Meinungen, Kommentare und Fragen bitte an ctornau@yahoo.de. Diese Zusammenfassung erhebt keinen Anspruch auf Vollständigkeit. Des weiteren plane ich auch, falls ich noch Zeit habe, einige Teile hinzuzufügen. Falls Ihr Fehler finden solltet, bitte ich darum, mir eine kurze Email zu schreiben. Beim lernen viel „Freude" und viel Glück in der Klausur! Viele Grüße Christoph Tornau

WS/SS 99/00

Kapitel 1:

Soll sich jeder selbst ansehen und feststellen, daß es nicht so wichtig ist. Hier in diesem Kapitel gibt es einige Informationen darüber, was Computer überhaupt sind oder was man unter Informatik verstehen kann. Hier gibt es auch eine Skizzierung der historischen Entwicklung der Computer.

Kapitel 2:

Definition: Alphabet

Ein Alphabet Σ ist eine endliche Menge von Symbolen.

Definition: Wort, String

Ein Wort (oder ein String) über Σ ist die Aneinanderreihung von endlichen vielen Symbolen aus Σ.

Definition: Formale Sprache

Eine (formale) Sprache L ist eine Menge von Worten über einem Alphabet. (L wie Language)

„Spezialsprachen": leere Menge: $\emptyset$ nur mit dem leeren Wort: $\{\varepsilon\}$
die Menge aller Worte über einem Alphabet: Σ^*

Darstellung von Binären Zahlen

Darstellung von ganzen positiven Zahlen:

Man dividiere immer ganzzahlig durch 2. Den Rest schreibe man hintereinander auf. Wenn man bei 0 angelangt ist, ist das Verfahren am Ende. Man lese dann die aufgeschriebenen Einsen und Nullen von hinten nach vorne. So erhält man die Binäre Zahl.

Beispiel: 20

	div 2	mod 2
20	10	0 ↑
10	5	0
5	2	1
2	1	0
1	0	1

Alternativ kann man auch immer Potenzen von 2 bilden. Man suche sich die größtmögliche Potenz aus, die noch in die darzustellende Zahl hineinpaßt. Für diese setzen wir eine 1 und ziehen sie von der Zahl ab. Danach gehen wir immer eine Potenz herunter. Paßt diese in die Restzahl, ziehen wir sie ab und setzen wiederum eine 1. Paßt sie nicht, setzen wir eine 0.

Beispiel 20:

$20-2^4=20-16=4$ → 1
$4-2^3=4-8$ ☹ → 0
$4-2^2=4-4=0$ → 1
$0-2^1=0-2$ ☹ → 0
$0-2^0=0-1$ ☹ → 0
↓

Betrag und Vorzeichen:

Man setzte bei positiven Zahlen eine 0 vor die ganze Zahl, bei negativen eine 1. Bei positiven Zahlen kann man die B+V Darstellung ignorieren, da auch in normaler Darstellung die richtige Zahl rauskommt. (Am Anfang wird ja nur eine 0 angehängt, die wertlos ist)

Zweierkomplement:

$$w(b_{n-1}...b_0) = \sum_{i=0}^{n-2} b_i 2^i - b_{n-1} 2^{n-1}$$

Bemerkung: Es gibt eine positive Zahl weniger als es negative Zahlen gibt.

Berechnungstrick: Binäre Zahl → Decimalzahl

1.invertieren
2.normal berechnen
3.addieren von 1
4.Minuszeichen davor

Einerkomplement:

$$w(b_{n-1}...b_0) = \sum_{i=0}^{n-2} b_i 2^i - b_{n-1}(2^{n-1} \boxed{-1})$$

Bemerkung: Es gibt 2 Nullen. Nämlich 0...0 und 1...1

Festpunktdarstellung:

Der Punkt wird fest festgelegt. Wenn man diesen festgelegt hat, kann man nach folgendem Schema die Festpunktzahl in eine Dezimalzahl umformen:

$b_{n-1}*2^{n-1}+b_{n-2}*2^{n-2}...+b_0*2^0 \,.\, b_{-1}\frac{1}{2^1}+b_{-2}\frac{1}{2^2}..b_{-m}\frac{1}{2^m}$

Gleitpunktdarstellung:

Zahl= ± Mantisse* Basis$^{\pm e}$

Normalisiert ist die Zahl, wenn folgendes gilt:

$$\frac{1}{Basis} \leq |\,Mantisse\,| < 1$$

Bei Basis=2 gilt offenbar für jede normalisierte Gleitpunktzahl, daß das höchstwertige Bit der Mantisse immer gleich 1 ist, da man nie unter ½ „rutschen“ darf. Wir benutzen deshalb dieses Bit zur zusätzlichen Speicherung eines Bits in der Mantisse und arbeiten mit einem **Hidden bit**.
Problem: Die Zahl 0 kann nicht mehr dargestellt werden. Lösung: Ist der Exponent 0, ist die dargestellte Zahl 0.

Weiterhin wurden nun das Oktalsystem, das Hexadezimalsystem und BCD, ASCII sowie UNICODE behandelt.

Verschlüsselungssysteme:

Verschlüsseln mit einem Schlüssel. Entschlüsseln mit einem passenden Schlüssel. Bei *asynchronen Verfahren* ist der Verschlüsselungsschlüssel öffentlich. Man darf mit akzeptablen Aufwand keinen passenden Entschlüsselungsschlüssel selbst finden können.

Das RSA Verfahren: (n ist in diesem Verfahren ein öffentlicher Schlüssel)

1. Wähle zwei beliebige Primzahlen p und q. (möglichst großen Zahlen)
2. Bestimme das Produkt n=pq
3. Berechne z = (p-1)(q-1)
4. Bestimme eine Zahl e, so daß e und z teilerfremd sind.
5. Bestimme d so, daß gilt: de mod z = 1 (d und e sollten möglichst nicht gleich sein, da sonst das Verfahren nicht verschlüsselt. Es funktioniert zwar aber es zeigt keine Wirkung.)

Zum Verschlüsseln verwendet man nun: P(M) = M^e (mod n)
Zum Entschlüsseln wieder : S(P(M))= S(C) = C^d (mod n)

Digitalisierung von Audiosignalen:

Digitalisieren:

Signal→Tiefpaß→Abtastung→Quantisierer zur Rundung→Codierer→Fertige Bitfolge

Analogalisieren:

Bitfolge→Decodierer→Tiefpaß zur Interpolation→Signal

Datenkompression:

Es gibt zwei Verfahren. Bei der **Irrelevanzreduktion** gehen **unwichtige Daten** bei dem Kompression und Dekompressionsvorgang **verloren**. Bei der **Redundanzreduktion** erhält man später wieder **das Original**.

Wichtige Erkenntnis über Information, die ich teile:

„Many people believe that if you collect enough information it will do your thinking for you and that the analysis of information leads to ideas. Both are wrong.“

Kapitel 3:

Algorithmus:

Ein Algorithmus ist eine geordnete Menge eindeutiger, ausführbarer Schritte, die eine (terminierende) Schrittfolge definieren.

Die **Abfolge der Schritte** unterliegt gewissen **Regeln**. Es muß nicht immer einen ersten, zweiten, dritten... Schritt geben. Es kann beispielsweise auch parallel gearbeitet werden.

Die Schritte müssen jeder einzeln **ausführbar** sein. Ein Algorithmus wie zum Beispiel
1. Erstelle eine Liste aller positiven ganzen Zahlen von der kleinsten bis zur größten
2. Gebe das erste Element der Liste aus

ist nicht ausführbar, weil der erste Schritt nicht ausführbar ist, obwohl wir wissen, daß das Ergebnis 1 sein muß.

Der einzelne Schritt muß **eindeutig** sein.

Der Algorithmus muß **terminieren** (d.h. keine Endlosschleifen) Ausgenommen sind Algorithmen, die so konzeptziert sind, daß sie endlos laufen müssen. (z.B. Ampelschaltung, Steuerung chemischer Anlagen)

Jeder Algorithmus gibt aufgrund eines Anfangszustandes einen Endzustand aus. Dabei muß der Anfangszustand gültig sein.

Euklidischer Algorithmus (ggT) in Java:

```
int euklidGgT (int a,b)
{
      r=a mod b;
      while (r>0)
      {
            a=b;
            b=r;
            r= a mod b;
      }
      return b;
}
```

*Besprochen wurde das große Kapitel der **Rekursion**. Ich denke in der Klausur wird es wieder wichtig sein, ein rekursives Programm zu entschlüsseln.*

Datentypen

Definition Datentyp:

Ein Datentyp umfaßt eine oder mehrere Wertemengen und die dazugehörigen Operatoren.
Beispiel: Menge der ganzen Zahlen,+,-,*,/

Ausdrücke liefern Datentypen:

Es besteht die Möglichkeit verschiedene **Variablen** oder auch **Literale** (Konstanten) miteinander zu verknüpfen. Dieser **Ausdruck** liefert wiederrum einen **Datentyp** zurück.

Operatoren zur Verknüpfung können häufig überladen sein. (d.h. unterschiedliche Funktion mit unterschiedlichen Datentypen)

Variablen (und auch Konstanten) können implizit oder explizit deklariert werden. Bei **implizierter** Deklarierung legt der Compiler beim ersten Aufruf dieser Variable automatisch den Datentyp fest (Beispiel: Basic, uuhahh...). Bei **explizierter** Deklarierung muß man vorher den Datentyp genau angeben. (Beispiel: Java: z.B. `int a;`)

Kurz: implizit = automatisch; explizit = durch Deklaration

Besprochen wurden unterschiedliche Datentypen in Java. Java sollte man für die Klausur kennen.

Type Casting:

Java führt eine automatische Typkonvertierung durch, wenn die Variablen nicht zueinander passen.
Beispiel:

```
int a=10; String aString = "Type Casting eines Integerwertes: "+a;
```

Manchesmal funktioniert das jedoch nicht so gut, da jeder Ausdruck einzeln konvertiert wird und erst später dieser Ausdruck zusammengebaut wird. Man muß dann **explizites Casting** anwenden:
Beispiel:

`int a=10; int b=6; double aDouble = 1.5+ a/b;` liefert die Ausgabe `aDouble= 2.5`
`int a=10;int b=6;double aDouble=1.5+(double)a/b;` liefert die Ausgabe `aDouble= 3.16`

Abstrakte Datentypen:

Man kann abstrakte Datentypen aus mehreren primitiven Datentypen „zusammenbasteln". Man kann zu diesen Datentypen sogar noch Programmcode hinzufügen. So werden Daten zu einem gesamten Objekt, auf dem auch operiert werden kann. In Java nennt man dies dann Klasse oder Objekt.

Abstrakte Datentypen sind zum Beispiel Mengen oder Listen.

Mengen kann man mit Hilfe des abstrakten Datentyps Liste implementieren.

Es gibt Stacks und Queues. Beim Stack (Kellerspeicher) wird das erste Element als letztes wieder ausgegeben. Bei der Queue (Schlange) das erste als erstes und das letzte als letztes.

Wir haben diese abstrakten Datentypen anhand von Pseudocode besprochen. Wir brauchen den Pseudocode jedoch nicht auswendig zu lernen. Dies kann man aus einer Email von Jens Tölle aus dem Mailverteiler entnehmen: _Wenn_ es eine Aufgabe zum Thema Pseudocode gibt _dann_ bekommt ihr dazu auch die (hoffentlich) bereits bekannte Folie mit den Elementaroperationen.

Jede Procedure (Methode oder auch „Algorithmus") braucht eine spezifische Ein und Ausgabe. Der Datentyp muß genau festgelegt sein.

Es gibt Unteralgorithmen. Unteralgorithmen vereinfachen ein Problem oft erheblich, da sie mehrmals ineinander verschachtelt aufgerufen werden können und somit ein Problem in immer kleinere Teilprobleme zerlegen, die dann immer einfacher gelöst werden. Man nennt dies **Rekursion**.

Beziehungen zwischen Objekten

Es gibt immer wieder Beziehungen zwischen Objekten. Beispielsweise gibt es durch ein Autobahnnetze eine Beziehung zwischen Städten. Diese kann man sogar mit einer Kilometerangabe oder ähnlichem beschriften. Man nennt die Objekte solcher Beziehungen **Knoten** , die Beziehungen selbst **Kanten**.

Das kartesische Produkt: Verknüpfung zweier Mengen. Jedes Element der einen Menge wird mit jedem Element der anderen Menge verknüpft. Folgende Gesetze gelten für das kartesische Produkt x:
L,M,N seinen Mengen ; Teilmengen seien $A \subseteq M$, $B \subseteq M$

1. $(L \cup M) x N = L x N \cup M x N$

2. (L∩M)xN= LxN∩MxN
3. (L \ M)xN =LxN \ MxN
4. AxN∩MxB=AxB

Eine **Relation** ist eine Teilmenge eines Kartesischen Produktes: $\rho \subseteq M \times M$

Grundbegriffe zweistelliger Relationen:

reflexiv	∀ x∈ M gilt	(x,x) ∈ ρ	Alle Elemente in M müssen dies erfüllen.
irreflexiv	∀ x∈ M gilt	(x,x) ∉ ρ	Kein Element in M darf auf dasselbe Element in M zweigen.
symmetrisch	∀ x,y∈ M gilt	(x,y) ∈ ρ ➔(y,x) ∈ ρ	Zu jedem (x,y) muß es ein (y,x) geben.
antisymmetrisch (=identitiv)	∀ x,y∈ M gilt	(x,y) ∈ ρ und (y,x) ∈ ρ ➔ x=y	
asymmetrisch	∀ x,y∈ M gilt	(x,y) ∈ ρ ➔(y,x) ∉ ρ	
transitiv	∀ x,y,z∈ M gilt	(x,y) ∈ ρ und (y,z) ∈ ρ ➔ (x,z) ∈ ρ	
alternativ	∀ x,y∈ M mit x≠y gilt	entweder (x,y) ∈ ρ oder (y,x) ∈ ρ	
Äquivalenzrelation		reflexiv, transitiv und symetrisch	

Sortierverfahren:

Ordnungen:

partielle Ordnung oder Halbordnung:
Eine Relation die reflexiv, transitiv und antisymmetrisch ist. (≤)

strenge Ordnung oder strenge Halbordnung:
Relation, die irreflexiv und transitiv ist (<)

totale oder lineare Ordnung:
Eine Relation, die eine alternative Halbordnung ist.

Sortieren:

Sortieren ist als eine Permutation zu verstehen. Als **Ausgang** haben wir eine Folge von Zahlen in einem Array. Das Sortierproblem ist es nun, diese richtig zu permutieren, daß als **Ergebnis** eine Ordnung entsteht.
Auf Listen kann auch sortiert werden.
Meistens werden „Mehrfachmengen" sortiert. In ihnen kann ein und dasselbe Element mehrfach vorkommen. (Man erhält so statt einer partiellen Ordnung eine strenge Ordnung)

Es gibt **interne** und **externe** Sortierverfahren. Bei **internen** benötigt man möglichst die Möglichkeit zufällig auf den Speicher zugreifen zu können. Bei **externen** sequentiell.

Bubblesort:

```
public static int [] bubbleSort (int[] defaultArray)
{
      for (int i=0;i<defaultArray.length; i++)
        {
          for (int j=0; j<defaultArray.length-1-i; j++)
          // "Hochbubbeln": Wir müssen nur bis i gehen, weil an oberster
          // Stelle jeweils im ersten Durchlauf die höchste Zahl kommt,
          // im zweiten Durchlauf die zweithöchste. Deshalb müssen wir
          // hier nicht mehr vergleichen.
          {
            if (defaultArray [j] > defaultArray [j+1])
            {
              //Tauschen
              int temp = defaultArray [j];
              defaultArray [j] = defaultArray [j+1];
              defaultArray[j+1]=temp;
            }
          }
        }
}
```

Wir haben noch diverse andere Sortierverfahren durchgenommen, u.a. ***Sortieren durch Auswahl*** *und* ***Sortieren durch Einfügen****.*

Graphen

Ein **Graph** G = (V,E) besteht aus einer endlichen **Menge von Knoten V** und einer **Menge von Kanten E**.

Es gibt **gerichtete** und **ungerichtete** Graphen
In einem Graphen dürfen keine zwei Kanten parallel sein (außer in einem gerichteten). Dies sind dann dieselben Graphen.

Induzierte Teilgraphen:
Man kann eine Teilmenge der Knoten nehmen. Läßt man alle übrigen Knoten weg und läßt man auch die Kanten zu diesen Knoten weg erhält man einen induzierten Teilgraphen.

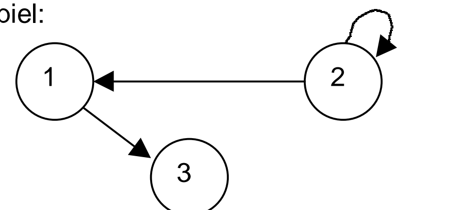
Ein Graph heißt **vollständig**, wenn jeder Knoten mit jedem verbunden ist.

Die **Summe aller Eingangsgerade** (Anzahl der ankommenden Kanten) ist **gleich** der **Summe aller Ausgangsgerade** aller Knoten.(Das ist logisch.)

Adjazenzmatrix:

Graphen kann man nicht so einfach in den Computer eingeben. Man kann sie zwar mit einem Malprogramm malen, doch da hat man wenig gewonnen. Da Graphen durch Knoten und Kanten definiert sind und nicht durch äußere Erscheinungsweise, kann man einen Graphen mit Hilfe einer **Adjazenzmatrix** definieren. Die Spalten und Zeilen repräsentieren die Knoten. Ist eine Kante zwischen den Knoten vorhanden, wird an der „Adresse“ der Tabelle eine 1 geschrieben ansonsten nicht.

Beispiel:



	1	2	3
1	0	0	1
2	1	1	0
3	0	0	0

Graphen kann man färben. Dies geschieht auf ganz bestimmt Weise, indem man sie so färbt, daß zwei benachbarte Knoten nicht die gleiche Farbe haben. Dieses „Färben“ kann man bei natürlichen Problemen benutzen, z.B. beim Herausfinden der besten Ampelschaltung.

Färben kann man mit Hilfe eines **Greedyalgorithmusses**:

1. Wähle eine erste Farbe und einen ersten Knoten. Färbe diesen ersten Knoten.
2. Betrachte nacheinander alle Knoten. Sind sie mit derselben Farbe färbbar (also gibt es keine direkte Nachbarschaft zu andern mit derselben Farbe gefärbten Knoten), färbe sie.
3. Wähle nun einen noch ungefärbten Knoten und färbe ihn mit einer neuen Farbe.
4. Färbe alle möglichen ungefärbten Knoten mit dieser Farbe.
5. Gibt es noch ungefärbte Knoten, springe zu Schritt 3, ansonsten Ende.

Haben wir den Graphen mit der minimalen Anzahl an Farben gefärbt? (Der Greedyalgorithmus ist nicht der beste, es gibt auch nicht den besten Algorithmus hierfür, da daß Verfahren NP-vollständig ist)

Man überprüfe mit einer Clique:
Eine k-Clique ist eine Teilmenge der Knotenmenge in der alle Knoten miteinander verbunden sind. (Vollständiger Graph) Findet man eine solche Menge mit k Knoten und hat man mit k Farben gefärbt, so liegt nahe, daß dies die minimale Anzahl ist, da man eine k-Clique nicht mit weniger als k Farben färben kann.

Wege in Graphen:

Sei G=(V,E) ein Graph.
Die Knotenfolge $(v_0,v_1,...v_k)$ heißt Weg oder Pfad von v_0 nach v_k, wenn $(v_i,v_{i+1})\in E$ für $0 \leq i < k$. k ist die Länge des Weges.

Wenn es einen Weg von v nach v‘ gibt, dann sagen wir, daß **v‘ von v erreichbar** ist. Spezialfall ist, daß ein Knoten als ein Weg der Länge 0 aufgefaßt werden kann. Jeder Knoten ist somit von sich selbst mit der Weglänge 0 aus erreichbar.

Ein ungerichteter Graph, in dem jeder Knoten von jedem Knoten aus erreichbar ist, heißt **zusammenhängend.** (Wie auch sonst, sonst würde ja eine Kante um einen Knoten zu erreichen fehlen und man hätte mehr zwei Graphen als einen)

Besondere Wege:

- Ein Weg heißt **einfach** oder **doppelpunktfrei** wenn kein Knoten auf ihm zweifach oder mehrfach besucht wird.
- Ein Weg heißt **Kreis** oder **Zyklus** genau dann, wenn die Länge > 0 ist (also ist ein Knoten in sich selbst kein Kreis) und **Anfang und Ende des Weges identisch** sind.
- Ein **einfacher** Kreis hat nur einen identischen Anfang und Ende**. Kein Knoten wird doppelt besucht.** (d.h. der Weg muß einfach sein)
- Ein Graph heißt **azyklisch** oder **kreisfrei**, wenn es in ihm keinen Kreis gibt.

Schluß aus dem Königsberger Brückenproblem:

Für jeden beliebigen ungerichteten Graphen gibt es einen Eulerschen Kreis**, wenn an jedem Knoten die gerade Anzahl von Kanten anliegt.** (In einem eulerschen Kreis wird jeder Knoten einmal besucht.)

Ein gerichteter Graph (Digraph) G* =(V,E*) ist die **reflexive, transitive Hülle (kurz: Hülle)** eines Digraphen G = (V,E), genau dann wenn $(u,v)\in E^*$ ist, wenn es einen Weg $u\rightarrow^* v$ in G gibt.
Die Hülle umfaßt neben den direkten auch alle indirekten Beziehungen.

Bestimmung der Erreichbarkeit mit Weglänge ≤ 2:

```
for (int i=0;i<n;i++)
{
  A [i][i]= 1; // Jeder Knoten ist direkt von sich selbst mit einem Weg 0
               // erreichbar.
}
for (int i=0;i<n;i++)
{
   for (int j=0;j<n;j++)
   {
      if (A[i][j] == 1)
       {
          for (int k =0; k<n;k++)
          {
               if (A[j][k]==1)
                {
                   A[i][k]=1;
                }
          }
       }
   }
}
```

Für alle Knoten

Wenn Verbindung zum ersten Knoten

und Verbindung zum zweiten Knoten

Dann Verbindung zwischen den Knoten

Die Adijazenzmatrix wird so im jeden Durchlauf verändert und Knoten, die über eine „Brücke" verbunden waren mit einer neuen Kante verbunden. Wenn sich nach einem Durchlauf nichts mehr geändert hat, ist der Algorithmus dort zu ende. (Was mit n-1 Kanten nicht erreichbar ist, ist gar nicht erreichbar)

Graphen können markiert werden. Jede Kante erhält so eine Markierung, z.B. eine Weglänge oder die Kosten. In der Adjazensmatrix muß man die Weglänge speichern. Hinzukommt, daß man noch eine extra Weglänge braucht (z.B. unendlich) für Kanten, die nicht verbunden sind.

Oft fragt man nach dem kürzesten Weg

Dieses Problem löst **Dijkstra's Algorithmus:**

Initialisiere die Menge S. Zunächst ist nur der Knoten 0 (Ausgangsknoten) in der Menge S.

Initialisiere jetzt das Entfernungsfeld D[n], das die minimale Entfernung von Knoten 0 zu allen Knoten des Graphen angibt. (Betrachte hierbei nur die direkten Verbindungen)

Solange S nicht alle Knoten enthält

- Bestimme die Menge der Knoten, für die die minimale Entfernung vom Knoten 0 noch nicht bekannt ist. (noch nicht in S vorhanden)
- Erweitere S um einen Knoten außerhalb von S, der die minimale aktuelle Entfernung von 0 hat.
- Aktualisiere für alle Knoten außerhalb von S (nicht mehr innerhalb) den Eintrag im Entfernungsfeld.

Ein Digraph G = (V,E) heißt **gerichteter Wald**, wenn der Graph azyklisch (kreisfrei) ist und wenn indeg ≤ 1 ist.

- Wurzel: Der Knoten ohne Vorgänger (indeg = 0)
- Blatt: Die Knoten ohne Nachfolger (outdeg = 0)
- Sohn: Der Nachfolger eines Knotens
- Vater: Der Vorgänger eines Knotens.

Das minimale Spannbaumproblem

Manchmal will man aus einem ungerichteten, zusammenhängenden Graphen einen Spannbaum erzeugen. Ein **Spannbaum** ist ein Baum in dem **Kanten** das Graphen **weggelassen worden sind**, so daß er ein **Baum geworden ist**.

Den minimalen Spannbaum erhält man, wenn man
1. Alle Kanten aus dem Graphen entnimmt und diese speichert (mit Start und Zielknoten und Kantenkosten)
2. Diese nach den Kantenkosten sortiert.
3. Diese Kanten nacheinander wieder in den Graphen einfügt. Dabei werden Kanten nicht eingefügt, wenn ein Kreis dadurch entstehen würde.

Datenbanken

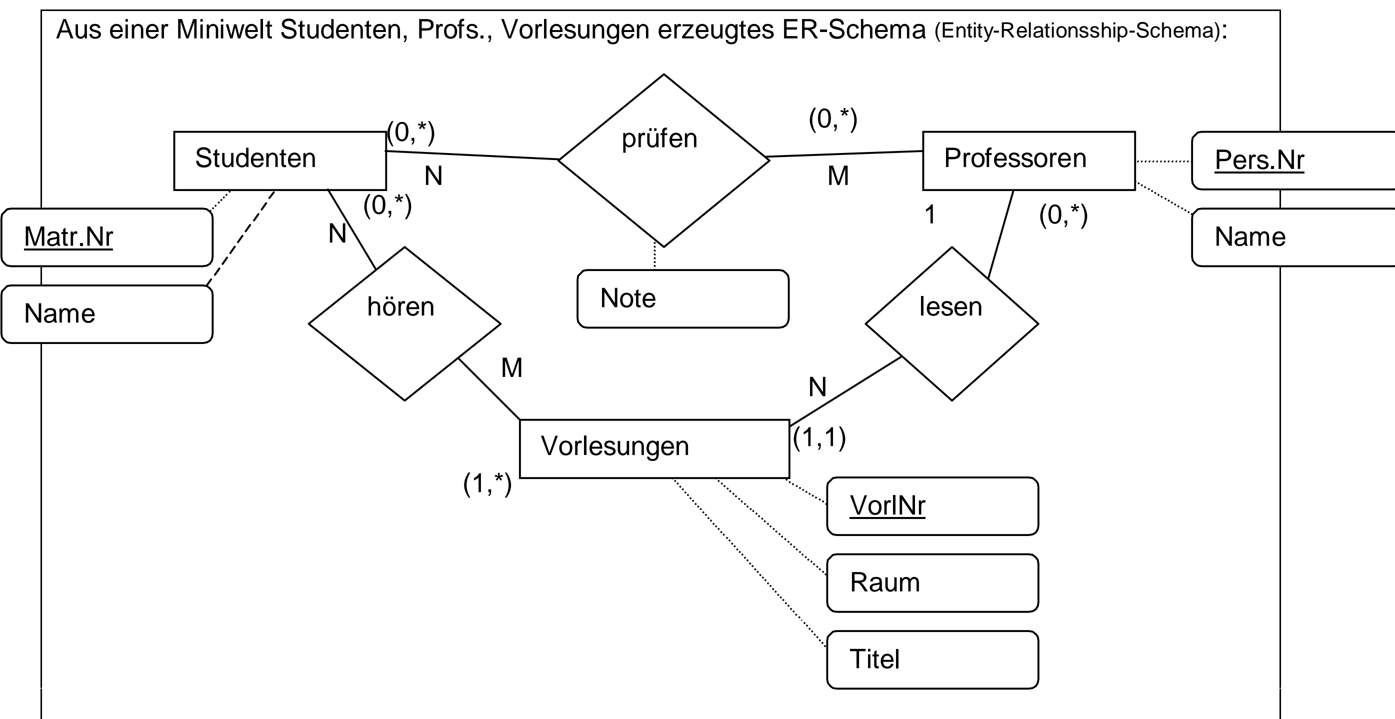
Die Welt kann man aus verschiedenen **Gesichtspunkten** sehen. Für verschiedene Gruppen können verschiedene Dinge interessant sein. Alles kann aber in einer Datenbank gespeichert sein, die ein **Datenbankverwaltungssystem (kurz:Datenbanksystem)** verwaltetet und zugänglich macht.

Man kann einen **Ausschnitt aus der realen Welt nehmen (Miniwelt)** Diesen kann man mit einem **Konzeptuellen Schema** (heute meinst ER-Schema (Entity Relationsship Schema)) **nachmodellieren**. Dann kann man im Computer ein **logisches Modell** erzeugen. Das **Relationale Schema** (marktbeherrschend (SQL,...)) **das Netzwerkschema** oder das **objektoriertierte Schema**. Das **Datenbankschema** legt nur die Struktur fest. Der aktuelle Zustand der Datenbank ist die **Datenbankausprägung**. Sie wird z.B. durch das ER-Modell nicht gezeigt.

Entitys sind wohlunterschiedene Gegenstände in einer Welt, die des weiteren über **Attribute** verfügen. **Ähnliche „Entitiys" werden zu Entitytypen zuammengefaßt.**

Graphische Darstellung:
- Entitytyp:Rechteck
- Beziehungstyp(Relationshiptyp): Raute
- Attribut: Ovale oder Kreise

Aus einer Miniwelt Studenten, Profs., Vorlesungen erzeugtes ER-Schema (Entity-Relationsship-Schema):

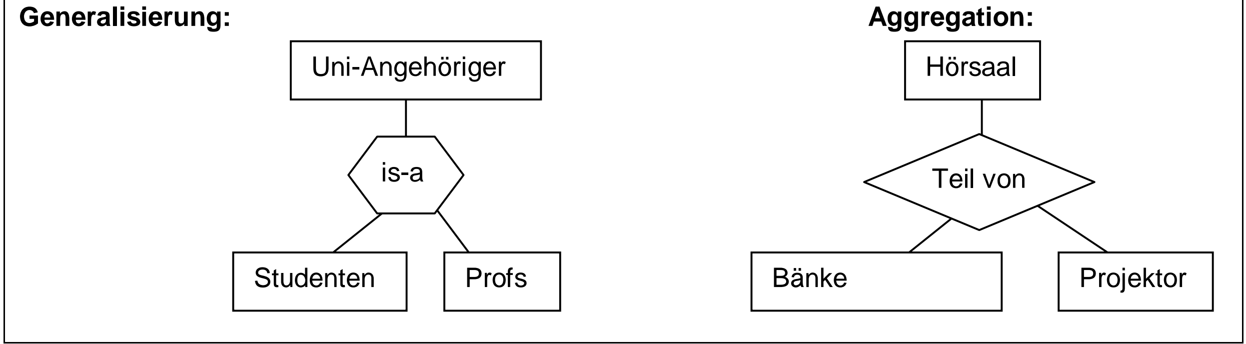


Funktionalität und **Kardinalität**:

- Die **Funktionalität** (1,N,M) bezieht sich auf die Beziehung selbst. Sie gibt an, wieviele Entitys an einer dieser einen Beziehungen hängen dürfen.
- Die **Kardinalität** ((0,*)(1,*)(*,*)bezieht sich auf einen Entitytypen. Sie gibt an wieviele Beziehungen die Entitys dieses Entitytypens eingehen müssen / können.

Es gibt **Schlüssel** (oben unterstrichen). Sie sind Unikate. Deshalb kann man die jeweilige Entity mit ihnen im Entitytyp **eindeutig identifizieren**. Manchmal gibt es auch mehr als einen Schlüssel.

Generalisierung: **Aggregation:**



Das konzeptuelle Modell kann schnell unübersichtlich werden. Deshalb wird es häufig in verschiedene Sichten aufgeteilt. Zu beachten ist jedoch, daß diese Schemata passen müssen, d.h. auch mit der realen Welt **konsolidiert** (integriert) werden müssen. Ein **globales** Schema soll **redundanzfrei**, **widerspruchsfrei**, **ohne Synonyme** (verschiedene Namen für gleiche Dinge), **ohne Homonyme** (ohne gleiche Namen für verschiedene Sachverhalte) und **ohne strukturelle Widersprüche** sein.

Sei E ein Entitytyp mit den Attributen $A_1,...A_n$.
Diesem Entitytyp entspricht im relationalen Modell ein
Relationsschema: schema (R) =($A_1,...A_n$)
ganz besonders in der aktuellen Datenbank sieht eine Entity folgendermaßen aus:
$R^t \subseteq dom(A_1) \times dom(A_2) \times ... \times dom(A_n)$

Tabellarische Darstellung von Relationen:

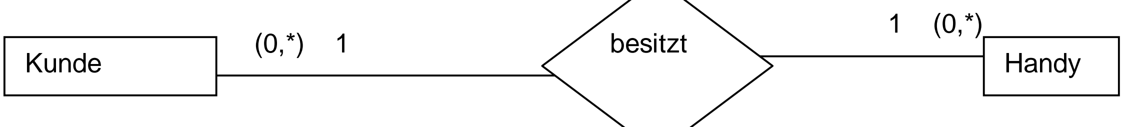
Beispiele:

Kunde		
Kunde.Nr	Name	Vorname
5425	Schmitz	Hugo
5426	Schmitz	Fridolin
4515	Ferdinand	Nadine
5427	Meier	Hans

Eine Entity

Handy		
Karten.Nr	Nummer	Vertrag
4	0170-445645	D1
5	0171-000000	D1
6	0177-456215	Eplus
7	0179-452354	E2

Beispiel: Beziehung „besitzten“



„besitzen“	
Kunde.Nr.	Karten.Nr.
5425	5
5426	4
5427	7

Rationale Algebra:

Die rationale Algebra wird benutzt um Daten aus der Datenbank zu selektieren und zu extrahieren. Operatoren der Rationalen Algebra sind folgende:

- Die Mengenoperationen
  - Vereinigung ($\cup$)
  - Durchschnitt ($\cap$)
  - Differenz (\)
- Selektion (Zeilen)
- Projektion (Spalten)
- Verbund (Join) (kartesisches Produkt und Auswahl)

Selektion:
Bei der Selektion werden die Entitys herausgefiltert, die bei einem Attribut eine bestimmte Eigenschaft erfüllen.

Handy		
Karten.Nr	Nummer	Vertrag
4	0170-445645	D1
5	0171-000000	D1
6	0177-456215	Eplus
7	0179-452354	E2

→

$\sigma_{Vertrag}$=“D1“(Handy)		
Karten.Nr	Nummer	Vertrag
4	0170-445645	D1
5	0171-000000	D1

Projektion:

Es verschwinden die nicht selektierten Spalten. Nur gewählte Spalten werden projiziert. Es gehen keine Entitys verloren.

Kunde		
Kunde.Nr	Name	Vorname
5425	Schmitz	Hugo
5426	Schmitz	Fridolin
4515	Ferdinand	Nadine
5427	Meier	Hans

→

$\Pi_{Kunde.Nr\ Name}$(Kunde)	
Kunde.Nr	Name
5425	Schmitz
5426	Schmitz
4515	Ferdinand
5427	Meier

Der natürliche Verbund (Join)

Zwei Relationen werden zusammengemischt, indem man über sogenannte Joinattribute geht. (Der Join Operator ist dazu noch assoziativ und kommutativ) **Zuerst wird jede Zeile mit jeder Zeile verknüpft. Danach streicht man die Zeilen, in der der Joinopperator auf beiden Seiten nicht gleich ist.**

Handy		
Karten.Nr	Nummer	Vertrag
4	0170-445645	D1
5	0171-000000	D1
6	0177-456215	Eplus
7	0179-452354	E2

Kunde		
Kunde.Nr	Name	Vorname
5425	Schmitz	Hugo
5426	Schmitz	Fridolin
4515	Ferdinand	Nadine
5427	Meier	Hans

„besitzen“	
Kunde.Nr.	Karten.Nr.
5425	5
5426	4
5427	7

Kunde ⋈ besitzen			
Karten.Nr.	Kunde.Nr	Name	Vorname
5	5425	Schmitz	Hugo
4	5426	Schmitz	Fridolin
7	5427	Meier	Hans

Kunde ⋈ besitzen ⋈ Handy					
Karten.Nr.	Kunde.Nr	Name	Vorname	Nummer	Vertrag
5	5425	Schmitz	Hugo	0171-000000	D1
4	5426	Schmitz	Fridolin	0170-445645	D1
7	5427	Meier	Hans	0179-452354	E2

Haben die beiden Relationen keine Attribute gemeinsam, dann wird das kartesische Produkt gebildet. Das heißt jeder mit jedem.

Entwurf von Algorithmen

Phase 1: Problem verstehen
Phase 2: Algorithmus erdenken
Phase 3: Formulierung (u.U. Programmierung) des Algorithmusses und Einsatz
Phase 4: Lösung bewerten, Fehlersuche, fragen, ob dieser Algorithmus nicht auch für andere Probleme geeignet ist.

Die Problemspezifikation soll möglichst vollständig, detailliert und unzweideutig sein. Oftmals ist jedoch dies nicht der Fall und die Spezifikation wird erst im Laufe der Entwicklung entwickelt. (Weil manche Leute von Computern keine Ahnung haben und sich arg waschig ausdrücken)

Vorbedingungen und Nachbedingungen:

Man kann Spezifikationen auch mit Hilfe von Vorbedingungen und Nachbedingungen beschreiben. Wenn zu gewissen Vorbedingungen {P} gewisse Nachbedingungen {Q} erfüllt sind, nach dem der Algorithmus gestartet worden ist. und dies die richtigen sind, schreiben wir: {P}A{Q}

Ein Standardalgorithmus:

Divide-and-Conquer:

1. **Devide** Das Problem wird in kleine Teilprobleme aufgeteilt.
2. **Conquer** Sind die Teilprobleme lösbar, löse sie, ansonsten in weitere Teilprobleme aufteilen.
3. **Combine** Kombiniere die Lösungen miteinander zu einer Gesammtlösung.

Divide-and-Conquer anhand von Mergesort:

1. **Devide** Das zu sortierende Feld, wird in der Mitte aufgeteilt. So entstehen zwei Teilprobleme, die einfacher zu lösen sind.
2. **Conquer** Sind die Teilarrays nun sortierbar, werden sie sortiert. Andernfalls wird wieder in zwei Felder aufgespalten. Die Sortierung eines einelementigen Arrays ist trivial, da ein einelementiges Array immer sortiert vorliegt. Soweit muß man es jedoch nicht kommen lassen, wenn man auch andere Sortierverfahren mit hineinmixt.
3. **Combine** Alle Lösungen werden zu einer kombiniert, indem das jeweils vorderste kleinste Element von zwei Arrays in das Zielarray geschrieben wird und in der Hierarchie nach oben weiter gegeben wird.

Ein Standardalgorithmus:

Gierige Algorithmen (Greedyalgorgithmen)

Der **Greedyalgorithmus** nimmt sich **gierig** immer das, was er momentan kann. Dieser Algorithmus führt zu einer schnellen Lösung des Problems. Es wird jedoch höchstwahrscheinlich bessere Lösungen geben, da die **nur lokal** gearbeitet wird und **nicht global.** Die von uns angesprochenen Greedyalgorithmen sind der **Wechselgeldalgorithmus** (Hier wurde immer die nächstbeste Münze genommen, um den Betrag auszuzahlen. Bei einigen Kombinationen hätte man den Betrag besser auszahlen können, d.h mit weniger Münzen), **Dijkstra's Algorithmus** (zum Finden des kürzesten Weges), **Graphfärben** (Wir haben immer den nächstbesten Knoten zum Färben genommen, wir haben nie eine Färbung zurückgenommen, wenn wir vielleicht gesehen haben, daß woanders diese Färbung günstiger gewesen wäre.)

Laufzeit von Algorithmen

Wir fragen vor allen Dingen nach der Laufzeit eines Algorithmusses. Dabei geht es nicht darum, wie schnell der Algorithmus auf einem bestimmten Rechner in Sekunden läuft. Es geht auch nicht um die Maximalzeit, die dieser Algorithmus läuft, sondern es geht um eine Durchschnittszeit abhängig von der Größe des Problems, die die Laufzeit dieses Algorithmusses allgemein vergleichbar macht. (Ein „größerer" Rechner kann größere Probleme lösen. Dabei verläuft aber die Größe des Problems in den seltesten Fällen linear zu der Stärke des Rechners.)

Groß-O-Notation:

Sei f,g : N→R Funktionen. Dann sagt man:
„**f hat die Ordnung von g**“ bzw. „**f ist (Groß) - O von g**“ (in Zeichen: **f(n)= O (g(n))**), wenn es Konstanten c und n_0 gibt, so daß für alle **n**∈ **N n** $\geq$ $\mathbf{n_0}$ gilt **|f(n)|** ≤ **c|g(n)|**

Die Summenregel

Seien $T_1(n)=O(f(n))$ und $T_2(n)=O(g(n))$ die Laufzeiten zweier Algorithmen. Dann ist $T_1(n)+T_2(n)=O(\max(f(n),g(n)))$

Kapitel 4:

Wichtige Eigenschaften von komplexen Systemen

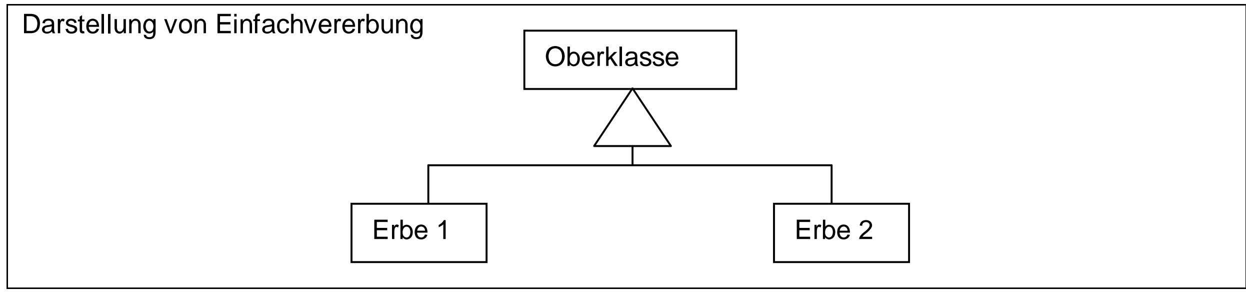
1. Komplexe Systeme sind in einer Hierarchie aufgebaut. Sie setzen sich aus kleineren weniger komlexen Systemen zusammen. (Wir haben das an unseren bekannten logischen Bausteinen später gesehen)
2. Der Aufbau eines komplexen Systems ist fast beliebig und hängt stark von der Sichtweise des Betrachters ab. (Eine Katze ist für eine Oma ein knurrendes Pelzknäul, für einen Tierarzt besteht sie aber aus Organen, die operiert werden können.)
3. Verknüpfungen innerhalb der einzelnen Teilobjekte sind im allgemeinen stärker, als Verknüpfungen zwischen Objekten. Jedes Objekt stellt mehr oder weniger eine funktionierende Einheit da. (z.B. ist eine Fahrradklingel selbst funktionierend. Sie hat wenige Verknüpfungen mit dem Fahrrad. Ohne Fahrrad ist sie aber selbstverständlich schon wieder fast nutzlos.)
4. Komplexe Systeme sind nur aus ein paar weniger komplexen Untersystemen konstruiert. (Beispielsweise gibt es unter Windows in verschiedenen Programmen soviele unterschiedliche Fenster. Sie alle wurden aber nur aus ein Paar Subobjekten kreiert, wie Button, Fenster, Checkbox...)
5. Komplexe Systeme werden von unten nach oben aufgebaut. Hat man keine vernünftige Grundlage zu einem komplexen System und funktioniert das komplexe System nicht, so kann man es vergessen. (Ein Fahrrad kann nicht richtig funktionieren, wenn die Kette nicht funktioniert)

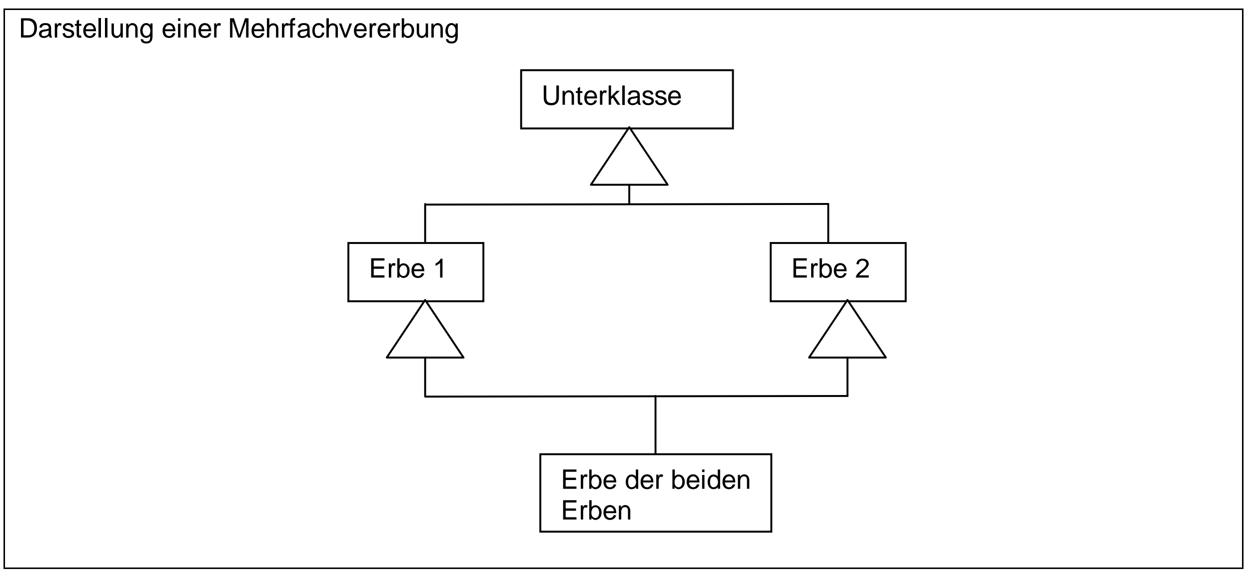
Von der Analyse bis zum Programm

- **Implementation**: Objektorierntierte Programmierung ist das Implementieren eines Problems durch Zusammenfügen von Objekten, welche aus bestimmten Klassen kommen, welche wiederrum von bestimmten Unterklassen abgeleitet sind.
- **Design**: Objektorientiertes Design ist eine Methode zum Auseinanderpflücken und eine logische und pysikalische Notation in statischen und dynamischen Objekten
- **Analyse**: Objektorierntierte Analyse ist eine Methode, die überprüft, welche Ansprüche das Problem an die Klassen und Objekte stellt.

UML: Klassendiagramme

Klassenname
Variablen in der Form Variable: Typ = startwert
Methoden: methode (Parameter:typ)





Zustand, Verhalten und Identität

Jedes Objekt hat zur Laufzeit einen **Zustand** (Variablen), verhält sich **eindeutig** (Methoden) und hat eine **Identität** (vergleichbar mit dem Zeiger auf dieses Objekt)

Einkapslung (so wird dies bei der Programmiersprache Java gemacht)

Attribut	**class**	**subclass**	**package**	**world**	**Erläuterung**
private	x				stärkste Geheimhaltung; Zugriff nur von Objekten derselben Klasse (nicht nur das Objekt, indem diese Variable ist)
protected	x	x	x		Familiengeheimnis: Unterklassen desselben Paketes haben auch Zugriff auch Oberklassen. Sind sie nicht mehr im selben Packet, entfällt dieser Zugriff. Die Klassen sind dann nur noch teilweise zugreifbar.
public	x	x	x	x	Öffentlich
package	x		x		Nur innerhalb desselben Paketet verfügbar

SS 2000
Kapitel 1:

Schaltfunktion

Eine Funktion f: $B^n \rightarrow B^m$ mit B={1,0}; n,m $\in N$; n,m ≥ 1 heißt **Schaltfunktion**.

Boolesche Funktion

Eine Funktion f: $B^n \rightarrow B$ mit B={1,0}; n $\in N$; n ≥ 1 heißt ***(n-stellige)*** **boolesche Funktion**.

Jede Schaltfunktion mit m Ausgängen ist darstellbar als m boolsche Funktionen.

Äquivalenz von Schaltnetzen

Wir bezeichnen zwei Schaltnetze als äquivalent genau, wenn sie für zulässige Eingaben dasselbe Ergebnis liefern.

Bausteinsystem

Ist es möglich, **aus einer Menge von Bausteinen jede beliebige Schaltfunktion** zusammen zu setzen, dann heißt **diese Menge Bausteinsystem**

Gattertypen

	Und Gatter (Konjunktion)	X•Y $X \wedge Y$
	Oder Gatter (Disjunktion)	X+Y $X \vee Y$
	Not Gatter	$\neg X$ $\overline{X}$

Fan-In / Fan-Out : Input/ Outputkanalzahl eines Gatters

Monom

Eine boolsche Funktion, die durch Konjunktion (Und-Verknüpfung) der sogenannten Literale (ggf. negierten Literale) entsteht, bezeichnet man als Monom.

Polynom

Durch Disjunktion (Oder- Verknüpfung) von Monomen entstehen Polynome.

Ein Monom heißt **vollständig** genau dann, wenn alle x_i ($1 \leq i \leq n$) in ihm vorkommen. Es heißt dann *Minterm*.

Ein Polynom heißt **vollständig** genau dann, wenn alle in ihm vorkommenden Monome vollständig sind.

Jede boolsche Funktion ist durch ein vollständiges Polynom darstellbar. Man nennt dieses Polynom dann **disjunktive Normalform (DNF)**

Erzeugen der DNF

Eine **DNF** wird erzeugt, indem die Zeilen der Wertetabelle, in denen eine 1 am Ausgang erzeugt wird genommen werden, die Eingänge für jede Zeile **konjugiert** werden und anschließend alle disjunkten „Zeilen“ **disjunkt** werden. ➔ $(a \cdot b \cdot c) + (a \cdot b \cdot \overline{c}) \cdots$

Eine **KNF** ist die kunjugierte Normalform. Sie wird **erzeugt**, indem die Zeilen der Wertetabelle, in denen der Ausgang 0 ist, genommen werden und **die Eingänge einmal gekippt werden** ($\overline{X}$ (wobei X der Eingang ist)) und dann **disjunkt** werden. Dann werden die einzelnen Zeilen zusammen **konjugiert.**

➔ $(a + b + c) \cdot (a + b + \overline{c}) \cdots$ (Dieses Beispiel steht nicht im Zusammenhang mit obigem Beispiel)

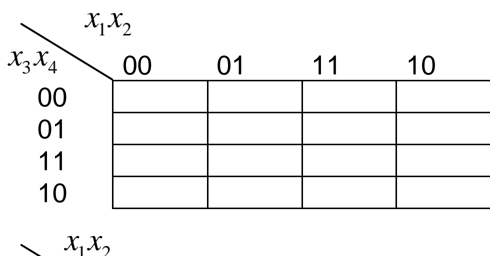
Vereinfachung von Schaltnetzen

Resulutionsregel

Kommen in eine disjunktiven Form zwei Summanden vor, welche sich in genau einer komplementären Variable unterscheiden, so können diese beiden Summanden durch ihren gemeinsamen Teil ersetzt werden.

Beispiel : $\overline{x_1} x_2 x_3 + x_1 x_2 x_3 = (\overline{x_1} + x_1)(x_2 x_3) = 1(x_2 x_3) = x_2 x_3$

Karnaugh -Diagramm



x_3 \ x_1x_2	00	01	11	10
0				
1				

Minimalpolynom

Ein Minimalpolynom einer boolschen Funktion ist ein **billigstes Polynom**, daß diese Funktion realisiert.

Implikant

Sei f: B^n → B eine Boolsche Funktion.

Dann heißt M Implikant von f genau dann, wenn M(x) $\leq$ f(x) für alle $x \in B^n$.

Primimplikant

Ein Implikant M heißt Primimplikant, wenn keine echte Verkürzung von M noch Implikant von f ist.

Zu lernen sind in diesem Kapitel noch

- Quine/McCkuskey
- R-S-Flipflop
- Getaktetes Flipflop
- J-K-Flipflop
- Schieberegister **(Master-Slave-Flipflop** (siehe Lösungen der Übungszettel)**)**
- Halbaddierer und Volladdierer
- Carry-Rippel-Addition
- Serielle-Addition
- Von-Neumann Addition
- Carry-Save-Addition
- „Adder Tree“

(bei diesem Themenbereichen gibt es viele kleine Schaltnetze, die man alle wissen sollte, für die ich aber keine Zeit habe, sie hier aufzuschreiben, da ich mich in den nächsten Tagen nicht mit dem positionieren von Grafiken beschäftigen möchte und mich mit Word 97 herumschlagen möchte.)

Wenn ein Übertrag zustande kommt, dann kann dieser beim Zweierkomplement ignoriert werden (das macht das Zweierkomplement sehr schnell). Beim Einerkomplement muß er hinten wieder drangeschoben werden.

Feststellung für den kompletten Überlauf, wenn wir das Ergebnis der Addition nicht mehr verwenden können. (Oben kann ein Überlauf stattfinden. Der ist dann allerdings egal, da wir uns immer noch im darstellbaren Zahlenbereich bewegen.)

Das **doppelte Vorzeichen** wird benutzt, um bei der Addition festzustellen, ob ein Überlauf zustande gekommen ist:

Das Doppelte Vorzeichen wird folgendermaßen interpetiert:

Komplementärdarstellung: (Einer und Zweierkomplement)

Wir verdoppeln das Vorzeichenbit. Wenn nach der Addition beide Bits, das Original und das Doppel **unterschiedlich** sind, ist ein **Überlauf** aufgetreten.

B+V (Betrag und Vorzeichen):

Wir fügen eine 0 vor dem Vorzeichenbit ein. Wenn diese 0 ungleich 0 nach der Addition ist, dann ist das ein Überlauf. (ist ja trivial ☺)

Kapitel 2:

<u>Interne Struktur eines Prozessors / Bestandteile:</u>

Akkumulator: (Accumulator)

Schnellspeicher mit kurzen Zugriffszeiten nimmt Operanden auf, die als nächstes von der Arithmetrischen-Logischen Einheit verarbeitet werden oder nimmt die Ergebnisse der Berechnung auf.

Arithmetisch-Logische Einheit (ALU): (Arithmetic Logic Unit, ALU)

Der eigentliche „Rechner“. Führt Befehle auf Operatoren, die im Akkumulator liegen aus.

Kontrolleinheit / Steuerwerk: (Control Unit)

Steuert die Vorgänge im Prozessor und sorgt somit für die Abarbeitung der Befehle. Das Steuerwerk unterliegt dem Systemtakt, sagt aber selbst, wann aus dem Speicher gelesen und geschrieben werden soll, wann die ALU rechnen soll usw.

Programmzähler: (Program Counter, PC)

Hier steht die Speicheradresse des als nächsten abzuarbeitenden Befehls.

Befehlsregister: (Current Instruction Register, CIR)

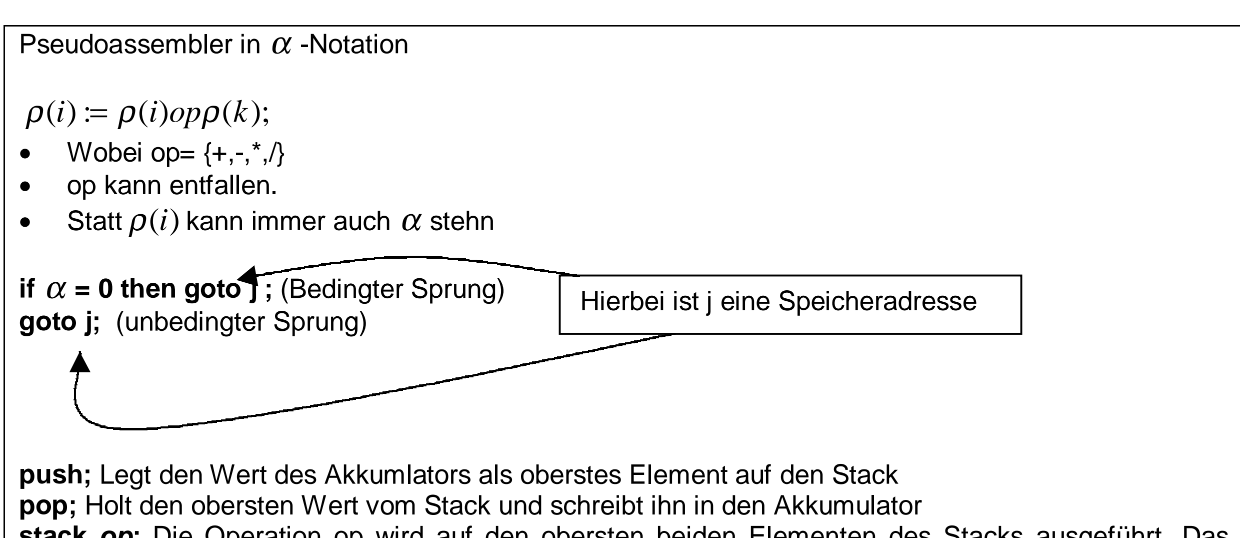
Hier steht der momentane Befehl der gerade abgearbeitet wird. Dieser wird von anderen Registern aus dem Speicher geholt und über den internen Datenbus in dieses Register geschoben.

Address-Register: (Memory Adress Register, MAR)

Aktuelle Adresse die aus dem Speicher geholt wird oder die in den Speicher geschrieben wird.

Datenregister: (Memory Date Register, MDR)

Hier stehen die Daten an der Position auf die das Adressregister zeigt. Dies können sowohl Daten sein, die gerade geholt worden sind, sowie auch Daten, die geschrieben worden sind.



Pseudoassembler in α -Notation

$\rho(i) := \rho(i) op \rho(k);$

- Wobei op= {+,-,*,/}
- op kann entfallen.
- Statt $\rho(i)$ kann immer auch α stehn

if α = 0 then goto j; (Bedingter Sprung)
goto j; (unbedingter Sprung)

push; Legt den Wert des Akkumlators als oberstes Element auf den Stack
pop; Holt den obersten Wert vom Stack und schreibt ihn in den Akkumulator
stack *op*; Die Operation op wird auf den obersten beiden Elementen des Stacks ausgeführt. Das Ergebnis landet hierbei im Akkumulator und wird auch wieder auf den Stack gelegt. (Aus 2 macht 1)

Wenn **drei Adressen** in jedem Befehl vorkommen, so ist der Pseudoassembler eine **3-Adressmaschine**.
Wenn **zwei Adressen** vorkommen, so ist es eine **2-Adressmaschine**.
Wenn nur **eine Adresse** da ist, so ist es eine **Einadressmaschine**.
Wenn gar **keine Adresse** da ist, bzw. **nur Adressen** verwendet werden, **um den Speicher zu laden**, so ist es eine **0-Adressmaschine**.

Es muß hierbei mit dem **Akkumluator** gearbeitet werden.

Bei der 0-Adressmaschine gibt es einen **Stack**, der benutzt werden muß.

Basisadresse und Displacement:

Adressen können mit Hilfe eines **Displacements** relativ zu einer **Basisadresse** angegeben werden. Die Adresse wird somit kürzer. Das Programm kann man im Speicher verschieben ohne zusätzliche Aufwendungen am Programmcode (in der Maschienensprache selbst) zu haben.

Indexregister:

$\rho(\rho(\gamma))$: Man kann eine Speichervariable auch über ein Indexregister ansteuern, der erst zur Programmausführung erzeugt wird. So lassen sich z.B. bei Arrays viele Programmzeilen sparen.
$\rho(\rho(i))$: Natürlich kann der Index auch aus einer Speicherzelle geholt werden.
Der Nachteil bei diesem Verfahren ist es, daß es länger dauert, bis der Befehl abgearbeitet ist, da der Index noch aus dem Speicher geholt werden muß.

Unterprogramme:

Einstufig, nur einmal aufrufbar: Rette Rücksprungandresse in einer Variablen oder in einem Register.
Mehrstufig, nur einmal aufrufbar: Rette Rücksprungadresse in der ersten Zeile des Unterprogramms. Diese muß leer sein, damit dort die Rücksprungadresse stehen kann.
Mehrstufig, erneut aufrufbar (Rekursion): Benutze einen Stack

Makros:

Ein Makro kann dazu eingesetzt werden, um meherere Befehle zusammenzufassen und als eine Einheit aufzurufen.

Der von uns verwendete Syntax:

MACRO;
Name &var1, &var2
...(mit &var1 oder &var2 können die Variablen eingesetzt werden)
MEND;

Aufruf: Name a,b;

Verschiedene Rechnertypen:

SISD: Single-Instruction Single-Data Ein Prozessor arbeitet einen Befehlsstrom auf einem Datenstrom ab.
SIMD: Single-Instruction Multiple-Data Viele Prozessoren arbeiten einen Befehlsstrom auf unterschiedlichen Datenströmen ab. Benutzt werden kann dies zum Beispiel für eine Matrixaddition. Man benötigt n^2 Prozessoren (für jede Zahl der Matrix einen), dann kann man eine nxn Matrix mit einem, auf allen Prozessoren laufenden, Additionsbefehl addieren. Die unterschiedlichen Datenströme beinhalten dann jeweils zwei Zahlen und als Ausgabe eine Zahl
MISD: Multiple-Instruction Single-Data Viele Prozessoren arbeiten mit vielen Instruktionen auf denselben Daten. Dies ist nahezu Unsinn und wird deshalb nicht angewandt. MISD-Computer haben keinerlei praktische Relevanz.
MIMD: Multiple-Instruction Multiple-Data Viele Prozessoren arbeiten auf vielen Datenströmen und unterschiedliche Befehle ab → Viele Rechner in einem

MIPS = Million Instructions per Second
FLOPS = Floating Point Operations per Second
} Als Maßeinheit für Prozessorleistung sehr fragwürdig

RISC: Reduced Instruction Set Computer – Wenige Befehle, historisch bedingt aber in einem einzigen Chip realisierbar und deshalb schnell.
CISC: Complex Instruction Set Computer – Viele Befehle. Das ging sogar fast dahin, daß für jeden Befehl in einer Hochsprache ein Assemblerbefehl gefordert wurde. (HLLCA)

Wie kommt die Laufzeit eines Programmes auf verschiedenen Prozessoren zusammen?:

$$\frac{time}{program} = \frac{instructions}{program} \bullet \frac{cycles}{instruction} \bullet \frac{time}{cycle}$$

→ CISC und RISC tun sich nicht viel, wenn eine Variable kleiner wird, die andere aber dafür größer.

Pipelining:

Ein Befehl wird zerpflückt. So kann man, wenn der erste Teil des Befehls abgearbeitet ist, schon einen zweiten loschicken, bevor der erste Befehl ganz fertig ist.

Superpipelining:

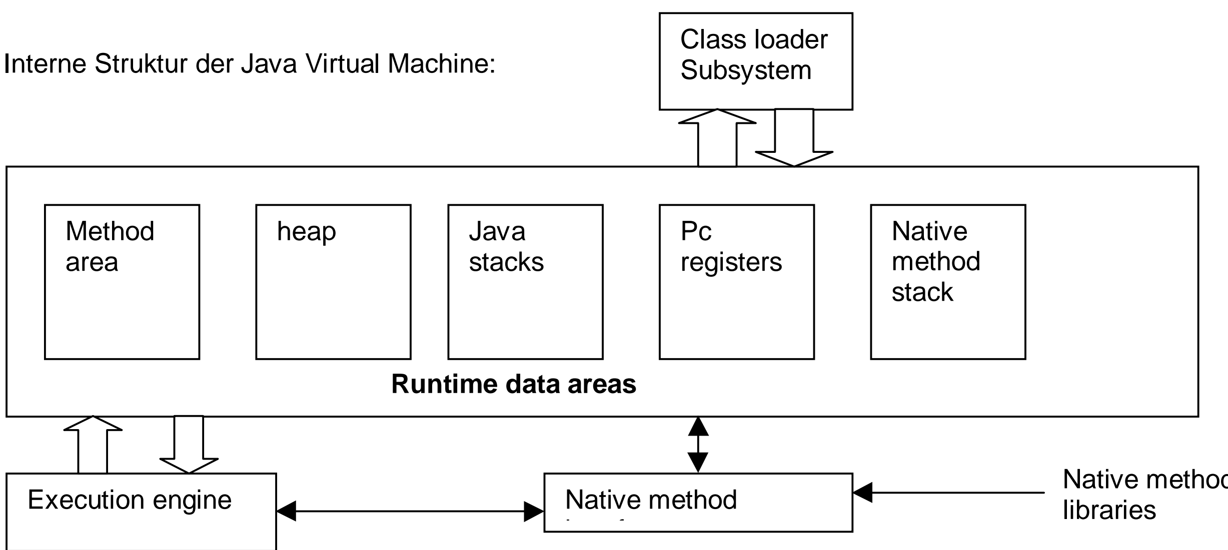
Der Befehl wird noch weiter zerpflückt und zerstückelt, indem die natürlichen funktionalen Einheiten noch weiter aufgepalten werden. Noch mehr Befehle können gleichzeitig durch den Prozessor.

Superskalartechnik:

Auch die einzelnen Phasen können gleichzeitig durch den Prozessor. Eigentlich sind es jetzt mehrere Prozessoren in einem Gehäuse. Dies kann zur Matrixmultiplikation mit einem Skalar gut verwendet werden.

Wenn man solche Verfahren anwendet, wird es jedoch schwierig den nächsten Befehl vorherzusagen. Wir wissen eventuell noch gar nicht, was der nächste Befehl sein soll. Deshalb ist ein sehr kompliziertes Kapitel die „**Branch Prediction**“.

Interne Struktur der Java Virtual Machine:



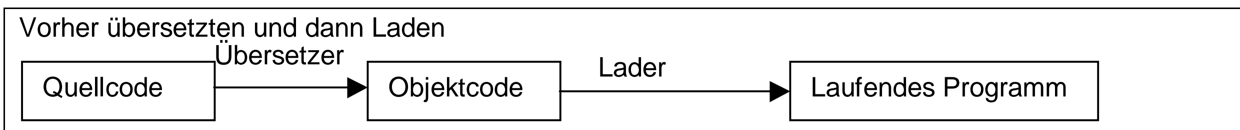
Kapitel 3:

Lader und Binder

Übersetzen und Laden:

Das Programm wird zur Laufzeit übersetzt. Der Lader reduziert sich auf den Starter. Der Nachteil dieses Systems ist es, daß relativ viel Speicherplatz im Speicher für den Übersetzter und auch für den Quellcode verloren geht. Viel Rechenzeit zum Übersetzten. Externe Referenzen kaum möglich.

Vorher übersetzten und dann Laden



Absoluter Lader

Schon beim Übersetzten steht die Startadresse fest. Diese kann beim Ladevorgang nicht mehr geändert werden. Wenn diese Adresse schon von einem anderen Programm belegt ist, ist das schlecht. Umschiffung allerdings mit Basisregistern.

Verschiebende Lader

Es gibt eine Verschiebungstabelle, die beim Übersetzten erzeugt wird, in der steht, wo die Variablen im Programm stehen. Auf diese wird dann ein Verschiebungswert aufaddiert.

Bindende Lader

Mehrere verschiedene Übersetzer arbeiten auf verschiedenen Quellcodes und übersetzen Teile des Programmes in Objektcode. Diese Teile werden dann gebunden und dann geladen. Dabei kann Binden zur Laufzeit geschehen oder auch schon vorher geschehen sein. (Java bindet zur Laufzeit, Turbo Pascal nicht)

Dynamisches Binden und Laden

Binden und Laden während der Laufzeit und nur dann wenn der Code auch wirklich benötigt wird.

Übersetzer / Compiler

Ein Übersetzter erzeugt zu einem Programm in Quellsprache ein äquivalentes Programm in der Zielsprache.

Definition: Kontextfreie Grammatik

Eine kontextfreie Grammatik (CFG) ist ein Quadrupel G=(T,N,P,S) mit
T: Endliche Menge von **Terminalsymbolen**
N: Endliche Menge von **Nicht-Terminalen** $N \cap T = \emptyset$
P: Endliche Menge von **Produktionen**
S: Eine spezielle Variable, als **Startsymbol**

Eine Produktion läßt sich schreiben als $A \rightarrow a$
a ist dabei ein String über Terminals und Nonterminals

Backus-Naur-Form (BNF)

Ersetztungsregeln: <Bezeichner> ::= ersetzterBezA | ersetzterBezB | ersetzterBezC | ε (Beispiel)

In **Syntaxdiagrammen** sind Elipsen bzw. Kreise **Terminale** und Rechtecke **Nicht-Terminale**.

Teile des Compilers:

Lexikalische Analyse:

Aufgaben der Lexikalischen Analyse:

Erkennen der Grundsymbole, ausblenden bedeutungsloser Zeichen und Codieren der Grundsymbole.

Definition: Reguläre Grammatiken

Definition: Rechts-lineare Grammatik

Eine kontextfreie Grammatik heißt „rechts-linear" genau dann, wenn alle Produktionen folgender Form sind:
$A \rightarrow w\,B$ oder $A \rightarrow w$, wobei $w = \varepsilon$ sein darf

Eine reguläre Grammatik ist eine Grammatik, die entweder links oder rechts-linear ist. Aus regulären Grammatiken lassen sich Automaten erzeugen.

Nun kommt ein wichtiges Themengebiet: *Endliche Automaten*

Definition: Endlicher Automat

Ein endlicher Automat ist ein Tupel $A = (Q, \sum, \delta, q_0, F)$ mit
Q: (endliche) Zustandsmenge
$\sum$: (endlicher) Zeichensatz (Eingabealphabet)
δ : Übergangsfunktion
q_0: Anfangszustand ($q_0 \in Q$)
F: Menge der Endzustände ($F \subset Q$)

Deterministischer Endlicher Automat:

In jedem Zustand gibt es für jedes eingegebene Zeichen höchstens einen Folgezustand. Es kann also nicht passieren, daß wir uns aussuchen dürfen, wo wir demnächst gerne lang wollen.

Nicht- Deterministischer Endlicher Automat:

Es gibt Zustände für die es Zeichen gibt, bei denen zu mehreren Folgezuständen verzweigt werden kann. **Wir möchten aber solcher Automaten nicht verwenden. In Übungen und in der Klausur wird ein solcher Automat falsch sein.**

Vom Syntaxdiagramm zum Automaten:

Alle Syntaxdiagramme sind in Automaten überführbar.

Algorithmus:

1. Alle Knoten des Syntaxdiagramms werden mit Nummern markiert (durchzählen) Der Ausgang bekommt die 0.
2. Der Anfangszustand wird bestimmt. q_1 in den Automaten eintragen. m sind alle Marken die erreicht werden können
3. Es wird ein Zustand aus dem schon existierenden Automaten gewählt. Dann wird aus der Markenmenge m dieses Zustandes die Inschrift a einer Marke geholt.
4. k ist die Menge der Marken aus m, die die Inschrift a besitzen. Wir bilden die Menge m' der Marken, die von irgendeinem Knoten aus k direkt erreichbar sind.
5. Gibt es den Zustand q' mit der Markenmenge m' noch nicht im Automaten, so kommt er zum Automaten dazu.
6. Ist $0 \in$ m' von q', so ist der Zustand ein Endzustand.
7. Der gewählte Zustand q und der neue Zustand wird durch eine Marke, die die Inschrift a trägt verbunden. Dies wird auch gemacht, wenn wir q' nicht neu erzeugt haben.
8. Sprung zu 3, wenn wir noch nicht alle Paare von konstruierten Zuständen betrachet haben.

Und nun ein Beispiel (Das Beispiel stammt von den Folien – Bitte dort nachschauen)

Hier die Tabelle nach dem Verfahren:

Zuerst selber machen!

q	m	a	k	m‘	q‘	Aktion
q_1	{1}	Ziffer	{1}	{1,2,4}	q_2	q_1—Ziffer→q_2
q_2	{1,2,4}	Ziffer	{1}	{1,2,4}	q_2	q_2—Ziffer→q_2
q_2	{1,2,4}	„.“	{2}	{3}	q_3	q_2—„.“→q_3
q_2	{1,2,4}	E	{4}	{5,6,7}	q_4	q_2—E→q_4
q_3	{3}	Ziffer	{3}	{3,4,0}	q_5	q_3--Ziffer→q_5(Endzustand)
q_4	{5,6,7}	+	{5}	{7}	q_6	q_4--+→q_6
q_4	{5,6,7}	-	{6}	{7}	q_6	q_4--„-“→q_6
q_4	{5,6,7}	Ziffer	{7}	{7,0}	q_7	q_4--Ziffer→q_7 (Endzustand)
q_5	{3,4,0}	Ziffer	{3}	{3,4,0}	q_5	q_5--Ziffer→q_5 (Endzustand s.o.)
q_5	{3,4,0}	E	{4}	{5,6,7}	q_4	q_5--E→q_4
q_6	{7}	Ziffer	{7}	{7,0}	q_7	q_6--Ziffer→q_7 (Endzustand s.o.)
q_7	{7,0}	Ziffer	{7}	{7,0}	q_7	q_7--Ziffer→q_7 (Endzustand s.o.)

Regel des Längsten Musters:

Kommt in einen Automaten ein String (Muster) hinein, daß nicht ganz erkannt werden kann, so wird es bis zu einem Zustand erkannt, wo es nicht mehr weitergeht. Das ist dann das längste Muster.

Syntaktische Analyse:

Aufgaben:

Ermittelt die Struktur des Programmes und gibt eine abstrakte Datenstruktur wieder zurück. Die lexikalische Analyse hat den Code in einzelne Wörter unterteilt. Die Syntaktische Analyse baut diese Wörter zusammen und gibt sozusagen einen Satzbau zurück.

Definition: Kellerautomat

In Kellerautomaten gibt es ε-Moves (Stackmodifikation und Zustandswechsel ohne Verbrauch von Input).

Ein Kellerautomat ist ein Tupel $A=(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
Q: (endliche) Zustandsmenge
Σ: (endlicher) Zeichensatz (Eingabe-Alphabet)
Γ: (endlicher) Zeichensatz (Stack-Alphabet)
q_0: Anfangszustand (q_0=Q)
Z_0: ein spezielles Stacksybol $Z_0 \in \Gamma$ (Startsymbol)
F: Menge der Endzustände ($Q \supset F$)
δ: Eine Abbildung von $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ zu einer endlichen Teilmenge $Q \times \Gamma^*$

Sätze:

Zu jeder regulären Grammatik gibt es einen endlichen Automaten (und umgekehrt)

Zu jedem nicht-deterministischen endlichen Automaten gibt es einen deterministischen Automaten, der die gleiche Sprache akzeptiert.

Zu jeder kontextfreien Grammatik gibt es einen Kellerautomaten, der die Sprache der Grammatik akzeptiert.

Deterministische Kellerautomaten akzeptieren nur Teilmengen der kontextfreien Sprachen.

Arithmetrische Ausdrücke:

Durch die kontextfreie Grammatik
N={E}
T={+,-,(,),id}
P={E→E+E,E→E-E,E→(E),E→id}
lassen sich Bäume erzeugen, die jeden arithmetrischen Ausdruck zerlegen.

Semantische Analyse:

Ein Syntaktisch korrektes Programm muß noch lange nicht ein gültiges Programm sein. Es folgt noch eine Semantische Analyse, da wir mit einer kontextfreien Grammatik nur eine Obermenge der Grammatik einer modernen Programmiersprache beschrieben haben.
Es gibt z.B. Überladene Operatoren, die auf richtige Anwendung überprüft werden müssen oder Variablen, die nicht im ganzen Programm gültig sind, sondern nur in Teilen des Programms. Das auf diese nicht außerhalb zugegriffen wird, muß geprüft werden.

Codeerzeugung:

Die Codeerzeugung ist sehr wichtig. Sie entscheidet später über die Geschwindigkeit des Programmes und ist ein weiterer komplizierter Teil. Sie umfaßt **die Abbildung der Speicherstrukturen (ganz besonders Register)** in der Zielsprache und **die Abbildung auf den Befehlssatz der Zielsprache**.

Registerzuteilung:

Ein Problem der meisten Programme ist es, daß wir mehr Variablen haben, als Register in der CPU vorhanden sind. Wir müssen so die Register der CPU irgendwie belegen. Das machen wir, indem wir **symbolische Register** verwenden. Die Register, die gerade aus den symbolischen Registern gebraucht werden, sind auch in den realen Registern vorhanden. Andere Werte sind im (langsamen) Hauptspeicher.

Das Problem ist nun, nach welcher Strategie wir bestimmen, wann ein Register aus den realen Registern verdrängt werden soll. Dies können wir rein zufällig machen, daß ist aber nicht so toll. Es gibt Strategien wie **FIFO** (First in First Out) und **LRU** (Last Recently Used). Hierbei können wir die Registerzuteilung „On-the-fly“ machen. Optimal ist aber das Verfahren mit **zukunftsorientierung**. Das Register, auf das am längsten nicht mehr zugegriffen werden wird, wird ausgelagert. Dazu müssen wir jedoch das Programm zwei mal durchgehen. Beschert hat uns dieses Verfahren der Herr **Belady**, der auch das Cacheseitenaustauschverfahren erfunden hat.

Registerzuteilungen machen übrigens auch nur in einem **Grundblock** Sinn. Grundblöcke sind Programmabschnitte, in die **nicht** von außen noch von innen heraus **gesprungen** werden kann.

Kapitel 4: (Kommunizierende und konkurrierende Prozesse)

Definition: Prozess

Ein Prozess ist ein in Ausführung befindliches Programm (=Code+Daten+Stack+Programmzähler...). Es ist die kleinste Einheit, der Betriebsmittel zugeordnet werden können.

Definition: Parallele Prozesse

Viele Prozesse werden echt parallel zueinander ausgeführt. Das muß auf mehreren Prozessoren geschehen

Definition: Verzahnte Prozesse

Viele Prozesse werden auf einem Prozessor ausgeführt, wobei in kurzen Abständen zwischen ihnen gewechselt wird. Es sieht so aus, als wären sie parallel, sie sind es aber nicht.

Definition: Nebenläufige Prozesse

In der Praxis ist es nicht so wichtig, ob ein Prozess nun echt oder „unecht" parallel ausgeführt wird. Man schmeißt alles in einen Topf und nennt es nebenläufigen Prozess. Nebenläufige Prozesse können auf einem Prozessor ausgeführt werden.

Definition: Threads

Threads sind sozusagen auch Prozesse. Hier jedoch wird auf Sicherheitsmechanismen zwischen den einzelnen Threads verzichtet (Fehlerschutzverletzung!). Das macht Threads schnell.

Multiprozessing:

Mehrere Prozessoren arbeiten mehrere Prozesse ab. (Parallele Prozesse)

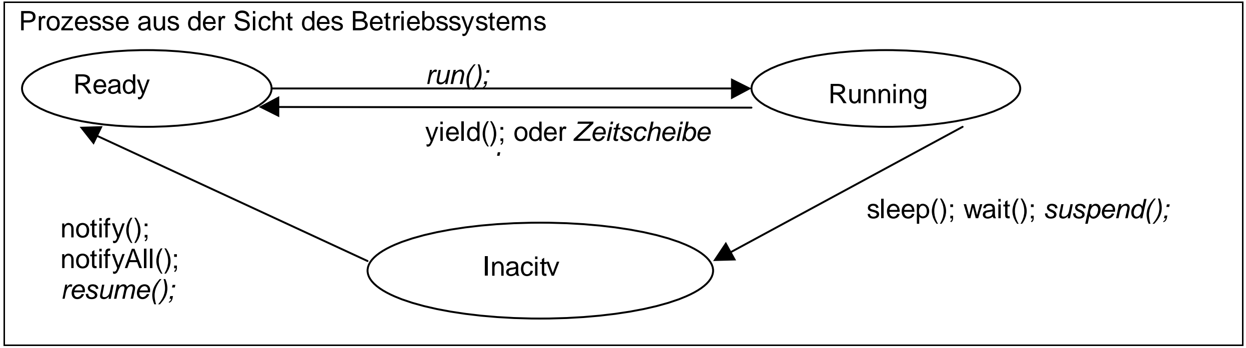
Multitasking:

Verzahnte Prozesse laufen auf einem Prozessor. Speicherverwaltung ist dem Anwender überlassen. Es gibt Multitasikingprozesse, die auf ein und denselben Speicher zugreifen.

Multiprogramming:

Verzahnte Prozesse. Speicherverwaltung durch das Betriebssystem.

Prozesse aus der Sicht des Betriebssystems



Erzeuger / Verbrauchersystem:

Ein **Erzeugerprozess** legt Einheiten in einen Puffer. Ist dieser schon zu voll wartet er. Ein **Verbraucher** holt Einheiten aus dem Puffer. Sind nicht mehr genügend Einheiten hier vorhanden, wartet er bis neue da sind. **Deadlock** wenn Erzeuger wartet, daß Puffer leerer wird, damit er neue Elemente hineinlegen kann, Verbraucher aber im Puffer nicht seine Anzahl der Elemente finden kann und ebenfalls wartet.

Dining Philosophers:

Deadlock, wenn alle eine Gabel hochheben und auf die zweite warten.

Prozesswechsel

Ein Prozesswechsel ist nicht trivial. Zuerst muß der alte Prozess gesichert werden. Das bedeutet das Register, Stack und Programmcounter ... gesichert werden müssen. Dann wird bestimmt, welcher Prozess als nächstes laufen soll. Dann werden alle die Daten dieses Prozesses wieder zurückgeschrieben. Prozesswechsel kosten Zeit, sind aber mit Hilfe von Spezialhardware deutlich zu beschleunigen. (Hierbei ist Spezialhardware - glaube ich - schon ein 386 und aufwärts. Er verwaltet mehrere virtuelle Maschinen.)

Threads:

Ein Thread ist eine Klasse. Sie muß von der Klasse Thread vererbt werden:

```
public class meinThread extends Thread {...}
```

In der Klasse ist folgende Methode enthalten. Sie wird aufgerufen, wenn ein neuer Thread dieser Klasse gestartet wird. Vergleichbar ist sie mit `public static void main(String Args []){}`:

```
public void  run() {...}
```

Wechselseitiger Ausschluß (Mutal Exclusion)

Damit sich zwei Prozesse nicht ins Gehege kommen, müssen sie sich wechselseitig aus einem kritischen Bereich aussperren. Dies funktioniert mit Hilfe von Monitoren.

Java bietet Monitore an:

`synchronized` vor jeder Methode, die nur zu jeder Zeit einmal gleichzeitig ausgeführt werden darf. In allen Methoden zusammen, die in einer Klasse mit `synchronized` bezeichnet sind, befindet sich höchstens ein Thread.

Es gibt noch folgende weitere Befehle. Sie dürfen nur in `synchronized` Methoden aufgerufen werden:

`wait()` Blockiert den aufrufenen Thread. Dieser kann dadurch reaktiviert werden, indem er von woanders mit `notify()` oder `notifyAll()` wieder aktiviert wird. `wait()` gibt die Betriebsmittel wieder frei. Es können andere Threads in die `synchronized` Methoden.
(Achtung: Diese Methode schleudert eine Exception. Folgendermaßen abfangen:
`try{wait();} catch (InterruptedException e) {}`
`notify()` Weckt einen vom Betriebssystem bestimmen Prozess. Aufrufen, wenn sich die Lage in den `synchronized` Methoden geändert hat und neue „action“ möglich ist.
`notifyAll()` Weckt alle anstehenden Prozesse, so daß sie ein Wettrennen veranstalten. Aufrufen, wenn sich die Lage in den `synchronized` Methoden geändert hat und neue „action“ möglich ist. Es ist besser `notifyAll()` statt `notify()` aufzurufen.

Man definiert als Monitor eine eigene Klasse, die sich um die Betriebsmittelvergabe kümmert und nur jeweils einen Thread in die kritischen Bereiche läßt. Auf dieser Klasse laufen dann verschiedene Threads, die auf sie zugreifen.

Deadlock

Ein Deadlock kommt zustande, wenn folgende Bedingungen erfüllt sind:

1. Exklusiver Zugriff auf einen Bereich
2. Halten von Betriebsmitteln, warten auf andere
3. Betriebsmittel können erst nach Erfolgreicher Benutzung zurückgegeben werden.
4. Circuläre Warteschlange: Prozess 1 hat Betriebsmittel 1 wartet auf Betriebsmittel 2, welches aber Prozess 2 hat, der auf Prozess 1 wartet, weil er Betriebmittel 1 benötigt. (Beliebig erweiterbar)

Scheduling-Verfahren:

Fast jedes Scheduling-Verfahren basiert im Grunde genommen auf ein Basis-Scheduling-Verfahren. Wir haben verschiedene Warteschlangen mit verschiedener Priorität, die dann mit dieser Priorität ausgeführt werden. Die Ausführung dauert eine gewisse Zeitscheibe lang. Der Prozess kann diese Zeitscheibe selbst unterbrechen, indem er sagt: „Ich will nicht mehr. Ich will ein Ereignis haben." oder der Scheduler unterbricht den Prozess, wenn die Zeitscheibe abgelaufen ist. Dann wird der Prozess wieder in eine Warteschlange mit gewisser Priorität eingeordnet. In die Warteschlangen können neue Prozesse von außen hereinkommen. Sie werden neu gestartet. Bei der Abarbeitung kann sich ein Prozess terminieren und wird somit entfernt. (Auch das Betriebssystem kann terminieren. Das interessiert uns jedoch momentan nicht so sehr.)

FIFO

Ältestes Verfahren. Die Prozesse, werden einer nach dem anderen abgearbeitet. Sie können entweder sagen, sie würden auf ein Ereignis warten und wieder eingeordnet werden oder sie können arbeiten. Wichtig bei diesem Verfahren ist, daß es keine Zeitscheibe gibt. Wenn ein Prozess im System hängt und abgestürzt ist und nicht mehr zum Scheduler zurückkehrt, dann ist Feierabend. Auch können Prozesse, die viel zu tun haben, das System ausbremsen.

Round-Robin-Scheduling

Es gibt zwei Prioritäten. Gestartete Prozesse können in beide eingeordnet werden. Dann gibt es eine Zeitscheibe. Wird der jeweilige Prozess in der Zeitscheibe fertig und gibt ans Betriebssystem zurück, dann kommt er zurück zur hohen Priorität. Wird er nicht fertig, kommt er in die niedrige Priorität. Dies hat den Vorteil, daß „Hard-working"-Prozesse im Hintergrund ablaufen, während im Vordergrund das System zügig weiterarbeitet. Meistens warten die Prozesse mit kurzer Laufzeit auch auf Ereignisse, was die Grafische Benutzeroberfläche dann beschleunigt.

E/A-orientiertes Scheduling

Es gibt drei oder mehrere Prioritäten. Dieses Schedulingverfahren bevorzugt die Prozesse, die gerne auf ein langsames E/A-Gerät zugreifen möchten (Druckerport,COM-Port,Streamer) Das hat den Vorteil, daß diese Geräte schon mal arbeiten und der Prozessor sich mit den Prozessen beschäftigen kann, die schnelle E/A-Geräte oder auch gar keine benötigen. So sind die Daten dann schnell da, weil sozusagen schon „vorbestellt" wurde.(Jeder COM-Port hat beispielsweise einen FIFO-Puffer. In diesen werden die Daten hineingeschrieben und dann ist der Bus wieder frei, während die Daten über die Leitung gehen)

SPT: Shortest Processing Time First

Der Prozess, der am schnellsten endet, wird zuerst bearbeitet:
Bearbeitungszeit Prozess a: 1
Bearbeitungszeit Prozess b: 9
Bearbeitungszeit Prozess c: 2
Zuerst a, dann c und dann b. Zusammen ist die Bearbeitungszeit von
Prozess a: 1
Prozess c: 1+2
Prozess b: 1+2+9
Somit ist $\frac{1+(1+2)+(1+2+9)}{3}=\frac{16}{3}=5,\bar{3}$ die mittlere Laufzeit eines Prozesses. Das ist ziemlich kurz

SRPT Shortest Remaining Processing Time First

Es werden auch Prozesse berücksichtigt, die gestartet werden.

SRPT ist die Optimalstrategie!

Lernen aus der Vergangenheit

Das **Problem** bei diesem Verfahren ist nur zu bestimmen, **wie lange ein Prozess noch läuft**, da das meistens nicht vorherbestimmt ist. Lösung: Schätzen!

Sei
T_i der aktuelle Schätzwert
X der Bedarf der CPU-Zeit im letzten Inactive-Inactive-Zyklus

Dann schätze folgendermaßen:
$T_{i+1}=aT_i+(1-a)X$
Durch a steuern wir die Vergeßlichkeit. Wählen wir a=0, so vergessen wir die Vergangenheit sofort (also T_i). a=1, so mißachten wir X. Dann ist dieses Verfahren wirklich schlecht.

Veranschaulichung:

Wenn gerade ein Task läuft, der ein schönes Apfelmännchen berechnet (natürlich auch hochauflösend), dann ist es sehr wahrscheinlich, daß dieser Task im nächsten Zyklus in den nächsten Millisekunden auch wieder viel Rechenzeit für sein Apfelmännchen benötigt. Hierbei können wir sogar a=0 setzen. Für Algorithmen, die stoßweise Rechenzeit benötigen muß der Vergeßlichkeitsfaktor höher gesetzt werden.

Speicherverwaltung:

Feste Partitionen:

Wir haben feste Partitionen im Speicher. Zu diesen gibt es entweder zu jeder eine Warteschlange oder eine Warteschlange für alle. Das Problem bei diesem Verfahren ist, daß eventuell die Paritionen nicht richtig ausgelastet werden, so daß z.B. das Betriebssystem gezwungen wird einen kleinen Prozess in eine große Partition zu legen.

Variable Partitionen:

Wenn ein Prozess gestartet wird, dann wird für ihn die passende Partition kreiert. Der Nachteil dieses Verfahrens ist es, daß der Speicher zerstückelt. Bewegungen innerhalb des Speichers sind sehr Zeitaufwendig.

Speicherplatzverwaltung: Bitmaps und verkettete Listen

Der Speicher läßt sich als Bitmap verwalten. Nachteil hierbei ist, daß das Bitmap relativ groß werden kann und somit auch wieder viel Platz benötigt.

Man kann aber auch eine verkettete Liste benutzen, in der in jedem Element eine „Wegstrecke" enthalten ist und gekennzeichnet ist, ob diese Wegstrecke belegt oder noch frei ist.

Algorithmen zur Speicherplatzvergabe:

First Fit: Wähle den ersten, wo der Prozess hineinpaßt.
Next Fit: Wähle den ersten, wo der Prozess hineinpaßt, ab der aktuellen Position
Best Fit: Wähle denjenigen aus allen, wo der Prozess am besten hineinpaßt, so daß wenig Platz drumherum ist. (Leider lange Suchzeit, es werden auch kleine Lücken im Speicher gemacht, die nicht genutzt werden können)
WorstFit: Wähle unter allen freien Bereichen den größten. Es werden keine kleinen freien Bereiche freigelassen, aber große Speicherbereiche werden konsequent vernichtet.

Virtueller Speicher

Wir können den Hauptspeicher (fast) beliebig erweitern, indem wir einen virtuellen Adressraum benutzen. Im Hintergrundspeicher befinden sich sogenannte Seiten, die bei Bedarf in den Hauptspeicher geladen werden und dann dort zur Verfügung stehen.

Virtueller Speicher:

Der virtuelle Speicher kann beschrieben werden durch ein Tripel (N,M,f) wobei
$N=\{0,1,...,n-1\}$ Menge der adressierbaren Seiten (logischer Adressraum)
$M=\{0,1,...,m-1\}$ Menge der Seitenrahmen (pysikalischer Adressraum)
Es gibt eine Funktion $f_t: N \rightarrow M \cup \{\perp\}$ die den zur Zeit t aktuellen Seitenzustand angibt.
Es ist $f_t(i)=j$ ($i \in N$, $j \in M$), falls die Seite i im pysikalischen Adressraum ist. Andernfalls $f_t(i)=\perp$

Behandlung von Seitenfehlern:

Die Seiten werden aus dem Hintergrundspeicher nachgeladen. Dabei werden genausoviele Seitenrahmen verdrängt, wie nachgeladen werden. $a=r_0r_1...r_{T-1}$ ist ein Referenzstring, der angibt, auf welche Seiten schon zugegriffen wurde (in der Theorie unendlich lang, in der Praxis werden irgendwann die Elemente vorne wieder gelöscht)

Belady-Anomalie

Verwenden wir als Strategie des Verdrängens von Seitenrahmen FIFO, so kann es bei mehr Seitenrahmen im Hauptspeicher dazu kommen, daß trotzdem mehr Seitenfehler auftreten.

Stackalgorithmen

Dies kann man genau verhindern, wenn man Stackalgorithmen benutzt

Definition: Stackalgorithmus

Sei **A ein Demand Paging Algorithmus** (also ein Algorithmus der Seiten nachlädt und alte verdrängt) **n** die Anzahl der **Seiten** und **m** die Anzahl der **Seitenrahmen** in M, **a** ein **Referenzstring** der Länge T. Bezeichne S(m,a) die Seitenmenge die nach a in den Rahmen im Hauptspeicher ist. Ein Demand Paging Algorithmus A ist ein Stack-Algorithmus, wenn für alle $m \in N$ und für alle a gilt
$\mathbf{S(m+1,a) \supset S(m,a)}$

Working Set

Jeder Prozess bekommt ein sogenannten Workingset. In ihm sind Seiten enthalten, die er in der Vergangenheit gebraucht hat. Da meistens immer nur dieselben Seiten benutzt werden, gibt es wenige Seitenfehler, wenn man das Workingset in den Rahmen im Hauptspeicher beläßt.

Java:

Zuerst ist sicherlich einmal wichtig, wie man aus der Standardeingabe einen Tokenstream macht, wobei ich mir nicht vorstellen kann, daß wir dies auswendig lernen müssen, aber sicher ist sicher:

```
try
{
    Reader r = new BufferedReader(new InputStreamReader(System.in));
    StreamTokenizer st = new StreamTokenizer (r);

    st.nextToken(); //Bevor überhaupt das erste Token kommt, muß man dies einmal aufrufen
    ... st.nval; //Richtig konvertieren
} catch (IOException e) {}
```

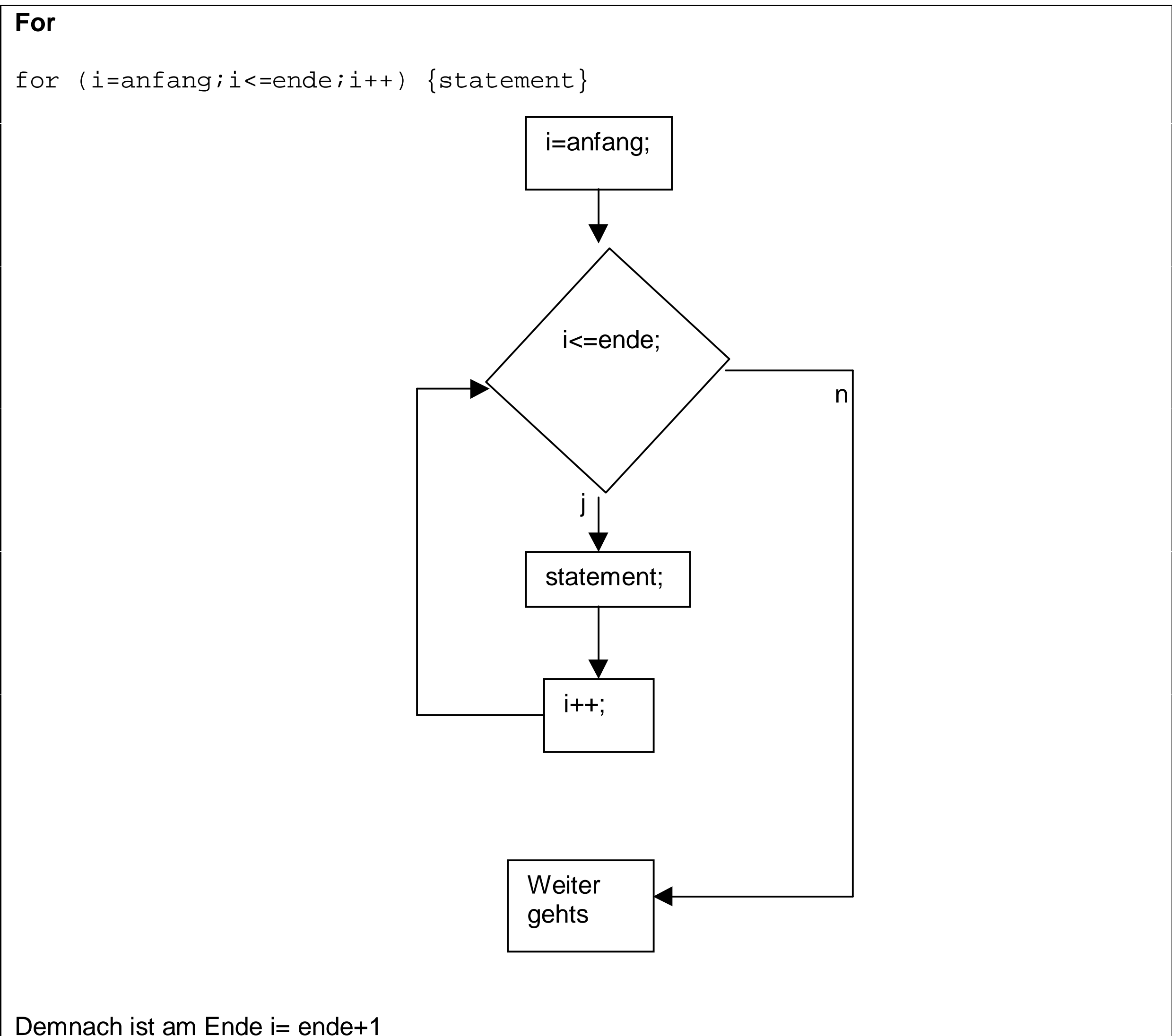
If

```
if (expression) {statements;} else {statements;} nextStatement;
if (expression) statement else statement; nextStatement;
```

Wichtig ist, daß man bei der Überprüfung, ob ein Wert gleich einem anderen Wert ist nicht = schreibt, sondern ==.

For

```
for (i=anfang;i<=ende;i++) {statement}
```



Demnach ist am Ende i= ende+1

While

```
while (expression) {}
```

Repeat until

```
do {statements} while !(expression)
```

Switch: (Wenn break weggelassen wird, dann werden die anderen case-Anweisungen auch noch durchgegangen. Java stoppt in diesem Falle nicht)

```
switch(expression)
{
    case value1: ... break;
    case value2: ... break;
    default: ... break;
}
```

Vererbung

Für die Klausur müssen wir unbedingt die Philosophie von Objekten und Vererbung kennen. Es würde hier zu weit ausschweifen, wenn ich dies alles noch einmal hier hinschreiben würde. Man lese sich die Sachen im Buch durch. Es sei nur gesagt, daß **`extends`** vererbt.

Interfaces:

Statt `class` `interface` schreiben. Alle zu verknüpfenden Methoden mit der Kopfzeile hineinschreiben. (So wie man es bei abstrakten Methoden macht) Abschließen mit Semikolon. Ein Interface benutzt man für eine Klasse, indem man im Klassenkopf **implements** schreibt.