarrow \mathbb{B}$

- $\text{and}, \text{or}, \leq \colon \mathbb{B}^2 \longrightarrow \mathbb{B}$
- $true\colon \longrightarrow \mathbb{B}$
- $false\colon \longrightarrow \mathbb{B}$

[Anmerkung: Gibt's nicht eine schönere Schreibweise für Funktionen ohne Argumente? Könnte man vielleicht 0-Tupel einführen?]

[Anmerkung: Endlich hat mir jemand `\stackrel` genannt. Danke, Michael.]

1.10 Noch ein wenig Mathematik: Quantoren

[Anmerkung: Kann sein, daß hier einige Erklärungen fehlen. Der Handapparat ist Dein Freund.]

Definition 22 (Allquantor). Um die Allaussage "Für alle x gilt $P(x)$" auszudrücken, schreiben wir:

$$\forall x P(x)$$

Definition 23 (Existenzquantor). Um die Existenzaussage "Es gibt ein x mit $P(x)$" auszudrücken, schreiben wir:

$$\exists x P(x)$$

[Anmerkung: Wieso schreiben wir jetzt plötzlich $\forall x$ und $\exists x$, wenn wir eben noch $\bigwedge_x$ und $\bigvee_x$ geschrieben haben?]

Regeln. Es gelten die folgenden Äquivalenzen:

$$\begin{aligned}
\neg\forall x P(x) &\Longleftrightarrow \exists x \neg P(x)\\
\neg\exists x P(x) &\Longleftrightarrow \forall x \neg P(x)\\
\forall x \forall y P(x,y) &\Longleftrightarrow \forall y \forall y P(x,y)\\
\exists x \exists y P(x,y) &\Longleftrightarrow \exists y \exists y P(x,y)\\
\forall x (p(x) \wedge Q(x)) &\Longleftrightarrow \forall x P(x) \wedge \forall x Q(x)\\
\exists x (p(x) \vee Q(x)) &\Longleftrightarrow \exists x P(x) \vee \exists x Q(x)
\end{aligned}$$

Nichtregeln. Dagegen gilt i.A. nicht:

$$\begin{aligned}
\forall x (P(x) \vee Q(x)) &\Longleftrightarrow \forall x P(x) \vee \forall x Q(x) \quad (*)\\
\exists x (P(x) \wedge Q(x)) &\Longleftrightarrow x P(x) \wedge \exists x Q(x)\\
\forall x \exists y P(x,y) &\Longleftrightarrow \exists y \forall x P(x,y)
\end{aligned}$$

Beweis (*). Gegenbeispiel:

$$\forall (n \in \mathbb{N})(2 \mid n \vee 2 \nmid n)$$

aber keineswegs

$$\forall (n \in \mathbb{N})(2 \mid n) \vee \forall (n \in \mathbb{N})(2 \nmid n)$$

$\square$

Kapitel 2

Funktionale Programmierung in Haskell

Abbildung 2.1 soll den Weg vom algorithmischen Problem zum ausführbaren Programm illustrieren. Wir werden die funktionale Programmiersprache Haskell verwenden. Als Implementation von Haskell steht Hugs zur Verfügung. 10. Nov.

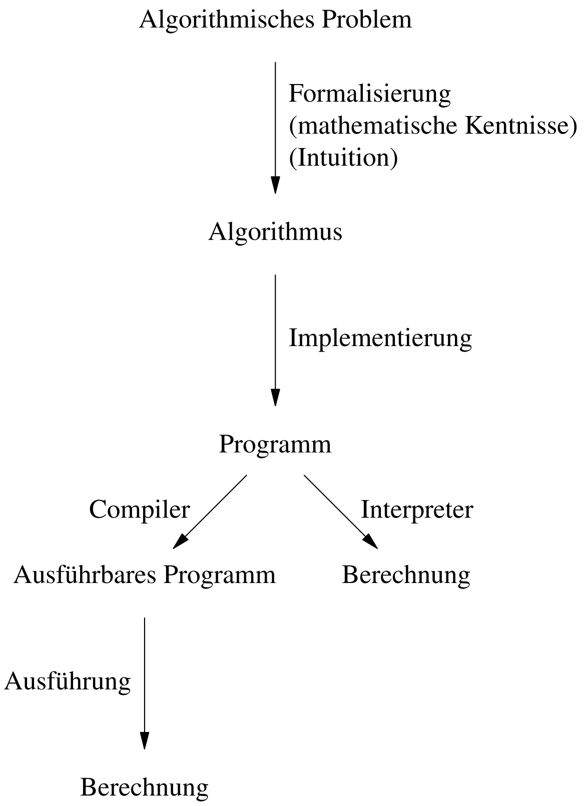


Abbildung 2.1: Schema für die Bearbeitung eines algorithmischen Problems

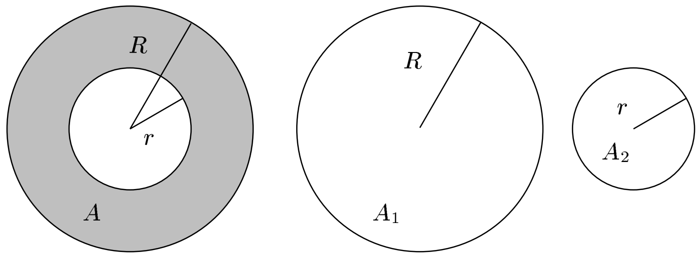


Abbildung 2.2: Fläche eines Kreisrings

2.1 Lauter Beispiele

Beispiel 1. Berechnung der Fläche eines Kreisrings mit Außenradius R und Innenradius r. Der Lösungsansatz ist in Abbildung 2.2 dargestellt. Man berechnet:

$$A = A_1 - A_2 = \pi R^2 - \pi r^2 = \pi(R^2 - r^2)$$

Algorithmus: Berechne R^2, r^2, $R^2 - r^2$, $\pi(R^2 - r^2)$.

Um ein Haskell-Programm zu schreiben, daß diese Berechnung durchführt, legt man etwa eine Datei `kreisringFl.hs` an. Die Abbildungen 2.3 und 2.4 zeigen mögliche Implementationen der Funktion `kreisringFl`.

Kommentare können entweder mehrzeilig zwischen `{- -}` oder einzeilig (`-- foo`) geschrieben werden. [Anmerkung: Einzeilige Kommentare sind aber zu bevorzugen!]

```
-----------------------------------------------------------
--                                                       --
--  kreisringFl.hs                                       --
--                                                       --
--  Berechnung der Flaeche eines Kreisrings mit          --
--  Aussenradius R und Innenradius r.                    --
--                                                       --
--  J. Random Luser                                      --
--  31.12.1999                                           --
--                                                       --
-----------------------------------------------------------


-- Funktion kreisringFl
--
-- (Float ist Typ fuer Gleitkommazahlen)
--
kreisringFl :: Float -> Float -> Float
kreisringFl R r = 3.14 * (R * R - r * r)
```

Abbildung 2.3: `kreisringFl.hs`

```
square :: Float -> Float
square x = x * x

kreisringFl :: Float -> Float -> Float
kreisringFl R r = 3.14 * (square R - square r)
```

Abbildung 2.4: `kreisringFl`: Zweite Fassung

Beispiel 2. Maximum von zwei ganzen Zahlen x, y. Algorithmus: Teste $x \geq y$. Falls erfüllt, gib x zurück; sonst y.

```
max :: Int -> Int -> Int
max x y
  | x >= y    = x
  | otherwise = y
```

Beispiel 3. Maximum von drei ganzen Zahlen.

```
threeMax :: Int -> Int -> Int -> Int
threeMax x y z
  | (x >= y) && (x >= z)  = x
  | (y >= z)              = y
  | otherwise             = z

-- Oder alternativ:
--
-- threeMax :: Int -> Int -> Int -> Int
-- threeMax x y z = max (max x y) z
```

Bemerkung. Wie in den Definitionen für `kreisringFl` und `threeMax` schon illustriert, beginnt man Funktionsnamen in Haskell mit kleinen Buchstaben. Ein weiteres Beispiel wäre `fourMax`. 12. Nov.

Beispiel 4a. Summe der Zahlen von 0 bis n unter Verwendung der Formel $\Sigma_{i=0}^{n} i = \frac{n(n+1)}{2}$. Die übergebene Anzahl wie der Rückgabewert sollen vom Typ `Int` sein, daher sollte nicht mit Floats gerechnet werden. `0.5 * n * (n + 1)` ist daher keine günstige Lösung. Ganzzahlige Division kann nur benutzt werden, wenn kein Rest übrigbleibt, so daß das Ergebnis falsch wird. $\left\lfloor \frac{n}{2} \right\rfloor (n+1)$ würde bei ungeraden, $n \left\lfloor \frac{n+1}{2} \right\rfloor$ bei geraden Zahlen falsch rechnen. Richtige Lösung ist aber

$$\left\lfloor \frac{n(n+1)}{2} \right\rfloor = \frac{n(n+1)}{2}$$

denn $n(n+1)$ ist gerade.

```
sumInt :: Int -> Int
sumInt n = div (n * (n + 1)) 2
```

Beispiel 4b. Summe der Zahlen von 0 bis n ohne Verwendung der obigen Formel. In diesem Fall addiert man in einer Iteration einfach alle natürlichen Zahlen von 0 bis n auf. In Haskell macht man das durch Rekursion. [Anmerkung: Was den Mathematikern bekannt vorkommen dürfte, denn so definiert man die Summenformel.] Zur Illustration, wie es in rein funktionalen Sprachen (und Haskell) *nicht* geht, siehe Abbildung 2.5.

Zur Erklärung der Rekursion definieren wir die Summen $s_n = \Sigma_{i=1}^{n} i$ derart, daß $s_0 =$ und $s_i = i + s_{i-1}$ für alle $i > 0$. Am Beispiel von s_4:

Beschreibung top down

$$\begin{aligned} s_4 &= 4 + s_3 \\ &= 4 + 3 + s_2 \\ &= 4 + 3 + 2 + s_1 \\ &= 4 + 3 + 2 + 1 + s_0 \\ &= 4 + 3 + 2 + 1 + 0 \end{aligned}$$

Auswertung bottom up

```
sumInt :: Int -> Int
sumInt n
  | n == 0    = 0
  | n > 0     = n + sumInt (n - 1)
  | otherwise = error "sumInt only defined for pos. Ints"
```

Beispiel 5. Berechnung des größten gemeinsamen Teilers (ggT) durch den Euklidischen Algorithmus. Dazu zunächst die mathematischen Grundlagen.

Definition (Teilbarkeit). Seien $a, d \in \mathbb{Z}$. Dann heißt a *teilbar durch* d falls es ein $c \in \mathbb{Z}$ gibt mit $a = cd$.

Satz (Divisionsalgorithmus). *Sei $a \in \mathbb{Z}$ und $d \in \mathbb{Z}^+$. Dann existieren eindeutig bestimmte $q, r \in \mathbb{Z}$, mit $a = qd + r$ und $0 \leq r < d$.*

Wir schreiben: $r = a \bmod d$ (r ist der Rest von a modulo d) und $q = \lfloor \frac{a}{d} \rfloor$ (q ist ganzzahliger Quotient aus a und d).

Definition (ggT). Seien $a, b \in \mathbb{Z}$. Dann heißt die Zahl $\mathrm{ggT}(a, b) = \max\{d \in \mathbb{Z} : d \mid a \wedge d \mid b\}$ der größte gemeinsame Teiler von a und b.

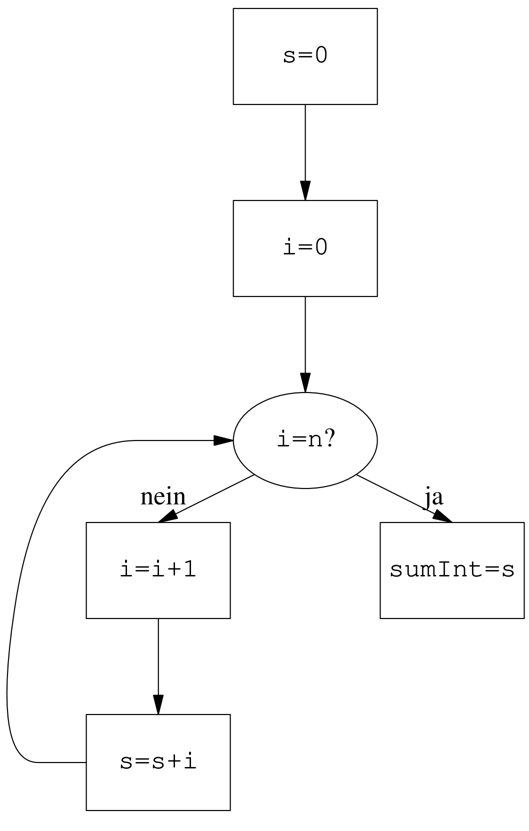


Abbildung 2.5: Iteration durch Sprung

Lemma 24. *Seien $a, b \in \mathbb{Z}^+$, $a > b$ und $r = a \bmod b$. Dann gelten:*

1. *Ist $r = 0$, dann $\mathrm{ggT}(a, b) = b$.*
2. *Ist $r > 0$, dann $\mathrm{ggT}(a, b) = \mathrm{ggT}(b, r)$.*

Beweis. Anhand der Definition der Teilbarkeit zeigt man:

1. Wegen $r = 0$ ist $a = qb + r = qb$, also $b \mid a$ und b ist der größtmögliche Teiler von b.

2. Beweisidee: Jeder gemeinsame Teiler von a und b ist auch gemeinsamer Teiler von b und r und umgekehrt:

$$\begin{aligned} d \mid a \wedge d \mid b &\Longrightarrow d \mid r \\ d \mid b \wedge d \mid r &\Longrightarrow d \mid a \end{aligned}$$

   Beweis des ersten Teils: $d \mid a \wedge d \mid b$, also gibt es a' und b', so daß $a = a'd$ und $b = b'd$, d.h.

$$\begin{aligned} & a = qb + r \\ \Longrightarrow\quad & a'd = qb'd + r \\ \Longrightarrow\quad & r = (a' - qb')d \\ \Longrightarrow\quad & d \mid r \end{aligned}$$

   Beweis der zweiten Teils analog: $d \mid b$ und $d \mid r$, es gibt also b' und r', so daß $b = b'd$ und $r = r'd$, also

$$a = qb + r = (qb' + r')d \Longrightarrow d \mid a$$

□

Wir setzen jetzt den Algorithmus in ein Programm um. Erster Ansatz:

```
-- Achtung!  Nur fuer a >= b > 0
--
ggT :: Int -> Int -> Int
ggT a b
  | (mod a b) == 0  = b
  | otherwise       = ggT b (mod a b)
```

Der Kommentar deutet es schon an: Dieser Ansatz funktioniert nicht einmal für alle $a \geq b \geq 0$, denn sogar der Fall $b = 0$ führt zum Fehler. Man schreibt stattdessen eine Funktion für geordnete a, b und eine darauf aufbauende, die sicherstellt, daß die Voraussetzung gegeben ist:

```
-- function ordggT for a => b => 0
--
ordggT :: Int -> Int -> Int
ordggT a b
  | b == 0          = a            -- neu
  | (mod a b) == 0  = b
  | otherwise       = ordggT b (mod a b)

-- function ggT for arbitrary integers a, b
--
ggT :: Int -> Int -> Int
ggT a b
  | (abs a) >= (abs b)  = ordggT (abs a) (abs b)
  | otherwise           = ordggT (abs b) (abs a)
```

Satz. *Für beliebige* $a, b \in \mathbb{Z}^+$ *gibt es* $s, t \in \mathbb{Z}$ *mit*

$$\mathrm{ggT}(a, b) = sa + tb$$

[Anmerkung: Kann mir irgendwer verraten, wozu das gut sein soll?]

Beweis. [Anmerkung: Naja, sowas in der Art jedenfalls. Wir zeigen einen Algorithmus am Beispiel und sagen, daß der auch im Allgemeinen funktioniert.] Umkehrung des Euklidischen Algorithmus am Beispiel von $\mathrm{ggT}(371, 168)$:

$$\begin{array}{rlrl}
371 &= 2 \cdot 168 + 35 & 35 &= 371 - 2 \cdot 168 \\
168 &= 4 \cdot 35 + 28 & 28 &= 168 - 4 \cdot 35 \\
35 &= 1 \cdot 28 + 7 & 7 &= 35 - 28 \\
28 &= 4 \cdot 7 + 0 & &
\end{array}$$

$$\begin{array}{lrl}
\Longrightarrow \mathrm{ggT}(371, 168) = 7 & 7 &= 35 \cdot 28 \\
& &= 35 - (168 - 4 \cdot 35) \\
& &= 5 \cdot 35 - 168 \\
& &= 5(371 - 2 \cdot 168) \cdot 168 \\
& &= 5371 - 11 \cdot 168
\end{array}$$

Also $s = 5$, $t = -11$. [Anmerkung: Empfohlene Lektüre: `http://www.cs.nwu.edu/~riesbeck/proofs.html`] □

[Anmerkung: Und noch eine abschließende Bemerkung: Einige von euch sind vielleicht schockiert von Haskells (für mit C, Java oder <insert-your-favourite-portable-macro-assembler-or-pseudo-OOP-language-here> vertraute Menschen) eher ungewohnten Eigenschaften. Zur Beruhigung: Keiner zwingt euch, eine rein funktionale Sprache später einzusetzen. Lohnenswert dürfte es allerdings sein, sich eine Sprache anzusehen, die mehrere Programmierstile zur Auswahl anbietet (und am Besten sogar erweiterbar ist). Ich frage mich sowieso, warum wir nicht gleich eine solche Sprache verwenden. Da kann man die Programmierstile schön vergleichen. Auf der Webseite zum Skript ist ein Link zu SICP, ein Buch, das die Sprache Scheme verwendet. Oder seht euch mal Dylan an.]

2.2 Fundamentale Datentypen

[Anmerkung: So ein paar Typen und Funktionen kennenzulernen, mag ganz praktisch sein, aber wenn ihr sie mal genau nachsehen wollt, das hier steht natürlich alles im Haskell 98-Report, insbesondere Kapitel 6: "Predefined Types and Classes". Auf der Webseite zum Skript ist ein Link zum Haskell 98-Report und Library-Report. Oder Kapitel 6 direkt unter: `http://www.haskell.org/onlinereport/basic.html`.]

2.2.1 Boolesche Werte

- Bezeichnung: `Bool`
- Werte: `True`, `False`
- Operationen: `not` (Negation), `&&` (Konjunktion), `||` (Disjunktion)

```
--
-- Beispielsdefinitionen fuer neue Operationen.
--

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

-- oder:
--
exOr True True = False
exOr True False = True
exOr False True = True
exOr False False = False

-- oder:
--
exOr True x = not x
exOr False x = x
```

2.2.2 Ganze Zahlen

- Bezeichnung: `Int`

- Werte: $-2^{29}, \ldots, 2^{29} - 1$ [Anmerkung: In der Vorlesung wurde das anders gesagt. Dies ist aber die Mindestanforderung von Haskell 98; es kann mehr sein, nur nicht weniger. Je nach Architektur sollte der Exponent heutzutage eigentlich 31 oder 63 sein; es sind vermutlich nur 29 gefordert, weil heutige Architekturen Sprachen wie Haskell leider nicht richtig unterstützen und man daher einige Bit abzweigen muß.]

- Infix-Operatoren: `+`, `-`, (Gibt es auch als Funktionen; dann Präfix: `a+b` $\equiv$ `(+) a b`).

Fehler #8

- Präfix:

`div a b`	$\left\lfloor \frac{a}{b} \right\rfloor$
`mod a b`	$a \bmod b$
`abs a`	$\lvert a \rvert$
`negate a`	$-a$
`- a`	$-a$

  (Es geht auch umgekehrt wieder Infix, etwa: `a ‘div‘ b`.)

- Vergleichsoperatoren: `>`, `>=`, `==`, `/=`, `<`, `<=`.

2.2.3 Gleitkommazahlen

- Bezeichnung: `Float`

- Bedeutung: Mit gewisser Genauigkeit können das endliche Dezimalbrüche sein (24.381, -0.014, 28.0) oder Zahlen in wissenschaftlicher (scientific) Notation (0.1835e6).

- Operatoren: `+`, `-`, `*`, `/`, `**` (alle `Float -> Float -> Float`), `^` (`Float -> Int -> Float`).

- Vergleichsoperatoren: Wie schon bei `Int`, nur eben `Float -> Float -> Bool`.

- Weitere Funktionen: `abs`, `sin`, `asin` (arcsin), `exp`, `log` (ln), `sqrt`
- Runden/Typkonversionen: `ceiling` (gegen $+\infty$), `floor` (gegen $-\infty$); alle `Float -> Int`. `fromInt :: Int -> Float`, etwa `fromInt 27 == 27.0`, und `signum :: Float -> Float`. Fehler #9

2.2.4 Zeichen

- Bezeichnung: `Char`
- Werte: "Zeichen auf der Tastatur" [laut Vorlesung]. Beispiel: `'x'`, `'1'`. Außerdem Sonderzeichen:

`'\t'`	tabulator
`'\n'`	Newline [Formfeed?]
`'\\'`	Backslash
`'\''`	Single (Forward) Quote
`'\"'`	Double Quote

- Kodierung: Die meisten Implementationen verwenden ASCII (American Standard Code for Information Interchange). Haskell 98 spezifiziert Unicode. Anhand dieser Kodierungen können Zeichen als Zahlen interpretiert werden. Beispiele: Die Ziffern $0, \ldots, 9$ entsprechen $48, \ldots, 57$; die Kleinbuchstaben a, ..., z entsprechen $97, \ldots, 122$. Funktionen zur Umwandlung: `ord :: Char -> Int`, `chr :: Int -> Char`. Fehler #10

- Vergleichsoperatoren: Schon bekannt. Beispiel: `'a'<='b'`.
- Prädikate. Zum Beispiel: `isDigit`.

2.2.5 Tupel

Mit Tupeln bezeichnet man zusammengesetzte Typen; etwa was man in Pascal Record, in C Structure nennt. Man bezeichnet den Typ mit `(t1,t2,t3,...,tn)` (mit Typen `ti`) und ein bestimmtes Tupel dieses Typs mit `(v1,v2,v3,...,vn)` (mit konkreten Werten `vi`).

[Anmerkung: Achtung! In der Vorlesung wurde `typ name definition` geschrieben. Ich weiß nicht, was das sein soll, aber gültiges Haskell ist es nicht. Daher werde ich `type name = definition` schreiben. Sollte sich das doch als falsch herausstellen, findet ihr dann einen Hinweis auf der Webseite.]

```
type Point2 = (Float, Float)
type ShopItem = (String, Int)
```

Nach dieser Definition könnte man etwa das `ShopItem` für Butter `("Butter, 250g", 199)` verwenden.

Der Zugriff auf den Tupelinhalt erfolgt durch Funktionen. Die kann man selbst definieren oder man kann (zum Beispiel für Paare) eingebaute Funktionen benutzen:

```
name :: ShopItem -> String
name (n, p) = n

```

```
price :: ShopItem -> Int
price (n, p) = p

-- oder auch:
-- fst (x, y) == x
-- snd (x, y) == y
```

Tupel sind zum Beispiel praktisch, um einige Algorithmen besser in Haskell formulieren zu können; für viele Rekursionen braucht man etwa Tupel, wenn sie effizient sein sollen.

Fibonacci-Zahlen Wieviele Paare von Kaninchen gibt es zu einem Zeitpunkt n, wenn es zum Zeitpunkt 1 genau ein Paar gibt, jedes Paar nach einem Monat geschlechtsreif wird, jeden Monat ein weiteres Paar zeugt und die Kaninchen niemals sterben? Aus dieser Frage ergeben sich die FIBONACCI-Zahlen: $\boxed{a_0 = 0, a_1 = 1}$ und $\boxed{a_n = a_{n-1} + a_{n-2}}$ Fehler #1

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

Diese Rekursion ist leider ineffizient. Merkt man sich immer zwei Werte gleichzeitig, so geht es auch besser: Fehler #2

```
-- ###: Meine Ergaenzungen.

```

```
fibStep :: (Int, Int) -> (Int, Int)
fibStep (a, b) = (b, a + b)

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

fib :: Int -> Int                       -- ###
fib n = fst (fibPair n)                 -- ###
```

Fehler #3

2.2.6 Strings

Strings (Zeichenketten) sind ein Beispiel für Listen. Alle Funktionen für Listen funktionieren auch mit Strings. Zu Listen später mehr.

Man schreibt Strings in "normalen" Anführungsstrichen und kann darin auch Sonderzeichen verwenden. Beispiele:

```
"Alice"
"\104a\116" ("hat")
" gewonnen gegen:\nBob\tCharles\tDave"
"\n1:0\t2:1\t5:2"
```

Man kann Strings ausgeben:

```
-- putStr :: String -> IO()

putStr "\104a\116"
-- gibt "hat" aus
```

Mit Hilfe des Operators `++` (Concatenation) kann man Strings aneinanderhängen:

```
> putStr ("Alice "++"\104a\116"++" gewonnen gegen:"
          ++"\nBob\tCharles\tDave\n1:0\t2:1\t5:3")
Alice hat gewonnen gegen:
Bob     Charles Dave
1:0     2:1     5:3
```

2.3 Zusammengesetzte Datentypen

Tupel und Strings sind Beispiele für zusammengesetzte Datentypen. Allgemein möchte man man mehrere Daten (eventuell unterschiedlichen Typs) in einer größeren Einheit zusammenfassen. In Haskell kann man Objekte verschiedener Typen, aber in fester Größe (Punkte im $\mathbb{R}^d$; Profil eines Kunden; Auswertung eines Fragebogens) zusammenfassen, oder Objekte gleichen Typs variabler Anzahl (Meßpunkte auf einer Landkarte; Kundendatei; Umfrageergebnis). Tupel sind ein Beispiel für die erste Möglichkeit, Listen für die zweite.

2.4 Bindungsstärken der Operatoren

Damit man Klammern einsparen kann, sind Bindungsstärken zwischen 0 und 9 definiert:

9 (selbst definierte Funktionen)

8 `**`, `^` (rechtsassoziativ: `a^b^c` $= a^{b^c}$)

7 `*`, `/`, `` `div` `` (linksassoziativ: `a/b/c` $= \frac{a}{b}/c$)

6 `+`, `-` (linksassoziativ)

5

4 Vergleichsoperatoren

3 `&&`

2 `||`

Beispiel. Klammern des folgenden Ausdrucks:

```
x + 3 * xy < 20 || x == 5 && x ^ 2 + 4 < 100
(x + 3 * xy < 20) || (x == 5 && x ^ 2 + 4 < 100)
((x + 3 * xy) < 20) || ((x == 5) && (x ^ 2 + 4 < 100))
((x + (3 * xy)) < 20) || ((x == 5) && ((x ^ 2 + 4) < 100))
((x + (3 * xy)) < 20) || ((x == 5) && (((x ^ 2) + 4) < 100))
```

Nachtrag

24. Nov. Anhand des Programms zur FIBONACCI-Zahlen-Berechnung, das auf Seite 43 schon dargestellt ist, soll illustriert werden, wie man die Komposition zweier Funktionen in Haskell realisieren kann.

```
-- Variante von Seite 44:
--
-- fastFib :: Int -> Int
-- fastFib n = fst (fibPair n)
```

```

-- Komposition von Funktionen
--
fastFib :: Int -> Int
fastFib = fst . fibPair
```

Fehler #12

Effizienz der Funktionen

Hier gehört ein Abschnitt zur Effizienz der beiden `fib`*-Fassungen hin. Leider werde ich trotz zusätzlicher Erklärungen von Herrn Kriegel nicht schlau aus meiner Mitschrift. Daher fehlt das hier noch. Glücklicherweise allerdings kommt es auf die Reihenfolge der Abschnitte wohl nicht an, und ich kann diesen einfach später nachholen.*

2.5 Listen

Am Beispiel der Strings wurden Listen schon eingeführt. Eine Liste ist eine (endliche) Folge von Daten gleichen Typs. Für einen Typ `t` bezeichnet `[t]` den Typ der Liste von Daten dieses Typs.

Beispiele. Listenbeispiele und dazugehöriger Typ:

```
[3, 2, 12, 5] :: [Int]
[3.14] :: [Float]
[] :: [Int]
[mod, div] :: [Int -> Int]
```

```
-- String ist Synonym fuer [Char]
"abc" :: String -- das Gleiche wie ['a', 'b', 'c']
```

Für Typen mit Aufzählung wie `Int` oder `Char`:

```
[2 .. 7] = [2, 3, 4, 5, 6, 7]
['i' .. 'n'] = ['i', 'j', 'k', 'l', 'm', 'n']
[3, 5 .. 10] = [3, 5, 7, 9]
[8.1 .. 11] = [8.1, 9.1, 10.1]
[7.9, 8.5 .. 9.5] = [7.9, 8.5, 9.1]
```

Listenmanipulation. Neue Listen können aus bestehenden Listen erzeugt werden. Dabei kann man bestimmte Elemente auswählen und diese durch eine Funktion auf die Elemente der neuen Liste abbilden.

1. Sei beispielsweise `li=[2, 4, 7]`. Dann ergibt

   ```
   [2 * n | n <- li]
   ```

   die Liste `[4, 8, 14]`. Diese Schreibweise wird analog zur in der Mathematik für Mengen verwendeten Schreibweise $\{2k \mid k \in A\}$ gebraucht.

2. Ist `isPrime :: Int -> Bool` eine Prädikat für Primzahlen, so ist

   ```
   [2 * n | n <- li, isPrime n]
   ```

   die Liste `[4, 14]`.

3. `[isPrime n | n <- li]` liefert `[True, False, True]`.

4. Anstelle von Variablen können auch andere Muster (Patterns) verwendet werden:

```
type Package = (Int, Int) -- (Anzahl, Gewicht/g)

totalWeight :: [Package] -> [Int]
totalWeight la = [n * w | (n, w) <- la]
-- ist das Gleiche wie:
totalWeight la = [(fst p) * (snd p) | p <- la]
```

Beispiel. Eine einfache Implementation einer Inventarliste mit dazugehörigen Funktionen ist ein Abbildung 2.6 dargestellt.

Funktionen Funktionen zur Listenmanipulation:

- `: :: a -> [a] -> [a]` 26. Nov.
  Element an den Anfang der Liste stellen
  Beispiel: `4 : [3, 5, 6]` → `[4, 3, 5, 6]`

- `++ :: [a] -> [a] -> [a]`
  Listen aneinanderhängen
  Beispiel: `"xyz"++"67"` → `"xyz67"`

- `!! :: [a] -> Int -> a`
  Zugriff auf ein Listenelement[1]
  Beispiel: `!! "Gummibaum" 5` → `'b'`

[1] Das erste Element hat den Index 0, ebenso wie ihr in eurem *ersten* Lebensjahr *Null* Jahre alt wart. (Ordinal- vs. Kardinalzahl.)

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

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

-- Funktion inWhichRooms
-- Gibt Liste der Raeume mit dem Objekt zurueck
inWhichRooms :: Database -> Object -> [Room]
inWhichRooms db obj = [r | (i, r, o) <- db, o == obj]

-- Funktion where
-- Wo befindet sich die Ware mit diesem Index?
where :: Database -> Index -> Room
where db k = head [r | (i, r, o) <- db, k == i]

-- Funktion where
-- Welches Objekt hat diesen Index?
where :: Database -> Index -> Object
where db k = head [o | (i, r, o) <- db, k == i]
```

Abbildung 2.6: Inventarliste

- `concat :: [[a]] -> [a]`
  Aneinanderhängen der Elemente
  `concat [[1, 2], [], [5, 7, 8]]` → `[1, 2, 5, 7, 8]`

- `length :: [a] -> Int`
  Länge der Liste
  Beispiel: `length "Baum"` → 4

- `head`, `last :: [a] -> Int`
  Erstes/letztes Element der Liste
- `tail`, `init :: [a] -> [a]`
  Gibt eine Liste ohne das erste/letzte Element zurück
  Beispiel: `tail "Baum"` → `aum`
  Beispiel: `init "Baum"` → `Bau`
- `replicate :: Int -> a -> [a]`
  "Vervielfachung" von einem Element zu einer Liste
  Beispiel: `replicate 3 'a'` → `"aaa"`

Fehler #13, #15

- `take :: Int -> [a] -> [a]`
  Gibt die ersten n Elemente der Liste zurück
  Beispiel: `take 4 "Gummibaum"` → `"Gumm"`
- `drop :: Int -> [a] -> [a]`
  Gibt alle außer den ersten n Elementen der Liste zurück
  Beispiel: `drop 4 "Gummibaum"` → `"ibaum"`
- `splitAt :: Int -> [a] -> ([a], [a])`
  Beispiel: `splitAt 3 "Baum"` → `("Bau", "m")`
- `reverse :: [a] -> [a]`
  Beispiel: `reverse "abcx"` → `"xcba"`
- `zip :: [a] -> [a] -> ([a, b])`
  Beispiel: `zip [1, 3, 7] "axcbd"`
  → `[(1, 'a'), (3, 'x'), (7, 'c')]`
- `unzip :: [(a, b)] -> ([a], [b])`
  Beispiel: `unzip [(1, 'a'), (3, 'x'), (7, 'c')]`
  → `([1, 3, 7], "axc")`

2.6 Polymorphie

Unter Polymorphie (Vielgestaltigkeit) versteht man ein Konzept, das es erlaubt, eine Funktion für verschiedene Typen zu verwenden.

Polymorphe (generische) Funktionen verwenden die gleiche Definition (den gleichen Algorithmus) für verschiedene Typen. (Beispiele sind die genannten Listenfunktionen sowie `fst` und `snd` für Paare).

[Anmerkung: Also, mit Haskell kenne ich mich nicht aus, aber in objektorientierten Sprachen wie CLOS oder Dylan, die Multimethods unterstützen, verwendet man den Begriff *Generic Function* genau umgekehrt.]

Das Sprachkonzept von Haskell unterstützt dieses Konzept durch Typvariable. Das geht so weit, daß Typangaben bei der Definition von Funktionen nicht notwendig sind. Als Beispiel betrachte man die Funktion `fib` ohne Typdeklaration:

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

Dann ergeben:

```
fib 4.0 ........... 3
fib (4.2 - 2.2) ... 1
fib (4.1 - 2.1) ... Fehler!
```

Die Funktion, die für `Ints` geschrieben wurde, funktioniert also auch mit ganzzahligen Gleitkommazahlen gut, obwohl sie dafür gar nicht gedacht war. Übergibt man aber eine nichtganzzahlige Gleitkommazahl, versagt sie (eigentlich erwartungsgemäß, aber vielleicht doch plötzlich).

[Anmerkung: Das Problem ist hier, daß 0.1 mit nur endlich vielen Stellen im Binärsystem nicht dargestellt werden kann ($0.1_{10} = 0.000\overline{1100}_2$). Rechnet man mit Doubles, so ist

$$\mathrm{rd}(4.1) \tilde{-} \mathrm{rd}(2.1) = 2 - 2^{-51} \neq 2$$

Das haut dann natürlich nicht ganz hin.]

Ein Beispiel für eine Funktion, bei der das Konzept sehr gut aufgeht, ist die Identität: `id x = x`.

Vor- und Nachteile im Vergleich:

- Vorteile:
  - Kompaktifizierung des Programmcodes
  - Wiederverwendbarkeit von Funktionen
- Nachteile:
  - Bei Typerkennung bei Laufzeit können Fehler an unerwarteten Stellen auftreten.

[Anmerkung: Also, bei mir liest sich dieser Vergleich so: Statische Sprachen wie C *zwingen* den Programmierer zu Premature Optimization.]

Ähnliche Eigenschaften wie Polymorphie treten auch beim Überladen von Operatoren auf, z.B. bei `+` oder `<=` für `Int`, `Float` und `Char`. Der Unterschied: Hier kommen verschiedene Algorithmen zur Anwendung.

[Anmerkung: Ein Artikel von Kent M Pitman geht auf den Unterschied zwischen dem Typ der Repräsentation eines Objekts und dem Typ, den das Objekt sinngemäß hat, ein. Hauptaussage des Textes ist, das eine Funktion zwar für unterschiedliche Typen verschiedene Algorithmen verwenden kann, aber eine einheitliche Aufgabe haben muß. Wer noch nicht zu verwirrt ist, sollte sich das ganze mal durchlesen: `http://world.std.com/~pitman/PS/EQUAL.html`.]

Monomorphe Listenoperationen. Beispiele:

- `and [Bool] -> Bool` (Konjunktion aller Listenelemente)
- `or [Bool] -> Bool` (Disjunktion aller Listenelemente)
- `sum [Int] -> Int` (Summe)
- `sum [Float] -> Float`
- `product [Int] -> Int` (Produkt)
- `product [Float] -> Float`

2.7 Pattern-Matching

Listenfunktionen sollen mit Hilfe von Pattern-Matching definiert werden. Am Beispiel von Paaren wurde ähnliches bereits durchgeführt. Muster (Patterns) sind Ausdrücke zur Darstellung der Argumente von Funktionen.

[Anmerkung: Apropos “Pattern Matching”: Wenn ihr mal eine Programmiersprache sehen wollt, die *wirklich* anders funktioniert als andere, dann seht euch Prolog an.]

Bisher wurden feste Werte (`True`, `'a'`) oder Variablen (`x`, `count`) oder auch Tupel-Pattern (`(x, y)`) verwendet.

Nun sollen Listen-Patterns verwendet werden: Es gibt die leere Liste `[]` und man kann den Konstruktor `:` verwenden. Die Funktion `:` hängt ja ein Element an der Anfang einer Liste. Man kann nun `x:xs` schreiben, um mit `x` das Kopfelement (head) und mit `xs` den Rest der Liste (tail list) zu bezeichnen.

Beispiel. Die Elementfunktion prüft, ob ein Element vom Typ `t` in einer Liste vom Typ `[t]` enthalten ist.

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

-- oder ausgeschrieben:
--
elem x []     = False
elem x (x:ys) = True
elem x (y:ys) = elem x ys
```

Fehler #11

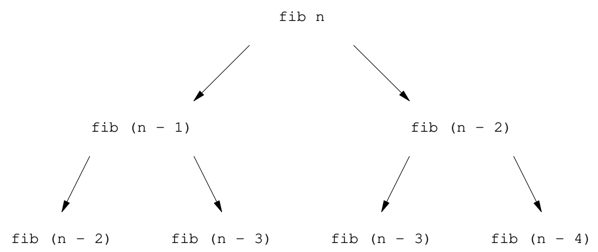
Dabei wird das erste passende Pattern ausgewählt. Die restlichen werden ignoriert.

2.8 Effizienz

[Anmerkung: Ich hoffe, das stimmt jetzt so.]

(24. Nov.) Die Effizienz der ersten Fassung von `fib` und der neuen Funktion `fastFib` sollen miteinander verglichen werden. Die Abschätzung wird anhand der Anzahl der Funktionsaufrufe durchgeführt.[2]

Ineffiziente Fassung von Seite 43. Diese Funktion ruft sich nach dem folgenden Schema auf:



Bezeichne b_k die Anzahl der Aufrufe von `fib k`. Wie man aus der Definition der Funktion `fib` sieht, werden (bei $k \geq 2$) zu jedem Aufruf von `fib k` sowohl `fib (k - 1)` als auch `fib (k - 2)` aufgerufen, d.h. `fib (k - 2)` bekommt seine Aufrufe von `fib (k - 1)` und von `fib k`.

Allgemein gilt also

$$b_{n-i} = b_{n-i+1} + b_{n-i+2} = b_{n-(i-1)} + b_{n-(i-2)}$$

[2] Ich gestatte mir, hier einige Formulierungen halbwörtlich von Herrn Kriegel zu borgen. Vielen Dank für seine Erklärungen.

für alle i von 2 bis $n-1$. Zusammen mit dem Anfang

$$b_n = b_{n-0} = 1$$

und $b_{n-1} = 1$ erfüllt b_{n-i} die Bedingungen der FIBONACCI-Definition (nur der Index ist um 1 verschoben; die FIBONACCI-Zahlen beginnen mit $0, 1, 1, \ldots$).

Letzendlich erhalten wir $b_1 = b_{n-(n-1)} = f_n$,[3] und schon die Anzahl der Aufrufe von `fib n` bis `fib 1` ist

$$\sum_{i=1}^{n} f_i$$

Effiziente Fassung von Seite 43. Die Aufrufe verlaufen hier folgendermassen:

```
      fastFib n       ——→   fibPair n
→ fastFib (n - 1) ——→   fibPair (n - 1)
→ fastFib (n - 2) ---→
```

Die Anzahl der Aufrufe ist also

$$2n + 2$$

Auswertung. Oben sind die verschiedenen `Fib`-Fassungen der Übersicht halber bereits als "ineffizient" und "effzient" bezeichnet. Die Bewertung aber natürlich erst anhand eines Vergleiches der ermittelten Abschätzungen möglich.

[3] Hier ist f_n die n-te FIBONACCI-Zahl.

Die Abschätzung der zweiten Fassung ist klar. Bei der ersten Fassung kann man anhand der Monotonie erkennen, daß

$$\left.\begin{array}{r} f_n = f_{n-1} + f_{n-2} \leq 2f_{n-1} \\ f_1 = 1 \end{array}\right\} \Longrightarrow f_n \leq 2^{n-1}$$

und

$$\left.\begin{array}{r} f_n = f_{n-1} + f_{n-2} \geq 2f_{n-2} \\ f_1 = f_2 = 1 \end{array}\right\} \Longrightarrow f_n \geq 2^{\lfloor \frac{n-2}{2} \rfloor}$$

Also $2^{\lfloor \frac{n-2}{2} \rfloor} \leq f_n \leq 2^{n-1}$.

Man kann beweisen, daß

$$f_n \sim \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^n$$

bzw.

$$\lim_{n \longrightarrow \infty} \sqrt[n]{\sqrt{5} f_n} = \left(\frac{1+\sqrt{5}}{2}\right)$$

2.9 Sortieren

Eine Liste soll sortiert werden. Wir besprechen die Algorithmen *Insertion Sort* und *Quicksort*. 1. Dez.

2.9.1 Insertion Sort.

Insertion Sort besteht aus den folgenden Schritten:

- erstes Element herausnehmen
- Restliste wird rekursiv sortiert
- Einordnen des ersten Elements an der richtigen Stelle

Die ersten beiden Schritte sind mit bekannten mitteln einfach zu lösen, der dritte wird realisiert durch die Funktion:

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

Beispiel. Ein Aufruf von `insert`:

```
   insert 3 [0, 2]
↪  0:(insert 3 [2])
↪  0:(2:[3])
↪  0:[2, 3]
↪  [0, 2, 3]
```

Vollständige Funktion für Insertion Sort:

```
iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = insert x (iSort xs)
```

2.9.2 Quicksort.

Vorgehensweise:

- Erstes Element herausnehmen.
- Erzeuge Teilliste der Elemente kleiner-gleich und Teilliste der Elemente größer-gleich zum ersten Element.
- Sortiere Teillisten rekursiv.
- Ordne erstes Element zwischen den sortierten Teillisten an und setze zusammen.

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

Beispiel. Ein Aufruf von `qSort`:

```
    qSort [5, 7, 1, 3]
↪  qSort [1, 3] ++ [5] ++ qSort [7]
↪  (qSort [] ++ [1] ++ qSort [3])
     ++ [5]
     ++ (qSort [] ++ [7] ++ qSort [])
↪  ... ++ (qSort [] ++ [3] ++ qSort []) ++ ...
↪  [1, 3, 5, 7]
```

2.9.3 Analyse der Algorithmen

Die Komplexität von Algorithmen dieser Art anzugeben ist nicht so einfach möglich wie bei der Berechnung der FIBONACCI-Zahlen, denn sie hängt von den erwarteten Eingaben ab.

Man kann verschiedene Analysen durchführen:

- Analyse des schlechtesten Falls (worst case)
- Durchschnittsanalyse (average case)

Die Analyse der durchschnittlichen Komplexität ist ohne geeignete mathematische Hilfsmittel nicht präzise durchzuführen, also untersuchen wir nur den schlechtesten Fall exakt und stellen eine (richtige) Vermutung über das durchschnittliche Ergebnis an.

Insertion Sort: worst case. Im schlechtesten Fall ist die Liste absteigend sortiert und bereits die `insert`-Funktionen benötigen $(n-1)(n-2)(n-3)+\cdots+1$ Aufrufe:

Im schlimmsten Fall liegt also ein Gesamtaufwand vor, der mindestens

$$\Sigma_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

beträgt (also quadratisch wächst).

Quicksort: worst case. Die Erzeugung der Teillisten mit $y \leq x$ und $y > x$ erfordert Aufwand, der jeweils der Listenlänge entspricht.

Im schlechtesten Fall ist die Liste auf- oder absteigend sortiert. Eine Teilliste ist dann leer; die andere enthält den gesamten Rest.

$$\begin{aligned}\text{Gesamtaufwand} &\geq 2(n-1) + 2(n-2) + \cdots + 2 + 1 \\ &= n(n-1)\end{aligned}$$

Also wieder ein quadratischer Gesamtaufwand.

[Anmerkung: Ja, es heißt $\geq$, nicht $\leq$, denn gemeint ist der Gesamtaufwand im schlechtesten Fall. Wir werden später eine exakte Notation für sowas kennenlernen.]

Durchschnittsanalyse. Wir nehmen an, daß ein Element “wahrscheinlich” in der Mitte der Liste liegt und erwarten:

- Insertion Sort: quadratische Laufzeit $\frac{n(n-1)}{4}$
- Quicksort: $n \log n$

2.10 Funktionen auf Listen

Bisher haben wir per Hand Funktionen geschrieben, die folgende Listenoperationen verwenden:

- `t` sei ein Typ, `f :: t -> t` und `xs :: [t]`. Dann kann man die Liste der Bilder der `xs` unter `f` erzeugen durch: `[ f x | x <- xs ]`.
- `t` ein Typ, `p :: t -> Bool` und `xs :: [t]`. Dann kann man die Liste der Elemente von `xs`, die die Bedingung `p` erfüllen, erzeugen durch: `[ x | x <- xs, p x]`.

Dies sind Beispiele für Listenoperationen, die man sehr oft durchführen möchte. Man unterscheidet: 3. Dez.

- Abbildungen (Mapping)
- Auswahl (Filtering)
- "Zusammenfügen" aller Elemente einer Liste durch eine Verknüpfung (Faltung, Folding) der Art `(op) :: t -> t -> t` (wie z.B. `(+) :: Int -> Int -> Int` zur Bildung der Summe aller Elemente).
- Primitive Rekursion als Spezialfall der Faltung
- Aufspalten von Listen (Filtern bis zu einer bestimmten Stelle)

Diese Prinzipien können für verschiedenste Listen und Funktionen (Eigenschaften/Operatoren) verwendet werden. Bisher haben wir immer von neuem eine Funktion per Hand geschrieben. Nun wollen wir sprachliche Mittel verwenden, die diesen Prozeß automatisieren. Funktionen dazu sind `map`, `filter`, `foldr1`, `foldr`. (Dabei steht das `r` für "rechts"; ensprechend gibt es auch `foldl1` und `foldl`.

Diese Funktionen haben als Argumente Listen *und Funktionen.*

Definition. Funktionen, die Funktionen als Argument haben, werden *Funktionen höherer Ordnung* genannt.

Die Funktion `map`. Definition:

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

-- oder: (so definiert in Prelude)
--
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

Beispiele zu `map` in Abbildung 2.7.

Die Funktion `filter`. Definition:

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

-- oder: (so definiert in Prelude)
--
```

```
 o Kubikzahlen


cube :: Int -> Int
cube n = n * n * n

-- berechne Liste der ersten 100 Kubikzahlen:
map cube [1 .. 100]


 o Plausibel machen von Grenzwerten:

--
-- sin'(a) -> 1  (a -> 0)
--

dq :: Float -> Float
dq x = (sin x) / x

-- Testen f"ur Folge, die gegen Null geht:
dq1 y = y * sin (1 / y)

> map dq1 [1.0 .. 100.0]
[..., 0.99...]


 o Misc.

> map length [[0, 1, 3], [], [5, 7, 2, 8]]
[3, 0, 4]
> map length [1, 3, 5]
Fehler
> map fastFib [1 .. 40]
Liste der Fib.-Zahlen
```

Abbildung 2.7: Beispiele zu `map`

```
filter p [] = []
filter p (x:xs)
  | p x        = x:(filter p xs)
  | otherwise  = filter p xs
```

Beispiele zu `filter` in Abbildung 2.8.

Fehler #14

Die Funktionen `foldr1` und `foldr`. Definition:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
-- ohne '1': liefert keinen Fehler fuer die leere Liste
--
-- (Summe ueber [] ist 0, Produkt ueber [] ist 1)
--
foldr :: (a -> b -> b) -> b -> [a] -> a
foldr f x [] = x
foldr f x (y:xs) = f y (foldr f x ys)
```

8. Dez.

Beispiel. Zwei Varianten für den Einsatz von `foldr`:

- `a` = `b`, d.h. `f` ist binäre Operation. `foldr` sollte dann verwendet werden (an Stelle von `foldr1`), wenn die Operation ein neutrales Element hat. Beispiele:

  `(+)` neutrales Element 0

  `(*)` neutrales Element 1

```
 o Primzahlen

-- wahr wenn Primzahl:
--
isPrime :: Int -> Bool

-- Hat x keinen echten Teiler >= y?
noDivGeq :: Int -> Int -> Bool
noDivGeq x y
  | y >= x          = True
  | (mod x y) == 0  = False  -- y ist einer
  | otherwise       = noDivGeq x (y + 1)

isPrime x = noDivGeq x 2

> filter IsPrime [1 .. 1000]
Liste der Primzahlen <= 1000


 o Beispiel:

-- Person mit Geburtsjahr
--
xs :: [(String, Int)]

isOver40 (String, Int) -> Bool
isOver40 (st, n) = n < 1959

> filter isOver40 xs
Liste der Personen aelter als 40

> map fst (filter isOver40 xs)
Liste der Personen ohne Geburtsjahr


 o Beispiel:

isSorted :: [Int] -> Bool
isSorted xs = xs == iSort xs

> filter isSorted [[3, 6, 4], [], [2, 4 6], [7, 3]]
[[], [2, 4, 6]]
```

Abbildung 2.8: Beispiele zu `filter`

`(||)` neutrales Element `False`, weil `x || False == x`

`(&&)` neutrales Element `True`, weil `x && True == x`

`foldr (+) 0 xs` gibt etwa die Summe der Listenelemente (für nichtleere Listen) und 0 für die leere Liste. `max` hat neutrales Element 0 für natürliche Zahlen; nicht aber, wenn beliebige ganze Zahlen betrachtet werden.

- $\mathtt{a} \neq \mathtt{b}$. Beispiele:

```
--
-- zweite Zahl um die Laenge des Strings erhoehen:
--

incrementByLength :: String -> Int -> Int
incrementByLength st x = (length st) + x
```

```
> foldr incrementByLength 0 ["was", "passiert"]
  --> incrementByLength "was"
                        (foldr incrementByLength
                               0 ["passiert"])
  --> incrementByLength "was" 8
  --> 11
```

Das gleiche leisten auch `foldr (+) 0 (map length xs)` und `sum (map length xs)`. Wir kennen außerdem die Funktion `insert :: Int -> [Int] -> [Int]`. `iSort` ist äquivalent zu `foldr insert [] xs`.

Diese Beispiele zeigen, daß primitive Rekursionen mit `foldr` beschrieben werden können, ohne daß explizit primitive Rekursion verwendet wird.

Beispiel. Aufspalten von Listen. Gegeben: ein Text (String); gesucht: Liste der Wörter im Text.

```
whitespace = ['\n', '\t', ' ']
getWord :: String -> String
getWord [] = []
getWord (x:xs)
  | elem x whitespace  = []
  | otherwise  = x:(getWord xs)

dropWord :: String -> String
dropWord [] = []
dropWord (x:xs)
  | elem x whitespace  = (x:xs)
  | otherwise    = dropWord xs
```

```
> getWord "was passiert"
"was"
> getWord " was passiert"
""
> dropWord "was passiert"
" passiert"
> dropWord " was passiert"
"was passiert"
```

```
dropSpace :: String -> String
dropSpace [] = []
dropSpace (x:xs)
  | elem x whitespace  = dropSpace xs
  | otherwise  = (x:xs)

type Word = String
split :: String -> [Word]
split [] = []
split st = (getWord st):(split (dropSpace (dropWord st)))

splitWords :: String -> [Word]
splitWords st = split (dropSpace st)

-- Verallgemeinerung:
--
-- (p x == elem x whitespace)
--
getUntil :: (a -> Bool) -> [a] -> [a]
getUntil p [] = []
getUntil p (x:xs)
  | p x  = []
  | otherwise  = x:(getUntil p xs)

-- Eingebaute Funktion: takeWhile
-- Kopiert Liste so lange, wie Bedingung erfuellt ist.
--
takeWhile :: (a -> Bool) -> [a] -> [a]

```

```
getWord xs = takeWhile p xs
  -- p muss vorher definiert werden, oder
  -- wir koennen es lokal definieren
  where p x = not (elem x whitespace)
```

2.11 Definitionen

Definitionen und lokale Definitionen

- Wir unterscheiden zwischen Deklaration einer Funktion (Beschreibung des Typs, Signatur) und der Definition (Beschreibung des Algorithmus')

- Definitionen beginnen mit Namen

- Erster Buchstabe des Namens definiert eine Box (der Definition) die Box endet beim ersten Zeichen unter oder links unter dem ersten Buchstaben:

```
fun :: Int -> Int -> Int      -- Deklaration
fun x y                       -- Beginn der Definition



                              -- Ende der Definition
funny
```

  Jedes `.hs`-file zerfällt in Boxen, also in Definitionen. Jede dieser Definitionen ist innerhalb der ganzen Datei gültig (unabhängig von der Reihenfolge also).

- zweite Möglichkeit zur Beendigung einer Definition ist das Semikolon
- Innerhalb einer Definition kann mit `where` eine lokale Definition eingeführt werden (die nur innerhalb der äußeren Definition gültig ist).

15. Dez. **Lokale Definition.** Innerhalb einer Definition `def1` kann durch `where` eine lokale Definition `def2` eingeführt werden. Sie ist nur innerhalb von `def1` gültig. Lokale Definitionen schränken den Gültigkeitsbereich von äußeren Definitionen mit gleichem Namen ein, d.h. die lokale Definition hat Vorraung.

```
maxsq x y                    -- maximal square
  | (sq x) >= (sq y)  = sq x
  | otherwise  = sq y
  where
    sq z = z * z
    -- oder:
    where sq x = x * x
```

`where` kann auch geschachtelt werden; die lokalen Definitionen sind nur in der jeweils aufrufenden Funktion, also nur eine Stufe höher (und darunter) sichtbar.

2.12 Programmierung mit Modulen

Das Modul. (Der Modul: Zahl, bezüglich derer der Rest bei einer Division gebildet wird. Engl. Module: Baustein.) Große

Softwareprodukte umfassen Hunderte oder gar Tausende von Funktionen und Typen. Deshalb braucht man Mittel, um diese Fülle sinnvoll organisieren und überschaubar strukturieren zu können. Im Software-Engeneering hat sich dafür der Begriff der Modularisierung eingebürgert.

Modularisierung. Man spaltet ein komplexes Programm (den Algorithmus) in Unterprogramme (Unterfunktionen) auf, von denen jedes einen Teil des gesamten Programms leistet.

Man unterteilt den Algorithmus also in unabhängig voneinander realisierbare und in sich abgeschlossene Komponenten (genannt Modul), die, setzt man sie zusammen, das Programm ergeben, und somit wie Bausteine wirken.

Die Vorteile der Modulnutzung:

1. Ein Modul ist eine in sich abgeschlossene Komponente eines größeren Algorithmus, der ihn aufruft. Der Entwurf des Moduls und der Entwurf des aufrufenden Algorithmus können getrennt erfolgen und vereinfachen so den Entwurfsprozeß.

2. Da der Algorithmusentwurf normalerweise schwierig ist, ist jede Vereinfachung zu begrüßen. Die Vorteile liegen im schnelleren Entwurf und in einer geringeren Fehlerwahrscheinlichkeit.

3. Um einen Modul in einen Algorithmus einzufügen, muß man nur wissen, was der Modul leistet und nicht wie er funktioniert. Was ein Modul leistet, kann als Kommentar in den Modulkopf eingetragen werden.

4. Ebenso wie Moduln den Algorithmusentwurf erleichtern, so vereinfachen sie auch das Verstehen eines Algorithmus. Das ist von Bedeutung, falls der Algorithmus verändernt werden soll, was häufig vorkommt. Das gilt besonders bei der Änderung durch Personen, die nicht die Autoren des Algorithmus sind.

5. Ist ein Modul erst einmal entworfen, so kann er in jeden Algorithmus eingefügt werden., der ihn benötigt. Es ist deshalb möglich und sinnvoll, eine Modul-„Bibliothek" aufzubauen, die Moduln für Sortieraufgaben, Gleichungslösungen, Steuerberechnungen usw. enthält.

Realisierung eines Moduls in Haskell. Funktionen in einer Datei werden zu einem Modul zusammengefasst, indem man ganz oben in die Datei schreibt:

```
module Modulname where
```

Die Datei ist dann mit dem Modulnamen und der Endung `.hs`[4] zu speichern. Der Name eines Moduls fängt mit Großbuchstaben an.

Möchte man an anderer Stelle das Modul benutzen, dann importiert man dieses mit:

```
import Modulname
```

Damit kann man nochfolgend alle Funktionen aus dem importierten Modul benutzen.

[4]`import` sucht nach Dateien mit dem Namen des Moduls (z.B. `import Ggt` sucht `Ggt`, `Ggt.hs`, `Ggt.lhs`) im Arbeitsverzeichnis und den Verzeichnissen im „Search Path" (siehe Hugs-Manual).

Eine Schnittstelle beschreibt, was von dem Inhalt des Moduls nach außen hin sichtbar sein soll. Damit wird also festgelegt, was später aus dem Modul exportiert werden darf.

Haskell sieht folgende Notation vor: Dem Modulnamen folgt eine Auflistung aller Elemente, die asu dem Modul exportiert werden sollen:

```
module Modulname (function1, ..., functionn) where
```

Das ist wichtig, wenn eine Modulhierarchie aufgebaut wird, und man explizit festlegen will/muß (Haskell hat keinen automatischen Reexport), wer was von wem erbt.

Will man den Import selektieren, so fügt man die Liste der zu importierenden Elemente an die `import`-Anweisung an:

```
import Modulname (function1, ..., functionn)
```

Beispiel. Ein paar Beispielsmodule:

```
module Division where
-- Modul fuer ganzzahlige Division
-- Funktionen divi und modu
divi ...
modu ...
```

```
module Ggt where
-- Modul fuer den ggT
-- importiert Modul Division
import Division

ggT ...
```

```
module Kgv where
-- Modul fuer das kgV
-- importiert Module Division und Ggt
import GgT
import Division

kgV :: Int -> Int -> Int
kgV ab = abs (divi (a * b) (ggT a b))
```

Oder man definiert den Export des Moduls `Ggt` explizit:

```
module Ggt (module Ggt, module Division) where
-- Klammer gibt an, welche Funktionen exportiert
-- werden, und man kann (wie hier gezeigt) auch
-- ganze Module exportieren.
```

Dann genügt `import Ggt` im Modul `Kgv`. Der Import kann auch eingeschränkt werden:

```
import Ggt hiding (modu)
```

2.13 Prinzipien der Programmentwicklung

Programmentwicklung als Zyklus. [Anmerkung: Man stelle sich hier ein tolles Diagramm vor.]

- Spezifikation: genaue Festlegung der Aufgabenstellung

  - Welche Grundbegriffe, Strukturen und Zusammenhänge sind wichtig?
  - Anfordungsdefinition (mathematische Formel)
  - Nebenbedingungen (zum Beispiel Geltungsbereich)

- Lösungsentwurf:
  - Rechenstrukturen (Typen)
  - Zerlegung in Teilaufgaben
  - Wahl der Datenstrukturen und Algorithmen

- Implementierung:
  - syntaktisch korrekte Umsetzung
  - gut lesbar und wiederverwendbar
  - Nutzung von Bibliotheken

- Test:
  - typische Eingaben, zufällige Eingaben, kritische Fälle
  - Fehleranalyse (Debugging)
  - Effizienzanalyse (Profiling)

- Verifikation: Beweis, daß das Programm die Anforderungen erfüllt
  - vollständige Verifikation erfordert eine komplette Beschreibung der Semantik einer Programmiersprache
  - Verifikation kann nur so gut sein wie die Spezifikation

  - ist aufwendig, sehr formal
  - in der Praxis (für größere Projekte) ist eine Verifikation sehr wünschenswert, aber schwer machbar

[Anmerkung: `C-h k C-M-x`.]

Verifikation ist eng mit dem Beweisprinzip der vollständigen Induktion verbunden. [Anmerkung: Vollständige Induktion erkläre ich jetzt hier nicht. Im folgenden gibt's ein Beispiel, und ansonsten ja auch noch den Handapparat.]

2.14 Korrektheitsbeweise

Korrektheitsbeweise durch strukturelle Induktion.
17. Dez. Sind die natürlichen Zahlen der Definitionsbereich, so kann man vollständige Induktion anwenden (strukturelle Induktion, Parameterinduktion) und teilweise offensichtliche Beweise führen:

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

Satz. `fac n` *berechnet* $\prod_{i=1}^{n} i$ *für alle* $n \in \mathbb{N}$.

Beweis. Beweis also durch vollst. Ind.:

IA richtig für $n = 0$.

IV `fac n` berechnet $\prod_{i=1}^{n} i$ für ein $n \in \mathbb{N}$

IB `fac (n + 1)` berechnet $\prod_{i=1}^{n+1} i$.

Beweis der Behauptung:

$$\begin{aligned}
\texttt{fac}\ (n+1) &= (n+1)\cdot \texttt{fac}\ (n+1-1) \text{ (nach Def.)}\\
&= (n+1)\cdot \texttt{fac}\ n\\
&= (n+1)\cdot \prod_{i=1}^{n} i\\
&= \prod_{i=1}^{n+1} i
\end{aligned}$$

□

[Anmerkung: Es ist 10 vor 1 und ich muß bis morgen noch einen LinA-Zettel machen. Daher fehlen die übrigen Induktionsbeispiele. Wer meint, sie dürfen auf gar keinen Fall fehlen, darf sie gerne abtippen und mir in TEX geben. Dann trage ich sie mit Vergnügen nach.]

Berechnungsinduktion. Berechnungsinduktion bezieht sich auf die induktive Auswertung (Interpretation) von Ausdrücken. Ein Ausdruck kann zum Zeitpunkt n ausgewertet werden, wenn all seine Bestandteile bereits früher ausgewertet wurden. Ist der Ausdruck (noch) nicht auswertbar, so wird das durch das Sondersymbol $\perp$ gekennzeichnet. Die Ausgabe des Programms ist auch durch einen Ausdruck beschrieben. Mit der Berechnungsinduktion muß man zeigen, daß zu irgendeinem Zeitpunkt die Auswertung dieses Ausdrucks mit der Spezifikation (also mit dem geünschten Ergebnis) übereinstimmt. Oft wird dieser Beweis in zwei Teile unterteilt:

- Beweis durch Induktion, daß, wenn das Programm für eine bestimmte Eingabe terminiert, dann der Wert korrekt ist. (Partielle Korrektheit)

- Beweis, daß das Programm für alle Eingaben aus dem Definitionsbereich terminiert. (Terminierungsbeweis)

Kapitel 3

Codierungs- und Informationstheorie

Information muß zur maschinellen Speicherung, Verarbeitung und Übertragung durch exakt festgelegte Formen repräsentiert (dargestellt) werden. 5. Jan.

Definition (Codierung). Codierung ist die Darstellung von Information durch Zeichenfolgen.

Folgende Aspekte sind dabei von Bedeutung:

1. Eindeutige Decodierbarkeit
2. Fehler bei der Übertragung/Verarbeitung sollten erkannt (ev. korrigiert) werden können.

3. Die codierte Information sollte möglichst kompakt sein. (Datenkompression)

4. Oft ist es wünschenswert, daß codierte Informationen nur durch berechtigte Personen decodiert werden können. (Kryptographie)

Die Punkte 2 und 3 stehen sich diametral gegenüber; ohne Redundanz ist keine Fehlererkennung möglich.

Codierung Ausgangspunkt: Die zu codierende Information ist durch eine Zeichenfolge gegeben (String aus A^*). Ziel ist die Darstellung der Information als String aus B^* (hier $B = \mathbb{B} = \{0, 1\}$).

- $c\colon A^* \longrightarrow B^*$ (Codierung von Wörtern)

- $c\colon A \longrightarrow B$ (Codierung von Buchstaben)
  $c^*\colon A^* \longrightarrow B^*$
  $c^*(a_1, \ldots, a_n) = c(a_1), \ldots, c(a_n)$

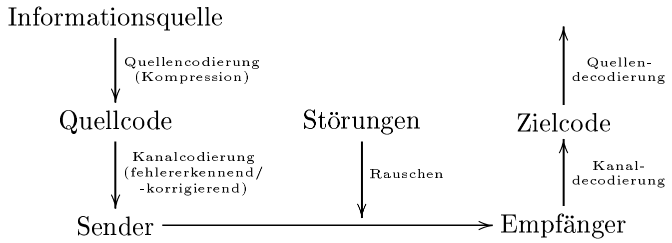


Abbildung 3.1: Codierung

Decodierung

- $d\colon \{c(w) \mid w \in A^*\} \longrightarrow A^*$
  $d \circ c(w) = w$
  (wobei $d \circ c(w) = d(c(w))$)

- $d\colon \{c^*(w) \mid w \in A^*\} \longrightarrow A^*$
  $d \circ c^*(w) = w$

3.1 Blockcodes

Definition 25. Ein Code $c\colon A \longrightarrow \mathbb{B}^n$ wird binärer Blockcode der Länge n genannt. (Bei Broy: Code eindeutiger Länge.)

Definition 26. Für zwei Binärwörter $b = (b_1, \ldots, b_n), b' = (b'_1, \ldots, b'_n) \in \mathbb{B}^n$ ist der Hamming-Abstand definiert als Anzahl der Stellen, an denen sich b und b' unterscheiden, d.h.:

$$d_H(b, b') = |\{i \mid 1 \leq i \leq n \wedge b_i \neq b'_i\}| = \sum_{i=1}^{n} d_i$$

wobei

$$d_i = \begin{cases} 1 & b_i \neq b'_i \\ 0 & \text{sonst} \end{cases}$$

Beispiel. $d_H((0, \underline{0}, 1, \underline{1}, 0, 1, 1), (0, \underline{1}, 1, \underline{0}, 0, 1, 1)) = 2$.

Satz 27. *d_H ist eine Abstandsfunktion in $\mathbb{B}^n$, d.h. es gilt für beliebige $b, b', b'' \in \mathbb{B}^n$:*

1. $d_H(b, b') = 0 \Longleftrightarrow b = b'$

2. $d_H(b, b') = d_H(b', b)$

3. $d_H(b, b') \leq d_H(b, b'') + d_H(b'', b')$ *(Dreiecksungleichung)*

Definition 28. Sei $c\colon A \longrightarrow \mathbb{B}^n$, dann definieren wir:

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

Definition 29. Ein Code $c\colon A \longrightarrow \mathbb{B}^n$ heißt k-fehlererkennend, wenn für beliebige $a \in A$ und $b \in \mathbb{B}^n$ gilt

$$1 \leq d_H(c(a), b) \leq k \Longrightarrow b \notin c(a)$$

Definition 30. Ein Code $c\colon A \longrightarrow \mathbb{B}^n$ heißt k-fehlerkorrigierend, wenn für beliebige $a, a' \in A$ mit $a \neq a'$ und $b \in \mathbb{B}^n$ gilt:

$$d_H(c(a), b) \leq k \Longrightarrow d_H(c(a'), b) > k$$

- Für jeden Code muß wegen der eindeutigen Decodierbarkeit $d_H(c) \geq 1$ gelten.
- $d_H(c) \geq k + 1 \Longrightarrow c$ ist k-fehlererkennend
- $d_H(c) \geq 2k + 1 \Longrightarrow c$ ist k-fehlerkorrigierend

3.2 Fehlerkorrektur

Einfache Methoden zur Fehlererkennung/-korrektur (sei $c\colon A \longrightarrow \mathbb{B}^n$ eine beliebige Codierung):

Paritätsbit. Sei $c_p\colon a \longrightarrow \mathbb{B}^{n+1}$ mit $c_p(a) = c(a)b_p(a)$. Dabei ist

$$b_p(a) = \begin{cases} 0 & \text{falls } d_H(c(a)), \mathfrak{o}) \text{ gerade} \\ 1 & \text{sonst} \end{cases}$$

(und $\mathfrak{o}$ ist der Nullvektor.) Alle Wörter haben also eine gerade Anzahl von Einsen und es gilt $d_H(c_p) \geq 2$.

Doppelcodierung. Sei $c^2\colon A \longrightarrow \mathbb{B}^{2n}$ mit $c^2(a) = c(a)c(a)$ und seien $a, a' \in A$ mit $a \neq a'$. Dann gelten: $d_H(c(a), c(a')) \geq 1$ und $d_H(c^2(a), c^2(a')) \geq 2$.

Doppelcodierung mit Paritätsbit Mit $c_p^2(a)\colon A \longrightarrow \mathbb{B}^{2n+1}$ bezeichnen wir die Doppelcodierung mit zusätzlichem Paritätsbit, d.h. $c_p^2(a) = c(a)c(a)b_p(a)$, wobei $b_p(a)$ wie oben von $c(a)$ abgeleitet wird.

Beispiel. Alle besprochenen Codes:

$a \in A$	$c(a)$	$c_p(a)$	$c^2(a)$	$c_p^2(a)$
a	00	000	0000	00000
b	01	011	0101	01011
c	10	101	1010	10101
d	11	110	1111	11110

Satz 31. $d_H(c_p^2) \geq 3$.

Beweis. Betrachte $a \neq a'$. Fallunterscheidung:

- $d_H(c(a), c(a')) \geq 2 \Longrightarrow d_H(c_p^2(a), c_p^2(a')) \geq 4$
- $d_H(c(a), c(a')) = 1 \Longrightarrow d_H(c^2(a), c^2(a')) = 2$

□

Da sich $c(a)$ und $c(a')$ genau an einer Stelle unterscheiden, ist $b_p(a) \neq b_p(a')$ und folglich $d_H(c_p^2(a), c_p^2(a')) = 3$.

Vergleich dieser Codes:

n Bits tragen die Information

t Bits dienen der Kontrolle

Informationsrate:

$\mathrm{IR}(c) = \frac{n}{n+t}$

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

Kreuzsicherungscodes. Ordne die Bits von $c(a)$ in einem Rechteck, möglichst einem Quadrat ($n = k^2$) an, und ergänze je ein Paritätsbit für jede Zeile und Spalte:

$$c(a) = \boxed{\begin{array}{llll} b_1 & b_2 & \dots & b_k \\ b_{k+1} & b_{k+2} & \dots & b_{2k} \\ & \vdots & & \\ b_{(k-1)k+1} & b_{(k-1)k+2} & \dots & b_{k^2} \end{array}}$$

Man erhält so die Codierung $c^{\times}$.

Satz 32. $d_H(c^\times) \geq 3$

Beweis. Seinen $a, a' \in A$ $(a \neq a')$. Fallunterscheidung:

- $d_H(c(a), c(a')) \geq 3 \Longrightarrow d_H(c^\times(a), c^\times(a')) \geq 3$
- $d_H(c(a), c(a')) = 2$. Nochmals Fallunterscheidung:
  - Die Stellen liegen in verschiedenen Zeilen und Spalten: Es gibt vier zusätzliche Unterschiede in den Kontrollbits, also: $d_H(c^\times(a), c^\times(a')) = 6$
  - Die Stellen liegen in einer Zeile (Spalte): Es gibt zwei zusätzliche Unterschiede in den Kontrollbits der Spalten (Zeilen): $d_H(c^\times(a), c^\times(a')) = 4$
- $d_H(c(a), c(a')) = 1$. Dann sind die Kontrollbits der Zeile und der Spalte verschieden: $d_H(c^\times(a), c^\times(a')) = 3$.

□

$$\mathrm{IR}(c^\times) = \frac{n}{n + 2\sqrt{n}} = 1 - \frac{2\sqrt{n}}{n + 2\sqrt{n}} \approx 1 - \frac{2}{\sqrt{n}}$$

Es gibt noch bessere Codierungen:

Hamming-Codes. HAM$(3, 2)$ bedeutet:

- 4 Informationsbits $(b_1, \ldots, b_4)$

- 3 Kontrollbits (b_5, b_6, b_7):
  $b_5 = b_2 \oplus b_3 \oplus b_4$
  $b_6 = b_1 \oplus b_3 \oplus b_4$
  $b_7 = b_1 \oplus b_2 \oplus b_4$
  (Bei vier Informationsbits braucht man mindestens drei Kontrollbits.)

3.3 Codes variabler Länge

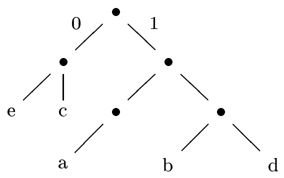
Wie erreicht man eindeutige Decodierbarkeit von Codes variabler Länge?

Definition 33. Ein Code $c\colon A \longrightarrow B^*$ wird Präfixcode genannt, wenn für beliebige $a \neq a' \in A$ gilt:

> $c(a)$ ist kein Präfix (Anfangsstück) von $c(a')$ und umgekehrt. (FANO-Bedingung)

Präfixcodes können durch einen Codebaum dargestellt werden:

$a \in A$	$c(a) \in \mathbb{B}^*$
a	100
b	110
c	01
d	111
e	00



Beispiel. 0100100111 gibt "cead".

(Würde man auf dem Weg zu einem Blatt an einem anderen Buchstaben vorbeikommen—stünde also in einem inneren Knoten ein Buchstabe—wäre es kein Präfixcode.)

Satz 34. *Präfixcodes sind eindeutig decodierbar.* 12. Jan.

Beweis. Angenommen, c ist nicht eindeutig decodierbar, d.h. es gibt $a_1 a_2 \dots a_n$ und $a'_1 a'_2 \dots a'_l$, die verschieden sind und $c(a_1 a_2 \dots a_n) = c(a'_1 a'_2 \dots a'_l)$. Sei i so gewählt, daß

$a_1 = a'_1, a_2 = a'_2, \dots, a_i = a'_i$ und $a_{i+1} \neq a'_{i+1}$
$\Longrightarrow c(a_1 \dots a_i) = c(a'_1 \dots a'_i)$
$\Longrightarrow c(a_{i+1} \dots a_n) = c(a'_{i+1} \dots a'_l)$

Dann folgt (Fallunterscheidung):

1. $|c(a_{i+1})| = |c(a'_{i+1})| \Longrightarrow c(a_{i+1}) \neq c(a'_{i+1})$
2. $|c(a_{i+1})| < |c(a'_{i+1})| \Longrightarrow c(a_{i+1})$ Präfix von $c(a'_{i+1})$
3. $|c(a_{i+1})| > |c(a'_{i+1})| \Longrightarrow c(a'_{i+1})$ Präfix von $c(a_{i+1})$

In allen drei Fällen ergibt sich also ein Widerspruch. Die Annahme ist falsch, c also eindeutig decodierbar. □

Ziel: Länge der Codewörter minimieren.

Satz 35 (Ungleichung von Kraft und MacMillen.). *Sei* $A = \{a_1, \dots, a_m\}$. *Für jeden eindeutig decodierbaren Code* $c \colon A \longrightarrow \mathbb{B}^*$ *gilt:*

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

Beweis. $n_i = |c(a_i)| = d_T(a_i)$, oBdA sei $n_1 \leq n_2 \leq \cdots \leq n_m$. (Länge des Codewortes ist die Tiefe des Blatts im Codebaum.) Sei $\alpha \in \mathbb{N}$. Dann folgt:

$$\begin{aligned}
\left(\sum_{i=1}^{m} \frac{1}{2^{n_i}}\right)^{\alpha} &= \sum_{k_1=1}^{m} \sum_{k_2=1}^{m} \cdots \sum_{k_\alpha=1}^{m} \frac{1}{2^{n_{k_1}} \cdot 2^{n_{k_2}} \cdots 2^{n_{k_\alpha}}} \\
&= \sum_{k_1=1}^{m} \sum_{k_2=1}^{m} \cdots \sum_{k_\alpha=1}^{m} \frac{1}{2^{n_{k_1}+n_{k_2}+\cdots+n_{k_\alpha}}} \\
&= \sum_{w \in A^\alpha} \frac{1}{2^{|c(w)|}}
\end{aligned}$$

Gruppierung: Sei $a_{n,\alpha}$ die Anzahl der $w \in A^\alpha$ (Strings mit α Buchstaben) mit $|c(w)| = n$. Dann erhalten wir:

$$\sum_{n=m\alpha}^{n_m \alpha} \frac{a_{n,\alpha}}{2^n} \leq \sum_{n=n_1 \alpha}^{n_m \alpha} \frac{2^n}{2^n} = \alpha(n_m - n_1) + 1 \leq \alpha n_m$$

Wir nehmen die α-te Wurzel:

$$\begin{aligned}
& \sum_{i=1}^{m} \frac{1}{2^{n_i}} \leq \sqrt[\alpha]{\alpha}\, \sqrt[\alpha]{n_m} \longrightarrow 1 \text{ (wenn } \alpha \longrightarrow \infty) \\
\Longrightarrow \quad & \sum_{i=1}^{m} \frac{1}{2^{n_i}} \leq 1 + \epsilon \text{ für alle } \epsilon > 0 \\
\Longrightarrow \quad & \sum_{i=1}^{m} \frac{1}{2^{n_i}} \leq 1
\end{aligned}$$

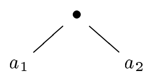
□

Definition 36. Ein binärer Baum wird vollständig genannt, wenn jeder Knoten entweder ein Blatt ist oder zwei Kinder hat.

Bemerkung. Jeder binäre Baum kann durch Kontraktion (über Knoten mit genau einem Kind) in einen vollständigen binären Baum umgewandelt werden.

Satz 37. *Codes, die durch vollständige binäre Bäume repräsentiert sind, führen zur Gleichheit in der KMM-Ungleichung, d.h. sie sind optimal.*

Beweis. Beweis durch Induktion nach Anzahl der Blätter. Induktionsanfang: Im Fall $m = 2$ liegt ein Baum der Art



vor, und erhalten wir als Summe $\frac{1}{2^1} + \frac{1}{2^1} = 1$. Induktionsvoraussetzung (IV): Sei die Gleichheit erfüllt für alle vollständigen binären Bäume mit m Blättern. Zu zeigen als Induktionsschritt: Dann folgt die Gleichheit für solche Bäume mit $m + 1$ Blättern.

Sei T ein Baum mit $m + 1$ Blättern, x tiefstes Blatt in T, v Vater von x. Dann hat v ein zweites Kind y, und y hat die gleiche Tiefe wie x.

$$\sum_{a_i \in A} \frac{1}{2^{d_T(a_i)}} = \sum_{a_i \neq x,y} \frac{1}{2^{d_T(a_i)}} + \frac{1}{2^{d_T(x)}} + \frac{1}{2^{d_T(y)}}$$

$T' = T \setminus \{x,y\}$, also hat T' m Blätter; v ist Blatt in T'.

$$\begin{aligned}\sum_{a_i \in A} \frac{1}{2^{d_T(a_i)}} &= \sum_{a_i \neq x,y} \frac{1}{2^{d_T(a_i)}} + \frac{1}{2^{d_T(x)-1}} \\ &= \sum_{a_i \neq x,y} \frac{1}{2^{d_T(a_i)}} + \frac{1}{2^{d_T(v)}} \overset{\mathrm{IV}}{=} 1\end{aligned}$$

□

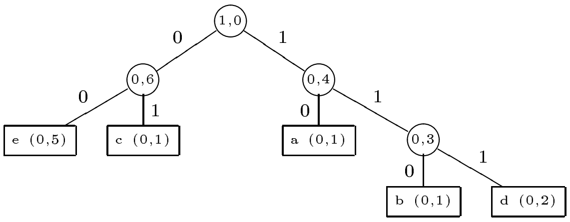
3.4 Codierung für Alphabete mit Wahrscheinlichkeitsverteilung

Sei $A = \{a_1, \ldots, a_n\}$, eine Wahrscheinlichkeitsverteilung gegeben als $p\colon A \longrightarrow [0,1]$ und $p_i = p(a_i)$ die relative Häufigkeit von a_i. Dabei soll $\sum_{i=1}^m p_i = 1$ sein. Die erwartete Länge eines Codeworts ist dann:

$$n(c) = \sum_{i=1}^{m} |c(a_i)| \cdot p_i = \sum_{i=1}^{m} n_i p_i$$

c ist optimal, wenn $n(c)$ minimal ist.

Vollständiger binärer Baum mit Gewichten:



Vollständige binare Bäume mit Gewichten:

- Blätter haben Label a_i und Gewicht p.
- Innere Knoten haben Gewicht, das sich als Summe der Gewichte der beiden Kinder zusammensetzt. (Hier ist $n(c) = 0,5 \cdot 2 + 0.1 \cdot 2 + 0.1 \cdot 3 + 0,2 \cdot 3 = 2,3$.)

Der Austausch von a (0.1) gegen d (0.2) ergibt $n(c') = 2.2$.

Huffman. Der Algorithms von HUFFMAN erzeugt optimale Codes.

- Gegeben seien A und p.
- Initialisierung: m Blätter a_i (p_i) nach absteigenden p_i geordnet als Liste von Bäumen.
- Wiederhole $(m-1)$-mal: Streiche die beiden Bäume minimalen Gewichts und füge einen neuen Baum ein, dessen Wurzel die zwei gestrichenen Bäume als Kinder hat.

Beispiel. Das Beispiel von oben ist in Abbildung 3.2 zu finden.

Optimalität des Huffman Codes. Dazu betrachten wir die folgenden beiden Lemmas: 14. Jan.

Lemma 38. *Sind $x, y \in A$ die Symbole mit der geringsten Wahrscheinlichkeit $(p(x) \leq p(y))$, dann gibt es einen optimalen Code in dem die Blätter von x und y Zwillinge sind, d.h. sie unterscheiden sich nur im letzten Bit.*

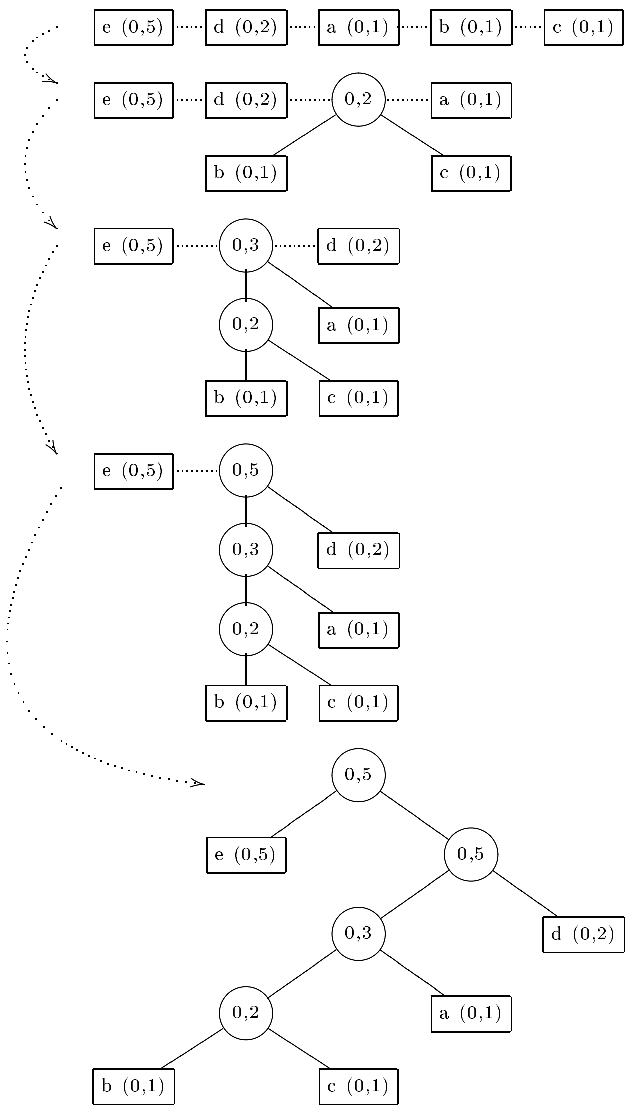


Abbildung 3.2: Huffman-Encoding

Lemma 39. *Wenn T einen optimalen Code für (A, p) definiert und x, y in T als Zwillinge auftreten, dann repräsentiert*

$$T' = T \setminus \{x, y\}$$

einen optimalen Code für (A', p'), wobei

$$\begin{gathered} A' = A \setminus \{x, y\} \cup \{v\}, \\ p'(v) = p(x) + p(y) \text{ und} \\ p'(a) = p(a) \text{ für alle } a \neq x, y. \end{gathered}$$

Beweis (Lemma 38). Sei $c\colon A \longrightarrow B^*$ optimaler Code. Seien a und b die zwei Blätter, am tiefsten liegen ($p(a) \leq p(b)$). $c'\colon A \longrightarrow \mathbb{B}^*$ sei wie c definiert, nur mit x bzw. y jeweils ausgetauscht durch a und b. Zu zeigen: $n(c') \leq n(c)$. Es gilt:

$$\begin{aligned} n(c') &= \sum_{i=1}^{m} p_i |c'(a_i))| \\ &= n(c) + d_T(a)(p(x) - p(a)) + d_T(x)(p(a) - p(x)) \\ &\quad + d_T(b)(p(y) - p(b)) + d_T(y)(p(b) - p(y)) \\ &= n(c) + \underbrace{(p(x) - p(a))}_{\leq 0} \underbrace{(d_T(a) - d_T(x))}_{\geq 0} \\ &\quad + \underbrace{(p(y) - p(b))}_{\leq 0} \underbrace{(d_T(b) - d_T(y))}_{\geq 0} \\ &\leq n(c) \end{aligned}$$

□

Beweisidee für Lemma 39. Angenommen, T' nicht optimal, dann gibt es ein S', das besser als T' ist; dann kann man einen

Baum S konstruieren, der besser ist als T, im Widerspruch zur Voraussetzung. $\square$

Kapitel 4

Fortgeschrittenes Haskell

4.1 Algebraische Typen

Zwei neue Ideen:

1. Konstruktoren (Funktionen,[1] die Objekte eines Typs konstruieren)

2. Alternativen (Objekte, die sich aus verschiedenen Parametersätzen ableiten, können zusammengefasst werden)

[1] oder sowas

4.1.1 Konstruktoren

Syntax. `data` eröffnet Definition eines algebraischen Typs:

```
data Typname = Konstruktorname Typ1... Typk
```

Die `Typ1` bis `Typk` sind die Typen der Parameter.

Beispiel. Eine Person wird beschrieben durch Namen und Alter:

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

(In Java heißen Konstruktor und Typ gleich, in Haskell kann man das machen, muß man aber nicht.)

Person funktionert dann so wie:

```
Person :: Name -> Age -> Person
... xs :: [Person] ...
... (Person "Hans Meier" 45):xs ...
```

Nutzen. Konstruktoren benutzt man für Typen, die wir bisher so geschrieben hätten:

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

Wozu sind Konstruktoren also gut?
Vorteile eines Konstruktors:

- Name des Konstruktors macht Angaben über den Typ (Lesbarkeit).
- Konstruktor muß explizit angegeben werden; eine zufällige Konstruktion von neuen Objekten ist ausgeschlossen.
- Bei Fehlermeldungen wird Name des Typs angegeben.

Nachteile:

- Etwas länger und aufwendiger.
- Polymorphe Funktionen wie z.B. `fst` und `snd` für Paare können nicht benutzt werden.

[Anmerkung: Warum will man das ganze dann? Ganz einfach: Die Sourcen werden *nicht* länger und aufwendiger dadurch, denn wenn sie länger als eine handvoll Zeilen sind, und man sie eventuell später noch verstehen will, dann schreibt man sich all dieses Zeug sowieso. Aber man möchte es eben nicht per Hand machen—und deshalb ist sowas eine *Vereinfachung*. (Lest euch mal was über Data Abstraction durch.) Qnf vfg qbpu xrva Crey uvre!]

4.1.2 Alternativen

Alternativen können zur Konstruktion von Aufzählungen verwendet werden:

```
data Season = Spring | Summer | Autumn | Winter
```

Erstes Wort jeder Alternative ist der Konstruktor; hier ohne Parameter.

4.1.3 Allgemeine Form algebraischer Typen

```
data Typname
  = Konstruktor1 Typ11... Typ1m
  | Konstruktor2 Typ21... Typ2m
  :       :         :         :
  | Konstruktorn Typn1... Typnm
```

Beispiel. Geometische Form:

```
data Shape
  = Circle Float          -- Radius
  | Square Float          -- Seite
  | Rectangle Float Float -- Width, Height
```

Bei Funktionen mit algebraischen Typen als Parameter können die Konstruktornamen für Muster in der Funktionsdefinition verwendet werden.

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

-- zu Michaels Idee:
```

```
--
-- isRound (Circle _) = True
-- isRound _ = False
--
-- funktioniert auch!

area :: Shape -> Bool
area (Circle r) = pi * r * r
area (Square a) = a * a
area (Rectangle a b) = a * b
```

[Anmerkung: Hier fehlt ein Tag. Soweit ich mich erinnere, wurden da fast nur die Sourcen zur HUFFMAN-Codierung angeschrieben, die es ja auch im Netz gibt. Möchte jemand, daß ich die hier noch einmal abdrucke? In diesem Fall bitte melden.] 19. Jan

4.2 Klassen in Haskell

Das Klassenkonzept in Haskell: 21. Jan

- Klasse ist Zusammenfassung von Typen, die gemeinsame Funktionen besitzen (Typklassen)
- Typen, die zu einer Klasse gehören, werden Instanzen dieser Klasse genannt.
- Es gibt eine Reihe vordefinierter Klassen
- Klassen können selbst definiert werden
- Typen könen als Instanz einer Klasse definiert werden.

Beispiel. Die Klasse `Eq` (vordefiniert). Eine mögliche Definition:

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

allEqual :: Eq a => a -> a -> a -> Bool
allEqual n m p = (n == m) && (m == p)

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

Allgemeine Definition einer Klasse.

```
class Classname a where
  f1 :: t11(a) -> ... -> t1k1(a) \
   :                             | Signatur
   :                             | der
   :                             | Klasse
  fn :: tn1(a) -> ... -> tnkn(a) /
```

Die wirkliche Definition von `Eq`.

```
class EQ a where
  (==), (/=), :: a -> a -> Bool
  x == y  = not (x /= y)
  x /= y  = not (x == y)
```

Zur Definition einer Instanz der Klasse `Eq` ist die Definition eines der beiden Operatoren ausreichend.

```
instance Eq (Int, Int) where
  (x1, y1)  == (x2, y2)  = (x1 == x2) && (y1 == y2)
```

Was passiert? Der Operator `==` in Eq wird überschrieben durch den Operator `==` für `(Int, Int)`.

Allgemeine Definition einer Instanz.

```
instance Classname typename where
  :
  (Definition von Funktionen aus der Signatur)
  :
```

Beispiel. Schon vordefiniert:

```
instance Eq a => Eq [a] where
  -- Voraussetzung: a ist Instanz von Eq
  -- Dann sind Listen van a Instanz von Eq
  [] == []  = True
  [] == xs  = False
  xs == []  = False
  (x:xs) == (y:ys)  = (x == y) && (xs == ys)
```

Selbst definierte Typen:

```
data Rect = Rect Float Float
instance Eq Rect
```

```
    (Rect x1 y1) == (Rect x2 y2)
      = (max x1 y1) == (max x2 y2)
     && (min x1 y1) == (min x2 y2)
```

```
> (Rect 1.5 2.0) == (Rect 2.0 1.5)
True
```

```
data Shape
  = Circle Float
  | Rectangle Float Float
  deriving (Eq)

-- vergleicht Konstruktoren und ann die Parameter und entscheidet
-- dann:
```

```
> (Rectangle 1.5 2.0) == (Rectangle 2.0 1.5)
False
```

Abgeleitete Klasse. Ist Signatur(`Cl1` $\subset$ Signatur(`Cl2`, dann kann `Cl2` aus `Cl1` abgeleitet werden (`Cl2` erbt von `Cl1`).

```
class (Cl1 a) => Cl2 a where
  zusaetzliche Funktionen der Signatur (Cl2)
```

Beispiel. Vordefiniert:

```
class (Eq a) => Ord a where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  -- natuerlich auch (==)
```

```
max, min :: a -> a -> a
compare :: a -> a -> Ordering
-- Definition von Ordering:
-- data Ordering = LT | EQ | GT
 :
 :
x <= y  = (x == y) || (x < y)
 :
 :
```

Zur Definition einer Instanz von `Ord` ist die Definition eines Operators bereits ausreichend.

Vordefinierte Instanzen von Eq.

- `Char`, `Int`, `Float`, `Bool`
- `Eq a => Eq (a, ..., a)`
- `Eq a => Eq [a]`

Funktionen wie `(Int -> Int)` sind keine Instanz von Eq.
Vordefierte Instanzen von Ord sind `Char`, `Int`, `Float` und `Bool`.

`Show`. Klasse zur Umwandlung von Werten in Strings.

```
class Show a where
  show :: a -> String

instance Show Bool where
```

```
  show True  = "True"
  show False  = "False"

instance (Show a, Show b) => Show (a, b) where
  show (x, y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

`Show` ist für Bildschirmausgabe wichtig:

```
putStr :: String -> IO()
putStrLn = putStr . (++ "\n")

print :: Show a => a -> IO()
print = putStrln . show
```

`Bit.`

```
data Bit = O | I  deriving (Eq)

instance Show [Bit] where
  show []  = ""
  show (O:xs)  = "O" ++ show xs
  show (I:xs)  = "I" ++ show xs
```

[Anmerkung: Irgendwas gibt's nur unter Haskell 98 (man verwende `/import/Haskell98/bin/hugs-98`:]

```
class Show a where
  show :: a -> String
   :
   :
```

```
    :
  print = PutStrLn.show

data Bit = O | I  deriving (Eq, Show)
type Coded = [Bit]
-- [O,I,O,O,I] soll OIOOI ergeben
```

`Num`. Numerische Typklasse.

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate :: a -> a
  abs, signum :: a -> a
  fromInt :: Int -> a
   :
   :
   :
  x - y = x + negate y
```

Inhaltsverzeichnis