Einführung in die Informatik II

Ein Skript zu der gleichnamigen Vorlesung von Prof. Dr. W. Brauer

Version 11

25. Oktober 1999

Inhaltsverzeichnis

Ein Einführendes Beispiel

Das Missionars Problem

Drei Missionare und drei Kannibalen stehen am Ufer eines Flusses und wollen diesen überqueren. Dazu steht ihnen aber nur ein Boot mit zwei Sitzplätzen zur Verfügung. Das wäre natürlich kein Problem würden die Kannibalen nicht die Bedingung stellen, das sie an keinem Ufer des Flusses in der Minderheit seinen wollen, da sie sonst befürchten missioniert zu werden.

Diese "Kanibalenbedingung" wird als Restriktion (engl. constraint) bezeichnet. Jedoch ist diese Restriktion einzuschränken da eine Anzahl von Null Kannibalen auf einen Ufer erlaubt sind. Auch wenn dies nicht direkt im Text steht, so ist es doch offensichtlich.

Ein möglicher Lösungsweg führt über den Zustandsbegriff:

Anfangszuztand:	3 Missionare und 3 Kannibalen auf dem linken Ufer
Endzustand:	3 Missionare und 3 Kannibalen auf dem rechten Ufer

Die einzelnen Zustände sind Paare (K_L, M_L), mit K_L als Anzahl der Kannibalen auf dem linken Ufer und M_L als Anzahl der Missionare auf dem linken Ufer. Mit dem Paar (K_R, M_R), für das rechte Ufer, verhält es sich analog, es ist aber eigentlich nicht erforderlich (wegen Symmetrie). Gewisse Zwischenzustände sind wegen der Restriktionen verboten. Das Problem läßt sich nun anschaulich durch eine Matrix darstellen, in der zunächst die verboten Zwischenzustände markiert werden.

M_L

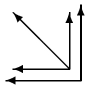
		0	1	2	3		
	0	Z				3	
K_L	1	⊗		×	×	2	K_R
	2	⊗	⊗		×	1	
	3				S	0	
		3	2	1	0		

M_R

Die mit × gekennzeichneten Zustände sind wegen der Bedingung der Kannibalen für das linke Ufer verboten. Analog sind wegen der Symmetrie die mit ⊗ gekennzeichneten Zustände für das rechte Ufer verboten. Das mit S markierte Feld ist der Startzustand und entsprechend ist das mit Z markierte Feld der Endzustand.

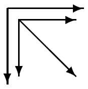
Nun gilt es zu definieren welche Aktionen (Züge, Bootsfahrten) möglich sind. Es gilt zu überlegen ob Einführung einer weiteren Zustandskomponente, nämlich der Position des Bootes, notwendig ist. Da dies aber die Lösung erheblich verkompliziert, werden statt dessen zwei Arten von Zügen (Fahrten) eingeführt, die jeweils abwechselnd ausgeführt werden müssen:

1. ungerade Züge: von links nach rechts



Diese Züge verringern die $K_L + M_L$ und erhöhen dementsprechend die $K_R + M_R$

2. gerade Züge: von rechts nach links



Diese Züge verringern die $K_R + M_R$ und erhöhen dementsprechend die $K_L + M_L$

Im folgenden sind die Züge in die Matrix eingetragen die nötig sind um durch die "Engstelle" zu gelangen.

M_L

K_L

	0	1	2	3	
0	Z				3
1	$\otimes$		$\times$	$\times$	2
2	$\otimes$	$\otimes$		$\times$	1
3				S	0
	3	2	1	0	

K_R

M_R

Bemerkungen: Eine Bootsfahrt ist eine atomare Aktion ohne Zeitverbrauch. Alle Personen steigen nach der Ankunft des Bootes aus und bleiben (eventuell auch länger) am Ufer. In dieser Zeit muß die Bedingung der Kannibalen erfüllt sein. Es werden mindestens 11 Fahrten benötig.

Eine andere Interpretation der Aufgabe

Ein Missionar bleibt vom Anfang bis zu Ende im Boot sitzen und steigt erst aus nach dem er alle Personen über den Fluß gebracht hat. Es zählen aber nur die Personen die sich auf dem jeweiligen Ufer befinden. Daher wird die Lösung des Problems um einiges Einfacher und es werden nur noch 9 Fahrten benötigt. Eine Mögliche Lösung ist:

$$
\begin{array}{ll}
M_1, M_2 \to r \; ; & M_1 \to l \\
M_1, K_1 \to r \; ; & M_1 \to l \\
M_1, K_2 \to r \; ; & M_1 \to l \\
M_1, K_3 \to r \; ; & M_1 \to l \\
M_1, M_2 \to r &
\end{array}
$$

Kapitel 1

Ergänzungen zu Theorie und Methodik

1.1 Prädikatenlogik

Die Prädikatenlogik ist eine Erweiterung der Aussagenlogik. Sie erlaubt es Aussagen über Individuen zu treffen (einzelne, alle, manche, ...). Dabei versteht man unter einem Individuum einen Gegenstand mit individuellen Eigenschaften. Die Eigenschaften der Individuen werden als Prädikate bezeichnet. Individuen, Gegenstände und Daten sind die (Grund-) Terme (oder auch Formeln) zu einer Signatur (eines ADT's). Sie können im folgenden enthalten:

- Konstante, Operatoren, Funktionen
- "Variable", also freie Identifikatoren
- Quantoren für All- und Existenzaussagen

Im folgenden seien diese Quantoren näher erklärt. Sei F eine Formel mit freien Identifikatoren aus der Familie von Mengen von Identifikatoren X und sei M die Menge der möglichen Werte für den Identifikator $x \in X$.

All-Aussagen: Für alle (jede/jedes) x gilt F:

$\forall xF$ $\forall$ ist All-Quantor   (oder auch $\forall x \in M : F$)

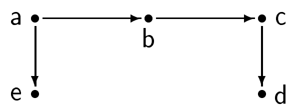
Existenzaussagen: Es existiert (mindestens) ein x für das F gilt:

$\exists xF$ $\exists$ ist Existenzquantor   (oder auch $\exists x \in M : F$)

Der Identifikator x tritt in den obigen Formeln nicht frei sondern gebunden auf. x heißt durch die Quantoren gebunden, weil er in einer Substitution von x durch Terme nicht ersetzt wird. Dies drücken die folgenden Regeln aus:

$$(\forall xF)[t'/x] = \forall xF$$
$$(\exists xF)[t'/x] = \exists xF$$

Ein n-stelliges Prädikat ist eine Abbildung auf n-Tuppeln von Termen auf {true, false}. Einfache Beispiele für zweistellige Prädikate sind "ungleich" und "gleich". Ein dreistelliges Prädikat wäre "a zwischen b und c". Ein etwas komplexeres Beispiel ist die Beschreibung eines gerichteten Graphen:



x, y, z seien Variablen für die Ecken
a,b,c,d,e sind Konstanten für die Ecken des Graphen

Das Prädikat kante(x,y) besagt es führt eine Kante von x nach y. Hierbei gilt es zu beachten das der Graph gerichtet ist. Zwei Beispiele:

kante(a,e) (ist true)
kante(a,b) (ist true)

Das Prädikat weg(x,y) bedeutet: "Es führt ein Weg von x nach y". Dieses Prädikat kann man mit Hilfe von kante(x,y) durch zwei Forderungen definieren.

1. $\forall x$ weg(x,x) (leerer weg)
2. $\forall x \forall y \forall z$: kante(x,z) $\land$ weg(z,y) $\underset{\text{impliziert}}{\Longrightarrow}$ weg(x,y)

Warum gilt nun weg(a,b), obwohl wir nur wissen das eine kante(a,b) existiert. Durch Substitution von a für x und b für y und z in 2 ergibt sich:

$$\text{kante(a,b)} \land \text{weg(b,b)} \quad \Longrightarrow \quad \text{weg(a,b)}$$

Ein weiters Beispiel liefern uns die Primzahlen. Die beiden Aussagen "es existieren unendlich viele Primzahlen" und "zu jeder Primzahl existiert eine größere" sind äquivalent. Oder auch:

$$\forall x\ p(x) \qquad \exists y\ (p(x) \land y > x)) \qquad p(x) = \text{Primzahl sein}$$

1.1.1 Syntax

In der Prädikatenlogik (erster Stufe) gehen wir von einem Alphabet $A = V \cup \Sigma \cup P$ aus. Darin ist:

1. V eine Menge von (Term-) Variablen
2. Σ eine Menge $\Sigma = \Sigma^{(0)} \cup \Sigma^{(1)} \cup ...$ von Funktionssymbolen.
3. P eine Menge $P = P^{(0)} \cup P^{(2)} \cup ...$ von Prädikaten.

Über $V \cup \Sigma$ bilden wir die Terme t. $p(t_1, ..., t_n)$ ist eine **atomare Formel**, wobei p ein n-stelliges Prädikat ist und die t_i Terme sind. Eine atomare Formel ist entweder true oder false. Die Menge der syntaktisch korrekten Formeln P wird Induktiv definiert mit:

1. Wenn F Element der Menge der atomaren Formeln ist, dann ist $F \in P$.
2. Wenn $F, G \in P$, dann auch $\neg F$, $F \lor G$ und $F \land G$.
3. wenn $F \in P$ und $v \in V$, dann gehören auch $\forall v F$ und $\exists v F$ zu P.

Variable können sowohl gebunden (an Quantoren) als auch frei vorkommen. Ein Beispiel hierfür ist:

$$\exists x\ (p(x) \land \forall \overset{z}{x}\ q(\overset{z}{x}, y) \lor q(x, x))$$

Und noch ein paar Beispiele für Formeln:

Für mindestens ein x gilt:	$\exists x \quad F$
Für höchstens ein x gilt:	$\forall x \forall y \quad (F(x) \wedge F(y) \rightarrow x = y)$
Für genau ein x gilt:	$\exists x \quad (F(x) \wedge \forall y \quad (x \neq y \rightarrow F(y)))$

Aus $\forall x F(x)$ folgt $F[t/x]$ für jeden Term $\neq \perp$.
Wenn $\forall x (F \rightarrow G)$, dann folgt $F[t/x] \rightarrow G[t/x]$ für jeden Term $\neq \perp$.
Gilt $F[t/x]$ für einen Term $\neq \perp$, so gilt $\exists x F$.

1.1.2 Semantik

Um die einzelnen atomaren Formeln mit Wahrheitswerten belegen zu können, ist es notwendig einen Weltausschnitt zu wählen für den die logischen Formeln gelten sollen. Dazu definiert man:

1. Die nichtleere Menge U, das **Universum** oder der Individuenbereich (für die Werte der Variablen).

2. Die Interpretation I der Funktions- und Prädikatensymbole als Funktionen bzw. Prädikate über U, mit $I : \Sigma \cup P$ (Menge der Funktionen und Prädikate über U).

$$\begin{array}{lll} f \in \Sigma^{(m)} & I(f): & U^m \rightarrow U \\ p \in P^{(m)} & I(p): & U^m \rightarrow \{\mathsf{true}, \mathsf{false}\} \end{array}$$

Zu einem vorgegebenen Alphabet $A = V \cup \Sigma \cup P$ heißt $\mathfrak{A} = U \cup I$ eine **Struktur**. Eine **Belegung** $\alpha : \quad V \rightarrow U$ ist eine Abbildung der Variablenmenge eines Alphabetes in das Universum einer Struktur $\mathfrak{A}$. Eine **Interpretation** $\mathfrak{I}$ ist ein Paar $(\mathfrak{A}, \alpha)$ aus einer Struktur $\mathfrak{A}$ und einer Belegung α. Sie definiert sowohl die Bedeutung von Funktionen und Prädikaten als auch die Belegung von Variablen.

Ein Beispiel für die Interpretation einer Formel:

Seien:

p, q:	Prädikate
f, g:	Funktionen
z:	ein freier Identifikator
a:	eine Konstante

$$F = \forall x \quad (p(x, f(x)) \wedge q(g(a, f(z))))$$

Eine zu F passende Interpretation:

$U = \mathbb{N} = \{1, 2, 3, ...\}$

$I(p) = \{(m, n) \in \mathbb{N}^2 | m < n\}$
(p wird angegeben durch die Menge der Paare, die p erfüllen)
Das heißt, $p(m, n) = \mathsf{true}$ genau dann, wenn $m < n; m, n \in \mathbb{N}$
$I(q) = \{n \in \mathbb{N} | n \text{ ist Primzahl}\}$
$I(f) = \mathsf{succ}(n)$ (Nachfolgerfunktion $\mathsf{succ}(n) = n + 1$)
$I(g) = \mathsf{add}$
$I(a) = 2$

$$F \rightarrow \forall x \quad ((x < \mathsf{succ}(x)) \wedge ((2 + (z + 1)) \text{ist Primzahl}))$$

Die Formel ist wahr für die Belegung α_1 mit $\alpha_1(z) = 2$: $\mathfrak{I}_1(F) = \mathsf{true}$.
Die Formel ist falsch für die Belegung α_2 mit $\alpha_2(z) = 1$: $\mathfrak{I}_2(F) = \mathsf{false}$.

Sei F eine prädikatenlogische Formel und $\mathfrak{I} = (\mathfrak{A}, \alpha)$ eine zu F passende Interpretation. Dann definiert $\mathfrak{I}$ eine Abbildung von F auf $\{\mathsf{true}, \mathsf{false}\}$ folgendermaßen:

$$\mathfrak{I}(p(t_1, ..t_n)) = I(p)(\alpha(t_1), ...\alpha(t_n))$$

($\alpha(t_i)$heißt: in t_i die Variablen gemäß der Belegung α ersetzen.)

$$\begin{aligned}
\mathfrak{I}(\neg F) &= \begin{cases} \mathsf{true}, \text{ wenn } \mathfrak{I}(F) = \mathsf{false} \\ \mathsf{false}, \text{ wenn } \mathfrak{I}(F) = \mathsf{true} \end{cases} \\
\mathfrak{I}(F \wedge G) &= \begin{cases} \mathsf{true}, \text{ wenn } \mathfrak{I}(F) = \mathsf{true} \text{ und } \mathfrak{I}(G) = \mathsf{true} \\ \mathsf{false}, \text{ sonst} \end{cases} \\
\mathfrak{I}(F \vee G) &= \begin{cases} \mathsf{true}, \text{ wenn } \mathfrak{I}(F) = \mathsf{true} \text{ oder } \mathfrak{I}(G) = \mathsf{true} \\ \mathsf{false}, \text{ sonst} \end{cases} \\
\mathfrak{I}(\forall x F) &= \begin{cases} \mathsf{true}, \text{ wenn für alle } a \in U \text{ gilt } \mathfrak{I}[a/x](F) = \mathsf{true} \\ \mathsf{false}, \text{ sonst} \end{cases} \\
\mathfrak{I}(\exists x F) &= \begin{cases} \mathsf{true}, \text{ wenn ein } a \in U \text{ existiert mit } \mathfrak{I}[a/x](F) = \mathsf{true} \\ \mathsf{false}, \text{ sonst} \end{cases}
\end{aligned}$$

Dabei ist $\mathfrak{I}[a/x]$ die Interpretation, die man erhält, wenn man die Belegung α für die Variable x durch $\alpha[a/x]x = a$ definiert.

$$\alpha[a/x](y) = \begin{cases} \alpha(y) \text{ für } x \neq y \\ a \text{ für } x = y \end{cases}$$

Die Interpretation $\mathfrak{I}$ heißt Modell für eine Formel F (im Zeichen: $\mathfrak{I} \models F$), wenn gilt $\mathfrak{I}(F) = \mathsf{true}$. Im weiteren gilt:

- F heißt erfüllbar, wenn ein Modell $\mathfrak{I}$ für F existiert ($\mathfrak{I} \models F$).
- F heißt unerfüllbar, wenn kein Modell $\mathfrak{I}$ für F existiert.
- F heißt allgemeingültig, wenn gilt $\neg F$ ist unerfüllbar, das heißt wenn jede Interpretation $\mathfrak{I}$ ein Modell von F ist.

Es gilt der folgende Satz:

1. Die allgemeingültigen Formeln der Prädikatenlogik erster Stufe (über festen Alphabeten) sind aufzählbar (per Programm erzeugbar). Das heißt, ob F allgemeingültig ist, ist semi-entscheidbar aber nicht voll entscheidbar.
2. Erfüllbarkeit ist nicht entscheidbar.

Bemerkung: es existieren Formeln der Prädikatenlogik erster Stufe die erfüllbar sind aber kein endliches Model besitzen.

1.1.3 Die Beziehung zur Aussagenlogik

Die Aussagenlogik erhält man eine Spezialisierung der Prädikatenlogik, in dem man alle Quantoren und Variablen verbietet. Atomare Formeln $p(t_1, ...t_n)$ sind bei fester Struktur $=(U, I)$ entweder true oder false, denn die t_i sind Grundterme ohne Variablen. Das heißt eine atomare Formel F ist eine Aussage. Die verschiedenen Belegungen der Aussagen entsprechen verschieden Aussagen.

Siehe auch: MANFRED BROY: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 2.4 Aussagenlogik und Prädikatenlogik; Seite 67ff und: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 4.2 Prädikatenlogik; Seite 188ff

1.2 Semantik von Programmen

Generell kann man zwei Arten von Semantik für Programme unterscheiden:

1. **Operative Semantik**: Beschreibung der schrittweisen Abarbeitung bei der Auswertung von Programmen oder von Termen.
2. **Funktionale Semantik**: Zuordnung mathematischer Objekte (Funktionen) zu Programm(-komponenten) oder (Teil-)Ausdrücken.

Beispiele hierfür sind Termersetzungssysteme (wie zum Beispiel Gofer) oder die Funktionssemantik für Gofer (siehe dazu auch Broy 4.3.1.).

1.2.1 Induktive Deutung rekursiver Funktionsdeklarationen

Sei eine rekursive Funktionsdeklaration gegeben mit:

$$\textbf{fct}\ \mathrm{f} = \underbrace{(\mathbf{m}\mathrm{x})}_{\text{Argument}} \qquad \underbrace{\mathbf{n}}_{\substack{\text{Ergebnis/}\\ \text{Funktionswert}}}:\mathrm{E}$$

Dabei seinen **n** und **m** Sorten der Trägermengen N bzw. M. Beide Mengen können vereinbarungsgemäß das Element $\perp$ enthalten. E ist ein Term (die Definition der Funktion) in dem f frei vorkommen darf.

1. Bemerkung: Sei T ein Term in dem f frei vorkommt. Wenn nun f nicht rekursiv ist, dann ist die Semantik von T: T[E/f].
2. Bemerkung: Sei f rekursiv. Somit kommt f frei in E vor. Wir treffen die Vereinbarung, daß außer x und f alle anderen Bezeichner korrekt in E implementiert sind.

In E kann für f jede beliebige Funktion der gleichen Signatur eingesetzt werden, das heißt E bestimmt ein Funktional.

$$\tau_E : \quad (\mathrm{M} \to \mathrm{N}) \to (\mathrm{M} \to \mathrm{N})$$

Das Funktional τ_E liefert für Funktionen g: M $\to$ N durch τ_E[g] wieder eine Funktion. Für Argumente a $\in$ M ist die Funktion τ_E[g] wie folgt definiert:

$$\tau_E[\mathrm{g}](\mathrm{a}) = \begin{cases} \mathrm{E}[\mathrm{g/f,\ a/x}] & \text{falls a} \neq \perp \\ \perp & \text{sonst} \end{cases}$$

3. Bemerkung: durch τ_E läßt sich induktiv eine Folge von Funktionen f_i: M $\to$ N mit i = 1, 2, 3, ... definieren. Seien alle a $\in$ M und alle i $\in \mathbb{N}$:

$$\begin{aligned} \mathrm{f}_0(\mathrm{x}) &= \perp && \text{überall nicht definiert} \\ \mathrm{f}_{(i+1)}(\mathrm{x}) &= \tau_E[\mathrm{f}_i](\mathrm{x}) \end{aligned}$$

   Die Folge von Funktionen f_i ist ausschließlich durch die iterierte Anwendung des Funktionals τ_E auf die Funktion f_0 gegeben. Es gilt:

$$\mathrm{f}_i = \tau_E^i[\mathrm{f}_0]$$

Man kann nun feststellen, daß unter den Bedingungen, daß alle i,j $\in \mathbb{N}$ und alle x $\in$ M sind,(durch den Terminierungsfall) gilt:

$$\mathrm{i} \leq \mathrm{j} \Rightarrow (\mathrm{f}_i(\mathrm{x}) = \perp) \vee (\mathrm{f}_i(\mathrm{x}) = \mathrm{f}_j(\mathrm{x}))$$

Man kann daher definieren:

$$\mathrm{f}_\infty(\mathrm{x}) = \begin{cases} \perp & \text{falls für alle i} \in \mathbb{N}\text{: } \mathrm{f}_i(\mathrm{x}) = \perp \\ \mathrm{f}_j(\mathrm{x}) & \text{falls für j} \in \mathbb{N}\text{: } \mathrm{f}_j(\mathrm{x}) \neq \perp \end{cases}$$

Dieses f_j ist die Funktion f die ursprünglich gemeint war.

1.2.2 Deutung als kleinster Fixpunkt

4. Bemerkung: neben der induktiven Deutung kann man eine rekursive Funktionsdeklaration auch rekursiv über eine Lösung der Funktionalgleichung

   $$\mathrm{f} = \tau_E[\mathrm{f}] \quad (*)$$

   deuten, wobei τ_E wie eben definiert ist. Wir suchen also eine Abbildung f: M $\rightarrow$ N mit der Eigenschaft, daß für alle x $\in$ M gilt:

   $$\mathrm{f(x)} = \tau_E[\mathrm{f}](\mathrm{x})$$

   Ein solches f heißt Fixpunkt von τ_E und $(*)$.

Man kann feststellen, daß f_∞ Fixpunkt von τ_E ist. Das heißt, die rekursive Funktionsdeklaration führt zur Fixpunktgleichung $(*)$. Die Lösung der Gleichung erfolgt induktiv (f_∞) und das ist die durch die Deklaration beschriebene Funktion. (?)

Problem: (*) könnte mehrere Lösungen haben, daher ist es notwendig eine Ordnung auf die Menge der Funktionen zu haben. f_0 ist kleinster Fixpunkt.

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 4.3 Rekursive Funktionsdeklarationen; Seite 110ff

1.3 Qualitätsbeurteilung von Programmen

1.3.1 Grundlegende Eigenschaften

- Korrektheit (partielle oder totale) (Hoare, Terminierung, Zusicherungen)
- Komplexität (Effizienz, Aufwand bei Ausführung) (Hauptsächlich diese beiden Eigenschaften werden in der InformatikEinführung behandelt. Die folgenden Eigenschaften werden ausführlicher in Vorlesungen über SoftwareEngineering behandelt).
- Für die Praxis wichtige Eigenschaften nach DIN ISO 9126

  Funktionalität
  Zuverlässigkeit
  Benutzbarkeit
  Effizienz
  Änderbarkeit
  Übertragbarkeit

  } Um das zu erreichen verwendet man in der Praxis Qualitätsmanagement und Qualitätssicherung (konstruktive und analytische Maßnahmen)

1.3.2 Effizienz und Programmierbarkeit

Als Beispiel dient uns der schrittweise Entwurf eines Programms, im Hinblick auf Verringerung der Ausführungszeit, basierend auf folgender Problemstellung: gegeben ist S eine Teilmenge der Ganzen Zahlen $\mathbb{Z}$ und ein Element c der Ganzen Zahlen. Es soll nun überprüft werden ob c in S liegt. Die Aufgabe ist es nun eine Funktion `contains` zu implementieren:

$$\texttt{contains}\ \gamma(\mathbb{Z}) \times \mathbb{Z}\ \texttt{->}\ \texttt{BOOL}$$

$$(\mathrm{s, c}) \rightarrow \begin{cases} \texttt{true} & \text{falls c} \in \text{S} \\ \texttt{false} & \text{sonst} \end{cases}$$

Eine erste Lösung:

1. Zunächst erfolgt keine Festlegung auf eine Datenstruktur für die Mengen S. Nehmen wir an, wir hätten eine Selektionsfunktion `select` zur Verfügung, die aus jeder Teilmengen M von $\mathbb{Z}$ mit $M \neq \varnothing$ irgendein $m \in M$ auswählt:

$$\texttt{select } \gamma(\mathbb{Z}) \texttt{ -> } \mathbb{Z}$$
$$M \to \begin{cases} m & \text{falls } M \neq \varnothing,\ m \in M \\ \perp & \text{sonst} \end{cases}$$

2. Aufbauend auf `select` lautet eine erste Lösung:

$$\texttt{cont}(S,c) \begin{cases} \texttt{cont}(S \backslash \texttt{select}(S), c) & \text{falls } S \neq \varnothing,\ c \neq \texttt{select}(S) \\ \texttt{false} & \text{falls } S = \varnothing \\ \texttt{true} & \text{falls } c = \texttt{select}(S) \end{cases}$$

Hierbei bezeichne $M \backslash m$ das Entfernen des Elements m aus der Menge M. Die Funktionen `contains`(S,c) und `cont`(S,c) entsprechen sich offenbar für alle $S \subseteq \mathbb{Z}$ und $c \in \mathbb{Z}$.

3. Wie effizient ist diese Lösung nun? Sowohl der Denkaufwand zur Lösungsfindung als auch der Zeitaufwand für dessen Erstellung (Größe des Programms) sind gering. Die Lesbarkeit (Verständlichkeit) des Programmes ist gut. Aber wie steht es mit der Ausführungszeit? Um sie zu bestimmen zählt man die Anzahl der Schritte vom Start bis zum Stop des Programms und bestimmt die Dauer der einzelnen Schritte. Unser Beispiel enthält die folgenden Schritte:

   - rekursiver Aufruf von `cont`
   - Aufruf von `select`
   - Entfernen des Rückgabewertes von `select` aus S (\-Ausführen)
   - $S \neq \varnothing$ prüfen
   - `select`(S) $\neq$ c prüfen

   Zur Vereinfachung nehmen wir an, daß für jeden Schritt die gleiche Zeit (1 Zeiteinheit) benötigt wird (gefährlich da zum Teil sehr unterschiedliche Ausführungszeiten auftreten können). T (n) sei nun Zahl der Schritte von Start bis Stop in Abhängigkeit von der Problemgröße n ($T \sim$ time). Die Problemgröße ist hier im wesentlichen die Anzahl der Elemente in S ($n = |S|$). Dabei kann man zwischen verschiedenen Fällen unterscheiden:

   - Der schlechteste Fall (worst case) ist der Maximal benötige Aufwand also die obere Schranke für Aufwand. Er tritt ein, wenn `select`M $\neq$ c für alle M mit $|M| > 1$, also wenn c falls es in S enthalten ist erst als letztes Element von `select` zurückgegeben wird:

     $$\texttt{time select}(M) \text{ solange } |M| \geq 2$$
     $$T_{WC} = \begin{cases} 5n - 1 & \text{falls } c \in S \\ 5n + 1 & \text{falls } c \notin S \end{cases}$$

     Diese Werte ergeben sich wie folgt:

(n-1) mal:	`cont`, `select`, \, $S \neq \varnothing$, `select`(S) $\neq$ c	5(n-1)
beim n-ten mal:	`cont`, `select`, `select`(S) $\neq$ c, $S \neq \varnothing$	+ 4
falls $c \notin S$ noch:	`cont`, $S \neq \varnothing$	+ 2

- Der beste Fall (best case), wenn c in S enthalten ist und als erstes Element von `select` zurückgegeben wird:

  $$\texttt{select}(\mathrm{S}) = \mathrm{c}, \text{ wenn } \mathrm{c} \in \mathrm{S}$$
  $$\mathrm{T}_{BC} = 4 \quad \text{falls } \mathrm{c} \in \mathrm{S}$$

  Nun steckt die Schwierigkeit des Problems aber im Finden einer Funktion select mit dieser schönen Eigenschaft.

Nehmen wir nun an, daß `select` zufällig ein Element aus S auswählt. Dabei können folgende Fälle unterschieden werden:

$c \notin S$
$c \in S$ nach n Schritten gefunden
$c \in S$ nach n-1 Schritten gefunden
...
$c \in S$ nach 2 Schritten gefunden
$c \in S$ nach 1 Schritt gefunden

Wenn alle Fälle gleich wahrscheinlich sind, läßt sich der Mittelwert des Aufwands (bei vielen Programmausführungen mit verschiedenen S, c) bestimmen (Erwartungswert):

$$\begin{aligned} & \tfrac{1}{n+1}((5-1)+(5*2-1)+....+(5n-1)+(5n+1)) \\ = \; & \tfrac{1}{n+1}\left(\left(\sum_{k=1}^{n+1}(5k-1)\right)-3\right) \\ = \; & \tfrac{5}{2}n+4-\tfrac{2}{n-1} \end{aligned}$$

Für große n ist das im wesentlichen 2,5n.

Verallgemeinernd läßt sich sagen, daß wir eine gute Datenstruktur für S brauchen um die Suche zu beschleunigen. Im folgenden seien verschiedene Lösungen vorgestellt.

1. Sei S nun ein Feld ganzer Zahlen (`int S[] = new int[n]`). Die Zugriffsfunktion `select` läßt sich nun durch `S[i]` ersetzen, mit i = 0, 1, 2, 3, ... `S\select(S)` wird durch `i++` ersetzt. Die Ausführungszeit hängt nun von der Position von c im Feld s ab. Man kann das Verhalten im Mittel abschätzen. Bei Gleichwahrscheinlichkeit ergibt sich der gleiche Wert wie oben. Allerdings wurden zwei Operationen vereinfacht.

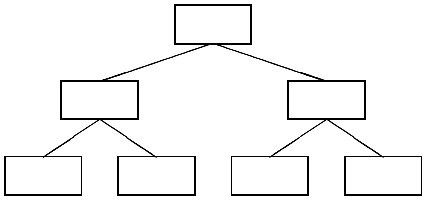
2. Weiteren Zeitgewinn erzielt man durch Entrekursivierung der Berechnung von `cont`, daß heißt den Übergang von funktionaler zu prozeduraler Programmierung (die funktionale Lösung dient jetzt als Spezifikation). Dies kann man durch die Verwendung einer `for`-Schleife erreichen. Der aufwendige rekursive Aufruf von `cont` wird durch einfaches Zählen ersetzt. Es ergibt sich folgendes Codefragment:

```
boolean b = false;
n = S.length

for(i=0;(b == false) && (i<n);i++) {
   if( S[i] == c) {
     b = true;
   }
}
```

Für den schlechtesten Fall ergibt sich die folgende Ausführungszeit: $\mathrm{T}_{WC} = 1 + 4n + 3 = 4(n+1)$. Dabei ist zu beachten das die Einzelschritte nun aber auch schneller ausgeführt werden.

3. Sei S ein sortiertes Feld mit `S[i] < S[i+1]`. Damit ist `S[0]` kleinstes und `S[n-1]` größtes Element. Nun kann man eine sogenannte Binäre Suche durchführen. Man testet dazu ob `S[0]` $\leq$ `c` $\leq$ `S[n-1]`. Ist diese Bedingung nicht erfüllt, so ist c kein Element von S. Ist sie erfüllt so wird getestet ob `c <` `S[`$\lceil \frac{n}{2} \rceil$`]`. Ist diese Bedingung erfüllt wird nur die linke Hälfte des Feldes weiter untersucht, andernfalls die rechte. Somit erhält man einen binären Baum.



Ist c in S enthalten, so muß man nur einen Weg von der Wurzel zum Blatt durchlaufen. Daher beträgt der Zeitaufwand in etwa $\log n$.

4. Sei S eine Teilmenge der natürlichen Zahlen $S \subseteq \mathbb{N}$ und sei ferner q eine obere Schranke für die Größe der Elemente aller zu untersuchenden Mengen S, d.h. es gelte stets s < q für s $\in$ S. Dann wählen wir als Datenstruktur ein Feld boolescher Werte der Länge q `boolean m[] = boolean[q]`. S wir nun dargestellt durch `m[s] = true` genau dann, wenn s $\in$ S. Das Feld m repräsentiert also die charakteristische Funktion der Menge S als Teilmenge von $\{0, 1, ..., \mathrm{q} - 1\}$. Damit gilt:

$$\texttt{cont}(\mathrm{S,c}) \begin{cases} \texttt{false} & \text{falls } \texttt{m[c] == true} \\ \texttt{true} & \text{falls } \texttt{m[c] == false} \text{ oder c} \geq \text{q oder c} < 0 \end{cases}$$

Diese Lösung ist sehr schnell, sie verfügt über einen konstanten Zeitverbrauch ($\mathrm{T}_{WC} = \mathrm{T}_{BC} = 2$), benötigt aber viel Speicherplatz in Abhängigkeit der Größe von q auch wenn die Menge S klein ist. Ähnlich funktioniert das Hashen (Streuspeicherung).

Beobachtung: das Sparen von Zeit kann Platz kosten. Außerdem sollte man den Arbeits- und Zeitaufwand des Entwicklers berücksichtigen. Man sollte nur die Funktionen entsprechend optimieren für die sich dieser Aufwand auch lohnt.

Zur Quantifizierung von Aufwand (Zeit, Platz, Kommunikation) kann man entsprechende Komplexitätsfunktionen verwenden ($T(n)$ time, $S(n)$ space, wobei n die "Größe der Eingabe" ist).

1.3.3 Die O - Notation

Für die Abschätzung von Komplexität führen wir die folgende Definition für das Wachstum oberer Schranken ein: Seien f und g Funktionen mit $f, g : \mathbb{N} \to \mathbb{R}^+$ ($\mathbb{R}^+$ sind die positiven reellen Zahlen). f wächst höchstens so schnell wie g, im Zeichen $f = O(g)$ (sprich f ist Element von Groß - O von g).

$$\exists n_0 \in \mathbb{N}, \quad \exists c \in \mathbb{R}^+, \quad \forall n \geq n_0 : f(n) \geq cg(n)$$

Ein mögliches Beispiel ist:

$$(\tfrac{5}{2}n + 4 - \tfrac{2}{n-1}) = O(n)$$

Typische Laufzeiten sind:

$O(1)$	= konstante Zeit
$O(\log n)$	= logarithmische Zeit (Suchen in einer Menge)
$O(n)$	= lineare Zeit (Bearbeiten jedes Elementes)
$O(n \log n)$	= (gute Sortierverfahren)
$O(n^2)$	= quadratische Zeit (primitive Sortierverfahren)
$O(n^k), k \in \mathbb{N}$	= polynomielle Zeit
$O(2^k)$	= exponentionelle Zeit

Im weiteren kann man definieren: Seien f und g Funktionen mit $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ ($\mathbb{R}^+$ sind die positiven reellen Zahlen). f wächst mindestens so schnell wie g, im Zeichen $f = \Omega(g)$ (sprich f ist Element von Groß - Omega von g).

Siehe auch: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 7.1.1 Der 0-Kalkül; 313f

1.3.4 Qualitätsmanagement und Qualitätssicherung

Produktorientiert:	Komponenten Zwischenergebnisse Produkte (Software)
Prozeßorientiert:	Erstellungsprozeß der Software

Konstruktive Maßnahmen

Produktorientiert:	Schema für Pflichtenheft und InfoGewinnung Richtige Programmiersprache Richtige Programmiermethodik Richtige Programmierwerkzeuge
Prozeßorientiert:	Richtlinien für Entwicklungsprozeß Softwareunterstützung des Entwicklungsprozesses Überwachungsprozeduren

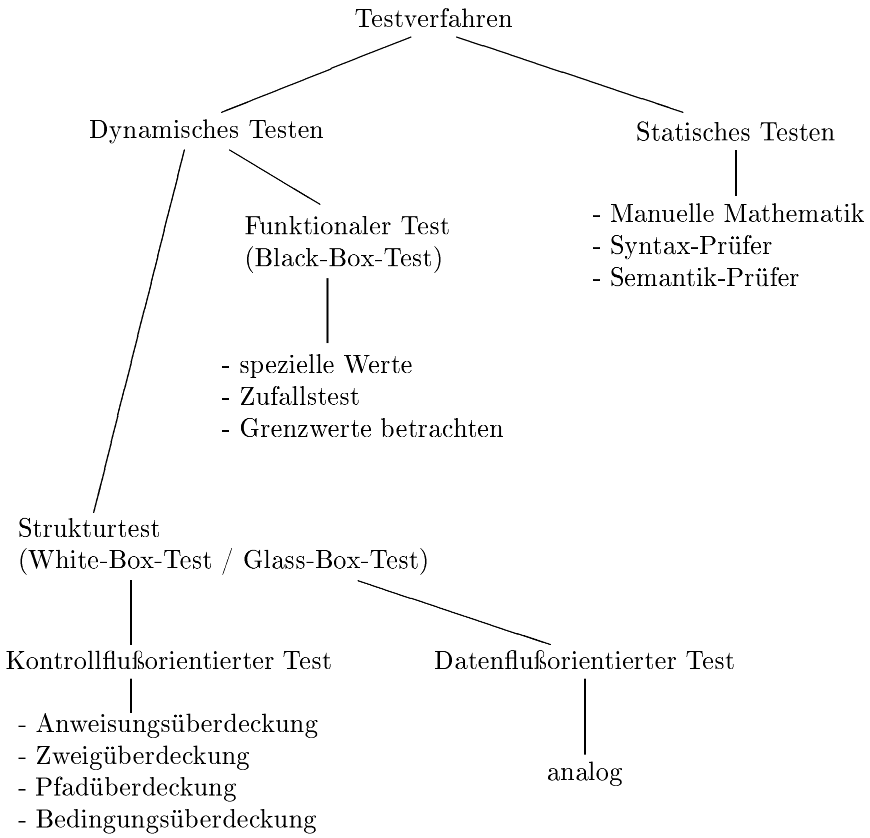
Analytische Maßnahmen

Produkt- bzw. Prozeß-Prüfung:	Automatisierung Nachvollziehbarkeit Einsatzbereich der Prüfmaßnahmen

Analytisches QualitätsManagement

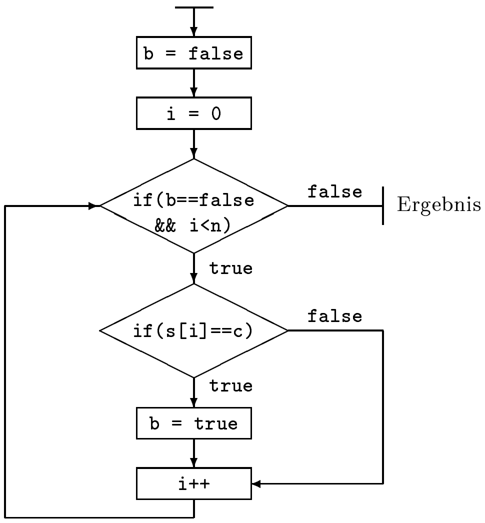
Untersuchende Verfahren	**Testende Verfahren**
Statische Analyse (Programmtext)	Dynamischer Test
Animation	Symbolischer Test
Programmverifikation (Korrektheitsbeweis)	Simulation
Review schriftlicher Dokumente (Gutachter)	SchreibtischTest
Inspektion (Detailprüfung)	
Walkthrough (schwaches Review)	
Durchsprache	
Audit	

Details: H. Balzert : **Lehrbuch der Softwaretechnik** SpektrumVerlag 1998



Problem: Konstruktion der Testfälle aber: Enge Beziehung zu Korrektheitsbeweisen (Hoare Kalkül/Zusicherungen, Terminierungsbedingungen).

Kontrollflußgraph für das Beispiel der cont(S,c) Berechnung:



Kapitel 2

Bemerkungen zum prozeduralen Programmieren

Wesentlich für das prozedurale bzw. zuweisungsorientierte Programmieren ist der Zustandsbegriff. Im folgenden sind funktionales und prozedurales Programmieren kurz gegenübergestellt:

- Funktionale Programmierung: hier entspricht das Programm einer Funktionsdefinition. Die Programmausführung ist die Funktionsanwendung auf die Funktionselemente. Das Ergebnis ist die Ausgabe des Funktionswertes. Im Rechner verbleibt keine "Erinnerung" an die Programmausführung (der Rechner transformiert).

- Zustandsorientierte Programmierung: Der Rechner speichert Resultate der Programmausführung und benutzt frühere (Zwischen-)Resultate. Ein und die selbe Funktion kann zu verschiedenen Zeiten bei gleichen Argumenten verschiedene Werte liefern, da sie keine "echte" Funktion (nur der Argumente) ist, sondern auch versteckte Argumente (z.B. globale Variablen) verwendet. Treten in Funktionen Zuweisungen an globale Variablen, die nicht als Parameter übergeben werden, auf spricht man von Seiteneffekten.

Die Zustände ermöglichen ein "Gedächtnis" des Programms. Sie werden durch Variable realisiert. Eine solche Variable besteht aus Referenz, Behälter und dem Wert.

- **Referenz**: (Bezeichner, Identifikator) ermöglicht den Zugriff auf die Variable (mittels einer Zugriffsfunktion)

- **Behälter**: bestimmt den Typ der Variable; hat zu verschiedenen Zeiten einen verschiedenen Inhalt (Zustand)

- **Wert**: ist der momentane Inhalt des Behälters; bei einer Zuweisung wird der alte Wert durch einen neuen Wert ersetzt

Die Weitergabe des Wertes einer Variable (`a` an eine andere Variable `b` durch Zuweisung `a = b;` wird als kopieren des Wertes unter Weitergabe der Kopie bezeichnet (daß heißt der Wert bleibt bei der Variablen `b`).

Ein Programm ist im wesentlichen eine Folge von Zuweisungen mit zusätzlichen "Regieanweisungen":

- **Vereinbarungen**: (Deklaration) von Variablen und zugehörigen Typ

- **Zusammenfassungen**: von (wieder zu verwendenden, mehrfach auszuführenden) Programmteilen: Blöcke, Prozeduren (Funktionen mit Seiteneffekten), Funktionen (Methoden)
- **Ablaufsteuerung**: Verzweigungen (Fallunterscheidungen), Wiederholungen (Schleifen), Sprünge
- **Werteberechnungen**: mittels Operatoren (Ausdrücke)

Beobachtungen: Vereinbarungen und Zusammenfassungen bestimmen die Gültigkeit und die Lebensdauer von (lokalen) Variablenvereinbarungen. Bei Werteberechnungen sind Operatorüberladung und Typanpassung möglich. Bei Funktionen und Prozeduren kann man bei der Parameterübergabe nach der Verwendung der Parameter unterscheiden:

- **Eingabeparameter**: tritt im Rumpf der Funktion nicht auf der linken Seite von Zuweisungen auf, sondern nur in Ausdrücken. Insbesondere wird sie in der Funktion nicht verändert.
- **Ausgabeparameter**: tritt im Rumpf der Funktion nur auf der linken Seite von Zuweisungen auf. Insbesondere ist der Wert des Parameters für die Ausführung der Prozedur bedeutungslos.
- **Transiente Parameter**: Der Parameter tritt im Rumpf der Funktion sowohl auf der linken Seite von Zuweisungen als auch in Ausdrücken auf.

Nach der Übergabeart unterscheidet man:

- **Wertübergabe**: Es wird nur der Wert der Variablen übergeben. Die Variable wird daher durch den Aufruf der Funktion auf keinem Fall verändert.
- **Referenzübergabe**: Es wird eine Referenz auf die Variable übergeben (es wird angegeben wo der Wert steht). Daher wird bei Zuweisungen innerhalb der Funktion der Wert der Variablen verändert.

Im weiteren kann man zwei Arten von Variablentypen unterscheiden:

- **primitive Typen**: (Grundtypen, Werttypen) Behälter der Variablen enthält Wert.
- **komplexe Typen**: (Referenztypen): Im Behälter der Variablen steht nur ein Verweis (Zeiger, Bezug) auf den "Inhalt","Wert" der gemeint ist.

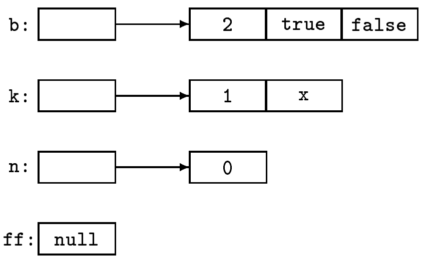
Sind `a` und `b` Referenztyp-Variablen, dann ist `a==b` nur `true`, wenn `a` und `b` dieselben Verweise enthalten. Einige Beispiele in JAVA:

```
boolean b[] = {true, false}

int k[] = new int[1]

int n[] = new int[0]

int ff[] = null
```



Kapitel 3

Objektorientierte Programmierung

3.1 Allgemeine Überlegungen

3.1.1 Unterschiede zur Prozeduralen Programmierung

Prozedurale Programmierung: Ein Rechner mit Speicher. Zustände, die von Programmen geändert werden.

Objektorientierte Programmierung: Mehrere Rechner , meistens verschiedener Bauart, die sich gegenseitig aufrufen und lokal prozedural arbeiten.

Zum Vergleich der beiden Programmierarten kann die Analogie zwischen sequentieller Programmausführung und einer einspurigen Autobahn verwenden.

Prozedurale Sichtweise

- die Autos entsprechen Datenobjekten
- eine globale Prozedur
  - prüft die Abstände
  - bestimmt die Geschwindigkeit
  - bestimmt die Positionen der Autos

Objektorientierte Sichtweise

- die Autos sind selbständige Objekte(Agenten, Akteure)
  - überprüfen selbst die Abstände (hinten und vorne)
  - regeln selbst die Geschwindigkeit
  - bestimmen die Position
  - geben auf Anfrage Position und Geschwindigkeit bekannt

3.1.2 Grundhaltung bei der OO - Programmierung

- in der OOP ist alles ein Objekt (wird als Objekt bezeichnet)
- **Objekt**: ist eine Art komplexe Variable, es speichert Daten (Attribute) und belegt Speicher im Rechner, es hat also Zustände. Es kann Operationen auf den Daten ausführen. Diese Algorithmen wurden in der prozeduralen Programmierung als Prozeduren oder Funktionen bezeichnet, in der OOP spricht

man von Methoden. Die Objekte sind relativ unabhängig voneinander.

- Ein OO - Programm ist ein Bündel von Objekten, die einander aufrufen, man spricht vom versenden von Nachrichten (Daten, Aufforderungen) (wie Funktions- oder Prozeduraufrufe).
- Jedes Objekt hat einen Typ, ist Instanzierung (Ausprägung eines Typs) einer Klasse.
- **Klasse**: ist gemeinsames Baumuster (- Schema) für die Struktur einer Familie von Objekten (Datentyp).
- Eine Klasse ist eine Beschreibungseinheit (ähnlich wie eine axiomische Spezifikation) zur Zusammenfassung von:
  - Funktionen, Prozeduren: als Methoden bezeichnet
  - Konstanten, Variablen: als Attribute bezeichnet

  Die Zusammenfassung führt zur Einkappselung der Daten (Zugriffsbeschränkung möglich).
- Jede Klasse enthält mindestens eine Methode zur Erzeugung von Objekten (Ausprägungen) der Klasse, die als Konstruktor (Generator) bezeichnet wird.

3.1.3 Aspekte der OO - Programmierung

- OO - Programme sind im Prinzip Zuweisungsorientierte Programme mit lokalen Zuständen in lose gekoppelten Objekten.
- Der Zugriff ein Objekt (dessen Methoden und Attribute) erfolgt ausschließlich über Referenzen (Adressierung ähnlich wie bei E-Mail Adressen). Es lassen sich Zugriffsbeschränkungen verschiedener Art festlegen.
- Ein wichtiger Aspekt der OOP ist die Wiederverwendbarkeit von Programmen, sowie das einfache durchführen von:
  - Modifikationen
  - Verfeinerungen
  - Ergänzungen

  an bestehenden Programmen.
- Programme (Klassen) werden in Bibliotheken organisiert.

3.2 Grundprinzipien der OO - Programmierung

1. Zwei weitere wichtige OOP - Aspekte sind die spezielle Entwurfsmethodologie, nämlich die objektorientierte Sicht der Welt, und die objektorientierte Strukturierung von Programmen.
2. Ein Ziel der objektorientierten Programmierung ist die Strukturierung von Programmen und die Minimierung des Konstruktionsaufwandes großer Programme, weniger die Minimierung von Laufzeit und Speicherplatz.
3. Die Programmeinheiten sind veränderbar (statische Wiederverwendung), man kann sie:
   - erweitern

- einschränken
- redefinieren
- abstrahieren (Zusammenfassung ähnlicher Dinge, Klassenbildung)
- polymorphisieren (dieselbe Funktion auf weitere neue Daten anwenden)

4. Dynamisches Binden von Methoden.
   Jedes Objekt enthält eine "Methodentabelle" mit den in der Klasse definierten Methoden. Haben verschiedene Objekte verschiedene Methoden gleichen Namens, so wird beim Aufruf des Objektes (zur Laufzeit) die passende Methode, anhand der Unterschiede in der Signatur, gewählt.

5. Subtypenbildung: Wenn A Subtyp von B ist, so ist jeder Ausdruck vom Typ A dort verwendbar, wo ein Ausdruck vom Typ erlaubt ist (Substitutivität).

6. Vererbung: (Inheritance) ist die Möglichkeit eine neue Klassen B (sogenannte abgeleitete Klasse) aus einer bestehenden Klasse A durch hinzufügen zu erzeugen. B erbt von A (B extends A). Man spart sich dadurch das Kopieren von Definitionen und das ändern der Kopien bei Änderungen in der Ausgangsklasse. Man muß allerdings alle Namen aus A kennen, um das Ererbte nutzen zu können. Es gilt:

   - B ist Subklasse von A
   - A ist Superklasse von B
   - B extends A

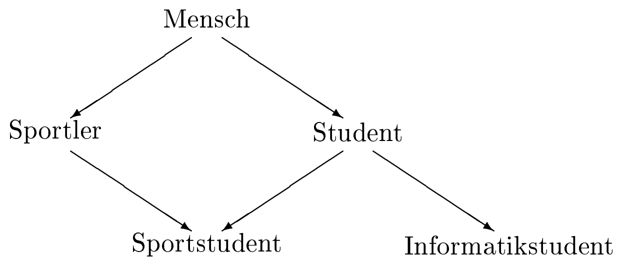
   Der Unterschied zwischen Vererbung und Besitz läßt sich mit der "Ist ein" - (Vererbung) bzw. "Hat ein" - Relation (Besitz) verdeutlichen. Ein Objekt einer Subklasse B von A:

   - "Ist ein" Objekt der Superklasse A
   - erbt deren Komponenten (Attribute und Methoden)
   - hat außerdem zusätzliche Komponenten der Subklassenbeschreibung B

   Es ist natürlich auch möglich Vererbung zu verweigern. Die Vererbung erlaubt verschiedene Beziehungen zwischen den Klassen:

   - B ist Unterklasse von A
   - B ist ein A (B übernimmt alles ohne Einschränkungen, Konformität)
   - B ist eine Spezialisierung von A (B hat spezielles Verhalten)
   - B implementiert A (B realisiert die Konzepte von A, zum Beispiel: Listen implementieren Mengen)
   - B verwendet A (rein technisch, B verwendet den Code von A)

   Durch mehrfaches Vererben entsteht eine Vererbungshierarchie, im allgemeinen ein azyklischer Graph:



Verfügt eine Klasse über mehrere verschiedene Superklassen, so spricht man von Mehrfachvererbung. In JAVA ist Mehrfachvererbung nicht erlaubt. Allerdings läßt sich diese fehlende Funktionalität mit Hilfe von Schnittstellen weitgehend kompensieren. Demzufolge ist die Vererbungshierarchie in JAVA ein Baum. Die oberste Klasse ist `Object` (jede Klasse ist Unterklasse von `Object`).

7. Zugriffskontrolle
   Zugriffskontrolle erhält man durch Kapselung in Klassen und Paketen sowie durch Angabe spezieller Schlüsselwörter (Modifikatoren). In JAVA sind das:

   - `public`: Zugriffsmöglichkeit besteht für alle
   - `protected`: Zugriffsmöglichkeit für alle Subklassen und Klassen der selben Package
   - `private`: Zugriffsmöglichkeit nur innerhalb der Klasse
   - ohne Angabe: Zugriffsmöglichkeit nur innerhalb der Package
   - `final`: Unterbinden von überschreibenden Zugriffen und von Vererbung

8. Persistenz
   Die Namen der Attribute sind nur innerhalb einer Klasse gültig. Die Lebensdauer der Attribute ist gleich der Lebensdauer des Objektes. Die Lebensdauer eines Objektes erstreckt sich vom Zeitpunkt der Erzeugung bis zur Zerstörung des Objektes. Das Objekt, insbesondere die Daten, ist persistent, daß heißt es existiert unabhängig von dem es erzeugenden Programm. In JAVA werden Objekte, wenn sie nicht mehr benötigt werden von der garbage collection zerstört.

3.3 JAVA - Beispiele

3.3.1 Klassendefinition

Eine denkbar einfache Klassendefinition:

```
public class Circle
{
   public double x, y, r;
   public double cf()
   {
     return 2 * 3.14 * r;
   }
   public double area()
   {
     return 3.14 * r * r;
   }
}
```

Gehen wir nun den Quellcode durch. Das Schlüsselwort `public` ist ein sogenannter Zugriffsmodifizierer (siehe oben). `class` ist das Schlüsselwort für eine Klassendefinition und wird vom Namen der Klasse gefolgt. Die Variablen `x,y,r` werden als Instanzvariable bezeichnet. Es ist allerdings ein äußerst schlechter Programmierstil Instanzvariablen als `public` zu definieren. Normalerweise werden Instanzvariable als `private` deklariert und der Zugriff auf sie über Methoden abgewickelt.

```
Circle c;
```

Hiermit wird eine Variable vom Typ `Circle` erzeugt. Das heißt, eine Variable die auf ein Objekt von Typ `Circle` zeigen kann. Dieses Objekt erzeugt man mit einer Konstruktormethode, man sagt, man erzeugt eine Instanz der Klasse.

```
c = new Circle();
```

Ein Konstruktor wird immer mit dem Schlüsselwort `new` aufgerufen. Die Instanzvariablen des Objektes `c` wurden nun mit Standardwerten belegt (für Zahlen ist das der Wert 0). Das ist natürlich wenig sinnvoll.

```
c.x = 1.4;
c.y = -1.0;
c.r = 2.5;
```

Mit diesen Anweisungen werden den Instanzvariablen des Objektes `c` Werte zugeordnet. Nun kann man mit dem Aufruf der Methode `area()` den Flächeninhalt des Kreises berechnen.

```
double a = c.area();
```

Die Methode `area()` verwendet ein sogenanntes implizites Argument, nämlich das Objekt `c`, also das Objekt auf dem die Methode arbeitet. Das implizite Argument wird vor dem Methodennamen angegeben. Explizite Argumente werden mit der Parameterliste übergeben.

Der Aufruf von `new Circle()` ist auch ein Methodenaufruf, nämlich der Aufruf der Konstruktormethode für die Klasse `Circle`. Da dieser Konstruktor keinerlei Argumente enthält wird er als Standartkonstruktor bezeichnet. Wird eine Klasse ohne Standartkonstruktor geschrieben (wie unsere Klasse `Circle`), so stellt JAVA automatisch den Standartkonstruktor bereit. Will man selbst einen Konstruktor schreiben, so ist zu beachten das der Name des Konstruktors gleich dem Klassennamen sein muß. Außerdem erzeugt JAVA keinen Standartkonstruktor, falls man selbst Konstruktoren definiert. Eine Konstruktordefinition für unsere Klasse `Circle` wäre:

```
public Circle(double x, double y, double r)
{
   this.x = x;
   this.y = y;
   this.r = r;
}
```

Konstruktormethoden haben keinen Rückgabewert sondern nur Seiteneffekte. In unserem Beispiel werden die Instanzvariablen des erzeugten Objektes mit den an die Konstruktormethode übergebenen Werten überschrieben. Das Schlüsselwort `this` ermöglicht den Zugriff auf das Objekt auf dem die Methode operiert. Jetzt kann man ein `Circle`-Objekt wie folgt erzeugen.

```
Circle c = new Circle(1.4,-1.0,2.5);
```

Mehrere Konstruktoren innerhalb einer Klasse sind erlaubt (Overloading), die Unterscheidung erfolgt anhand der Signaturen (verschiedene Parameterlisten). Man könnte für die `Circle`-Klasse also noch einen Standartkonstruktor definieren der den Einheitskreis erzeugt.

```
public Circle()
{
   x = 0.0;
   y = 0.0;
   r = 1.0;
}
```

Weitere mögliche Konstruktordefinitionen sind:

```
public Circle(double r)
{
   x = 0.0;
   y = 0.0;
   this.r = r;
}

public Circle(Circle c)
{
   x = c.x;
   y = c.y;
   r = c.r;
}
```

3.3.2 Vererbung

Nehmen wir an, das wir die Klasse mit der mathematischen Definition des Kreises um eine farbige Bildschirmausgabe des Kreises erweitern wollen. Dazu definieren wir eine neue Klasse `GraphicCircle`, welche die Klasse `Circle` erweitert. `Circle` ist somit Superklasse und `GraphicCircle` ist Subklasse. Es gilt die "Ist ein"-Beziehung zwischen beiden Klassen, denn ein gezeichneter Kreis "Ist ein" Kreis. Wir nehmen an, daß die Klassen `Color` und `DrawWindow` gegeben sind.

```
public class GraphicCircle extends Circle
{
   private Color outline, fill;
   private double r;

   public GraphicCircle(double x, double y, double radius,
     Color o, Color f)
   {
     super(x,y,radius);
     outline = o;
     fill = f;
   }

   public void setResolution(double resolution)
   {
     r = resolution;
   }
```

```
    public void draw(DrawWindow dw)
    {
      dw.drawCircle(x, y, super.r, this.r, outline, fill);
    }
}
```

Gehen wir nun näher auf den Code ein. Die Klasse enthält Instanzvariablen vom Typ `Color` und eine Instanzvariable `r`, die das Radius-`r` aus der Klasse `Circle` verschattet.

In der Konstruktormethode der Klasse wird mittels `super( .. )` der Konstruktor der Superklasse aufgerufen. Ein solcher Konstruktoraufruf muß stets am Anfang der Deklaration der Konstruktormethode stehen. Wird in der Konstruktordefinition der Konstruktor der Basisklasse nicht explizit aufgerufen, dann ruft JAVA automatisch den Standartkonstruktor der Basisklasse auf. Hat die Superklasse keinen selbst definierten Konstruktor, so muß `super( .. )` verwendet werden.

Mit `super.r`, wird auf das `r` der Superklasse zugegriffen, während mit `this.r` auf das `r` der aktuellen Klasse zugegriffen wird. Generell gilt für den Variablenzugriff innerhalb einer Vererbungshierarchie:

Gelte: A ist Basisklasse
B ist Unterklasse von A
C ist Unterklasse von B

die Variable `x` ist in allen drei Klassen als Instanzvariable definiert

`this.x`	x in C
`super.x`	x in B
`((B)this).x`	x in B
`((A)this).x`	x in A

Im folgenden sei kurz dargelegt was eine Unterklasse von ihrer Basisklasse erbt.

1. Die Unterklasse erbt alle Instanzvariablen der Superklasse die erreichbar sind. Das sind alle Variablen, die als `public` oder `protected` deklariert sind oder die keine Angabe der Zugriffsbeschränkung haben (`package`). Auf die als private deklarierten Variablen der Basisklasse hat die Subklasse keinen Zugriff. Um auf verschattete Variablen zuzugreifen bestehen verschiedene Möglichkeiten (siehe oben).

2. Die Unterklasse erbt alle Methoden der Superklasse die erreichbar sind. Das sind alle Methoden, die als `public` oder `protected` deklariert sind oder die keine Angabe der Zugriffsbeschränkung haben (`package`). Auf die als private deklarierten Methoden der Basisklasse hat die Subklasse keinen Zugriff.

   Die überschriebenen Methoden können nicht durch das Voranstellen des Oberklassennamens aufgerufen werden. Die überschreibende Methode ersetzt die überschriebene Methode in allen Objekten der Unterklasse. Aber der Aufruf einer überschriebenen Methode `m` ist in der Unterklasse, die die Überschreibung vornimmt, durch `super m` möglich.

Das Ableiten einer Klasse kann durch die Deklaration als `final` verhindert werden. Es ist ebenso möglich einzelne Methoden als `final` zu deklarieren, diese Methoden können von der Unterklasse dann nicht überschrieben werden.

3.3.3 Klassenvariablen und Klassenmethoden

Alle Instanzierungen (Ausprägungen) einer Klasse verwenden die gleiche Implementierung einer Methode, aber jedes Objekt hat eine eigene Version der Instanzvariablen.

Als `static` deklarierte Variablen einer Klasse werden als Klassenvariablen bezeichnet. Klassenvariablen werden nur einmal für alle Objekte der Klasse erzeugt.

Als `static` deklarierte Methoden (Klassenmethoden) können verwendet werden ohne das eine Instanz der Klasse erzeugt wurde. Solch Methoden operieren auf Klassenebene. Eine Klassenmethoden kann nur auf Klassenvariablen und andere Klassenmethoden der Klasse zugreifen. Auf die Instanzvariablen der Klasse besteht kein Zugriff. Klassenmethoden können die `this`-Referenz nicht verwenden (`this` ist objektbezogen).

Ein Beispiel für die Verwendung von Klassenmethoden und -variablen liefert die Klasse `Math` des Paketes `java.lang`. Sie stellt die Klassenkonstanten `E` und `PI` (als `public static final` deklariert) sowie verschiedene Klassenmethoden für mathematische Standardfunktionen, wie zum Beispiel `sqrt()`, zur Verfügung. Die Klasse arbeitet ausschließlich auf primitiven Typen, genauer auf Zahlen.

Eine mögliche Klassenmethode für die Klasse `Circle` ist:

```
public static isBigger(Circle a, Circle b)
{
   if(a.r > b.r)
     return a;
   else
     return b;
}
```

Der Aufruf der Funktion gestaltet sich wie folgt:

```
Circle a = new Circle();
Circle b = new Circle(3,4,2);

Circle c = isBigger(a,b);
```

Man kann die Funktionalität dieser Funktion natürlich auch mit einer Instanzmethode implementieren:

```
public Circle isBigger(Circle a)
{
   if(a.r > this.r)
     return a;
   else
     return this;
}
```

Der Aufruf der Instanzmethode gestaltet sich folgendermaßen:

```
Circle a = new Circle();
Circle b = new Circle(3,4,2);

Circle c = a.isBigger(b);
Circle d = b.isBigger(a);
```

Dabei ist aber folgendes zu beachten, wenn die beiden Kreise `a` und `b` verschiedene Radien haben, so zeigen die Variablen `c`, `d` und eine der Variablen `a` oder `b` auf ein und das selbe `Circle`-Objekt.

Auch eine Klassenvariable kann man zu der Klasse `Circle` hinzufügen. Um diese aber auch halbwegs sinnvoll zu verwenden, müssen die Konstruktormethoden der Klasse angepaßt werden.

```
static int num_of_circs = 0;

public Circle()
{
   x = 0.0;
   y = 0.0;
   r = 1.0;
   num_of_circs++;
}
```

Mit jedem Konstruktoraufruf wird nun die Variable `num_of_circs` um eins erhöht. Im Testprogramm kann man dann die Zahl der erzeugten Kreise ausgeben:

```
System.out.println("Erzeugte Kreise:  "+ Circle.num_of_circs)
```

Im folgenden sei ein Programm dargestellt das die Auswirkungen von Verschattung und Überschreiben demonstriert:

```
class A {
   int i = 1;

   int f() {
     return i; }
}

class B extends A {
   int i = 2;

   int f() {
     return -i; }
}

public class Override
{
   public static void main(String args[])
   {
     B b = new B();

     System.out.println(b.i);    //gibt 2 aus
     System.out.println(b.f());  //gibt -2 aus
     A a = (A)b;
     System.out.println(a.i);    //gibt 1 aus
     System.out.println(a.f());  //gibt -2 aus
     }
}
```

Der Grund für dieses Verhalten ist folgender. Der Objektzeiger `b` ist vom Typ `B` und zeigt auch auf ein solches Objekt. Allerdings wird der Typ des Objektzeigers auf `A` geändert (upcasting). Der eigentliche Typ des Objekts wird heruntergespielt. Der Aufruf `a.i` führt dazu das JAVA auf die öffentliche Variable `i` des Objektes `a` zugreift und demzufolge wird 1 ausgegeben. Beim Aufruf von `a.f()` hingegen kommt zum tragen, das Methoden fest an ein Objekt gebunden sind. JAVA greift also auf die Methode `f()` des Objektes `B` zu. Diese Methode gibt die Instanzvariable des Objektes `B` aus.

Nun wollen wir unsere Klasse `Circle` zu einer Klasse `Ellipse` erweitern. Auch wenn sich dies auf den ersten Blick irrsinnig anhört. Eine verkürzte Version wäre:

```
public class Ellipse extends Circle
{
   double r2 = 1.0;
   public Ellipse()
   {
     super();
   }

   public Ellipse(double x, double y, double r, double r2)
   {
     super(x,y,r);
     this.r2 = r2;
   }

   public double area()
   {
     return 3.14 * r * r2;
   }
}
```

Nun könnte man ein Array erzeugen das Kreise und Ellipsen enthält und sich dann die Fläche der einzelnen Objekte mittels `area()` berechnen lassen. Die Bindung der richtigen `area()`-Methode für das entsprechende Objekt im Array erfolgt erst zur Laufzeit. Dieses Verhalten wird als dynamisches (spätes) Binden bezeichnet.

```
Circle cFeld[] = new Circle[3];

cFeld[0] = new Circle();
cFeld[1] = new Ellipse(5,6,1,2);
cFeld[2] = new Ellipse(2,2,4,2);

System.out.println(Fläche:  "+ cFeld[0].area());
System.out.println(Fläche:  "+ cFeld[1].area());
System.out.println(Fläche:  "+ cFeld[2].area());
```

3.3.4 Abstrakte Klassen und Methoden

Abstrakte Klassen werden für sehr allgemeine Konzepte verwendet. Sie dienen eher als eine Art "Schablone" für die abgeleiteten Klassen. Eine Klasse ist muß immer dann als `abstrakt` deklariert sein, wenn sie mindestens eine abstrakte Methode enthält. Eine als `abstrakt` deklarierte Methode enthält keinen Methodenrumpf. Eine abstrakte Methode verspricht, das alle nicht abstrakten Subklassen diese abstrakte

Methode implementieren.

Eine abstrakte Klasse ist nicht instanzierbar (keine Objekte als Ausprägungen). Aber sie kann natürlich sowohl konkrete als auch abstrakte Unterklassen haben. Eine Klasse kann selbst dann als `abstrakt` deklariert sein wenn sie keine abstrakten Methoden enthält.

Dazu nun ein Beispiel an unserer altbekannten Klasse `Circle`:

```
public abstract class Shape
{
   public static final double Pi = 3.14159265358979323846;

   public abstract double area();
   public abstract double cf();
}

public class ExtCircle extends Shape
{
   private double x=0, y=0, r=1;

   public ExtCircle() { }

   public ExtCircle(double r) {
     this.r = r; }

   public ExtCircle(ExtCircle c) {
     x = c.x;
     y = c.y;
     r = c.r; }

   public ExtCircle(double x,double y,double r) {
     this.x = x;
     this.y = y;
     this.r = r; }

   public double cf() {
     return 2 * Shape.Pi * r; }

   public double area() {
     return Shape.Pi * r * r; }

   public void setRadius(double r) {
     this.r = r; }
   public double getRadius() {
     return r; }

   public void setCoord(double x,double y) {
     this.x = x;
     this.y = y; }
   public double getxCoord() {
     return x; }
   public double getyCoord() {
     return y; }
}
```

Nun könnte man natürlich auch eine Klasse `ExtRectangle` aus `Shape` ableiten:

```
public class ExtRectangle extends Shape
{
   private double x=0, y=0, h=1, w=1;

   public ExtRectangle() { }
   public ExtRectangle(double h,double w) {
     this.h = h; this.w = w; }
   public ExtRectangle(ExtRectangle c) {
     x = c.x; y = c.y; h = c.h; w = c.w; }
   public ExtRectangle(double x,double y,double h,double w) {
     this.x = x; this.y = y; this.h = h; this.w = w; }

   public double cf() {
     return 2 * w + 2 * h; }

   public double area() {
     return w * h; }

   public void setWidth(double w) {
     this.w = w; }
   public double getWidth() {
     return w; }

   public void setHight(double h) {
     this.h = h; }
   public double getHight() {
     return h; }

   public void setCoord(double x,double y) {
     this.x = x; this.y = y; }
   public double getxCoord() {
     return x; }
   public double getyCoord() {
     return y; }
}
```

Nehmen wir nun an wir wollten die Gesamtfläche einer Liste von `Shape`-Objekten berechnen. Dies könnte nun folgender Code tun:

```
double total = 0;
Shape shapes[] = new Shape[4];

shapes[0] = new ExtRectangle();
shapes[1] = new ExtCircle();
shapes[2] = new ExtRectangle(6,7,3,6);
shapes[3] = new ExtCircle(-1,2,4);

for(int i = 0; i < shapes.length; i++) {
   total += shapes[i].area();
   System.out.println("+ "+ df.format(shapes[i].area())); }

System.out.println(- "+ df.format(total));
```

3.3.5 Methodenauswahl und Bindung

Es existieren in JAVA zwei Möglichkeiten wie Methoden an Objekte gebunden werden:

- **statische Bindung**: (frühe Bindung) die Bindung der Methoden erfolgt während der Übersetzung durch den Compiler.
- **dynamische Bindung**: (späte Bindung) die Bindung der Methoden erfolgt dynamisch während der Laufzeit des Programms.

Im folgenden sei das Verfahren der Methodenauswahl näher erklärt. Dabei werden der Vereinfachung wegen die folgenden Einschränkungen gemacht:

- die Methoden sind parameterlos (sonst Überladen)
- alle Klassen sind als `public` deklariert und es werden keine Schnittstellen verwendet
- der Rückgabetyp der betrachteten Methoden ist `void` (sonst wird Typüberprüfung durchgeführt)

Die Methodenauswahl erfolgt in 2 Schritten. Der erste Schritt erfolgt während der Übersetzungszeit. Diese sogenannte Bindungsvorbereitung wird aus Effizienzgründen durchgeführt. Der zweite Schritt erfolgt zur Laufzeit. Der erste Schritt der Methodenauswahl zerfällt in drei Teilschritte:

1.a Es wird die zum Methodenaufruf gehörige Deklaration gesucht. Kriterien sind der Typ des aufrufenden Objektes (Klasse des Objektes) sowie der Bezeichner der Methode.

1.b Es erfolgt die Prüfung des Zugriffsrechtes. Dieses ist abhängig von den Zugriffsmodifikatoren der Methodendeklaration.

1.c Es erfolgt die Festlegung des Aufrufsmoduses. Für Methoden die als `private`, `static` oder `final` deklariert sind ist der Aufrufmodus "nonvirtual" (statische Bindung). Für alle anderen Methoden ist der Aufrufmodus "virtual" (dynamische Bindung).

Der zweite Schritt der Methodenauswahl:

2.a Es erfolgt die Bestimmung des sogenannten Zielobjektes, also die Bestimmung des Objektes auf dem die Methode ausgeführt werden soll (Auswerten des Aufrufausdruckes), und seiner Klasse, also der Klasse die bei der Erzeugung des Objektes (mit `new`) verwendet wurde.

2.b Nun erfolgt die Auswahl der auszuführenden Methode. Wenn der Aufrufmodus "nonvirtual" ist, also statische Bindung vorliegt, wird die entsprechende Methodendeklaration genutzt.

Ist der Aufrufmodus hingegen "virtual", liegt also dynamische Bindung vor, so erfolgt eine systematische Suche nach der Methodendeklaration mit passender Signatur. Die Suche startet bei der Klasse des Zielobjektes und wird, falls keine passende Methode gefunden wurde in der direkten Oberklasse fortgesetzt usw.. Es wird maximal bis zu einer statischen Methodendeklaration gesucht. Die erste Methode deren Signatur stimmt wird verwendet.

Hieraus ergeben sich die Folgerungen:

1. Die ausgewählte Methode muß mindestens die gleichen Zugriffsrechte gewähren, wie die überschriebene Methode, da der Zugriff nur statisch geprüft wird.

2. Wenn in einer Unterklasse eine Methode mit gleicher Signatur wie eine in der Oberklasse verborgene Methode deklariert ist, so ist dies kein Überschreiben sondern Neudefinition.

3.3.6 Polymorphismus

Im folgenden sei kurz der Unterschied zwischen funktionalen und objektorientierten Sprachen in Bezug auf Polymorphismus herausgearbeitet:

- Bei funktionalen Sprachen wie zum Beispiel GOFER: Es existieren Typkonstanten (`Bool`, `Int`, ...) und Typvariable (`[t]`, `t` ist Typvariable). Typvariable sind in Funktionssignaturen erlaubt, damit werden polymorphe Funktionen erzeugt. Eine polymorphe Funktion in einer funktionalen Sprache besitzen die gleiche Signatur und gleichen Funktionsrumpf. Es variieren nur die Typen, damit ist die Funktion auf verschiedene Typen anwendbar (Hilfsmittel der Typinferenz).

- In objektorientierten Sprachen besitzen polymorphe Funktionen die selbe Signatur aber verschiedene Rümpfe (Überschreiben). Die Typvariablen der funktionalen Sprachen entsprechen in etwa den Oberklassen der objektorientierten Sprachen (Typ des Objektes).

3.3.7 Schnittstellen

In Java gibt es keine Mehrfachvererbung von Klassen. Es kann aber nötig sein, Methodendeklarationen von mehr als einer Klasse zu erben. Schnittstellen bieten hierfür eine Ersatzmöglichkeit.

Eine Schnittstelle (Interface) ist eine besondere Form einer Klasse, die ausschließlich abstrakte Methoden und Konstanten enthält. Anstelle von `class` wird zur Definition eines Interfaces das Schlüsselwort `interface` verwendet. Alle Methoden sind daraufhin standardmäßig abstrakt. Eine Schnittstelle kann keine Konstruktoren enthalten. Demzufolge sind Schnittstellen nicht instanzierbar, wohl können Variablen aber vom Typ einer Schnittstelle sein. In diesem Fall kann man ihnen Objekte zuweisen, die aus Klassen abstammen, die dieses oder einer der daraus abgeleiteten Schnittstelle implementieren.

Schnittstellen lassen sich vererben. Wird eine Schnittstelle `A` aus einer Schnittstelle `B` abgeleitet, so erbt sie alle Deklarationen von `B` und kann diese um eigene Erweiterungen ergänzen.

Was man bei der Vererbung von Klassen als Ableitung bezeichnet, wird bei Schnittstellen Implementierung genannt. Durch das Implementieren einer Schnittstelle verpflichtet sich die Klasse, alle Methoden, die in der Schnittstelle definiert sind, zu implementieren. Fehlt eine Methode, so gibt es einen Compilerfehler. Die Implementierung einer Schnittstelle wird durch das Schlüsselwort `implements` bei der Klassendefinition angezeigt.

Ein Beispiel wäre:

```
public interface Drawable
{
   public void setColor(Color c);
   public void setResolution(double r);
   public void draw();
}
```

```
public class DrawRect extends ExtRectangle implements Drawable
{
   private Color c;
   private double r = 100;

   public DrawRect(double x, double y, double w, double h)
   {
     super(x, y, w, h);
   }

   public void setColor(Color c)
   {
     this.c = c;
   }

   public void setResolution(double r)
   {
     this.r = r;
   }

   public void draw()
   {
   }
}
```

Schnittstellen lassen sich wie Datentypen verwenden:

```
Drawable[] drawables = new Drawable[3];
```

3.4 Objektorientiertes Modellieren

Für den objektorientierten Entwurf sind folgende Fragestellungen von Bedeutung:

1. Welche Objekte sind nötig?
2. Welche Merkmale sollen die Objekte besitzen?
3. Wie sollen die Objekte Klassifiziert werden? Welche Beziehungen soll es zwischen den Klassen geben?
4. Wie sollen die Objekte verwendet werden?
5. Welche Zustände sollen die Objekte annehmen können? Welche Ereignisse führen zu einer Zustandsänderung? Was tun die Objekte? Wie wirken sie zusammen?

Für die Darstellung der Antworten bedient man sich der folgenden Hilfsmittel.

- **Objektmodel**: (für Fragen 1 - 3) Klassen und Objektdiagramme
- **Anwendungsfälle**: (für Frage 4) typische Anwendungen, Tätigkeiten und Interaktionen von Objekten (use cases)
- **Verhaltensmodelle**: (für Frage 5) Darstellung des zeitlichen Verhaltens der Objekte bzw. Methoden (endliche Automaten). Eine sehr verbreitete Modellierungsmethode sind UML-Diagramme.

Ein Beispiel: Ein Student bestellt einen Rechner bei einer Firma RW (Kunde - Anbieter). Auf die Frage welche Objekte man verwenden sollte, sind verschiedene Antworten möglich. Eine Lösungsmöglichkeit besteht darin nach Kooperationen der Personen oder Geräte zu suchen. Also Entwurf nach Zuständigkeit. Die Objekte werden bezüglich ihrer Rollen festgelegt. Als Hilfsmittel können die sogenannten KZK-Karten (Klasse- Zuständigkeit - Kooperation) verwendet werden.

<table>
<tr><td>Klassenname ...</td><td rowspan="2">Zusammenarbeit mit ...</td></tr>
<tr><td>Zuständig für ...</td></tr>
</table>

Beobachtungen:

1. Es treten oft Überspezifikationen auf, so sollte man zum Beispiel besser eine Klasse Kunde statt Student definieren.

2. Das erzeugte Objektmodell ist oft noch Unvollständig, so fehlen zum Beispiel noch Klassen wie Transport oder Paket.

Ein anderes allgemeines Verfahren zum finden von Klassen und Objekten ist die Substantiv - Verb - Analyse. Die Substantive entsprechen den Objekten oder Daten und die Verben den Methoden.

Kapitel 4

Codierung und Informationstheorie

In diesem Kapitel beschäftigen wir uns mit der Darstellung von Informationen und Daten, mit technischen Mitteln, in Computern. Das heißt die Darstellung erfolgt durch Folgen aus Nullen und Einsen. Sie sind nicht nur Zahlen für das Rechnen sondern auch Informationen die übertragen und gespeichert werden. Wichtige Kriterien für die Darstellung der Informationen im Computer sind:

1. Ökonomie und Effizienz bei der Speicherung (möglichst kompakt) und der Verarbeitung (möglichst schnell).
2. Sicherheit bei der Übertragung, in erster Linie sind das Fehlererkennung und Fehlerkorrektur, weniger Abhörsicherheit

4.1 Codes und Codierung

Sei im folgenden A eine endliche Menge von Zeichen. Die Menge A wird auch als Zeichenvorrat bezeichnet. Ist A geordnet heißt es Alphabet. $\mathbb{B}$ ist die Menge der Binärzeichen.

$$\mathbb{B} = \{\mathbf{L}, \mathbf{0}\}$$

A^n und B^n sind die Menge aller Wörter der Länge $n \in \mathbb{N}$ über A beziehungsweise B. Ein Code ist eine Abbildung:

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

Oft entsprechen Codes partiellen Abbildungen, das heißt, sie sehen nicht für alle Wörter aus A^* Darstellungen vor. Besteht der Zeichenvorrat A nur aus Einzelzeichen, so spricht man von Chiffrierung. Eine solche Abbildung ist:

$$c' : A \to B^*$$

Man codiert Wörter endlicher Länge über A ($c : A^* \to B^*$) dann mit:

$$c(a) = c'(a) \qquad a \in A$$

(Die Fortsetzung erfolgt mit Homomorphie[1] Bedingungen)

$$c(ab) = c(a) \circ c(b) \qquad a, b \in A$$

$$c(wa) = c(w) \circ c(a) \qquad a \in A, w \in A^*$$

$$c(\varepsilon) = \varepsilon$$

[1]vergleiche HM 1, lineare Abbildungen: $f(u \bullet v) = f(u) \bullet f(v)$

Es gilt also: "Das Bild der Zusammensetzung ist die Zusammensetzung der Bilder".

Im allgemeinen ist es eine wichtige Eigenschaft von Codes, daß die Abbildung c injektiv ist, das heißt, daß die Decodierung (Umkehrabbildung) eindeutig ist. Es gilt dann:

$$d\{c(a)|a \in A\} \to A \quad \text{und damit} \quad d(c(a)) = a \quad \forall a \in A$$

Ein Spezialfall ist die Binärcodierung $B = \mathbb{B}$ von Alphabeten, wobei A ein vorgegebenes Alphabet ist.

$$c' : A \to \mathbb{B}^*$$

Im folgenden werden wir uns aber fast ausschließlich mit diesem Fall beschäftigen.

Manchmal treten auch Abbildungen der Form: $c'' : A \to \mathbb{B}$ auf (Umbenennen und Zusammenlegen).

4.1.1 Codes einheitlicher Länge

Bei Codes einheitlicher Länge wird jedes zu codierende Zeichen auf ein Binärwort gleicher Länge n mit $n \in \mathbb{N}$ abgebildet.

$$c' : A \to \mathbb{B}^n$$

Ein Beispiel für einen solchen Code ist der 4-Bit-Cray-Code für Dezimalziffern. A ist in diesem Fall die Menge der Ziffern $\{0, ..., 9\}$. Von i zu $i+1$ ($i \in \mathbb{N}_0; i < 10$) ändert sich beim Cray-Code immer nur genau ein Bit.

z	$c(z)$
0	**OOOO**
1	**OOOL**
2	**OOLL**
3	**OOLO**
4	**OLLO**
5	**OLLL**
6	**OLOL**
7	**OLOO**
8	**LLOO**
9	**LOOO**

Aufgrund dessen, daß sich die Codewörter für zwei benachbarte Zeichen immer um jeweils ein Bit unterscheiden wird dieser Cray-Code auch als einschrittiger Cray-Code bezeichnet. Unterscheiden sich in einem Cray-Code die Zeichen des ersten und des letzten Zeichens in nur einer Stelle, so spricht man von einem zyklischen Cray-Code.

Hammingabstand

Sind a und b Binärwörter gleicher Länge, so bezeichnet man die Anzahl der Zeichen in denen sich die beiden Wörter unterscheiden als Hammingabstand. Der Hammingabstand entspricht einer mathematischen Abbildung:

$$\mathsf{Ham\_dis}(< a_1, ..., a_n >, < a_1, ..., a_n >) = \sum_{i=1}^{n} d_i,$$

$$\text{wobei} \quad d_i = \begin{cases} 1 & \text{falls } a_i \neq b_i \\ 0 & \text{sonst} \end{cases}$$

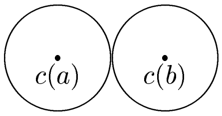
Der Hammingabstand einer Codierungsabbildung $c : A \to \mathbb{B}^*$ der Codelänge n ist durch den kleinsten Hammingabstand zwischen den Codewörtern gegeben.

$$\mathsf{Ham\_dis}(c) = \min\{\mathsf{Ham\_dis}(c(a), c(b)) : a, b \in A \wedge a \neq b\}$$

Ist c injektiv, so ist der Hammingabstand von c größer oder gleich Eins. Beim Cray-Code ist der Hammingabstand Eins.

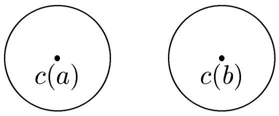
Ein Hammingabstand größer Eins erlaubt das erkennen von Fehlern in Codewörtern, wie sie Beispielsweise bei Übertragungen auftreten können. Dies liegt darin begründet, das beim Auftreten eines Ein-Bit-Fehlers ein Binärwort entsteht dessen Hammingabstand zum Original Eins ist. Da nun aber alle Codewörter mindestens den Hammingabstand zwei haben, kann das fehlerhafte Wort als Nichtcodewort identifiziert und damit auch der Fehler festgestellt werden.

Dies läßt sich auch graphisch darstellen, indem man die Binärwörter als Punkte mit einem Abstand zueinander, der dem Hammingabstand entspricht, zeichnet und den Fehler als Kreis mit entsprechenden Radius einzeichnet:



Bei einem Ein-Bit-Fehler haben die Kreise den Radius Eins (entspricht Hammingabstand). Ändert sich das Codewort durch einen Ein-Bit-Fehler, so entsteht ein Nichtcodewort, da die Ein-Bit-Änderung an den Kreisrand führt.

Wird der Hammingabstand eines Codewortes größer Zwei gewählt, so ist es möglich Ein-Bit-Fehler zu korrigieren, denn es existieren damit keine verschiedenen Codewörter die durch einen Ein-Bit-Fehler zu einem Binärwort gleicher Bitfolge werden könnten:



Ob ein Ein-Bit-Fehler oder ein n-Bit-Fehler auftritt, kann man im allgemeinen nicht feststellen. Man geht aber davon aus das n-Bit-Fehler mit $n \geq 2$ nur äußerst selten auftreten.

Es existieren unterschiedliche Ansätze, um einen Code so zu erweitern das er einen bestimmten Hammingabstand aufweist. Ein einfacher Ansatz ist es die Codewörter um zusätzliche Bits zu erweitern. Im einfachsten Fall, wird einfach ein Prüfzeichen, das Parity-Bit, an den Code angehangen.

Sei nun $c_1 : A \to \mathbb{B}^n$ ein Code mit Hammingabstand 1. Ein Code $c_2 : A \to \mathbb{B}^{n+1}$ mit Hammingabstand 2 läßt sich aus c_1 konstruieren, mit:

$$c_2(a) = c_1(a) \circ pb(a) \quad \text{mit} \quad pb(a) \begin{cases} \mathbf{L} & \text{falls } qs(a) \text{ gerade} \\ \mathbf{O} & \text{sonst} \end{cases}$$

$qs(a)$ ist die Anzahl der Zeichen $\mathbf{L}$ in $c(a)$, also die binäre Quersumme.

z	$c(z)$	$pb(z)$
0	**OOOO**	**L**
1	**OOOL**	**O**
2	**OOLO**	**O**

Der Beweis dazu: Seien $a_1, a_2 \in A$ mit $a_1 \neq a_2$. Sei weiterhin der Hammingabstand $\mathsf{Ham\_dis}(c_1(a_1), c_1(a_2)) = 1$. Dann folgt aus $qs(c_1(a_1))$ ist gerade, $qs(c_1(a_2))$ ist ungerade, und damit $pb(c_1(a_1)) \neq pb(c_1(a_2))$. Damit gilt:

$$\mathsf{Ham\_dis}(c_1(a_1) \circ pb(c_1(a_1)), c_1(a_2) \circ pb(c_1(a_2))) = 1 + 1 = 2$$

Im weiteren gilt: $\mathsf{Ham\_dis}(c_1(a_1), c_1(a_2)) > 1 \quad \Rightarrow \quad \mathsf{Ham\_dis}(c_2(a_1), c_2(a_2)) > 1$.

Im folgenden sei ein Beispiel für die Konstruktion eines Fehlerkorrigierenden Codes für 1-Bit-Fehler aus einem 4-Bit-Code gegeben. Sei $w =< b_1, b_2, b_3, b_4 >$ ein Codewort. Daraus wird durch das Hinzufügen der Bits b_5, b_6 und b_7 ein 7 Bit langes Codewort konstruiert. Jedes dieser Zusatzbits ist Parity für einen Teil von w:

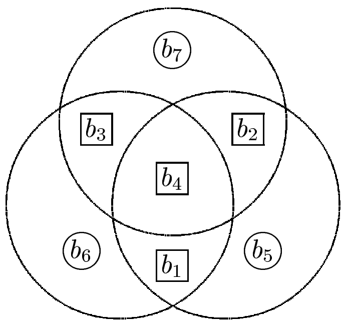
b_5 für b_1, b_2, b_4
b_6 für b_1, b_3, b_4
b_7 für b_2, b_3, b_4

Also wird in etwa das Codewort **LOLL** zu **LOLLOLO** erweitert. Tritt nun in w ein 1-Bit-Fehler auf, dann stimmen mindestens 2 der Paritybits (b_5, b_6, b_7) nicht. Die Lokalisierung des Fehlers erfolgt durch Fallunterscheidung:

b_5, b_6, b_7 falsch	$\Rightarrow$	b_4 fehlerhaft
b_5, b_6 falsch	$\Rightarrow$	b_1 fehlerhaft
b_5, b_7 falsch	$\Rightarrow$	b_2 fehlerhaft
b_6, b_7 falsch	$\Rightarrow$	b_3 fehlerhaft

Tritt ein 1-Bit-Fehler in einem der Paritybits (b_5, b_6, b_7) auf, so ist genau eines der Paritybits falsch.

Das hier vorgestellte System für Fehlerkorrektur läßt sich anschaulich mit einem Mengendiagramm für die Bitdarstellung darstellen:



Ist der Code korrekt, so weist jeder der Kreise eine gerade Anzahl von **L**'s auf.

Codierung ganzer Zahlen

Sollen nicht nur Ziffern sondern ganze Zahlen allgemein codiert werden, so wird häufig ein sogenanntes Stellenwertsystem verwendet. Hierbei wird die Umkehrabbildung betrachtet und den einzelnen Binärstellen des Binärwortes werden Gewichtsfunktionen g_i zugeordnet. Für die Binärzeichen werden die Gewichte $w(\mathbf{L}) = 1$ und $w(\mathbf{O}) = 0$ definiert. Die Zahl die einem Binärwort:

$$d =< d_n, d_{n-1}, ...d_1, d_0 > \quad d_i \in \mathbb{B}$$

zugeordnet wird ergibt sich dann mit:

$$\sum_{i=0}^{n} g_i * w(d_i)$$

Ein Beispiel ist die Gewichtung $g_i = 2^i$; $0 \leq i \leq n$. Sie ergibt für die Ziffern 0 bis 9 den sogenannten direkten 4-Bit-Code. Ein anderes Beispiel ist der Aiken-Code, hier betragen die Gewichtungen $g_i = 2|4|2|1$.

Tritt in einem Binärwort der Länge n genau k mal das Zeichen **L** auf so spricht man von einen k-aus-n-Code. Ein Beispiel hierfür ist der 1-aus-10-Code für Dezimalziffern.

z	$c(z)$
0	**OOOOOOOOOL**
1	**OOOOOOOOLO**
2	**OOOOOOOLOO**
3	**OOOOOOLOOO**
4	**OOOOOLOOOO**
5	**OOOOLOOOOO**
6	**OOOLOOOOOO**
7	**OOLOOOOOOO**
8	**OLOOOOOOOO**
9	**LOOOOOOOOO**

Ein weiteres Beispiel ist der Kettencode, er wird durch schrittweises Vorbeiwandern eines Auslesefensters an einer Zeichenfolge gebildet. Man wählt also eine Zeichenfolge der Länge n, zum Beispiel s = **OOOLLLOL** und ein Fenster der Größe m wobei natürlich gelten muß $n > m$, zum Beispiel $n = 4$. Schritt für Schritt schiebt man nun s über das Fenster, in jeder Fensterposition wird der Sichtbare Abschnitt des Binärwortes als Codewort für die Nummer der Fensterposition gewählt. Ein Beispiel ist der 4-Bit-Kettencode:

z	$c(z)$
1	**OOOL**
2	**OOLL**
3	**OLLL**
4	**LLLO**
5	**LLOL**
6	**LOLO**
7	**OLOO**
8	**LOOO**

4.1.2 Codes variierender Länge

Codes variierender Länge sind technisch schwieriger zu handhaben als Codes fester Länge. Sie bieten aber Vorteile, wenn bekannt ist mit welcher mittleren Häufigkeit die zu codierenden Zeichen auftreten. Zeichen die häufig auftreten erhalten kurze

Codes und Zeichen die nicht so oft auftreten längere. Ein Beispiel hierfür ist der Morsecode. Er verwendet die Codezeichen $\bullet$ (kurz), $-$ (Lang) und $\sqcup$ (Lücke). Einige Beispiele:

e:	$\bullet$
t:	$-$
a:	$\bullet -$
w:	$\bullet - -$

Zahlen werden 5-stellig dargestellt, Sonderzeichen 6-stellig. Schwierig ist es bei Codes variierender Länge die Trennung zwischen den einzelnen Codewörtern zu erkennen, wenn ganze Sequenzen von Codewörtern vorliegen. Beim Morsecode wird daher zwischen den einzelnen Zeichen eine Lücke $\sqcup$ gelassen.

4.1.3 Übertragen und Decodieren

1. **Beschreibung von Codes**: Unter der Voraussetzung das A endlich ist und auch die Chiffrierung c eine endliche Länge aufweist, ist die Chiffrierung c durch eine Tabelle darstellbar. Andernfalls existieren verschiedene Verfahren bzw. Algorithmen zur Berechnung von $c(w)$, $w \in A^*$. Solche Algorithmen können zum Beispiel für die Datensicherheit eingesetzt werden (im Hinblick auf "Spione").

2. **Übertragung**: Es existieren zwei Verfahren um codierte Informationen zu Übertragen. Bei Codes fester Länge n wird im allgemeinen eine parallele Übertragung der Bits über n Leitungen (je eine pro Zeile) verwendet. Die Taktungen auf allen Leitungen sind dabei gleich. Das zu übertragende Wort läßt sich als Spaltenvektor darstellen:

$$\begin{pmatrix} \mathbf{O} \\ \mathbf{L} \\ \mathbf{O} \\ \mathbf{L} \end{pmatrix}$$

   Für Codes variabler Länge ist hingegen die serielle Übertragung geeigneter. Denn bei der Parallelübertragung würden viele Leitungen ungenutzt bleiben. Zudem bräuchte man noch ein zusätzliches Signal das anzeigt welche Leitungen gerade zur Übertragung des Codewortes genutzt werden und welche nicht.

3. **Decodierung**: Bei Codes fester Länge n ist die Decodierung trivial, hier braucht man nur in der entsprechenden Tabelle nachzuschauen.

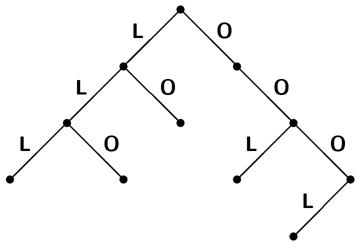
   Bei Codes Variabler Länge, die seriell übertragen werden, ist diese Aufgabe wesentlich komplizierter, da sich kein Zwischenraum oder Trennzeichen zwischen den einzelnen Codewörtern befindet. Um sicherzustellen, daß die Codewörter eindeutig decodierbar sind, wird die folgende nützliche Einschränkung, die sogenannte Fano-Bedingung, getroffen:

   Kein Codewort darf der Anfang eines anderen Codewortes sein.

   Die Fano-Bedingung ist keine notwendige Voraussetzung dafür, daß Codes decodierbar sind.

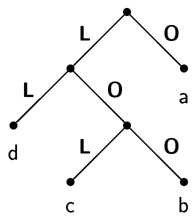
4. Schnellere bzw. einfachere Verfahren der Decodierung: Bei der seriellen Übertragung, gestattet jedes neu empfangene Bit, die Menge der in Frage kommenden Codewörter weiter einzuschränken. Es entsteht ein Entscheidungsbaum

für die schrittweise Decodierung, der auch als **Codebaum** bezeichnet wird. Ein Codebaum ist ein binärer Baum, dessen einzelne Kanten mit **O**, **L** beschriftet sind.



Wenn die Aneinanderreihung der Kantenbeschriftungen auf dem Weg von der Wurzel zu den Blättern ein Codewort $c(x)$ ergibt, so wird der erreichte Knoten mit x beschriftet. Ein Beispiel dazu, anhand eines Präfixcodes. Beim **Präfixcode** stehen die ursprünglichen Zeichen, aus A, ausschließlich an den Blättern des Codebaumes.

z	$c(z)$
a	**O**
b	**LOO**
c	**LOL**
d	**LL**

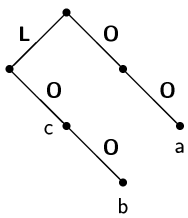


Würden die $x \in A$ im Codebaum an einem inneren Knoten stehen, also vor einem Blatt, dann wäre die Fano-Bedingung nicht erfüllt.

Bei Präfixcode ist es möglich eine Online-Decodierung durchzuführen. Beim Decodieren des Präfixcodes wird der eingehende Bitstring gelesen und beim lesen Zeichen für Zeichen im Codebaum weitergegangen. Gestartet wird an der Wurzel. Tritt an einem Knoten ein Zeichen aus A auf, so ist dies das codierte Originalzeichen. Um weiter zu decodieren startet man wieder von neuem bei der Wurzel. Bemerkung: jeder endliche Binärbaum stellt einen Präfixcode dar.

Ein Beispiel für einen Nicht-Präfixcode ist der im folgenden dargestellte. Hier muß gezeigt werden, daß der Code injektiv ist. Die Decodierung erfolgt mit Backtracking oder Vorausschau.

z	$c(z)$
a	**OO**
b	**LOO**
c	**LO**



Im folgenden sei gezeigt, wie sich ein Bitstring dieses Codes decodieren läßt. Dabei gilt die Voraussetzung, daß der Bitstring aus einer korrekten Folge von Bits, also ausschließlich aus Codewörtern, besteht.

- Beginnt der Bitstring mit einer **O**, so wird ein Paar von zwei Nullen (**OO** jeweils zu "a" decodiert. Man fährt damit solange fort bis man auf ein **L** trifft oder man an das Ende des Bitstrings gelangt.
- Tritt zu Beginn des zu decodierenden (Rest-) Bitstrings ein **L** auf, so müssen die Nullen bis zum nächsten **L** oder bis zum Ende gezählt werden. Ist die Anzahl der **O** gerade, dann wird der Anfang des Bitstrings **LOO** zu "b" decodiert. Ist die Anzahl der **O** ungerade, so wird der Anfang des Bitstrings **LO** zu "c" decodiert. Die verbleibenden **O** werden zu "a" decodiert. Falls der Bitstring noch nicht zu Ende decodiert ist, also noch ein **L** vorkam, wird dieser Punkt wiederholt.

Ein solches Verfahren wird auch als Offline-Decodierung bezeichnet.

Es existieren aber auch Beispiele für Nicht-Präfixcodes die Online Decodierbar sind, allerdings ist Vorausschau nötig, das heißt es wird ein Fenster über den Bitstring geschoben und der jeweilige Inhalt betrachtet. Ein Beispiel für einen solchen Code ist:

z	$c(z)$
a	**O**
b	**OL**

In diesem Fall wird ein zweistelliges Fenster über den Bitstring geschoben. Ist der Inhalt des Fensters **OO** so wird die erste **O** zu "a" decodiert und das Fenster um eine Stelle weitergeschoben. Ist der Inhalt des Fensters **OL** so wird der Inhalt zu "b" decodiert und das Fenster um zwei Stellen weitergeschoben. Ein weiters Beispiel ist:

z	$c(z)$
a	**OL**
b	**OLL**
b	**LOOL**

4.2 Codes und Entscheidungsinformation

Codes sind dann besonders kompakt, wenn man bei der Konstruktion der Codierung die Wahrscheinlichkeit (mittlere Häufigkeit) des Auftretens von Zeichen $a \in A$ berücksichtigt. Kompaktere Codes erlauben zum Beispiel ein wesentlich schnelleres Übertragen der Informationen(Morse Code).

Wir verwenden im folgenden die abstrakte Vorstellung, das ein Strom von Zeichen ($\in A$) von einer **Quelle** Q nach gewissen Gesetzmäßigkeiten erzeugt wird. Dabei spielt die Bedeutung des Codes jetzt eine untergeordnete Rolle wichtig ist nur die Wahrscheinlichkeit des Auftretens der Zeichen. Die Quelle Q erzeugt die Zeichen mit einer bestimmten Wahrscheinlichkeit. Es gilt:

- der verwendete Zeichenvorrat ist endlich
- die einzelnen Zeichen werden zu disjunkten Zeitpunkten erzeugt
- Die Quelle ist stationär, das heißt die wahrscheinlichkeitstheoretischen Eigenschaften der Quelle ändern sich im Laufe der Zeit nicht. Die Häufigkeit der Erzeugung ist für jedes Zeichen bekannt.
- Zu jedem Zeitpunkt ist die Wahrscheinlichkeit für die Erzeugung eines Zeichens $a \in A$ unabhängig von der Vorgeschichte.

Eine solche Quelle wird als **stochastische** oder **Shannonsche**[1] Quelle bezeichnet.

Ein anschauliches Modell für eine stochastische Quelle für einen Zeichenvorrat A ist ein Korb voller gleichartiger Kugeln. Jede der Kugeln ist mit einem bestimmten Zeichen $a \in A$ beschriftet. Dasselbe Zeichen darf dabei durchaus auf mehreren Kugeln auftreten. Die Erzeugung eines Zeichens durch die Quelle entspricht dem Ziehen einer Kugel mit anschließendem Zurücklegen und neuem Durchmischen.

Die mittlere Häufigkeit für die Erzeugung eines Zeichens $a \in A$ durch die Quelle Q ergibt sich wie folgt:

$$\frac{\text{Anzahl der mit } a \text{ beschrifteten Kugeln}}{\text{Anzahl aller Kugeln}}$$

Es gilt zu beachteten das die Zeichenfolgen der natürlichen Sprache nicht in diesem Sinne stochastisch sind, und die hier getroffenen Aussagen nicht für sie gelten.

Nun ist es die Aufgabe, stochastische Quellen mit einer "vernünftigen" Codierung der Zeichen zu suchen. Gehen wir nun ferner davon aus das eine Quelle Nachrichten mit **Informationswert** sendet. Der Informationswert eines Zeichens $a \in A$ läßt sich anschaulich wie folgt definieren: Der **Informationsgehalt** eines Zeichens ist um so größer, je seltener das Zeichen auftritt (Neuigkeitswert). Die Informationsgehalte aufeinanderfolgender Zeichen addieren sich - Additivität. Sei p_a die Erzeugungswahrscheinlichkeit des Zeichens a durch die Quelle Q, dann berechnet sich der Informationsgehalt des Zeichens a wie folgt:

$$\text{Informationsgehalt von } a = \mathrm{ld}\left(\frac{1}{p_a}\right)$$

Dieses Verständnis des Informationsgehaltes von Zeichen wird als **Shannonscher Informationsbegriff** bezeichnet. Der mittlere Informationsgehalt pro Zeichen berechnet sich dann wie folgt:

$$H = \sum_{a \in A} p_a * \mathrm{ld}\left(\frac{1}{p_a}\right)$$

H wird als **Entropie** der Quelle bezeichnet. Sie gibt ein Maß für die Schwankung der mittleren Häufigkeit der Zeichen. Sind alle Zeichen gleich wahrscheinlich, so ist H maximal.

$$H = \sum_{a \in A} p_a * \mathrm{ld}\left(\frac{1}{p_a}\right) = \sum_{a \in A} \frac{1}{n} * \mathrm{ld}(n) = n * \frac{1}{n} * \mathrm{ld}(n) = \mathrm{ld}(n)$$

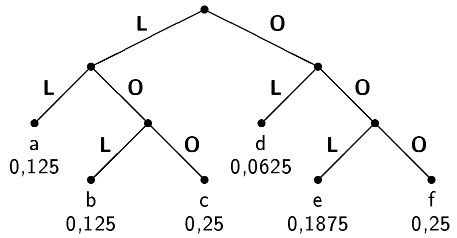
Ein Beispiel mit unterschiedlichen Wahrscheinlichkeiten ist:

$$A = \{\mathrm{a}, \mathrm{b}\} \qquad p_{\mathrm{a}} = 0,75 \quad p_{\mathrm{b}} = 0,25$$

$$H = \frac{3}{4} * \mathrm{ld}(\frac{4}{3}) + \frac{1}{4} * \mathrm{ld}(4) = \frac{3}{4} * (2 - \mathrm{ld}(3)) + 0,5 = 0,81$$

Betrachten wir nun eine Codierung (Chiffrierung) $c : A \to \mathbb{B}^*$. Von der Quelle wird seriell eine Zeichenfolge aus Zeichen $a \in A$ erzeugt und dann mittels der Codierung c codiert. Der Empfänger erhält eine Bitfolge und muß diese wieder decodieren. Es findet eine Nachrichtenübertragung statt. Um die übertragene Nachricht zu decodieren wird ein Codebaum verwendet. Dabei bringt jedes neu übertragene Bit den Empfänger bei der Decodierung eines Codewortes näher an ein Blatt des Codebaumes. In jedem Knoten sieht es so aus, als ob der Sender eine Entscheidung getroffen hätte ob er **O** oder **L** sendet. Für den Sender der codierten Nachricht ist der Codebaum ein Entscheidungsbaum. Der Empfänger stellt die Häufigkeiten des Auftretens von **O** oder **L** fest. Ein Beispiel hierfür ist:

[1] C. E. Shannon geboren 1916, amerikanischer Informatiker, begründete Informationstheorie



Die Wahrscheinlichkeit das in einem Codewort dieses Codes ein **L** an erster Stelle steht ist gleich $(p_a + p_b + p_c) = 0,5$.

Die Wahrscheinlichkeit mit der die Bits der Codewörter erzeugt werden, ist also eindeutig aus der Wahrscheinlichkeit der Erzeugung der $a \in A$ berechenbar (mittels Codebaum).

Nun ist es die Aufgabe eine möglichst kompakte Codierung zu finden, das heißt die mittlere Wortlänge L soll möglichst klein sein. Die mittlere Wortlänge eines Codebaumes L_C erhält man mit:

$$L_C = \sum_{a \in A} p_a * |c(a)|$$

Wobei $|c(a)|$ die Länge des Codewortes $c(a)$ ist und p_a die mittlere Häufigkeit des Zeichens a. Wir können nun versuchen die Binärcodierung so zu wählen, daß die Wortlänge L möglichst klein wird.

Es gilt der folgende Satz: zu einer stochastischen Quelle mit den Zeichen $a_i \in A$ und der Wahrscheinlichkeit p_i existiert ein Präfixcode c, $c : A \to \mathbb{B}^*$, so daß:

$$H \leq L_C < H + 1$$

Um einen kompakten Präfixcode zu konstruieren geht man wie folgt vor: Zu a_i wählt man die Länge des Codewortes l_i so, daß gilt:

$$\operatorname{ld}\left(\frac{1}{p_i}\right) < l_i < \operatorname{ld}\left(\frac{1}{p_i}\right) + 1$$

Zu jedem a_i wählt man nun ein Codewort $c(a_i)$ der Länge l_i so, daß die Fano-Bedingung erfüllt ist. Ein Beispiel dafür ist:

A	a_1	a_2	a_3	a_4	a_5	a_6
p_i	0,25	0,25	0,2	0,15	0,10	0,05
l_i	2	2	3	3	4	4
$c(a_i)$	**LO**	**OL**	$\mathbf{OO\underline{O}}$	**LLL**	**LLOL**	$\mathbf{LLOO\underline{O}}$

Die unterstrichenen **O** sind überflüssig und können entfernt werden. Die Entropie H der Quelle ist gleich 2,423.

Ein optimaler (Präfix-) Code c ist der **Huffmancode**. Optimal bedeutet in diesem Fall das L_C minimal ist. Es existiert stets ein Präfixcode c, $c : A \to \mathbb{B}^*$, mit minimalen L_C. Die Eigenschaften eines solchen Präfixcodes $c : A \to \mathbb{B}^*$ mit minimalen L_C sind:

1. Sei p_i die Wahrscheinlichkeit von a_i und $l_i = |c(a_i)|$ so gilt: $p_i > p_j$ impliziert $l_i \leq l_j$.

2. Mindestens zwei Codewörter haben maximale Länge, ansonsten fällt beim längsten Codewort das letzte Zeichen weg.

3. Von den Codewörtern maximaler Länge L_k stimmen mindestens zwei der Codewörter in den ersten $k-1$ Zeichen überein. Ansonsten läßt man die letzten Zeichen weg.

Für Chiffrierungen $c : A \to \mathbb{B}^*$, die die Fano-Bedingung erfüllen, gilt: $1 \leq L_C$. Dies ist natürlich von Nachteil. Eine Verbesserungsmöglichkeit besteht darin, nicht Zeichen sondern Zeichengruppen bzw. Wörter zu codieren. So könnte man zum Beispiel "sch" als Makrozeichen codieren.

Für eine Zeichenmenge A mit den Häufigkeiten p_a werden dann Wörter (mit einer Länge die ein Vielfaches von $m \in \mathbb{N}$ beträgt) aus A^* mit Hilfe einer Codierungsfunktion:

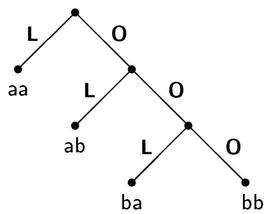
$$c' : A^m \to \mathbb{B}^*$$

codiert. Man bezeichnet dies als Block- oder **Wortcodierung**. Die Länge N_w eines Binärwortes, das ein Wort $w \in A^m$ codiert, ist mit $N_w = |c'(w)|$ gegeben. Die mittlere Wortlänge des Codes c' pro Zeichen des Ausgangswortes ist durch die folgende Formel gegeben:

$$L_C = \frac{1}{m} \sum_{w \in A^m} q_w * N_w$$

Dabei berechnet sich die Wahrscheinlichkeit q_w eines Wortes w aus dem Produkt der Wahrscheinlichkeiten der Einzelzeichen.

Ein Beispiel:



Das folgende Theorem gibt Auskunft über die unteren Grenzen für die mittleren Wortlängen auch im Fall der Wortcodierungen.

Das **Shannonsche Codierungstheorem**:

1. Für beliebige Wortcodierungen c' gilt:

   $$H \leq L_C$$

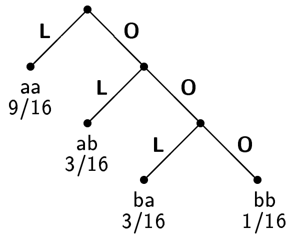
2. Jede stochastische Quelle kann durch Wortcodierungen c' so codiert werden, daß der positive Wert

   $$L_C - H$$

   beliebig klein wird.

Bemerkungen: $L_C - H$ ergibt die **Redundanz** des Codes. Eine große Redundanz ist gut für die Fehlerbehandlung. Weiterhin gilt L_C wird klein wenn in jedem Knoten des Codebaumes die "Gewichte" des linken und rechten Teilbaumes gleich sind. Die "Gewichte" sind hierbei die Summen der Wahrscheinlichkeiten des jeweiligen Teilbaumes. Das heißt der Codebaum sollte so beschaffen sein, daß an einem Knoten die Wahrscheinlichkeit, daß **L** oder **O** kommt, in etwa gleich groß ist.

Ein Beispiel:



Hier ist die mittlere Wortlänge L_C in etwa 0,84 und die Entropie H der Quelle ist etwa gleich 0,81.

4.3 Sicherung der Übertragung

Durch Übertragungsfehler ("Störungen") können die übertragenen Nachrichten verfälscht werden. Wir nehmen an, das jedes Bit mit einer bestimmten Wahrscheinlichkeit gegen Störungen anfällig ist und zwar unabhängig von den zuvor übertragenden Zeichen. Man sagt auch der Kanal hat kein Gedächtnis. Ein wichtiger Spezialfall ist im folgenden genauer charakterisiert. Beim **Binärkanal** entstehen durch Übertragungsfehler stets wieder Binärzeichen, also aus **L** wird durch eine Störung **O** und umgekehrt. Die beiden Zeichen **L** und **O** haben die Irrtumswahrscheinlichkeiten p_L beziehungsweise p_O. Bei der **symmetrischen Störung** gilt, daß $p_L = p_O$. Extrem wäre zum Beispiel $p_L = p_O = 0,5$. Hier erfährt der Empfänger nichts.

4.3.1 Übertragungssicherung

Der Kanal hat eine bestimmte Übertragungsgeschwindigkeit. Im folgenden sei:

- s_0 ist die Anzahl der Zeichen die der Sender pro Zeiteinheit übertragen muß
- s_1 ist die Anzahl der Zeichen die der Kanal pro Zeiteinheit übertragen kann

Die Quellen- oder **Senderate** R berechnet sich nun folgendermaßen: $R = \frac{s_0}{s_1}$ und man kann zwei Fälle bezüglich der Größe von R unterscheiden.

1. Ist $R < 1$, dann ist Redundanz der Codierung möglich. So könnte man zum Beispiel bei $R = \frac{1}{3}$ jedes Zeichen drei mal senden und bei der Decodierung eine Mehrheitsentscheidung durchführen. Des weiteren könnte man natürlich auch einen 4-Bit-Code zu einen fehlerkorrigierenden 7-Bit-Code erweitern.

2. Ist $R \geq 1$, dann ist Packen in Übertragungsblöcke notwendig. Ist zum Beispiel $R = 3$, so könnte man die Nachrichten in Blöcke von je drei Binärzeichen (b_1, b_2, b_3) aufteilen. Das Zeichen das am häufigsten in diesem Trippel vorkommt wird übertragen. Bei der Decodierung werden einfach aus jedem übertragenen Bit drei Bit gemacht (**L** $\rightarrow$ **LLL** und **O** $\rightarrow$ **OOO**). Somit werden nur die Bitfolgen **LLL** $\rightarrow$ **L** und **OOO** $\rightarrow$ **O** korrekt übertragen. Bei allen anderen sechs Möglichkeiten kommt es hingegen zu einem 1-Bit-Übertragungsfehler (**LOL** $\rightarrow$ **L** $\rightarrow$ **LLL**). Damit liegt die Fehlerwahrscheinlichkeit bei einem Viertel Fehler pro Bit.

Kapitel 5

Schaltwerke und Schaltnetze

5.1 Schaltnetze

Schaltnetze sind die technische Realisierung boolescher Funktionen. Schaltnetze entsprechen gerichteten azyklischen Graphen.

1. Boolesche Ausdrücke lassen sich als Boolesche Funktionen interpretieren. Sie können mittels Wahrheitstabellen darstellt werden.

   Zum Beispiel: $x \vee y \quad \rightarrow \quad f(x,y) = \mathbf{O} \quad \Rightarrow \quad x = y = \mathbf{O}$
   $x \wedge y \quad \rightarrow \quad g(x,y) = \mathbf{L} \quad \Rightarrow \quad x = y = \mathbf{L}$

2. Eine n-stellige Boolesche Funktion ist eine Abbildung: $\mathbb{B}^n \rightarrow \mathbb{B}$, mit $\mathbb{B} = \{\mathbf{O}, \mathbf{L}\}$. Eine Boolesche Funktion läßt sich durch eine Wertetabelle eindeutig darstellen:

$x_1, ..., x_n$	$f(x_1, ..., x_n)$
$a_1, ..., a_n$	e_1
...	..
$b_1, ..., b_n$	e_m

   Um aus der Darstellung der Booleschen Funktion durch eine Wertetabelle zu einer Darstellung der Funktion durch Boolesche Terme zu gelangen, geht man wie folgt vor: Man betrachtet nur die Zeilen der Tabelle in denen als Funktionswert ein **L** steht. Für jede dieser Zeilen gibt man die Belegung der Eingangswerte x_i an. Dazu schreibt man x_i falls $x_i = \mathbf{L}$ und $\neg x_i$ falls $x_i = \mathbf{O}$. Die einzelnen x_i bzw. $\neg x_i$ werden durch $\wedge$ miteinander verknüpft. Die Ausdrücke z_j die man für die einzelnen Zeilen erhält werden durch $\vee$ verbunden.

   $$z_j = y_1 \wedge y_2 \wedge ... \wedge y_n \quad \text{mit } y_j = \begin{cases} x_i & \text{für } b_i = \mathbf{L} \\ \neg x_i & \text{für } b_i = \mathbf{O} \end{cases}$$

   $$f(x_1, ..., x_n) = z_1 \vee z_2 \vee ... \vee z_m$$

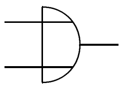
   Denn Term den man damit erhält bezeichnet man als vollständige **disjunktive Normalform**. Durch vereinfachen des Termes entsprechend der Gesetze der Booleschen Algebra erhält man die vereinfachte disjunktive Normalform. Analog erhält man die **konjunktive Normalform**, hier Vertauschen sich jeweils die Rollen von $\vee$ und $\wedge$ sowie von **L** und **O**.

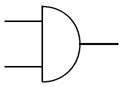
   Ein Beispiel:

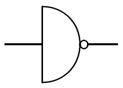
x_1	x_2	x_3	$f(x_1, x_2, x_3)$	
L	**L**	**L**	**L**	$\Rightarrow z_1 = x_1 \wedge x_2 \wedge x_3$
O	**L**	**L**	**O**	
L	**O**	**L**	**O**	
O	**O**	**L**	**L**	$\Rightarrow z_2 = \neg x_1 \wedge \neg x_2 \wedge x_3$
L	**L**	**O**	**O**	
O	**L**	**O**	**O**	
L	**O**	**O**	**O**	
O	**O**	**O**	**L**	$\Rightarrow z_3 = \neg x_1 \wedge \neg x_2 \wedge \neg x_3$

$$f(x_1, x_2, x_3) = z_1 \vee z_2 \vee z_3 = x_1 \wedge x_2 \wedge x_3 \vee \neg x_1 \wedge \neg x_2$$

3. Wie erhält man nun aus einem Booleschen Ausdruck ein Schaltnetz? Die Booleschen Operatoren werden im Schaltnetz durch die **Schaltglieder** (Gatter) repräsentiert. Sie sind die Bauelemente des Schaltnetzes. Im folgenden sind einige Symbole für Schaltglieder angegeben:

   
   Disjunktion

   
   Konjunktion

   
   Negation

   Aus einem Booleschen Ausdruck erhält man ein Schaltnetz, indem man die Schaltglieder nach der Struktur des Booleschen Ausdruckes zusammensetzt. Ein Schaltnetz entspricht somit der graphischen Darstellung eines Booleschen Ausdrucks. Beispiele für typische Schaltwerke sind Halb- und Volladdierer.

5.2 Schaltwerke

Schaltwerke sind Schaltnetze die um ein oder mehrere speichernde Elemente erweitert sind (Verzögerungselemente oder Speicherelemente). Die Ausgabe eines Schaltwerkes hängt von der Vorgeschichte ab, also früheren Eingaben die in Zuständen gespeichert wurden.

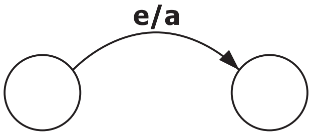
Die Darstellung einer Schaltwerksfunktion mit m Eingängen und n Ausgängen erfolgt durch die Abbildung $(\mathbb{B}^m)^* \rightarrow (\mathbb{B}^n)^*$. Es werden Eingabefolgen auf Ausgabefolgen abgebildet.

Schaltwerke lassen sich als Zustandsautomaten mit Eingabe und Ausgabe auffassen. Ein solcher Zustandsautomat ist durch die Abbildung

$$(\underbrace{\mathbb{B}^m}_{\substack{\text{Eingabe}\\ m\text{-Tupel}}} \times \underbrace{Z}_{\substack{\text{momentaner}\\ \text{Zustand}}}) \rightarrow (\underbrace{\mathbb{B}^n}_{\substack{\text{Ausgabe}\\ n\text{-Tupel}}} \times \underbrace{Z}_{\substack{\text{nächster}\\ \text{Zustand}}})$$

bezeichnet als **Zustandsfunktion**, gegeben. Es ist natürlich auch möglich eine Tabelle anzulegen.

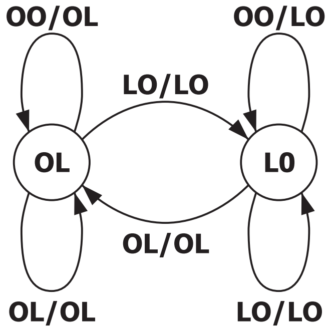
Faßt man die Schaltwerke als gerichtete Graphen auf so entsprechen die Knoten des Graphen den Zuständen und die Übergänge mit Eingabe e und Ausgabe a werden als Pfeile eingezeichnet die zwei Knoten (entsprechend des Zustandsüberganges) miteinander verbinden.



Betrachten wir als Beispiel die folgende Schaltwerksfunktion: $ff : (\mathbb{B}^2)^* \to (\mathbb{B}^2)^*$. Dieses Schaltwerk speichert die Zustände **LO** und **OL**.

Eingabe	Ausgabe	nächster Zustand
OO	gespeicherter Zustand	unverändert (lesen)
LO	**LO**	**LO**
OL	**OL**	**OL**

Über die Eingabe **LL** wird nichts ausgesagt. Die Zustandsübergangsfunktion definiert einen endlichen Automaten. Sowohl Zustände als auch Eingaben und Ausgaben sind als Binärwörter dargestellt.

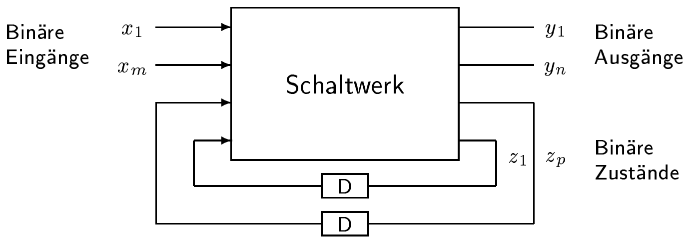


(Mealy - Automat)

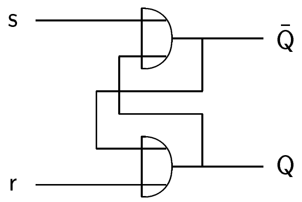
Werden die Zustände Z als binäre p-Tupel $\in \mathbb{B}^p$ codiert, so erhält man die Automatenabbildung

$$\underbrace{\mathbb{B}^m \times \mathbb{B}^p}_{=\mathbb{B}^{m+p}} \to \mathbb{B}^n$$

Man erhält damit ein Schaltnetz (da nur Boolesche Funktionen), allerdings muß man die Zeit als Faktor berücksichtigen.



D steht für Delay, also für Verzögerung. Es ist für spezielle Schaltwerke auch möglich auf die Verzögerung zu verzichten. Zum Beispiel für Flip-Flops. Im folgenden ist der Schaltplan für ein RS-Flip-Flop angegeben:



Es gilt:

$$\bar{\mathsf{Q}} = \neg(\mathsf{Q} \vee \mathsf{s})$$
$$\mathsf{Q} = \neg(\bar{\mathsf{Q}} \vee \mathsf{s})$$

Das RS-Flip-Flop ist stabil, wenn nicht gilt, daß $\mathsf{s} = \mathsf{r} = \mathbf{L}$ (zeigen mit Rekursion und Fixpunkten).

Kapitel 6

Aufbau von Rechnern und maschinennahe Programmierung

6.1 Zahlendarstellung im Rechner

6.1.1 Stellenwertsysteme für Natürliche Zahlen

Natürliche Zahlen z mit $z \in \mathbb{N}$ und $z \neq 0$ lassen sich darstellen durch:

$$z = \sum_{i=0}^{\beta} \alpha_i n^i \quad \text{mit: } \alpha_i, \beta \in \mathbb{N},\ 0 \leq \alpha_i < n \text{ und } i = 1, ..., \beta$$

Je nach Wahl der Basis n spricht man von binärer ($n = 2$), oktaler ($n = 8$), dezimaler ($n = 10$) oder hexadezimaler ($n = 16$) Codierung der Zahlen. Läßt man die führenden Nullen weg so gilt: $n^{\beta+1} > z \geq n^{\beta}$. Die Darstellung der Null, also $z = 0$, erhält man mit: $z = 0 * n^0$.

Beispiele:	dezimal	binär	oktal	hexadezimal
	$(15)_{10}$	$(1111)_2$	$(17)_8$	$(F)_{16}$
	$(16)_{10}$	$(10000)_2$	$(20)_8$	$(10)_{16}$
	$(1000)_{10}$	$(1111101000)_2$	$(1750)_8$	$(3E8)_{16}$

Die Umwandlung binär codierten Zahlen, also mit Basis 2, zu Zahlen die über einer Basis 2^p codiert sind erhält man, indem man je p Ziffern zusammenfaßt und als Ziffer α_i schreibt. Ebenso Einfach ist der umgekehrte Weg, also die Umwandlung von Zahlen die über einer Basis 2^p codiert sind in binär codierte Zahlen, man stellt die α_i einfach binär dar.

6.1.2 Darstellung negativer Ganzzahlen

Für die Darstellung negativer ganzer Zahlen durch Binärwärter fester Länge n existieren insgesamt drei Möglichkeiten.

b_0		b_{n-1}

1. Die einfachste Möglichkeit besteht darin b_0 als Vorzeichen zu verwenden. Dabei steht $b_0 = \mathbf{O}$ für das positive Vorzeichen und $b_0 = \mathbf{L}$ für das negative.

Allerdings existieren dann zwei Darstellungen der Null, nämlich **OOO** ... **OO** für die "positive Null" und **LOO** ... **OO** für die "negative Null".

2. **Einerkomplement**: hier erfolgt die Einteilung der Menge der 2^n möglichen Bitmuster in zwei symmetrische Teile, so daß $-x \cong (2^n - 1) - x$ für alle $x \geq 0$. Auch hier existieren aber zwei Darstellungen der Null, nämlich **OOO** ... **OO** für die "positive Null" und **LLL** ... **LL** für die "negative Null". Einen Vorzeichenwechsel bei einer Zahl erhält man durch bitweises Invertieren (Komplementieren).

3. **Zweierkomplement**: Hier erhält man die negative Darstellung einer positiven Zahl x mit $-x \cong 2^n - x$ für alle $x > 0$. Beim Zweierkomplement ist der darstellbare Zahlenbereich asymmetrisch. Es gibt aber nur eine Darstellung der Null. Es gilt:

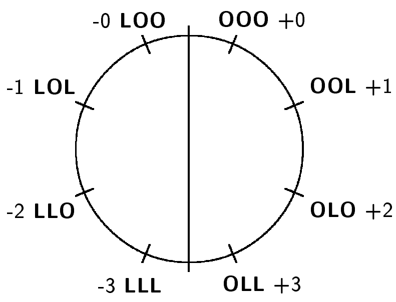
   - Für positive Zahlen x, also $x > 0$, ist die Dualdarstellung von x in b_1 ... b_{n-1} untergebracht und $b_0 = \mathbf{O}$ ist das Vorzeichen.
   - Für negative Zahlen mit Betrag x, ist die Dualdarstellung von $2^n - x$ in b_1 ... b_{n-1} untergebracht und $b_0 = \mathbf{L}$ ist das Vorzeichen.

   Aus der Zweierkomplement Darstellung b_0 ... b_{n-1} einer Zahl $x > 0$ erhält man die Zweierkomplement Darstellung b_0 ... b_{n-1} von $-x$ durch stellenweises Invertieren und Addition von 1.

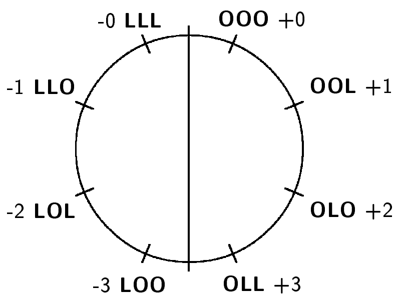
Beispiele:

Dezimal	Vorzeichen Dual	Einer-Komplement	Zweier-Komplement
$+7$	**OLLL**	**OLLL**	**OLLL**
-7	**LLLL**	**LOOO**	**LOOL**
-8	–	–	**LOOO**

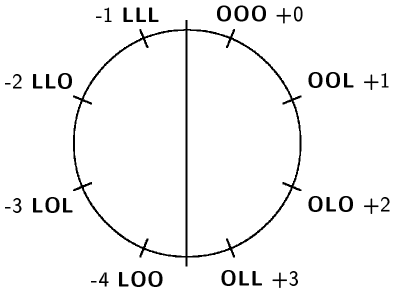
Ein Hilfsmittel zur Veranschaulichung ist eine Kreisdarstellung des Zahlenbereiches:



Vorzeichen-Dual



Einerkomplement



Zweierkomplement

Die bekannten mathematischen Gesetze für die Addition, Subtraktion, Multiplikation und Division gelten aufgrund des beschränkten Zahlenbereiches nur Eingeschränkt (Überlauf). Ein Beispiel hierfür ist $(a-b)+c=(a+c)-b$. So gilt **OLL** $-$ **OLO** $+$ **LLO** $=$ **LLL** aber **OLL** $+$ **LLO** $-$ **OLO** $=$ **OOO**, da bei der Addition von **OLL** und **LLO** ein Überlauf auftritt, denn das Zwischenergebnis **LOLO** ist länger als 3 Bit.

6.1.3 Gleitpunktzahlen

Für die Darstellung Reeller Zahlen werden in der Informatik Gleitpunktzahlen verwendet. Auch bei Gleitpunktzahlen gilt das der Wertebereich der darstellbaren Zahlen eingeschränkt ist. Zudem ist aber auch die Genauigkeit beschränkt. Eine Gleitpunktzahl besteht aus Vorzeichen, **Exponent** und **Mantisse**. Eine Gleitpunktzahl wird durch ein Binärwort wie folgt dargestellt:

$$b = \underbrace{b_0}_{\text{Vorzeichen}} \quad \underbrace{b_1 \;\; \ldots \;\; b_n}_{\text{Exponent}} \quad \underbrace{b_{n+1} \;\; \ldots \;\; b_{n+m}}_{\text{Mantisse}}$$

Die durch das Binärwort b gegebenen Gleitpunktzahl bestimmt sich wie folgt:

$$\mathsf{wert}(b) = (-1)^{\mathsf{wert}(b_0)} * 2^{\mathsf{exp}} * \mathsf{man}$$

Für die Bestimmung der Werte für den Exponenten exp und die Mantisse man verwenden wir die Hilfswerte v und e.

$$e = \sum_{i=1}^{n} 2^{n-i} * \mathsf{wert}(b_i) \qquad 0 \le e < 2^n$$

$$v = \sum_{i=1}^{m} 2^{-i} * \mathsf{wert}(b_{n+i}) \qquad 0 \le v < 1$$

Man setzt nun:

$$\mathsf{exp} = e - 2^{n-1} + 1$$

$$\mathsf{man} = 1 + v$$

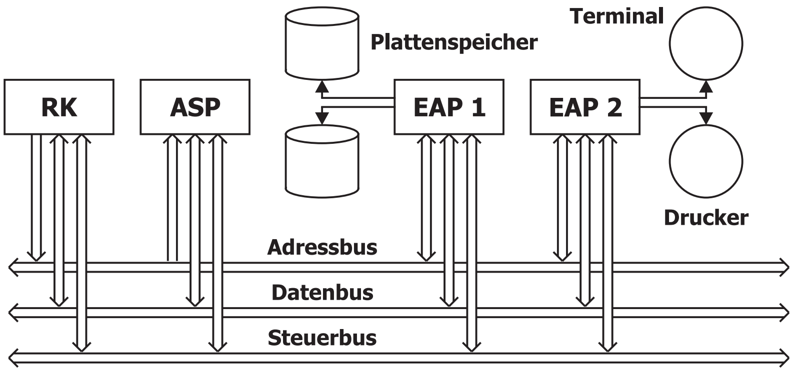
Der Umgang mit Gleitpunktzahlen bei arithmetischen Operationen ist nicht unproblematisch. Es ist nötig das die Gleitpunktzahlen Normalisiert werden. Eine Gleitpunktzahl heißt **normalisiert**, wenn gilt: $1 \le \mathsf{man} < 2$ und $-2^{n-1}+1 \le \mathsf{exp} \le 2^{n-1}$.
Folgende Sonderfälle können auftreten.

1. Darstellung der Null: Die Null wird durch die kleinste darstellbare Zahl repräsentiert, also $v = 0$ und $e = 0$.
2. Unterlauf: hier gilt $e = 0$ aber $v \neq 0$
3. Überlauf: es gilt $e = e_{max} = 2^n - 1$ und $v = 0$, die dargestellte Zahl entspricht der symbolischen Darstellung von Unendlich.
4. Fehlerfall: es gilt $e = e_{max} = 2^n - 1$ und $v \neq 0$ der Wert des Binärwortes b ist undefiniert.

Je nach Fall können die Bits im Kontrollwort gesetzt werden.

6.2 Struktur der MI

In heute vorherrschenden Rechnerarchitekturen findet sich eine grundlegende Strukturierung in Rechenerkern und Arbeitsspeicher. Die Steuerung des Ablaufes der Informationsverarbeitung erfolgt durch Programme. Sowohl Daten als auch Programme stehen im Arbeitsspeicher. Ein Programm ist eine Folge von Maschinenbefehlen. Die Architektur der hypothetischen Maschine MI ist im folgenden kurz skizziert.



Im folgenden werden die nachstehenden Abkürzungen verwendet:

- RK bezeichnet die Rechnerkerne der MI Maschine. Wir gehen im folgenden nur auf die Version mit einem Rechnerkern ein.
- ASP bezeichnet den Arbeitsspeicher
- EAP 1 Bezeichnet den EA (Eingabe/Ausgabe) Prozessor 1 für den Betrieb von zwei Plattenlaufwerken.
- EAP 2 Bezeichnet den EA Prozessor 2,der unter anderem für den Betrieb von Druckern und Terminals zuständig ist.

Ein Bus ist ein Leitungsstrang auf dem nach Bedarf Verbindungen zwischen verschiedenen Einheiten geschaltet werden können.
Wesentlich an der MI-Maschine ist der Rechenkern, er besteht aus 2 Einheiten:

1. **Rechenwerk**: (Arithmetic Logic Unit, ALU), führt die eigentlichen Operationen aus
2. **Steuerwerk**: (auch Leitwerk, mit Taktgeber und Steuersignalen) für die Steuerung des Verarbeitungsvorganges

Das Rechnen ist stark vereinfacht eine sequentielle Schaltung, die stets den folgenden Zyklus ausführt: Maschinenbefehl lesen (und interpretieren) — Befehl ausführen.
Das Rechenwerk der MI-Maschine enthält 16 Register für 32 Bit Werte.

$$\underbrace{\texttt{R0,R1, ... ,R12,R13}}_{\text{frei verwendbar}},\underbrace{\texttt{R14}}_{\texttt{SP}},\underbrace{\texttt{R15}}_{\texttt{PC}}$$

Im Register `PC` (Befehlszähler, Programm Counter) steht die Adresse (bezüglich des Arbeitsspeichers) des nächsten auszuführenden Befehls bzw. Befehlsteils. Der Stack Pointer `SP` enthält den Zeiger für den Keller. Auch das Steuerwerk enthält zwei wichtige Register. Das Instruktionsregister `IR` und das Prozessorstatusregister `PSL` (auch `PSW`), das eine Reihe von wichtigen Abgaben für den Ablauf von Programmen enthält.

Die Speichereinheit besteht aus zwei Registern für die Ansteuerung der Speicherzellen und einem Feld von Speicherzellen:

- das Speicherregister `MBR`
- das Speicheradressenregister `MAR`
- den Speicher `M` mit einem Adreßraum von 2^{32} Speicherzellen zu je 8 Bit.

Wird der Inhaltes einer Speicherzelle s durch die Angabe der Adresse a aus dem Speicher angefordert, so wird zunächst die Adresse a in das Speicheradreßregister geladen. Der Inhalt der Speicherzelle s wird nun in das Speicherregister und dann in den Rechnerkern übertragen.

Wird durch den Rechnerkern eine Speicherzelle über eine Adresse mit einem Wert überschrieben, so wird die Adresse in das Speicheradreßregister geladen der zu speichernde Wert wird in das Speicherregister und dann in die entsprechende Speicherzelle übertragen.

6.2.1 Zahlen in der MI

Für die Darstellung ganzer Zahlen werden in der MI Zweierkomplement Zahlen verwendet. Mittels einer Wortlängenkennung wird die Länge des Operanden und damit auch dessen Interpretation festgelegt. Die folgenden Kennungen sind für ganze Zahlen und natürlich für auch andere Binärdaten verfügbar:

Sorte	Kennung	Wortlänge
B	Byte	8 Bit
H	Halbwort	16 Bit
W	Ganzwort	32 Bit

Die MI unterstützt sowohl Gleitpunktzahlen mit einfacher und doppelter Genauigkeit. Auch hierfür existieren spezielle Kennungen:

Sorte	Kennung	Wortlänge
F	Float	32 Bit
D	Double	64 Bit

6.2.2 Der Aufbau der Maschinenbefehle

Für die Darstellung des Syntax der Maschinenbefehle verwenden wir im folgenden eine etwas modifizierte BNF. Der Ausdruck $\{\ ...\ \}_t^{i,k}$ bedeutet, daß der in den geschweiften Klammern stehende Ausdruck i bis k mal angegeben werden kann, als Trennzeichen zwischen den Wiederholungen dieses Ausdrucks wird t verwendet.

<Maschinenbefehl> ::= <Operationsteil>
|<Operationsteil>␣{<Operandenspezifikation>}$_,^{1,4}$

Die Operandenspezifikation dient der Lokalisierung der Operanden. Der letzte angegebene Operand ist immer der Zieloperand.

<Operandenspezifikation> ::= <absolute Adresse>
|<immediater Operand>
|<Registeradressierung>
|<relative Adressierung>
|<indirekte Adressierung>
|<indizierte relative Adressierung>
|<indizierte indirekte Adressierung>
|<Kelleradressierung>

1. **Absolute Adresse**: hier wird die Speicheradresse (vorzeichenlose ganze Zahl) des Bytes in dem der Operand beginnt angegeben.

   Beispiel: `ADD W 12, 14`

2. **Direkter (immediater) Operand**: Die angegebene Zahlenfolge wird als Zahlenwert interpretiert (nicht als Addresse). Der Syntax ist: `I`␣<Operand>.

   Beispiel: `ADD W I 32, R4`

3. **Registeradressierung** : hier wird die Nummer `x` eines Registers `Rx` angegeben. Bei den Kennungen `H` und `B` gilt es zu beachten, das jeweils rechte Byte bzw. Halbwort des Registers zugegriffen wird.

   Beispiel: `ADD H R0, R4, R5`

4. **Relative Adressierung**: Syntax: {<Ausdruck für ganze Zahl>+}^{0,1}!<Rx> Die ganze Zahl ist dabei eine Adresse r. Der Speicherplatz des Operanden errechnet sich mit r + Inhalt von `Rx`. So könnte zum Beispiel r die Anfangsadresse r eines Feldes sein und `Rx` eine Komponentennummer.

   Beispiel: `MOVE W 46+!R4, R5`

5. **Indirekte Adressierung**: Syntax: !(< relative Adressierung>)|!!<Rx> Der Inhalt der Speicherzelle mit der angegebenen Adresse wird wiederum als Adresse interpretiert, an der dann der Operand steht.

   Beispiel: `ADD H !(46+!R4), R5` oder `SUB H !!R7, R3`

6. **Indizierte relative Adressierung** : Syntax < relative Adressierung> /<Rx>/ Bei dieser Adressierungsart wird zu der relativen Adresse LOP $*$ `Rx` addiert. LOP ist die Anzahl der Bits des Operanden gemäß der Kennung.

   Beispiel: `MOVE W 66+!R8/R7/, R5`

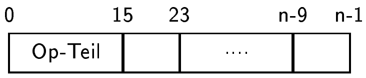
7. **Indizierte indirekte Adressierung** : Syntax < indirekte Adressierung> /<Rx>/ Bei dieser Adressierungsart wird zu der indirekten Adresse LOP $*$ `Rx` addiert.

   Beispiel: `MOVE W !(166+!R2)/R3/, R4`

Am Beispiel des Befehles `MOVE` X a_1`,` a_2 erklären wir nun die einzelnen Adressierungsarten. Dieser Befehl überträgt ein Binärwort b von der Adresse a_1 zu der Adresse a_2. Der Inhalt von a_1 bleibt unverändert, der Inhalt von a_2 wird überschrieben.

`MOVE H 7,13`	Der Inhalt der Speicherzellen mit den Adressen 7 und 8 wird in die Speicherzellen mit den Adressen 13 bzw. 14 geschrieben.
`MOVE B PC,R1`	Das rechte Byte des `PC` (Befehlszähler) wird in das rechte Byte des Registers `R1` geschrieben.
`MOVE H R1,13`	Das rechte Halbwort des Registers `R1` wird in die Speicherzellen mit den Adressen 13 bzw. 14 geschrieben.
`MOVE B 7,5+!R0`	Steht im Register `R0` die Zahl 10, so wird der Inhalt der Speicherzelle mit der Adresse 7 nach der Speicherzelle mit der Adresse $5 + 10 = 15$ transportiert.
`MOVE H 7,5+!R0/R1/`	Steht nun im Register `R1` zusätzlich die Zahl 3, so errechnet sich die Zieladresse des Transportbefehles wie folgt: Zieladresse = 5 + `R0` + `LOP` * `R1`. Da es sich bei den Operanden um ein Halbwort handelt ist `LOP` = 2. Somit gilt: Zieladresse = 5 + 10 + 2 * 3 = 21
`MOVE B !(5+!R0),7`	Hier wird der Inhalt der Speicherzelle mit der Adresse 15 wieder als Adresse interpretiert und somit wird das Byte transportiert, dessen Adresse in der Speicherzelle mit der Adresse 15 steht.
`MOVE B !!R0/R1/,7`	Der Inhalt der Speicherzelle mit der Adresse 10 sei 100, damit wird das Byte dessen Speicheradresse 100 + 1 * 3 ist zu der Speicherzelle mit der Adresse 7 transportiert.

6.2.3 Das Befehlsformat der MI



Der Operationsteil des Befehles enthält die auszuführende Operation, er wird von der Operandenspezifikation gefolgt sofern der Befehl über Operanden enthält. Der Operationsteil des Befehles wird in das Instruktionsregister geladen.

6.3 Maschinennahe Programmierung

Die Maschinennahe Programmierung erfolgt in der Sprache des Rechners. Grundbestandteile einer solchen Sprache sind die **Maschinenbefehle**. Unter Maschinenbefehlen versteht man Bitmuster verschiedener Länge. Ein Programm ist eine Folge von Maschinenbefehlen, die der Reihenfolge nach abgearbeitet werden, eine Ausnahme hiervon bilden Sprungbefehle. Die Daten stehen mit dem Programm im direkt im Arbeitsspeicher oder werden als immediate Operanden (Konstanten) angegeben.

6.3.1 Befehlsvorrat der MI

Im folgenden wird eine Grobe Einteilung der Befehle der MI gegeben:

1. **Transportbefehle**: schiebt den Inhalt von einer Speicherzelle oder einen Register in ein anderes Register oder eine andere Speicherzelle. Der Wert der transportiert wird, wird bei einigen speziellen Befehlen zusätzlich verändert. Im folgenden seien einige wichtige Transportbefehle dargestellt:

`MOVE` X a_1,a_2	Der Inhalt von a_1 wird nach a_2 geschrieben.
`CLEAR` X a_1,a_2	Der Inhalt von a_1 wird auf `00 ... 00` gesetzt. Dieser Befehl ist äquivalent zu `MOVE` X `I 0`,a_2
`MOVEN` X a_1,a_2	(move negated) Der Inhalt von a_1 wird als Zahl gedeutet und negiert in a_2 gespeichert.
`MOVEC` X a_1,a_2	(move complemented) Der Inhalt von a_1 wird als Bitmuster betrachtet und bitweise komplementiert in a_2 gespeichert.
`MOVEA` a_1,a_2	(move address) Aus a_1 wird die direkte Adresse berechnet und nach a_2 geschrieben.

2. **Arithmetische Befehle**: Bei den arithmetischen Befehlen kann man zwischen einer 2 Adreßform und einer 3 Adreßform unterscheiden. Bei der Ausführung der arithmetischen Operationen werden im `PSL` (`PSW`) bestimmte Bits (Condition Codes) gesetzt. Gesetzt werden: `Z` (zero) wenn das Ergebnis Null ist; N (negative) wenn das Ergebnis kleiner Null ist; V (overflow) wenn ein Überlauf auftritt. `Z` und `N` werden auch bei Transportbefehlen gesetzt.

`ADD H` a_1,a_2,a_3	Der Inhalt von a_1 wird mit dem Inhalt von a_2 addiert und nach a_3 geschrieben.
`ADD D` a_1,a_2	Der Inhalt von a_1 wird mit dem Inhalt von a_2 addiert und nach a_2 geschrieben.

3. **Vergleichsbefehle**: Der Vergleichsbefehl `CMP` X a_1,a_2 vergleicht a_2 mit a_1 und setzt die Bits im `PSW`. `Z` wird gesetzt wenn $a_1 = a_2$ und `N` wird gesetzt wenn $a_1 < a_2$

4. **Sprungbefehle**: Bei dem unbedingten Sprung `JUMP` a_1 wird die Adresse a_1 als Adresse des als nächsten auszuführenden Befehls in den Befehlszähler `PC` geladen. Beim Bedingten Sprung J<Bedingungscode> a_1 erfolgt der Sprung nur dann, wenn die durch den Bedingungscode codierte Bedingung erfüllt ist. Ansonsten erfolgt eine sequentielle Adreßfortschaltung. Diese Bedingung bezieht sich auf das Resultat `RES` des zuletzt ausgeführten Maschinenbefehls. Beispiele für Bedingungscodes:

EQ	wenn `RES` $= 0$ (`Z` $=$ **L**) erfolgt Sprung
NE	wenn `RES` $\neq 0$ (`Z` $=$ **O**) erfolgt Sprung
LT	wenn `RES` < 0 (`N` $=$ **L**) erfolgt Sprung
LE	wenn `RES` ≤ 0 (`N` $\vee$ `Z` $=$ **L**) erfolgt Sprung

Ein Beispielprogramm: Im Arbeitsspeicher stehen an der Adresse a stehen unmittelbar nacheinander 4 Bitmuster (m_1, ... ,m_4) von je 32 Bit Länge. In das Wort mit der Adresse a wird das Resultat der Disjunktion der Bitmuster m_1, ... ,m_4 geschrieben (Bitweises Oder).

```
SEG
CLEAR W R0
MOVE  W a,R0
OR    W a+4,R0
OR    W a+8,R0
OR    W a+12,R0
MOVE  W R0,a
HALT
END
```

Datendefinitionen

<Datendefinition> ::= {DD⊔}^{0,1}<Datengruppe>
<Datengruppe> ::= {<Kennung>⊔}^{0,1}{<Datenelement>}$^{1,\infty}$
<Datenelement> ::= <absolute Adresse>
|<Gleitpunktzahl>
|<Ausdruck für ganze Zahl>
|<String>
|(<Datengruppe>) * <vorzeichenlose ganze Zahl>

Beispiele für Datendefinitionen sind:

```
DD W 3            ⇒ 0003
DD W -1           ⇒ FFFF
DD (12,H 17)*2,35 ⇒ 0C 0011 0C 0011 23
```

Um die definierten Daten verwenden zu können benötigt man eine Bezeichnung:

<marke>:{⊔}^{1,n}<Datendefinition>

Dazu ein Beispielprogramm:

```
        SEG
start:  CLEAR W R0
        OR    W a1,R0
        OR    W a2,R0
        OR    W a3,R0
        HALT
a1:     DD    W 3
a2:     DD    (B1)*2,(B15)*2
a3:     DD    W -14
        END
```

6.4 Unterprogrammtechniken

Ein **Unterprogramm** ist eine Folge von Befehlen, die mehrfach in einem Programm (Hauptprogramm) verwendet aber im Quelltext nur einmal eingegeben wird. Unterprogramme dienen der Strukturierung des Codes sowie der Vereinfachung der Schreibarbeit (einfache Form der Wiederverwendung von Code). Man kann zwei Arten von Unterprogrammen unterscheiden:

1. **geschlossene Unterprogramme**: Der Aufruf des Unterprogrammes erfolgt durch Ansprung der Anfangsadresse. Nach der Ausführung des Unterprogrammes erfolgt der Rücksprung zu der gespeicherten Rücksprungadresse.

Das Unterprogramm steht nur einmal im Speicher. Mit Hilfe dieser Art von Unterprogrammen ist Rekursion möglich.

2. **offene Unterprogramme**: Im Quelltext des Hauptprogrammes steht ,dort wo das Unterprogramm benötigt wird, der Bezeichner des Unterprogramms. Während der Übersetzung durch den Compiler wird an dieser Stelle der Code für das Unterprogramm eingefügt (mit den nötigen Änderungen an den Adressen). Man bezeichnet dies auch als Makrotechnik oder Makroersetzung. Die Vorteile dieser Technik liegen darin, daß die Verwaltung und Sicherung des Programmstatus entfällt und sich kürzere Ausführungszeiten ergeben. Allerdings wird das Programm sehr lang zudem ist keine Rekursion möglich.

Beim Springen in ein geschlossenes Unterprogramm sind im allgemeinen folgende Operationen auszuführen:

- Übergeben der Parameter
- Übergeben der Rücksprungadresse
- Sichern des Status

Beim Rücksprung aus dem Unterprogramm müssen dann folgende Operationen ausgeführt werden:

- Übergeben des Ergebnisses
- Wiederherstellen des Status

Eine Möglichkeit für die Übergabe von Parametern und Ergebnissen sowie das Sichern des Status besteht darin sie im **Keller** abzulegen. Der Keller der MI ist ein Teil des Arbeitsspeichers er beginnt an dessen höchster Adresse und wächst nach "unten". Der **Kellerpegel** ist die Speicheradresse des zuletzt auf den Keller gelegten Elementes. Er steht im Register `R14`, dieses Register wird daher auch als `SP` (Stack Pointer) bezeichnet.

0		$2^{31}-1$
Arbeitsbereich Programme, Daten	frei	Keller

Der Ansprung eines Unterprogrammes erfolgt auf der MI mittels `CALL`. Dieser Befehl legt den Inhalt des `PC` auf den Keller und setzt den `PC` anschließend auf die Startadresse des Unterprogrammes, das Register SP wird um 4 dekrementiert. Der Rücksprung erfolgt mittels `RET` dabei wird der Befehlszählerinhalt wieder aus dem Keller in das Register PC zurückgeschrieben und das Register SP wird um 4 inkrementiert.

Ein Beispielprogramm, das allerdings keine Sicherung des Status vornimmt, ist folgende Implementierung der Fakultätsfunktion:

```
MOVE W I H'10000',SP   -- SP auf Anfangswert setzen
MOVE W a,R0            -- a als Parameter uebergeben
CALL fac
MOVE W R2,..           -- Ergebnis aus R2 holen
```

```
fac:    MOVE W I 1,R1
        MOVE W I 1,R2
        JUMP while
loop:   MULT W R1,R2
        ADD  W I 1,R1
while:  CMP  W R1,R0
        JLE  loop
        RET
```

Die MI-Maschine stellt spezielle Befehle für das Retten und Wiederherstellen der Registerinhalte (Status) zur Verfügung. Der Befehl `PUSHR` legt die Inhalte der Register `R0` bis `R14` der Reihenfolge nach nacheinander auf den Keller und dekrementiert das Register `SP` entsprechend. Mittels Maschinenbefehlen ließe sich dieser Befehl wie folgt implementieren:

```
SUB  W I 4,SP    - SP um vier dekrementieren
MOVE W SP,!SP    - SP (=R14) sichern
SUB  W I 4,SP
MOVE W R13,!SP   - usw.  ...
```

Das Wiederherstellen der Register erfolgt analog mit `POPR`. Hier würde sich die Implementierung mittels Maschinenbefehlen wie folgt darstellen:

```
MOVE W !SP,R0   - SP (=R14) sichern
ADD  W I 4,SP   - SP um vier dekrementieren
MOVE W !SP,R1
ADD  W I 4,SP   - usw.  ...
```

6.4.1 Standardschnittstelle für Unterprogramme

Die Standardschnittstelle für Unterprogrammaufrufe wird wie folgt definiert:

1. Das aufrufende Programm reserviert zuerst Platz für die Ergebnisse auf dem Keller, dann legt es die Parameter darauf ab. Nun wird das Unterprogramm mittels `CALL` aufgerufen. Dabei wird der Inhalt des `PC` auf dem Keller gesichert.

2. Unmittelbar zu Beginn des Unterprogramms wird zunächst mittels `PUSHR` der Status gerettet. Der erreichte Kellerpegel wird nun in das Register `R13` geschrieben. `R13` gibt damit an wo lokale Variable des Unterprogramms im Keller plaziert werden können. Weiterhin wird die Basisadresse der Parameterliste errechnet und nach `12` geschrieben.

   Der Zustand Kellers stellt sich nunmehr schematisch wie folgt dar:

	!SP = R13					64+!SP = R12			
frei	R0	···	R13	SP	PC	P_1	···	P_n	Erg

3. Das eigentliche Unterprogramm wird ausgeführt. Und das Ergebnis an den dafür reservierten Speicher geschrieben.

4. Am Ende des Unterprogramms wird zunächst der Kellerpegel aus dem Register `R13` restauriert und dann wird mittels `POPR` der ursprüngliche Status wiederhergestellt. Mittels `RET` erfolgt nun der Rücksprung in das Hauptprogramm.

5. Nach der Ausführung des Unterprogramms werden im Hauptprogramm die auf den Keller gelegten Parameter gelöscht und der SP entsprechend angepaßt. Nun sind nur noch die Ergebnisse des Unterprogramms auf dem Keller.

Ein Beispielquelltext für diese Spezifikation könnte wie folgt aussehen:

```
     SEG
     MOVE  W I H'10000',SP  -- Anfangsadresse für Keller
     ...
     CLEAR W -!SP           -- Platz für Ergebnis reservieren
     MOVE  W parm,-!SP      -- Parameter uebergeben
     CALL  up               -- Unterprogramm aufrufen
     ADD   W I 4,SP         -- Parameter loeschen
     MOVE  W !SP+,erg       -- Ergebnis vom Keller holen
     ...
     HALT

up:  PUSHR                  -- Status retten
     MOVE  W SP,R13         -- Kellerpegel sichern
     MOVEA 64+!R13,R12      -- Basisadresse für Argumente
     ...                    -- Unterprogramm ausfuehren
     MOVE  result,4+!R12    -- Ergebnis sichern
     MOVE  W R13,SP         -- Kellerpegel wiederherstellen
     POPR                   -- Status wiederherstellen
     RET                    -- zurueck zum Hauptprogramm

     END
```

6.5 Übersetzung blockstrukturierter Programme

6.5.1 Speicherverteilung

Bezüglich der Verwaltung läßt sich der Arbeitsspeicher in drei unterschiedlich zu handhabende Bereiche einteilen:

1. **statischer Speicher**: hier werden diejenigen Daten abgelegt deren Speicherplatzbedarf bereits zur Übersetzung feststeht und sich zur Laufzeit nicht verändert. Hierunter fallen Konstanten, globale Variable und der Programmcode.

2. **dynamischer Speicher** (Kellerspeicher): zum Abspeichern von Daten in Programmen mit Blockstruktur sowie Aufruf von Unterprogrammen mit Parametern und lokalen Variablendeklarationen. Hier wird für die lokalen Variablen nur solange Speicherplatz reserviert wie die Abarbeitung des entsprechenden Blockes dauert.

3. **Halde** (Heap): wird bei der Speicherbelegung keine Einteilung in Blöcke berücksichtigt, so wird der Speicher in Form einer sogenannten Halde organisiert. Ähnlich wie beim Keller wird der Speicher hier nach und nach vollgeschrieben, allerdings liegen nicht mehr benötigte Speicherabschnitte im allgemeinen nicht am Ende des genutzten Speicherbereiches, so das eine Speicherbereinigung notwendig ist.

6.5.2 Speicherverwaltung Blockstrukturierter Programme

Der Begriff Block sei im folgenden definiert wie bereits aus JAVA bekannt, allerdings verwenden wir im folgenden eine etwas andere Terminologie. Ein Block ist mit `begin` und `end` geklammert. Ein Block definiert jeweils einen:

- **Bindungsraum** (scope): dies ist der Bereich, der die Lebensdauer der im Block deklarierten Variablen bestimmt. Die Lebensdauer bestimmt, wie lange Speicher für die Variablen reserviert werden muß.
- **Gültigkeitsbereich**: eine Variable kann innerhalb eines Bindungsbereiches verschattet werden. Der Gültigkeitsbereich eines Bezeichners, definiert sich somit als der Bindungsraum der Variablen abzüglich der Bindungsräume mit Verschattung.

Umsetzung von Prozeduraufrufen und Blockschachtelung

1. Jeder Block kann durch folgende Nummern bezüglich seiner Lage im Programm gekennzeichnet werden:
   - **Blockzählernummer BZN**: Anzahl der `begin`'s vom Programmanfang bis zum Blockanfang
   - **Blockschachtelungstiefe BST**: Anzahl der einen Block umfassenden Blöcke addiert mit eins
   - **Blocknummerung**: besitzt ein Block die BZN b und die BST t, so ist die Blocknummerung ein t-Tupel, das aufsteigend geordnet die BZN aller umfassenden Blöcke enthält.

   Dazu ein Beispiel:

Struktur	BZN	BST	Blocknummerung	Lebensdauer			Gültigkeit		
				x_1	y_1	x_2	x_1	y_1	x_2
`begin int x,y`	1	1	1	$\star$	$\star$		$\star$	$\star$	
`  begin int x`	2	2	1.2	$\star$	$\star$	$\star$		$\star$	$\star$
`    begin`	3	3	1.2.3	$\star$	$\star$	$\star$		$\star$	$\star$
`    end`									
`    begin`	4	3	1.2.4	$\star$	$\star$	$\star$		$\star$	$\star$
`    end`									
`  end`									
`  begin`	5	2	1.5	$\star$	$\star$		$\star$	$\star$	
`  end`									
`end`									

2. Zur platzsparenden Verwaltung des Speichers, werden beim betreten eines Blockes zur Laufzeit die dort lokal deklarierten Variablen auf den Keller gelegt und der Kellerpegel wird entsprechend angepaßt. Beim verlassen des Blockes wird der durch die Variablen verwendete Speicher wieder freigegeben, der Kellerpegel also wieder zurückgesetzt. Es gibt somit keine Probleme mit der Gültigkeit von Variablen. Der Zugriff auf Variablen die in einen umschließenden Block deklariert sind (diese stehen logischerweise auch im Keller) erfolgt über die BST, BZN oder die Blocknummerung.
3. Diese Technik kann nun auf den Aufruf von Prozeduren oder Funktionen verallgemeinert werden. Dazu wird der Aufruf der Prozedur als Block gewertet.

Die Parameter der Prozedur oder Funktion werden wie lokal deklarierte Variable behandelt, sie werden also beim Aufruf der Funktion oder Prozedur auf den Keller gelegt. Somit kann im Prinzip die Unterprogrammtechnik aus dem vorhergehenden Abschnitt weiter verwendet werden.

4. Treten in Prozeduren globale Variablen auf, so existieren zwei Möglichkeiten der Zuordnung dieser globalen Variablen zu Deklarationen und damit zu Bindungsräumen:

   - **statische Bindung**: Die Bindung globaler Variablen richtet sich ausschließlich nach der Stelle, an der die Prozedur vereinbart ist. (geschlossene Unterprogramme).
   - **dynamische Bindung**: Die Bindung globaler Variablen bestimmt sich aus der Aufrufstelle (offene Unterprogramme).

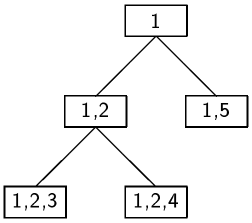
   Dazu ein Beispiel:

Programm	statische Bindung		dynamische Bindung	
`begin` $_1$				
`nat n := 1`	1		1	
`proc add n := n + 1`	1		1	
`begin` $_{1,2}$	1		1	
`nat n := 3`	1	3	1	3
`call proc add`	2	3	1	4
`end` $_{1,2}$	2		1	
`end` $_1$				

5. Für blockorientierte Sprachen wird die statische Bindung verwendet. Man kann hierfür also das Prinzip der Unterprogrammtechnik mit Kellerspeicherung der MI verwenden. Man erhält folgende Speicherorganisation:

   > Für jeden Block und jede Prozedur wird bei Betreten eine Anzahl organisatorischer Speicherzellen vorgesehen. Wird eine Prozedur P aufgerufen wird in der folgenden Weise Speicherplatz belegt und die folgenden Werte werden abgespeichert.
   >
   > 1. Für den statischen Vorgänger, das ist der die Prozedur P direkt umfassende Block, wird die Anfangsadresse gespeichert.
   > 2. Für den dynamischen Vorgänger, das ist der Block aus dem der Aufruf der Prozedur P erfolgte, wird ebenfalls die Anfangsadresse gespeichert.
   > 3. Die BST des dynamischen Vorgängers des Aufrufes von P (nicht bei allen Compilern).
   > 4. Weiterhin werden gespeichert: die Rücksprungadresse, das Resultat (falls erforderlich) Parameter sowie lokale Variablen und Hilfsgrößen.
   >
   > Bei Verlassen des Blockes wird der Keller entsprechend zurückgesetzt.

6. Eine Anschauliche Möglichkeit die Struktur von blockstrukturierten Programmen darzustellen stellen **Blockstrukturbäume** dar. Die Knoten eines solchen Baumes sind die Blöcke des Programmes, gekennzeichnet mit der Blocknummerung. Die direkten Nachfolger eines Knotens, sind alle von dem entsprechenden Block direkt umfaßten Blöcke.



Ein Block A umfaßt einen Block B genau dann, wenn die Blocknummerung von B gleich dem Anfang der Blocknummerung von A ist.

Die Speicherverwaltung des Kellers bei der Abarbeitung eines Programms entspricht dabei folgendem Baumdurchlauf: Gestartet wird an der Wurzel. Nun wird der Ast mit den jeweils am weitesten links liegenden Knoten (mit den kleinsten Blocknummern), die noch nicht abgearbeitet wurden bis zum Blatt durchgearbeitet. Anschließend geht man soweit zurück bis man auf einen Knoten trifft, der noch nicht abgearbeitete Nachfolger besitzt und beginnt dort erneut mit der Abarbeitung.

Es ist möglich in einen solchen Blockstrukturbaum Aufrufpfeile für den Aufruf von Prozeduren einzuzeichnen.

6.5.3 Felder und Referenzen

Felder lassen sich am einfachsten mittels indizierter Adressierung implementieren. Gehen wir zunächst auf eindimensionale Felder ein. In Java wird ein solches Feld mit k Elementen bekanntlich folgendermaßen definiert:

```
int a[] = new int[k]
```

Der zugriff auf das $i+1$ Element erfolgt folgendermaßen:

```
int[i]
```

Bei der Implementierung eines Feldes auf Maschinenebene geht man so vor, daß die einzelnen Werte des Feldes nach Indizes geordnet unmittelbar aufeinanderfolgend im Speicher stehen. Dabei steht das Element mit dem kleinsten Index an der kleinsten Speicheradresse. Die Anfangsadresse eines solchen Feldes sei nun A. Der Zugriff auf ein Elementes läßt sich nun mittels indizierter relativer Adressierung wie folgt auf der MI implementieren: Die Anfangsadresse A des Feldes steht im Register `R0`, der Index des Elementes steht im Feld `R1`. Damit errechnet sich die absolute Adresse des Elementes aus:

```
!R0/R1/
```

Die Länge der Elemente wird von der MI bei indizierter Adressierung automatisch berücksichtigt.

Der Zugriff auf mehrdimensionale Felder gestaltet sich etwas komplizierter. Als Beispiel für ein zweidimensionales Feld dient uns zunächst wieder Java:

```
int a[][] = new int[k1][k2]
int[i1][i2]
```

Wenn man nun davon ausgeht das dieses Feld zeilenweise abgespeichert wird, so erfolgt der Zugriff auf ein Element folgendermaßen: Wie gehabt steht in `R0` die

Anfangsadresse des Feldes. Der Inhalt von R1 berechnet sich folgendermaßen:

$$\texttt{R1} := i_1 * k_1 + i_2$$

Die Absolute Adresse berechnet sich nun genau so wie bei eindimensionalen Feldern.

Referenzen also Zeiger lassen sich relativ einfach über indirekte Adressen implementieren.

Index