Materialien zur Informatik II

R.Deissler

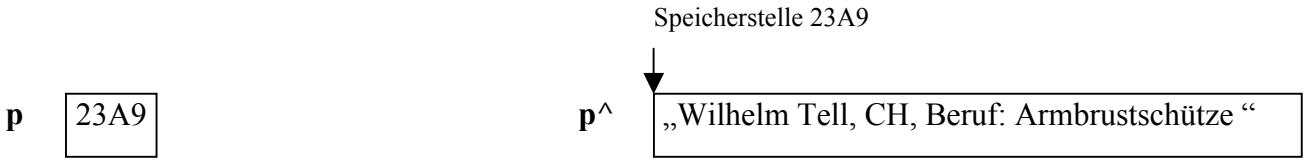
SS 2003

Inhalt

1 Dynamische Variablen und Zeiger

Wird Speicherplatz für Daten erst reserviert, wenn er zur Laufzeit eines Programms tatsächlich benötigt wird, dann spricht man von dynamischen Daten und dynamischen Variablen. Um auf Daten im Speicher zugreifen zu können, muss man wissen, an welcher Speicherstelle die Daten beginnen und wie viel Speicher sie umfassen. Weiter benötigt man einen Verwaltungsmechanismus für den dynamisch belegten Speicher, der zweierlei leistet: Er reserviert auf Anforderung entsprechenden Speicherplatz und gibt ihn wieder frei, wenn er nicht mehr benötigt wird.
Je nach Programmiersprache wird die Verwaltung vom System implizit übernommen (z.B. Java) oder der Programmierer muss die Verwaltung explizit selbst besorgen (C, Pacscal).

Um die Vorgänge richtig verstehen zu können, sollen hier die Prozesse am Beispiel von **Pascal** dargestellt werden, wo der gesamte Mechanismus klar zu Tage tritt.



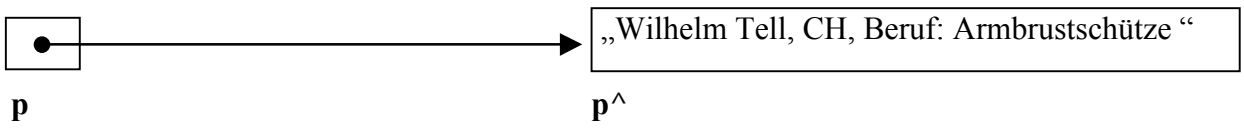
Zeigervariable (statisch),
enthält die **Adresse** der Bezugsvariablen.

Platz dauernd reserviert.

Bezugsvariable (dynamisch),
enthält die zu speichernden dynamischen **Daten**.

Platz nur reserviert, solange er benötigt wird.

Man sagt: Die Zeigervariable **p zeigt auf die Daten** und stellt den Zusammenhang schematisch oft folgendermaßen dar:



Die Prozedur, die Speicherplatz für die Daten reserviert und die Startadresse in der Zeigervariablen p speichert, heißt **`new(p)`**, die Prozedur, die den zuvor reservierten Platz wieder freigibt, heißt **`dispose(p)`**.
Der Platzbedarf für eine Zeigervariable beträgt so viele Bytes, wie zur Angabe einer Adresse nötig sind, in PCs sind das 4 Bytes.
Der Bereich, in dem ein Programm dynamischen Speicherplatz reserviert, heißt **Heap** (Halde, Haufen).

Hier folgt das einfachste Pascal-Programm, das den gesamten Prozess zeigt. Man beachte den Unterschied zwischen p und p^ sowie die unterschiedliche Stellung des Caret-Zeichens „^“ in der Typendeklaration `^string[20]` „Zeiger auf einen String mit maximal 20 Zeichen“ und bei der Bezugsvariablen `p^` „das, worauf p zeigt“.

```
program test;
uses crt;

var p:^string[20];

begin

  new(p);


  p^:='Hans';


  writeln(p^);


  dispose(p);

end.
```

Reserviert **zur Compilierzeit** Platz für einen Zeiger (4 Bytes)

Reserviert **zur Laufzeit** Platz auf dem Heap für einen String mit maximal 20 Zeichen und gibt den Zeiger darauf (die Adresse) in p zurück.

Weist der dynamischen Variablen einen Wert zu.

Gibt den Wert der dynamischen Variablen aus.

Gibt den reservierten Speicherplatz auf dem Heap wieder frei, so daß andere dynamische Variablen ihn wieder nutzen können.

Beachte:

- Nirgendwo braucht der Programmierer über die wirklichen Adressen Bescheid zu wissen, die gesamte Speicherverwaltung wird vom System über die Prozeduren `new` und `dispose` übernommen.
  Es gibt nur **einen** bestimmten festen Wert für eine Zeigervariable: **nil**. nil bezeichnet den Zeiger der nirgendwohin zeigt!!
  Symbolisch: **nil** oder •
- Die Zeigervariable p gibt an, bei welcher Adresse die dynamischen Daten beginnen. Der Umfang der Daten (also die Angabe, wie viele Bytes von der Anfangsadresse an zu den Daten gehören) ergibt sich in Pascal aus dem Typ der Bezugsvariablen p^. Im Beispiel weiß man durch die Angabe `var p:^string[20]` , dass von der Anfangsadresse an 21 Bytes für die Daten reserviert wurden.
  Es ist hier auch ein anderes Verfahren denkbar, das in der objektorientierten Programmierung an Stelle der Typenangabe für die Bezugsvariable benutzt wird: Aus den Daten selbst ist ihr Platzbedarf zu ermitteln (z.B. aus den ersten 2 Bytes oder ähnliches). Dann könnte ein Zeiger auf Daten von ganz verschiedenen Typen zeigen, was beim bisher diskutierten Modell nicht möglich ist.
- Wenn die Prozedur `new` mit dem Aufruf `new(p)` neuen Speicherplatz reserviert hat, dann stehen in diesem Speicherbereich noch keine sinnvollen Daten, eine Ausgabe mit `writeln(p^)` würde irgendwelchen zufällig auf dem Heap liegenden Daten ausgeben, allerdings würde keine Katastrophe eintreten.
- Wenn allerdings versucht wird, auf die Bezugsvariable p^ schreibend zuzugreifen, etwa mit `p^:=´Hans´`, ohne dass zuvor mit `new(p)` Speicherplatz reserviert wurde und p nun auf diesen zeigt, dann zeigt p irgendwo in den Speicher des Rechners (also auf eine zufällige Speicherstelle) und man überschreibt die dort stehenden Daten. Damit können Teile des Programms oder sogar, bei manchen Systemen, des Betriebssystems überschrieben werden. In diesem Fall wird sich der Rechner eventuell mit einem **Totalabsturz** verabschieden.
  Auch in Fällen, in denen von der Verwendung von Zeigern nicht explizit Gebrauch gemacht wird (z.B. bei den Programmiersprachen Java oder VBA) kann dieser Fall eintreten, wenn man keine der Prozedur `new` entsprechende Methode anwendet bevor dynamische Daten benutzt werden.

Zusammenfassung der Konstanten, Prozeduren und Funktionen, die für das Operieren mit Zeigern nötig sind (Pascal):

nil	Konstante für den Zeiger „nirgendwohin“ (Nullzeiger) . nil zeigt definiert nirgendwo hin, dies ist nicht zu verwechseln mit einem undefinierten Zeiger ohne bestimmten Wert, der zufällig irgendwohin zeigt und bei falscher Verwendung großen Schaden anrichten kann.
new	Prozedur zum Reservieren von Speicherplatz (in neueren Pascalversionen und in Delphi ist diese Prozedur durch eine Funktion gleichen Namens ersetzt worden)
dispose	Prozedur zum Freigeben von Speicher
memavail	Funktion, mit der festgestellt werden kann, wie viel Speicherplatz noch auf dem Heap verfügbar ist.

Deklaration von Zeigertypen:

type
  <typname> = ^ <name des Bezugstyps>

Bemerkung:

Nicht alle Programmiersprachen erfordern eine *explizite* Verwendung von Zeigervariablen, um dynamische Datenstrukturen aufzubauen. Gerade in den objektorientierten Sprachen (z.B. Java) werden die zuvor angesprochenen Prozesse vom System durchgeführt. Objekte werden meist dynamisch zur Laufzeit erzeugt. Dazu werden nur die Bezugsvariablen verwandt, nicht die Zeigervariablen, und die Reservierung des benötigten Speicherplatzes geschieht durch einen Aufruf einer new-Funktion mit der Bezugsvariablen. Dies gestaltet die Programmierung einfacher, verführt allerdings dazu zu vergessen, dass nicht mehr benötigter Speicherplatz auch wieder freigegeben werden muss, damit das Programm während einer längeren Ausführungszeit nicht nach und nach den gesamten Speicher des Rechners belegt. Manche Systeme (z.B. Java) bieten auch dafür eigene Mechanismen, die automatisch nicht mehr benötigten Speicherplatz freigeben (automatic „garbage collection“, „Speicherbereinigung“).

1.1 Ein einfaches Beispiel

Das folgende Beispiel zeigt nochmals die grundlegenden Schritte beim Umgang mit dynamischen Variablen. Es ist in dieser Form noch nicht sehr sinnvoll.

```
Program point1;
type
   kurzTyp=string[20];
```

```
var
  pname:^kurztyp;

begin
 new(pname);              {Speicherplatz für einen string vom kurzTyp auf dem „Heap“ reserevieren und Zeiger
                           darauf in der Zeigervariablen pname zurückgeben }
 writeln(pname^);         {Ausgabe der Referenzvariablen: keine sinnvolle Anweisung, da zwar der Zeiger
                           pname definiert ist, nicht jedoch der Inhalt der Referenzvariablen pname^ }
 pname^:='Hans';          {Zuweisung eines Wertes an die Referenzvariable}
 pname^:=pname^+' Fritz'; {Arbeiten mit der Referenzvariablen wie mit einer statischen Variablen}
 writeln(pname^);
 dispose(pname);          {Freigeben des reservierten Speicherplatzes}
end.
```

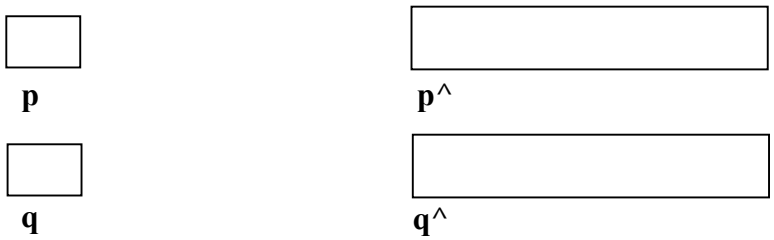
1.2 Übungen zu Zeigern

Obwohl nur sehr wenige Elemente zum Umgang mit Zeigern benötigt werden, sind die zugrunde liegenden Konzepte doch relativ schwierig und subtil. Daher sollen vor einer echten Anwendung von Zeigern noch einige Übungen zu Zeigern die Grundlagen festigen.
Wir nehmen an es seien in einem Programm zwei Zeigervariablen auf Strings durch die Zeile

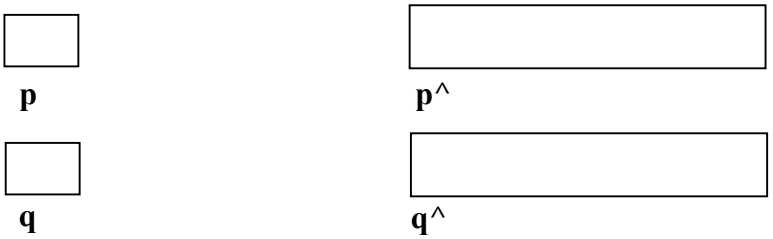
```
var p,q:^string[20];
```

deklariert. Weiter sei noch nichts geschehen.

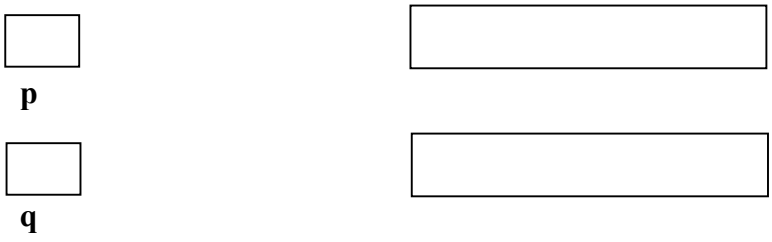
Beschriften Sie die folgenden symbolischen Darstellungen sinnvoll wo dies möglich ist, schreiben Sie „?“ wenn man zu dem gegebenen Zeitpunkt noch keine Angabe machen kann.



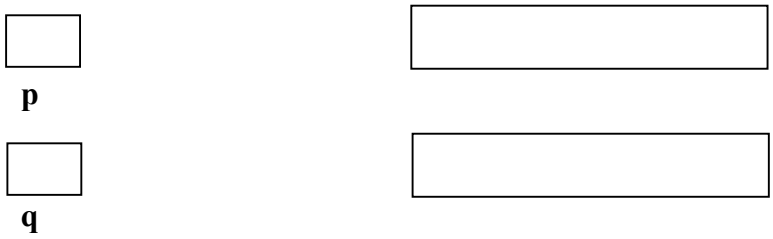

```
new(p); new(q);
```



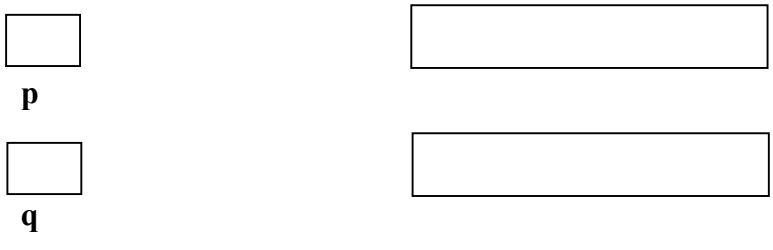

```
p^:=´Evelyn´; q^:=p^;
```



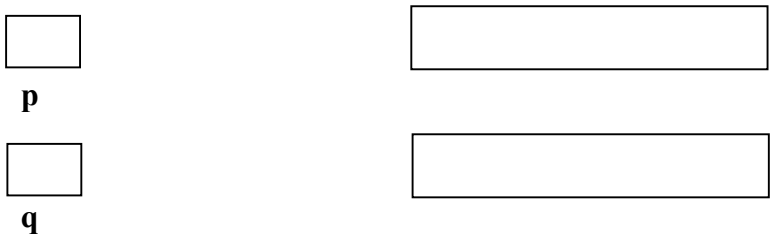

```
q^:=´Nora´;
```



```
p:=q;
```




```
p^:='Fred';
```



`writeln(q^);` Was wird ausgegebeen?

`dispose(q); dispose(p);` Welches Problem ergibt sich?
(mit dem zuvor gesagten ist das wohl nicht eindeutig zu beantworten)?
Was hätte man wohl anders machen müssen?

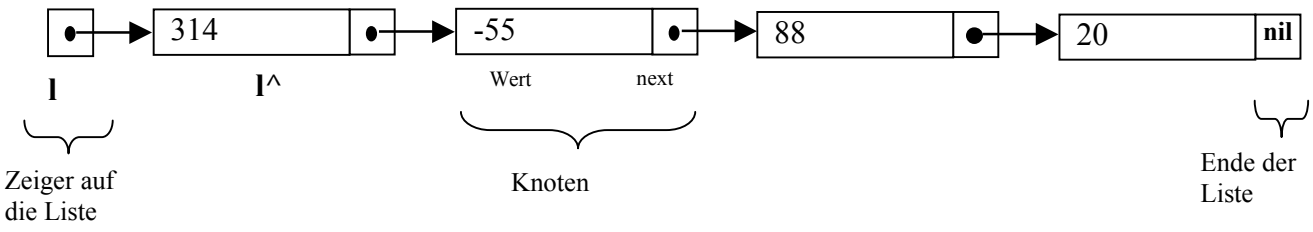
Beachten Sie im Vorangehenden die Unterschiede `p:=q` und `p^:=q^` !

1.3 Aufbau von Listen mit Hilfe von Zeigern

Hier soll eine wesentliche Anwendung des Zeigerkonzeptes gezeigt werden: der Aufbau und die Bearbeitung von dynamisch veränderbaren Listen von Daten. Im Rahmen der Behandlung der objektorientierten Programmierung (OOP) werden wir weitere Anwendungen des Zeigerkonzeptes kennenlernen. Auch Listen werden dort eine große Rolle spielen. Je nach Programmiersprache wird man mit Zeigern explizit umgehen müssen (C, C++) oder das Konzept implizit in andere Konzepte integriert finden (Delphi, Java, Java Skript, Visual Basic). Im ersten Fall ist ein Verständnis der grundlegenden Ideen unerläßlich, im zweiten zumindest sehr hilfreich.

Überall in der Informatik treten Baumstrukturen auf, etwa als Verzeichnisbäume eines Betriebssystems. Baumstrukturen können entweder als Erweiterung von Listenstrukturen aufgefaßt werden oder sie können mit Hilfe von Listen aufgebaut werden. Diese Tatsache zeigt die Bedeutung von Listen.

Schematische Darstellung einer (einfach verketteten) Liste:



Dabei ist ein Knoten ein Record, der aus dem eigentlichen Listeneintrag (Wert) und einem Zeiger (next) auf den nächsten Knoten besteht. Das Ende der Liste wird durch einen Knoten bestimmt bei dem der Zeiger den Wert nil hat. Die Liste wird zugänlich durch einen Zeiger l, der auf den ersten Knoten zeigt. Die Listeneinträge können von beliebige Datentypen sein, sollen aber zunächst einmal alle vom gleichen Typ sein.
Im Beispiel sollen die Listeneinträge alle vom Datentyp `longint` sein. Es wird die Liste

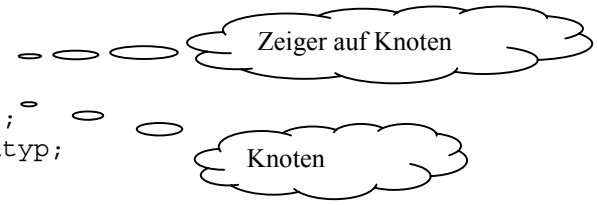
[314 , -55 , 88 , 20]

von Zahlen dargestellt.

Um die Konstruktion besser zu verstehen ist es am besten, gleich die syntaktisch korrekten Bezeichnungen von Pascal einzuführen, statt weitere umgangssprachliche Umschreibungen zu verwenden.

Dazu müssen wir zwei neue Datentypen definieren: einen Typ für Knoten und einen für Zeiger *auf* Knoten.

```
type
  PKnotentyp = ^Knotentyp;
  Knotentyp= record
               wert:longint;
               next:PKnotentyp;
             end;
```



Beachten sie die scheinbar zirkuläre Definition! Tatsächlich ist zur Bestimmung des Platzbedarfs für einen Zeiger zunächst unwichtig zu wissen, wie viel Speicherplatz die Daten benötigen, *auf* die der Zeiger zeigt.

Ist `k` eine Variable für einen Knoten vom Typ `Knotentyp` , dann bezeichnet

`k.wert`	den Wert dieses Knotens,
`k.next`	den Zeiger auf denFolgeknoten.

Dabei ist es möglich, dass `k.next` den Wert nil hat, wenn es sich um den letzten Knoten der Liste handelt.

Jetzt wird's etwas kompliziert:
Die gesamte Liste ist zugänglich über eine einzige Zeigervariable l vom Typ `PKnotentyp` !
Wir legen im folgenden das Beispiel oben zugrunde.

Ist l gegeben, dann bezeichnet

l^	den ersten Knoten,
l^.wert	314
l^.next	den Zeiger auf den zweiten Knoten,
l^.next^	den zweiten Knoten,
l^.next^.wert	-55,
l^.next^.next	den Zeiger auf den dritten Knoten,
l^.next^.next^	den dritten Knoten,
l^.next^.next^.wert	88,
l^.next^.next^.next	den Zeiger auf den vierten Knoten,
l^.next^.next^.next^	den vierten Knoten,
l^.next^.next^.next^.wert	20,
l^.next^.next^.next^.next	nil !!!!!!!!

Obwohl diese Notation in Pascal zulässig ist, wird man natürlich kaum in dieser unübersichtlichen Weise auf Listenelemente zugreifen. Außerdem ist noch völlig offen, auf welche Weise eine solche Liste im Rechner angelegt werden kann. Dies mögen die folgenden schematisch dargestellten Schritte veranschaulichen, bevor danach der formale Pascal-Code angegeben wird.

Wie wird wohl die einfachste Liste aussehen?
Antwort: Das ist die leere Liste, die kein einziges Element hat und einfach durch den Zeiger `nil` dargestellt wird.

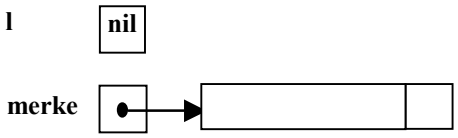
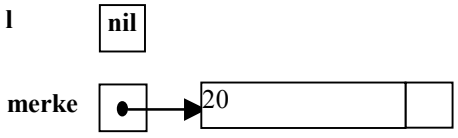
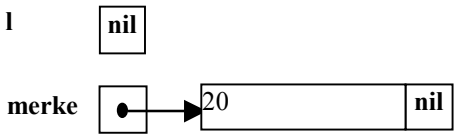
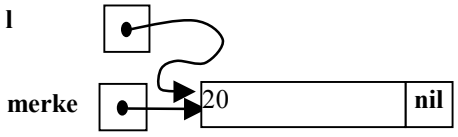
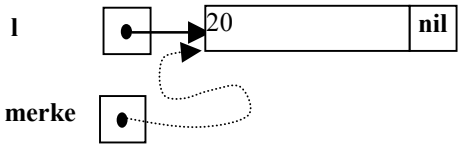
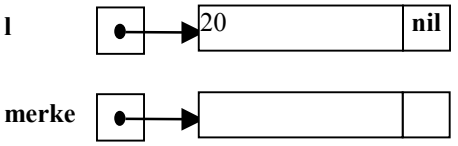
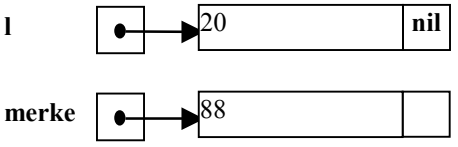
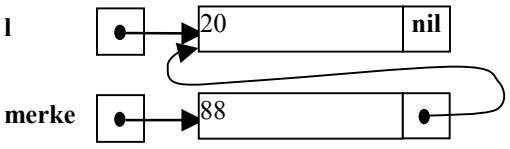
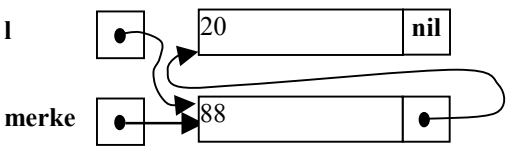
1.3.1 Aufbau einer Liste im einfachsten Fall (schematisch).

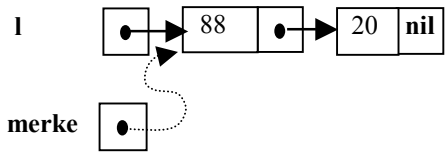
Wir benötigen zwei Variablen vom Zeigertyp (`PKnotentyp`):

`l`	steht für den Zeiger auf die aufzubauende Liste,
`merke`	steht für einen zwischendrin benötigten Hilfszeiger auf Knoten

Die Liste wird *vom Ende her aufgebaut*, indem man zunächst mit der leeren Liste beginnt und mit jedem Schritt *ein neues Element vorne an die Liste* anfügt. Es werden die folgende Listen schrittweise erzeugt: [] (leere Liste), [20] , [88 , 20] , ...

	Schematisch	Erklärung	Code
1.	**l** **nil**	Start : `l` als leere Liste erzeugen. Der erste Schritt ist fertig. `l` repräsentiert jetzt die leere Liste [ ]	`l:=nil`

2.		`l` ist noch immer die leere Liste Platz für einen neuen Knoten reservieren und die Adresse für diesen Speicherplatz in der Hilfsvariablen `merke` festhalten	`new(merke)`
		`l` ist noch immer die leere Liste Letzes Listenelement als Wert für den neuen Knoten zuweisen	`merke^.wert:=20`
		`l` ist noch immer die leere Liste. Das Ziel von l als next-Zeiger für den erzeugten Knoten definieren. `next` zeigt dorthin, wo l hinzeigt (das ist die im vorangehenden Schritt erzeugte Liste)	`merke^.next:=l`
		`l` jetzt auf den neuen Knoten zeigen lassen: „Zeiger umhängen“. `l` und `merke` zeigen gegenwärtig auf den gleichen Speicherplatz. `l` repräsentiert jetzt die Liste [20]	`l:=merke`
3.		Situation zu Beginn des nächsten Schrittes, nur etwas anders dargestellt: `l` repräsentiert die Liste [20]. `merke` zeigt auch noch auf dieselbe Stelle, das ist aber nicht mehr wichtig.	
		`l` repräsentiert noch die Liste [20] Platz für einen neuen Knoten reservieren und die Adresse für diesen Speicherplatz in der Hilfsvariablen `merke` festhalten	`new(merke)`
		`l` repräsentiert noch die Liste [20] Nächstes Listenelement als Wert für den neuen Knoten zuweisen	`merke^.wert:=88`
		`l` repräsentiert noch die Liste [20] Das Ziel von l als next-Zeiger für den erzeugten Knoten definieren. `next` zeigt dorthin, wo l hinzeigt. `merke` repräsentiert jetzt die Liste [88 , 20]	`merke^.next:=l`
		Zeiger `l` „umhängen“, so dass l die Liste [88 , 20] repräsentiert.	`l:=merke`

4.		Situation zu Beginn des nächsten Schrittes, nur etwas anders dargestellt: `l` repräsentiert die Liste [88 , 20]. `merke` zeigt auch noch auf dieselbe Stelle wie zuvor, das ist aber nicht mehr wichtig. Jetzt geht's weiter wie zuvor.	

Es sollte klar sein, dass ein solcher Listenaufbau üblicherweise in einer Wiederholungsschleife durchgeführt wird, wobei die Listenelemente beispielsweise über die Tastatur eingegeben werden.

1.3.2 Zugriff auf Listen – LIFO- und FIFO-Strukturen

Repräsentiert ein Zeiger `l` eine Liste, beispielsweise die Liste [314 , -55 , 88 , 20] , dann bedeutet

`l^.wert` **das erste Element** der Liste, also den **Wert** 314,
`l^.next` **die Restliste ohne das erste Element** der Liste, also die **Liste** [-55 , 88 , 20].

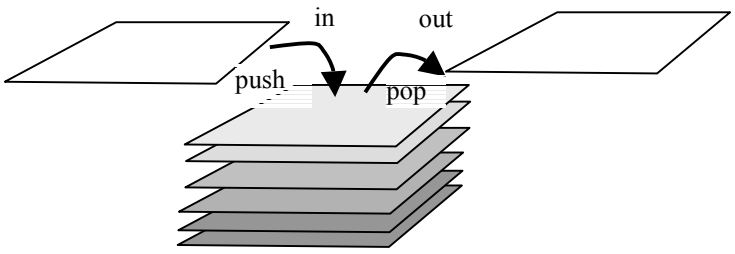
Dies zeigt, dass man auf die Elemente eine Liste nur schrittweise zugreifen kann, und zwar über die beiden Operationen „**erstes Element der Liste**" und „**Restliste ohne erstes Element**". Genau diese Operationen stehen in den listenorientierten Sprachen LISP und LOGO als Basisoperationen zur Verfügung. Das zweite Element einer Liste `l` wird so angesprochen als „erstes Element von Restliste von `l`" (in der Syntax des deutschen LOGO als „erstes ohneerstes :l", vergl.2.3). Wenn man noch eine Operation hinzu nimmt, die den in 1.3.1 skizzierten Prozess zum Listenaufbau durch „**vorne Anfügen eines Wertes e an die Liste l**" unterstützt, dann sieht man, dass man bei einer Liste auf denjenigen Wert **zuerst** zugreift, der **zuletzt** hinzugefügt wurde. Solche Strukturen nennt man **LIFO**-Strukturen als Abkürzung von „**L**ast **I**n, **F**irst **O**ut". Listen sind also LIFO-Strukturen.

Standardbeispiel für den Einsatz von LIFO-Strukturen ist der in der Informatik häufig gebrauchte **Stapel** (**stack**). Man denkt dabei an einen Stapel von Blättern, die man übereinander ablegt. Beim Hinzufügen eines Blattes wird dies oben auf dem Stapel abgelegt, und beim Zugreifen nimmt man das zuletzt abgelegte Blatt als erstes wieder auf.

Unter diesem Aspekt des Anfügens und Zugreifens betrachtet man auch **FIFO**-Strukturen, Abkürzung von „**F**irst **I**n, **F**irst **O**ut". Das Modell für diese Struktur ist die **Warteschlange** (**queue**), bei der immer derjenige zuerst bedient wird, der zuerst gekommen ist. Diese Struktur entspricht dem einer Liste, bei der Werte hinten angefügt und vorne entnommen werden. Die Konstruktion einer FIFO-Struktur ist etwas komplizierter als die einer Liste, da man zum Anfügen und zum Entnehmen zwei Zeiger für den Anfang und das Ende der Struktur nötig sind.

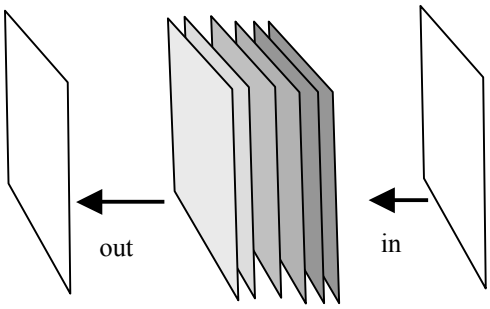
Ein sehr bekanntes Beispiel für eine FIFO-Struktur ist der Tastaturpuffer eines Computers: Wenn ein Benutzer Zeichen über die Tastatur eingibt, die der Computer nicht unmittelbar verarbeiten kann, dann werden diese in eine Warteschlange gestellt, aus der der Computer die zuerst eingegebenen Zeichen auch zuerst entnimmt, wenn er wieder Zeit dafür hat.

LIFO-Struktur



Stapel

FIFO-Struktur



Warteschlange

1.3.3 Beispielprogramm für einfachste Listenverarbeitung

- Typendeklaration
- Liste einlesen

- Liste ausgeben (ohne die Liste zu „verlieren“)
- Liste löschen (Speicherplatz freigeben)

```
program list1;
{ Programm für den Umgang mit einfach verketteten Listen }

uses crt;

type
  PKnotentyp = ^Knotentyp;
  Knotentyp= record
               wert:longint;
               next:PKnotentyp;
             end;

var
  liste:PKnotentyp;

{-------------------------------------------------------------------}

procedure liste_einlesen(var l :PKnotentyp);
var
  merke :PKnotentyp;
  zahl:longint;
begin
 l:=nil;
 repeat
   write('Zahl : ');readln(zahl);
   if zahl<>0 then
     begin
       new(merke);
       merke^.wert:=zahl;
       merke^.next:=l;
       l:=merke;
     end;
 until zahl=0;
end;

{-------------------------------------------------------------------}

procedure liste_ausgeben(l :PKnotentyp);
{ Zerstört die Liste l im Hauptprogramm nicht, da l kein Var-Parameter ist }
begin
 while l<>nil do
   begin
     writeln(l^.wert);
     l:=l^.next;
   end;
end;

{-------------------------------------------------------------------}

procedure liste_loeschen(l :PKnotentyp);
var
  merke:PKnotentyp;
begin
    while l<>nil do
      begin
        merke:=l^.next;
        dispose(l);
        l:=merke;
      end;
end;
```

```
{-------------------------------------------------------------------}

begin { Hauptprogramm }

 clrscr;
 writeln(sizeOf(Knotentyp));
 writeln('Heap : ',MemAvail);
 liste_einlesen(liste);
 liste_ausgeben(liste);
 writeln('Heap : ',MemAvail);
 liste_loeschen(liste);
 writeln('Jetzt Liste wieder gelöscht : ');
 writeln('Heap : ',MemAvail);
 readln;
 end.
```

1.3.4 Beispielprogramm für fortgeschrittenere Listenverarbeitung

Zusätzlich zu den einfachen Prozeduren aus Programm list1 findet man hier

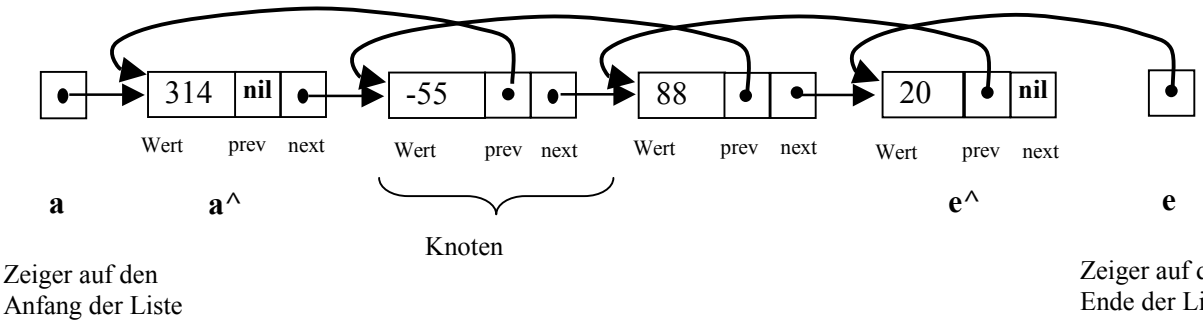
- ein Element in eine Liste an geeigneter Stelle einfügen, wenn die Listenelemente in natürlicher Weise geordnet sind (wie Zahlen oder Namen).
- Eine Liste sortieren.

```
program list2;
{ Programm für den Umgang mit einfach verketteten Listen }
uses crt;
type
  PKnotentyp = ^Knotentyp;
  Knotentyp= record
               wert:integer;
               next:PKnotentyp;
             end;
var
  liste:PKnotentyp;
  a:integer;
{-------------------------------------------------------------------}

procedure liste_einlesen(var l :PKnotentyp);
Wie in Programm list1
{-------------------------------------------------------------------}
procedure liste_ausgeben(l :PKnotentyp);
Wie in Programm list1
{-------------------------------------------------------------------}
procedure liste_loeschen(l:PKnotentyp);
Wie in Programm list1
{-------------------------------------------------------------------}
```

```
procedure einfuegen(a:integer;var l:PKnotentyp);
{ Mit Rekursion : Viel einfacher als ohne Rekursion ! }
var
  merke:PKnotentyp;
begin
  if l=nil then
    begin
     new(merke);
     merke^.wert:=a;
     merke^.next:=nil;
     l:=merke;
    end
  else
    if a<=l^.wert then
      begin
       new(merke);
       merke^.wert:=a;
       merke^.next:=l;
       l:=merke;
      end
    else einfuegen(a,l^.next);
end;

{--------------------------------------------------------------------}

procedure sortieren(var l:PKnotentyp);
{ Rekursiv : Viiiel einfacher !!                                     }
var
 lmerke:PKnotentyp;
begin
  if l<>nil then
   begin
    sortieren(l^.next);
    einfuegen(l^.wert,l^.next);
    lmerke:=l;
    l:=l^.next;
    dispose(lmerke);        { Nötig,damit kein Platz reserviert bleibt ! }
   end;
end;

{--------------------------------------------------------------------}

begin { Hauptprogramm }
 clrscr;
 liste_einlesen(liste);
 liste_ausgeben(liste);
 write('Element einfügen. Zahl : ');readln(a);
 einfuegen(a,liste);
 writeln('Eingefügt : ');
 liste_ausgeben(liste);
 writeln('Sortieren : ');
 sortieren(liste);
 liste_ausgeben(liste);
 liste_loeschen(liste);
 writeln('Jetzt Liste wieder gelöscht : ');
 readln;
end.
```

1.4 Zweifach verkettete Listen , Bäume

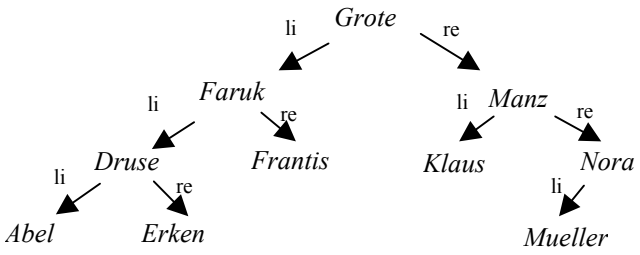
Ähnlich wie die zuvor definierten einfach verketteten Listen lassen sich auch komplexere Strukturen mit Hilfe von Zeigern definieren. Will man etwa eine FIFO-Struktur mit Hilfe von Zeigern aufbauen, dann ist eine einfach verkettete Liste nicht geeignet, da man dort nur von einem Ende unmittelbar auf die Listenelemente zugreifen kann. Hier bietet sich eine **doppelt verkettete Liste** an, die unten schematisch dargestellt ist.



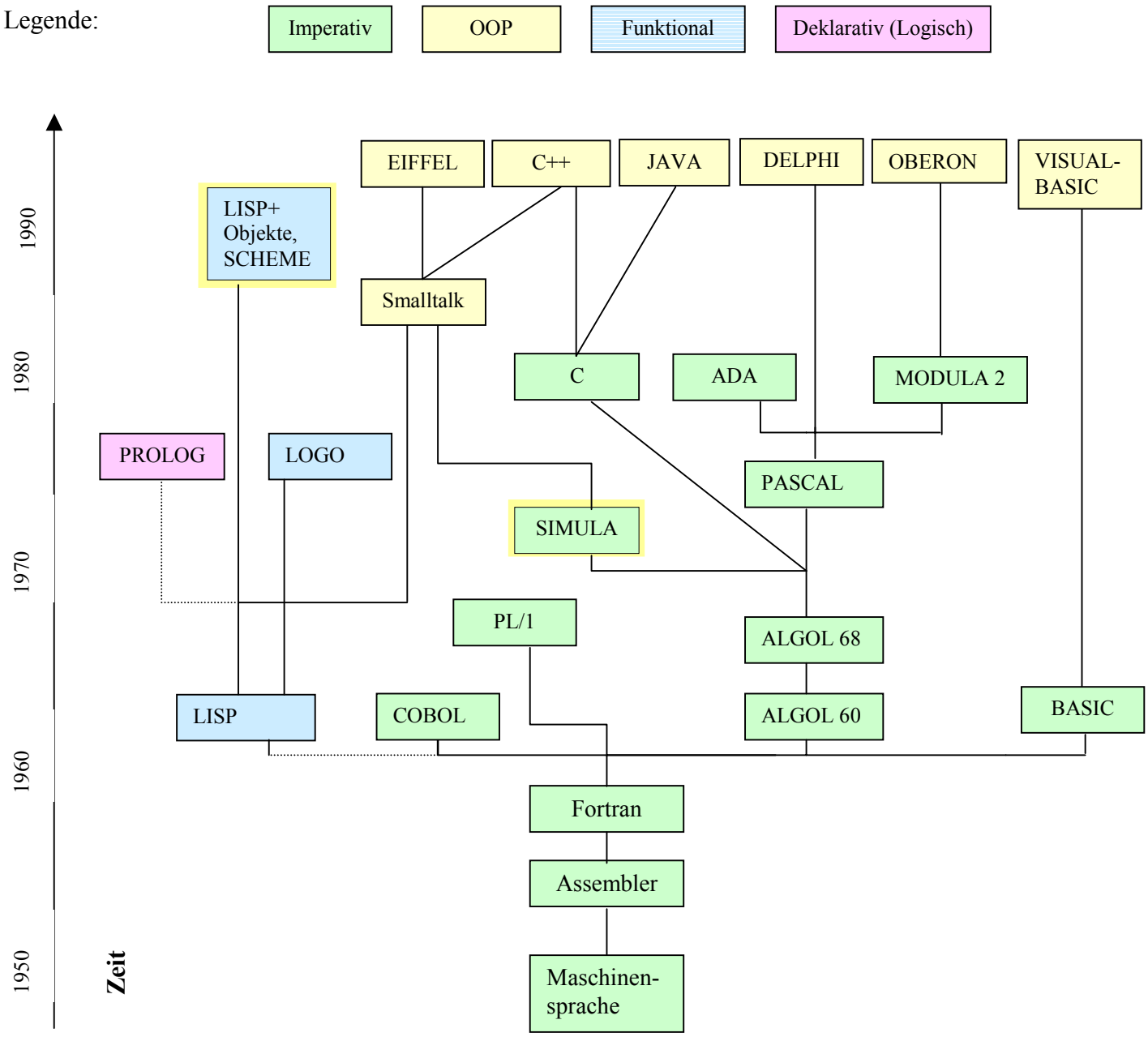
Geben Sie die notwendigen Deklarationen in PASCAL an. Beachten Sie, dass nun eine Liste durch zwei Zeiger a und e gegeben ist. Wie wird das erste, wie das letzte Element der Liste angesprochen? Skizzieren Sie die Operationen „Wert vorne an die Liste anfügen“ und „Wert hinten an die Liste anfügen“.

Binäre Bäume

Analog zu doppelt verketten Listen lassen sich Baumstrukturen definieren. Versuchen Sie, die im folgenden Bild gezeigte Struktur mit Zeigern darzustellen und die entsprechenden Definitionen zu geben.



2 Übersicht über Programmiersprachen



2.1 Maschinensprachen, Assemblersprachen.

Im Vordergrund stehen die Anweisungen, die die Zentraleinheit eines Rechners ausführen kann, nicht das zu lösende Problem.
Vorteil: Sehr **effiziente Programme** unter Ausnutzung der Eigenheiten der Maschine.
Nachteil: Sehr **unübersichtlich**, **schwer zu programmieren**, von Maschine zu Maschine verschieden, **kaum** gute **Strukturierungsmöglichkeiten**. Nur noch für sehr zeitkritische Programmteile verwandt.

Unter Maschinensprachen versteht man meist den Binärcode der Anweisungen, unter Assemblersprache eine eins-zu-eins Umsetzung des Maschinencodes in sogenannte mnemonische Codes (Codes, die man sich leicht merken kann, wie z.B. *add* für einen Additionsbefehl). Assembler (auch Assemblierer) sind Programme, die mnemonischen Code in Maschinencode übersetzen. Heute bieten Assembler zusätzlich zur reinen Übersetzung noch viele Hilfen zum einfacheren Programmieren an (Makros, Variablendefinitionen).

2.2 Problemorientierte Sprachen.

Hierunter fallen alle heute gebräuchlichen Sprachen. Die Sprachen sind weitgehend unabhängig von der Maschine, auf der sie laufen sollen. Ein Compiler oder ein Interpreter übernimmt die Übersetzung in die der Maschine verständliche Sprache.

2.2.1 Imperative Sprachen, prozedurale Sprachen.

Im Vordergrund stehen Anweisungen, mit denen ein Algorithmus ausgeführt wird. Algorithmen werden in einzelne Prozeduren zergliedert, die einzelne Teilaufgaben übernehmen und die bei neueren Sprachen zu Modulen sinnvoll zusammengehörender Prozeduren zusammengefasst werden. In neuerer Zeit erlauben viele dieser Sprachen auch mit Klassen und Objekten zu arbeiten, um den erzeugten Code später wieder verwenden zu können (→ 2.2.3).

2.2.2 Funktionale Sprachen.

Hierher gehören alle Versionen von LISP (**LIS**t **P**rocessing), als Vereinfachung davon LOGO, und ein spezieller LISP-Dialekt SCHEME. Ein LISP Programm ist eine Funktion, die Eingabedaten auf Ausgabedaten abbildet und selbst dazu wieder viele Hilfsfunktionen benutzt. LISP ist eine logisch sehr strenge Sprache und wird daher oft als Einstiegssprache im Informatikstudium eingesetzt. Grundlage von LISP ist der einzige Datentyp *Liste*, aus dem alle übrigen Typen aufgebaut werden. LISP Programme sind selbst Listen und können sich daher selbst verarbeiten und modifizieren. Das macht sie zur Standardsprache für die Forschung auf dem Gebiet der *Künstlichen Intelligenz*.

2.2.3 Objektorientierte Sprachen.

Die erste reine OO-Sprache war Smalltalk, die auch heute noch weiterentwickelt wird. Die meisten übrigen weiter verbreiteten Sprachen sind Mischsprachen, die sowohl imperative als auch OO Programmierung erlauben. Das OO Programmierparadigma scheint sich gegenwärtig fast überall durchzusetzen, es gibt internationale Standards und Normen, die den Austausch von Objekten regeln (Schnittstellen für Client-Server Anwendungen). Allerdings realisieren nicht alle unten als OO-Sprachen bezeichneten Sprachen sämtliche Prinzipien der OO Programmierung, sondern sind eher als Sprachen aufzufassen, die als Objekte bezeichnete *Komponenten* statt *Klassen* und *Objekte* mit Vererbung im strengen Sinn benutzen. Auf diese Unterschiede soll hier zunächst nicht eingegangen werden.
Weitere reine OO Programmiersprachen sind Java, Eiffel, und Oberon, wovon die letzteren beiden eher im Ausbildungsbereich als in der Praxis eine Rolle spielen. Die Skriptsprache Javascript (nicht zu verwechseln mit Java) die ebenfalls objektorientierte Züge trägt, hat zur einfachen Manipulation von Internet-Seiten große Bedeutung erlangt. Auch die Sprache **V**isual **B**asic arbeitet mit Objekten, wie auch **V**isual **B**asic for **A**pplications (VBA), die zur Programmierung in den Programmsystemen MS-Word, MS-Excel und MS-Access dient und mit diesen Systemen mitgeliefert wird,. Die Entwicklungsumgebung DELPHI der Firma Inprise (früher Borland) besteht aus der Mischsprache Object Pascal zusammen mit einer sehr umfangreichen Sammlung von vorgefertigten Objekten zur Gestaltung von Datenbankanwendungen unter Windows oder für das Internet.
Ein weiteres Beispiel aus dem Bildungsbereich ist das Autorensystem TOOLBOOK.
Die OOP Sprachen werden später noch ausführlich diskutiert werden.

2.2.4 Deklarative Programmiersprachen.

Hierunter fällt im wesentlich nur die Sprache PROLOG (**PRO**gramming in **LOG**ic). PROLOG ist eine Sprache für die künstliche Intelligenz und hat in den 80er Jahren große Hoffnungen auf „intelligente“ Computerprogramme geweckt. Japan hat bei einem großen Entwicklungsprojekt für intelligente Programme (the 5th generation project) auf PROLOG gesetzt. Das Projekt ist leider Anfang der 90er Jahre sang- und klanglos in der Versenkung verschwunden. PROLOG ist heute eine Sprache zur Entwicklung von Expertensystemen und wird in diesem Zusammenhang auch bei der Entwicklung von intelligenten tutoriellen Programmen eingesetzt, bei denen das Programm die Rolle eines Tutors mit Expertenwissen zu den Fehlern und Lernstrategien des Lernenden übernehmen muss.
Die Programmierung in PROLOG unterscheidet sich grundlegend von der Programmierung in allen anderen Sprachen, auch wenn der PROLOG-Mechanismus teilweise in LISP integriert werden kann. Ein Problem wird nur mit Hilfe von Fakten und Regeln beschrieben, ein Lösungsweg aber nicht angegeben. Das PROLOG System soll selbst aus dieser Wissensbasis mit Hilfe der mathematischen Logik Lösungen des Problems generieren. Man hat für PROLOG geworben mit dem Schlagwort von der „eingebauten Intelligenz“. Es gilt in PROLOG (leider nur im Prinzip):

Es ist nur nötig zu sagen, *was* das Problem ist, nicht aber *wie* man es löst.

2.3 LOGO: Beispiel für eine funktionale Sprache

2.3.1 Die Datenstruktur Liste

Die Datenstruktur **Liste** ist eine Datenstruktur, die als Grundlage zur Definition weiterer, sehr komplexer Datenstrukturen dienen kann. Die dazu benötigten Mechanismen sind sehr einfach und grundlegend und finden sich in vielen Programmiersprachen, entweder explizit (LISP, LOGO, PROLOG) oder durch Definition mit Hilfe von Zeigern (s. Kapitel 1.3). Beschränkt man sich nur auf die grundlegenden Funktionen zum Aufbau und zur Zerlegung von Listen, dann lassen sich die Konstruktionen leicht in jede Sprache übertragen. Dies ist ein Grund, weshalb Listenstrukturen bedeutsam sind.

Grundlegende Vorstellung:

Liste: 1.Element 2.Element 3.Element 4.Element ..... n.Element

Listenkopf **Restliste**

1. Zum **Aufbau von Listen** braucht man
   1.1 Die **leere Liste**, damit man überhaupt beginnen kann, meist zusammen mit einer Menge von Atomen, die keine Listen sind (Zahlen, Wörter).
   1.2 Eine **Konstruktionsfunktion**, die aus einem beliebigen Element E und einer Liste L eine Liste mit Listenkopf E und Restliste L macht
2. Zum Zerlegen von Listen braucht man
   2.1 Eine **Zugriffsfunktion**, die aus einer nichtleeren Liste L den **Listenkopf** liest.
   2.2 Eine **Zugriffsfunktion**, die aus einer nichtleeren Liste L die **Restliste** liest.

Notationen für diese grundlegenden Funktionen:

Sprache	leere Liste	Konstruktions-funktion	Zugriffsfunktion Listenkopf	Zugriffsfunktion Restliste
LOGO englisch	[ ]	FPUT :E :L	FIRST :L	BUTFIRST :L BF :L
LOGO deutsch	[ ]	MITERSTEM :E :L ME :E :L	ERSTES :L	OHNEERSTES :L OE :L
LISP, SCHEME	( ) NIL	(CONS E L)	(CAR L)	(CDR L)
PROLOG(*)	[ ]	[E | L]	[E | L]	[E | L]

(*) Dabei muß auf die grundlegenden Unterschiede der logischen Programmiersprache PROLOG zu allen übrigen Sprachfamilien hingewiesen werden.

Mit Hilfe des Datentyps Liste können alle komplexen Datentypen aufgebaut werden. Aus diesen wenigen Funktionen werden mit Hilfe von

- expliziten Termen
- Fallunterscheidungen
- Rekursionen

alle weiteren benötigten Funktionen aufgebaut.

2.3.2 Die Liste als abstrakter Datentyp

Für Listen sollen folgende Axiome gelten:
1. Listen sind nur solche Objekte, die aus der leeren Liste und den Atomen mit Hilfe der Konstruktionsfunktion aufgebaut werden.
2. FIRST und BUTFIRST sind für Atome und die leere Liste nicht definiert.
3. FIRST FPUT :E :L...= :E
4. BUTFIRST FPUT :E :L = :L

Werden Funktionen in einer Sprache implementiert, die diesen Axiomen genügen, dann kann alles, was nur auf den Grundfunktionen basiert, ohne Kenntnis der internen Details übertragen werden.

2.3.3 Einige Übungen zu Listen und Bäumen.

Im folgenden soll die Notation des englischen BERKELEY-LOGO (MSW-LOGO) benutzt werden. Dies kann kostenlos aus dem Internet bezogen werden, z.B. aus Verzeichnis http://www.ph-ludwigsburg.de/mathematik/personal/klaudt/logo/logo.htm . Dort findet man auch eine Einführung und Aufgaben mit Lösungen zu LOGO.

Listen:

Die unten beschriebenen Funktionen sollen definiert werden. Falls die Werte nicht sinnvoll definiert sind, sollen die Funktionen den Wert [ ] zurückgeben. Viele dieser Funktionen sind in LOGO schon vordefiniert.

ZWEITES :L	das zweite Element von :L
LETZTES :L	das letzte Element von :L
LÄNGE :L	die Länge von :L (Anzahl der Elemente)
NTES :N :L	das :N-te Element von :L
INSERT :N :L	die Liste, die man aus der Liste :L von **Zahlen** erhält, wenn die Zahl :N an der richtigen Stelle eingefügt wird. :N soll vor dem ersten Element von :L eingefügt werden, das größer oder gleich :N ist
SORT :L	wenn :L eine Liste von Zahlen ist, soll SORT :L die daraus entstehende aufsteigend sortierte Liste sein
ELEMENT :E :L	TRUE, wenn :E ein Element der Liste ist, FALSE sonst
MITLETZTEM :E :L	die Liste, die aus :L entsteht, wenn :E am Ende angehängt wird
INVERS :L	Die Liste :L rückwärts

Einfaches Beispiel: Sortieren einer Liste

Es werden zwei Funktionen INSERT und SORT definiert. Im Gegensatz zu PASCAL ist der Datentyp *Liste* in LOGO vordefiniert. Er ist der grundlegende Datentyp in den funktionalen Sprachen.

- Struktur einer Definition in LOGO:

```
TO  <name>  <Var1> <Var2> ...
.......
END
```

- Um einer Variablen X einen Wert zuzuweisen schreibt man MAKE "X <wert> . "X ist der Name der Variablen.
- Meint man den Wert einer Variablen X, dann schreibt man :X.
- Verwendet man einen Namen X ohne die Zeichen **:** oder **"** davor, dann meint man die Funktion X. Ist diese nicht definiert, dann meldet LOGO „I DO'NT KNOW HOW TO X" oder „Ich kann das Wort <X> nicht deuten".
- Listen werden mit eckigen Listenklammern [2 7 78] notiert.
- Funktionswerte einer Funktion werden mit OP ... zurückgegeben (**O**ut**P**ut).
- Die Standard-Listenfunktionen sind

FIRST :L	erstes Element der Liste L
BF :L	Restliste von L ohne erstes Element (engl. **B**ut**F**irst, dt. OE)
FPUT :X :L	Liste, die man aus L durch vorne anfügen von X erhält.
(LIST a b c d...)	Liste mit den Elementen a, b, c, d, ..., wobei a, b, c, d, ... beliebige Terme sein dürfen. Im Gegensatz zu [a b c d] werden die Terme zuerst ausgewertet.

Die folgenden Funktionen erledigen das Sortieren einer Liste:

```
TO INSERT :X :L
IF :L=[]    [OP (LIST :X)]
IF :X < FIRST :L    [OP FPUT :X :L]
OP FPUT FIRST :L INSERT :X BF :L
end

TO SORT :L
IF :L=[] [ OP [] ]
OP INSERT FIRST :L SORT BF :L
END
```

Ein Bildschirmausdruck mit MSW-LOGO[1], der die Anwendung dieser Funktionen zeigt:

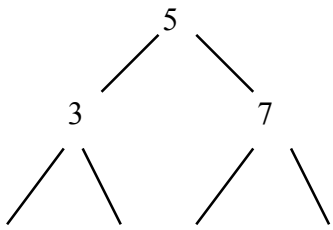
2.3.4 Bäume

Definition Binärbaum:
1. Die leere Liste ist ein Binärbaum
2. Sind B1 und B2 Binärbäume und ist :E ein beliebiges Element, dann ist auch die Liste
   [E B1 B2] ein Binärbaum. E soll Wert des Knotens [E B1 B2] heißen, B1 der linke Teilbaum und B2 der rechte Teilbaum.

Beispiel:

[5 [3 [ ] [ ]] [7 [ ] [ ]]] ist ein Binärbaum.
Der Wert des Knotens ist 5, der linke Teilbaum [3 [ ] [ ]], der rechte Teilbaum [7 [ ] [ ]].
Der linke Teilbaum wiederum hat den Wert 3 und die beiden Teilbäume [ ] und [ ], analog der rechte Teilbaum.

Graphische Idee dahinter:



Aufgabe:
Die unten beschriebenen Funktionen für Binärbäume sollen definiert werden. Falls die Werte nicht sinnvoll definiert sind, sollen die Funktionen den Wert [ ] zurückgeben

1. KNOTENWERT :B	Wert des Knotens :B
2. LIBA :B	Linker Teilbaum von :B
3. REBA :B	Rechter Teilbaum von :B
4. EINFBAUM :N :B	der Baum, die man aus dem Baum :B von **Zahlen** erhält, wenn die Zahl :N an der richtigen Stelle im Baum eingefügt wird, d.h. • ist der Baum :B leer, dann soll der Baum [:N [ ] [ ]] zurückgegeben werden.

[1] MSW-LOGO ist eine frei verfügbare LOGO-Version für Windows.

	• sonst :N soll in den linken Teilbaum von :B eingefügt werden, wenn :N kleiner oder gleich :N ist, andernfalls in den rechten. Dies soll auch für alle Teilbäume gelten. Hat man den leeren Teilbaum erreicht, so ist oben schon erklärt, was einfügen bedeutet.
5. INORDER :B	Gibt die Werte der Knoten des Baumes :B folgendermaßen auf dem Bildschirm aus: • ist der Baum :B leer, dann soll nichts ausgegeben werden • sonst soll der linke Teilbaum ausgegeben werden, dann der Wert des Wurzelknotens, dann der rechte Teilbaum Im Beispiel oben müsste ausgegeben werden 3, 5, 7
6. ZUFALLSBAUM :N	Zufälliger Baum mit :N Knoten, der "geordnet" ist

Listen und Bäume in MSW-LOGO:

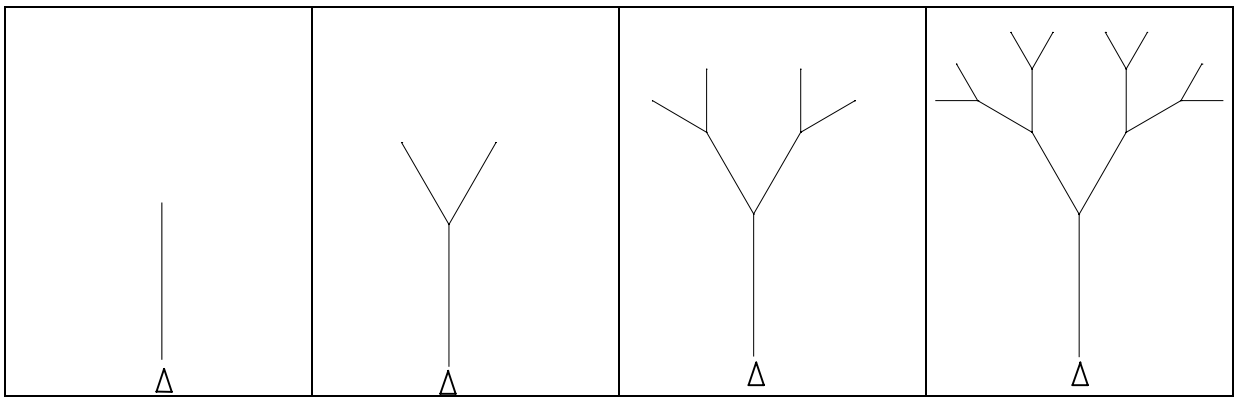
LOGO-Datei Listen.lg

```
; Listen  und Bäume ------------------------------------------------------------

; Listen -----------------------------------------------------------------------

TO LISTAUS  :L
        IF :L = []  [STOP]
        TYPE FIRST :L
        LISTAUS BF :L
ENDE

TO SORTL :L
 IFELSE   :L = []    [ OP :L ]    [ OP INSERT FIRST :L  SORTL BF :L]
ENDE


TO INSERT :E :L
;  Fügt in die geordnete Liste :L die Zahl :E an der richtigen Stelle ein
 IF :L=[]   [ OP FPUT :E [] ]
 IFELSE  NOT ((FIRST :L) < :E )  [ OP FPUT :E :L ]     [ OP FPUT FIRST :L INSERT :E BF :L ]
ENDE

; Binärbäume -------------------------------------------------------------------
; Struktur:  [ Knotenwert LinkerTeilbaum RechterTeilbaum ]  ----------------------

TO KNOTENWERT :B
;  Wert des Wurzelknotens des Baumes :B
 OP FIRST :B
ENDE

TO LIBA :B
;  Linker Teilbaum von :B
 OP FIRST BF :B
ENDE

TO REBA :B
;Rechter Teilbaum von :B
 OP FIRST BF BF :B
ENDE


TO EINFBAUM :N :B
; Fügt in den geordneten Baum :B die Zahl :N an der richtigen Stelle ein
 IF :B = []    [OP (LIST :N [] []) ]
 IFELSE  (:N < KNOTENWERT :B) ~
        [ OP (LIST (KNOTENWERT :B) (EINFBAUM :N LIBA :B) (REBA :B)) ]  ~
        [ OP (LIST (KNOTENWERT :B) (LIBA :B) (EINFBAUM :N REBA :B)) ]
ENDE
```

```
TO INORDER :B
; Aufruf immer gefolgt von PRINT '', sonst keine Ausgabe!
; Gibt einen Baum in der Art      Linker Teilbaum - Wert - rechter Teilbaum     aus
  IF :B = []  [STOP]
  INORDER LIBA :B
 TYPE KNOTENWERT :B TYPE Zeichen 32
  INORDER REBA :B
ENDE

TO ZUFALLSBAUM :N
  IFELSE  :N=0      [  OP []   ]  ~
                    [   OP EINFBAUM RANDOM 100 ZUFALLSBAUM :N-1  ]
ENDE

TO BAUMAUS0 :B :wi :L :T
  ;Grafische Ausgabe eines Baumes
  ;mit variablem Winkel wi, Länge :L , Pausezeit :T
  IF :B=[] [ STOP ]
  LEFT 90 LABEL  KNOTENWERT :B  RT 90
  WAIT :T*60
  IF NOT EQUALP LIBA :B []  ~
           [ ~
             RIGHT :wi  FD :L LEFT :wi  ~
             BAUMAUS0 LIBA :B  :wi*0.7  :L*0.8 :T ~
             RIGHT :wi  BACK :L LEFT :wi ~
           ]
  IF NOT EQUALP REBA :B []  ~
           [ ~
             LEFT :wi  FD :L  RIGHT  :wi ~
             BAUMAUS0 REBA :B :wi*0.7 :L*0.8 :T ~
             LEFT :wi  BACK :L RIGHT  :wi ~
           ]
ENDE

TO BAUMAUS :B
  obenhin
  BAUMAUS0 :B 70 100 1
ENDE

TO obenhin
  PENUP  SETXY 0 150 SETHEADING 180 PENDOWN
ENDE


; Daten ----------------------------------------------------------------------------------------
; Zwei Listen l, m :
Setze "l [5 [2 [] [4 [] []]] [8 [] [22 [] []]]]
Setze "m [5 [2 [-20 [] [-10 [] []]] []] [100 [50 [] []] [122 [] []]]]

; Ein geordneter Binärbaum  von Zahlen b :
Setze "b [79 [56 [18 [16 [15 [] []] []] [42 [36 [] []] [42 [] []]]] [67 [60 [] []] []]] [100 [] [125 [110 [] [117 [] []]][]]]]
```

2.3.5 Rekursive Graphiken (mit Hilfe der Turtle-Graphik)

Binärbaum

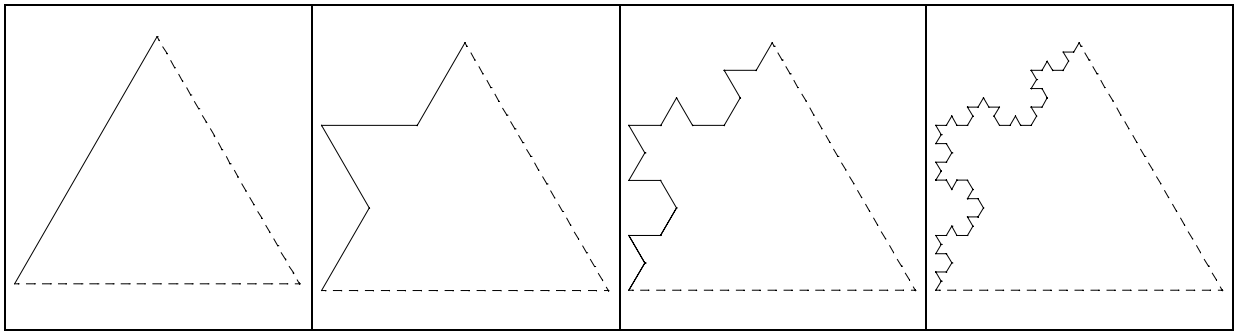


Baum 0 Baum 1 Baum 2 Baum 3

∆ : Position der Turtle vor und nach dem Zeichnen des Baumes.

Definieren Sie rekursiv eine Funktion **BAUM :L :N** , die den Baum N mit einer Stammlänge von L zeichnet.
Variieren Sie den Verkürzungsfaktor und die Verzweigungswinkel. Versuchen Sie auch, mit Hilfe des Zufallsgenerators die Winkel und die Verkürzungsfaktoren innerhalb gewisser Grenzen beim Zeichnen eines Baumes zu variieren.
Zufallsgenerator: random 10 ist eine Funktion, die bei jedem Aufruf eine neue, zufällige ganze Zahl zwischen 0 (eingeschlossen) und 10 (ausgeschlossen) zurückgibt (genauso natürlich random 30).

Schneeflockenkurve



Seite 0 Seite 1 Seite 2 Seite 3

Definieren Sie rekursiv eine Funktion **seite :L :N** , die die Seite N einer Schneeflocke mit einer Gesamtlänge von L zeichnet. Benutzen Sie diese Funktion, um die gesamte Schneeflocke zu zeichnen.
Hinweis zur Rekursion: Wie kann man den N-ten Schritt beschreiben, wenn man schon weiß, wie der (N-1)-te Schritt geht?

3 Objektorientierte Programmierung (OOP)

Ziele:

- **Andere** (natürlichere und einfachere ?) **Sichtweise der Welt und ihre Abbildung im Rechner.**
  Die Welt besteht aus Objekten, zwischen denen gewisse Beziehungen bestehen, die miteinander kommunizieren und auf Botschaften aktiv reagieren. → Kommunikationsorientierte Sprache.
- **Herstellen großer Programme ohne feste Ablaufstruktur.**
  Der Ablauf eines Programms folgt nicht einem festen Plan (z.B. Programmablaufplan), sondern wird durch "Ereignisse" gesteuert. Ereignisse werden dabei von außen durch Geräte oder Programmbenutzer oder programmintern durch Objekte ausgelöst.
- **Wiederverwendbarkeit von Programmcode.**
  Programmcode soll ohne Neucompilierung und ohne Kenntnis des Quellcodes wiederverwendet werden können. (z.B. Programmieren unter Windows). Einfache Erweiterbarkeit von Programmen. Dieser letzte Aspekt fehlt bei vielen Systemen, die Objekte benutzen (z.B. VBA, Toolbook, Mediator)

Prinzipien:

- **Objekte als Verbindung von Daten und Funktionen** (Eigenschaften und Methoden)
- **Klassen als Abstraktionen von Objekten** (Typen von Objekten)
- **Kapselung** (Geheimnisprinzip)
- **Vererbung**
- **Polymorphie**
- **Ereignissteuerung** statt Ablaufsteuerung (Objekte reagieren auf Ereignisse und agieren selbst)
- **Späte Bindung** (virtuelle Methoden)

3.1 1.Musterbeispiel: Ein Fenstersystem (z.B. Windows)

Fenster sind "Objekte".
Objekte sind Zusammenfassungen von Daten **und** Prozeduren.
Ein Fenster kann ein Eingabefenster, Meldungsfenster, Fenster mit Laufbalken... sein.
Was braucht jedes Fenster?

Daten: Ort, Größe, Rahmenart.....
+ **weitere spezielle Daten.**
Prozeduren: sich_zeigen, sich_verstecken,
sich_vergrößern, sich_verschieben
+ **weitere spezielle Prozeduren.**

Umsetzung Prozedur-orientiert (wie in BORLAND-PASCAL mit Objekten)**:**

Fenster sind Records, enthalten also nur Daten.
Prozeduren dafür:
- zeige_Fenster(F)
- verstecke_Fenster(F)
- verschiebe_Fenster(F,Ort)
- starte_Fenster(F)

Problem:

Diese Prozeduren müssen darauf Rücksicht nehmen, von welcher Art das Fenster ist. Das bedeutet:
- Fallunterscheidungen innerhalb der Prozeduren sind nötig.
- wird eine neue Fensterart (neuer Typ) hinzugefügt, dann müssen die Prozeduren **verändert und neu übersetzt** werden. Der Fensterverwaltungsteil des Programms kann also nicht in fertig compilierter Form (z.B. als Unit) für später zu entwickelnde Programme übernommen werden.

- man kann in diese Prozeduren wegen der strengen Typisierung von TP nicht ohne weiteres Fenster von verschiedenen Typen einsetzen.

Umsetzung Objekt-orientiert:

Fenster sind Objekte, d.h. Zusammenfassungen von Daten **und** Prozeduren (Methoden). Es gibt mehrere Klassen von Fenstern (entspricht den verschiedenen Typen).

Daten: Ort, Größe, Rahmenart.....
+ **weitere spezielle Daten**.
Methoden: sich_zeigen, sich_verstecken,
sich_vergrößern, sich_verschieben
+ **weitere spezielle Methoden**.

Jedes Fenster-Objekt F kennt jetzt seine eigenen Methoden:

- F.zeigeDich
- F.versteckeDich
- F.verschiebeDich(Ort)
- F.starteDich

Der Fensterverwaltungsteil eines Programms kann jetzt unter Verwendung dieser Methoden entworfen und endgültig übersetzt werden. Neu hinzukommende Fensterobjekte werden zusammen mit den allen Fenstern gemeinsamen Daten und Methoden definiert (Vererbung). Dabei nehmen diese Methoden auf die speziellen Eigenarten des neuen Fenstertyps Rücksicht. Soll z.B. ein neues Fenster sich immer mit einer speziellen Meldung zeigen, dann ist dies sicher bei der Methode 'zeigeDich' zu berücksichtigen. Benutzt die Methode 'verschiebeDich' nur die Methoden 'zeigeDich' und 'versteckeDich', dann muß sie für die neue Fensterart auch nicht mehr neu definiert werden, sondern erbt diese Methode vom allgemeinen Fenstertyp (Vorgängertyp).

3.2 2.Musterbeispiel: Ein Graphiksystem.

Es sollen geometrische Figuren auf einem Graphikbildschirm dargestellt, verschoben, gezeigt und versteckt werden. Geometrische Figuren sollen dabei Punkte, Strecken, Dreiecke, Vierecke und Kreise sein. Später sollen eventuell noch weitere Figuren hinzukommen.

Umsetzung Prozedur-orientiert (in TURBO-PASCAL)**:**

Geometrische Figuren sind Records, enthalten also nur Daten. Ein Punkt z.B. enthält seine Ortskoordinaten seine Farbe und seine Sichtbarkeit, eine Strecke die Ortskoordinaten ihres Anfangspunktes, ihre Farbe, ihre Sichtbarkeit, ihre Linienstärke und die **relativen** Koordinaten ihres Endpunktes bezüglich des Anfangspunktes. Ein Dreieck unterscheidet sich von einer Strecke dadurch, dass noch die relativen Koordinaten eines dritten Punktes hinzukommen. Ein Kreis enthält die Koordinaten seines Mittelpunktes, seine Farbe, seinen Radius und seine Linienstärke

Prozeduren dafür:
- zeige_Figur(F)
- verstecke_Figur(F)
- verschiebe_Figur(F,Ort)

Die Probleme sind dieselben wie bei den Fenstern im ersten Beispiel. Jede Figur muß neu definiert werden, alle Prozeduren müssen unterscheiden, welcher Figurtyp behandelt wird und in TP können wegen der strengen Typisierung ohnehin nicht ohne weiteres solche Prozeduren für Daten unterschiedlichen Typs definiert werden (obwohl man sich mit Listenstrukturen behelfen könnte)

Umsetzung Objekt-orientiert (in TURBO-PASCAL)**:**

Das Grundobjekt ist der Punkt, andere Objekte **erben** Eigenschaften von diesem Objekt.

Daten: Koordinaten des Anfangspunktes, Farbe, Sichtbarkeit
+ **weitere spezielle Daten**.

Methoden: sich_zeigen, sich_verstecken,
sich_verschieben
+ **weitere <u>spezielle</u> Methoden**.

Jedes Figur-Objekt F kennt jetzt seine eigenen Methoden:

- F.zeigeDich
- F.versteckeDich
- F.verschiebeDich(Ort)

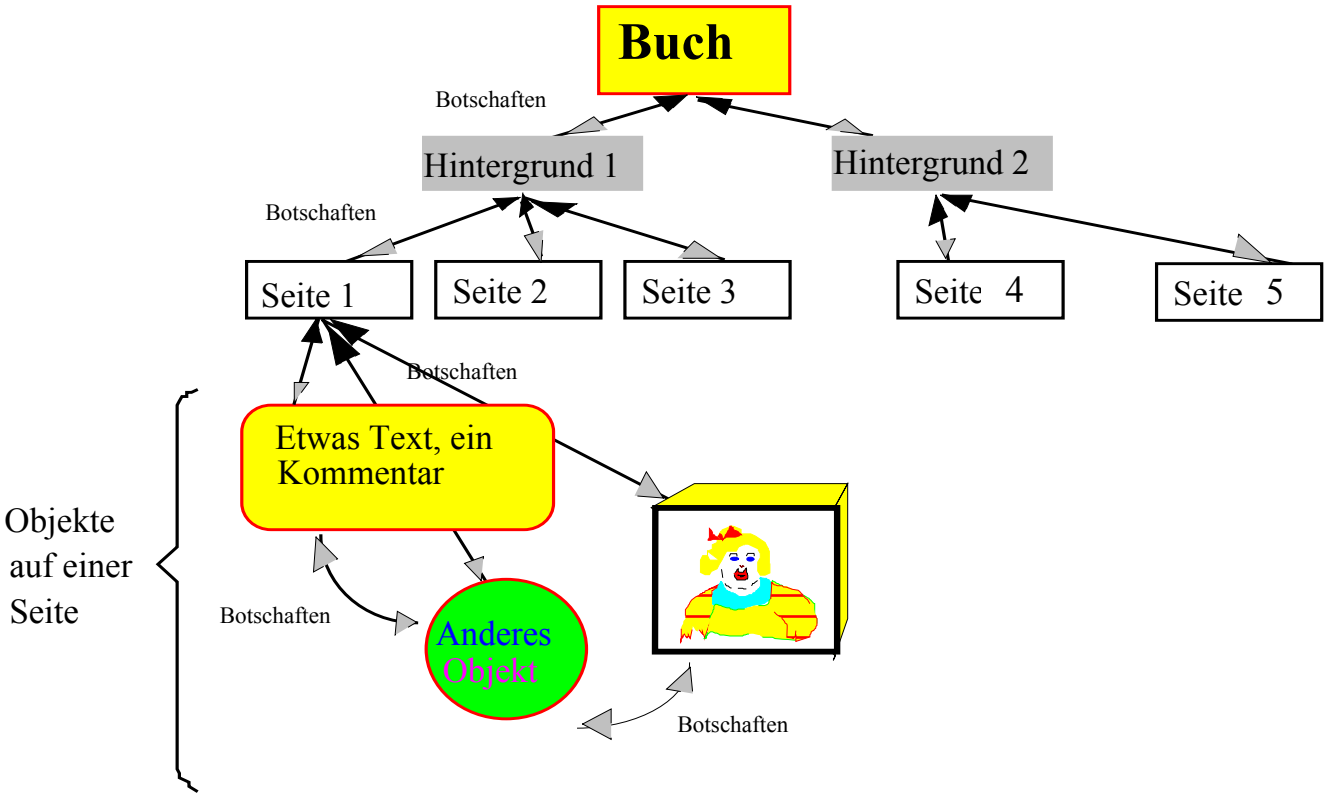
Konkret können die Definitionen in TP folgendermaßen lauten:

```
type
  TPunkt=class
      x, y , farbe: integer;
      visible: boolean;
      Constructor Init(x0,y0:integer);
      procedure zeigeDich;virtual;                    {Späte Bindung erzwungen}
      procedure versteckeDich;virtual;
      procedure verschiebeDich(xneu,yneu:integer);
      procedure setze_x(xneu:integer);          {Kapselung}
      procedure setze_y(yneu:integer);
      procedure setze_farbe(fneu:integer);
      function hole_x : integer;
      function hole_y : integer;
      function hole_farbe : integer;
  end;

  TStrecke=class(TPunkt)                          {Vererbung vom Vorgängertyp}
      x2,y2:integer;
      linienart:integer;
      Constructor Init(x0,y0,x20,y20:integer);
      procedure zeigeDich;virtual;                    {Späte Bindung erzwungen}
      procedure versteckeDich;virtual;
      procedure setze_x2(x2neu:integer);                    {Kapselung}
      procedure setze_y2(y2neu:integer);
      procedure setze_linienart(linartneu:integer);
      function hole_x2 : integer;
      function hole_y2 : integer;
      function hole_linienart : integer;
  end;
.
.
.
  Constructor TPunkt.Init(x01,y01:integer);
  begin
        x:=x01; y:=y01;
        farbe:=black;                             {Standardwerte}
        visible:=true;
  end

  procedure TPunkt.zeigeDich;
  begin  ..... end;

  procedure TPunkt.versteckeDich;
  begin  ..... end;
```

```
procedure TPunkt.verschiebeDich(xneu,yneu:integer);
begin
      versteckeDich;
      setze_x(xneu);
      setze_y(yneu);
      zeigeDich;
end;

Constructor TStrecke.Init(x0,y0,x20,y20:integer);
begin   ........  end;

procedure TStrecke.zeigeDich;
begin   ........  end;

procedure TStrecke.versteckeDich;
begin   ........  end;
```

3.3 Beispiel für ein Objektorientiertes System: Das Autorensystem Toolbook

Schema für den Aufbau des Systems:



Zweck des Programmiersystems: Erstellen von Lehr- und Lernprogrammen in Buchform.

Wesentliche Eigenschaften:

Objekt-Orientiert:

- Alle Komponenten sind Objekte mit Eigenschaften und Methoden. Sie werden graphisch erstellt (wie Graphikprogramm).
- Für jedes Objekt können die Eigenschaften einfach auf einer Karteikarte ausgewählt werden, und die Methoden werden als „Skripten“ in einer Skriptsprache (ähnlich Pascal) auf der Karteikarte aufgeschrieben.
- Objekte sind hierarchisch angeordnet. Die in der Hierarchie höher stehenden „besitzen“ die tiefer stehenden (s.Abbildung).

Ereignisgesteuert:

- Alle Objekte können auf „Meldungen“ mit den entsprechenden Methoden reagieren.
- Meldungen werden durch „Ereignisse“ ausgelöst oder von einem Objekt an ein anderes geschickt.
- Ereignisse sind z.B.: Mausbewegungen über ein Objekt, Mausklicks über ein Objekt, Drücken einer Taste....
- Ereignisse treten in die Objekthierarchie auf der untersten Stufe ein und werden bei Bedarf nach oben weitergereicht: Wenn z.B. auf einer Schaltfläche die linke Maustaste gedrückt wird, dann „sieht“ zuerst die Schaltfläche dieses Ereignis und kann mit seiner Methode *buttonDown* reagieren. Bei Bedarf kann sie dieses Ereignis an die zugehörige Seite, den Hintergrund und das Buch weiterleiten.

- Ein „Programm“ besteht in dieser Sichtweise aus einer Reihe von Objekten, die nur auf das Eintreten von Ereignissen warten, mit ihren Methoden darauf reagieren und gegebenenfalls Meldungen an andere Objekte schicken. Prozeduren sind dabei ganz in die Methoden verpackt.

Beispiele für Meldungen:

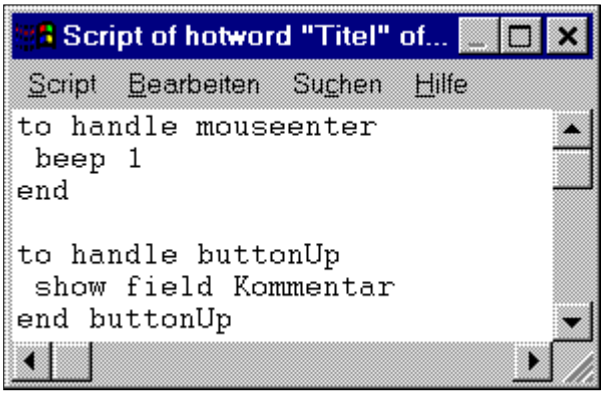
Meldungen für Betreten- und Verlassen-Ereignisse

enterSystem	enterBook	enterBackground	enterPage	enterButton	enterField
leaveSystem	leaveBook	leaveBackground	leavePage	leaveButton	leaveField

Meldungen für Maus- und Tastaturereignisse

mouseEnter	mouseLeave	buttonDown	buttonUp	buttonStillDown	buttonDoubleClick
rightButtonDoubleClick	rightButtonDown	rightButtonUp	keyDown	keyUp	keyChar

Ein typisches Skript eines „Schlüsselwortes“:



Eine typische Seite mit Schaltflächen und einem Schlüsselwort (hotword) „Titel“.

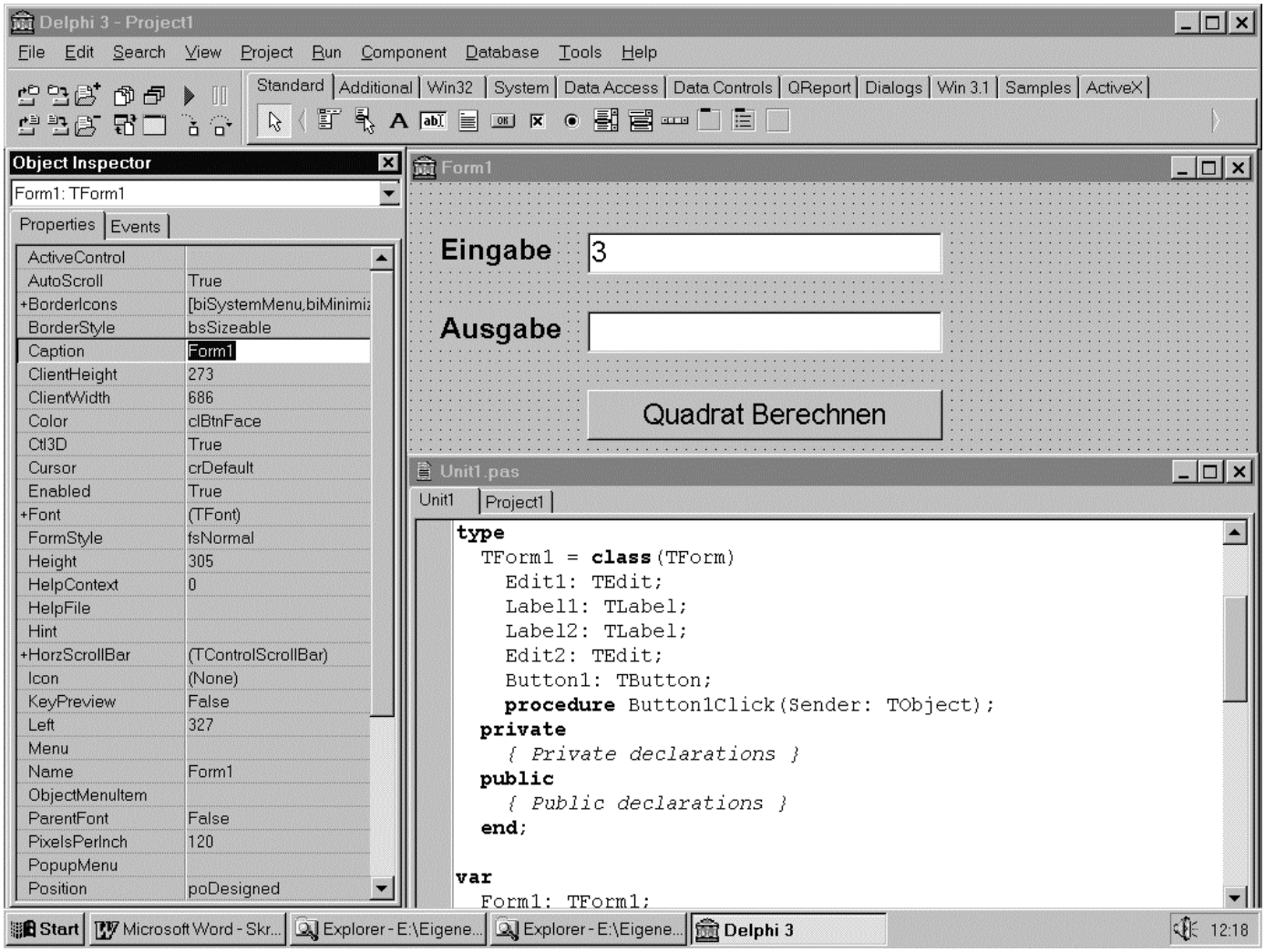
3.4 Beispiel für eine graphische, objektorientierte Programmierumgebung: Delphi.

Delphi (von der Firma Borland/Inprise) ist ein Programmier-System, das auf der objektorientierten Programmiersprache *Borland Pascal mit Objekten* aufbaut. Neben den Elementen der objektorientierten Programmierung sind folgende weiteren Eigenschaften zu nennen:

- Graphische Programmieroberfläche mit automatischer Code-Erzeugung. Diese Eigenschaft erleichtert die Programmierung, insbesondere der Benutzerschnittstelle, ungemein. Aus einem Vorrat von vorgefertigten Komponenten (Klassen von Objekten), die in einer Komponentenliste wie in einem Werkzeugkoffer zur Verfügung gestellt werden, zieht der Programmierer Kopien auf den Bildschirm. Dieses Vorgehen ist dem Zeichnen mit einem Zeichenprogramm ähnlich. Solche Komponenten sind Eingabefelder, Beschriftungsfelder (Labels), oder Schaltflächen (Buttons). Eigenschaften (Properties) und Ereignisse (Events) werden in einer Karteikarte für jedes Objekt eingetragen. Automatisch wird der zur Behandlung der Komponenten nötige Programmcode erstellt.
  Im unten aufgeführten Beispiel wurden zwei Eingabefelder, zwei Beschriftungsfelder und eine Schaltfläche in ein Formularobjekt eingefügt.
- Verwaltung der Module eines Programms und Debugging-Hilfen. Hierzu stellt das System ein umfangreiches Repertoire an Hilfsmitteln zur Verfügung. Der erzeugte Code kann unmittelbar getestet werden wobei vielfältige Unterstützung der Fehlersuche (Unterbrechungspunkte, schrittweise Ausführung, Überwachung von Variablen) geboten wird.
- Umfangreiche Komponenten zum Zugriff auf Datenbanken.

Das folgende Beispiel möge diese Art der Programmentwicklung verdeutlichen.

Der Bildschirm während des Programmentwicklung:



Der *Object Inspector* zeigt die Karteikarten für die Eigenschaften und Methoden des gerade ausgewählten Objekts, das Formular *Form1* zeigt die Entwurfsansicht des späteren Fensters der Anwendung, die *Unit1* zeigt den automatisch erzeugten Programmcode und am oberen Rand ist die Leiste mit den vorgefertigten Komponenten zu sehen.

Der Programmcode:

```
program Project1;
uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Hauptprogramm:

- Initialisiert ein Anwendungsprogramm-Objekt,
- erzeugt darin ein neues Formular-Objekt und
- sendet die Botschaft Run an das Anwendungsprogramm-Objekt („Starte dich“).

```
unit Unit1;

interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Edit2: TEdit;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
begin
  edit2.Text:=FloatToStr(strToFloat(edit1.text)* strToFloat(edit1.text));
end;

end.
```

Modul 1:

- Definiert im *Interface*-Teil eine neue Klasse *TForm1* von Formularen, die alle Eigenschaften von *TForm* erbt.
  Diese Formulare enthalten Edit-Felder, Labels und einen Button sowie eine Methode zur Behandlung des Ereignisses *Button1Click*.

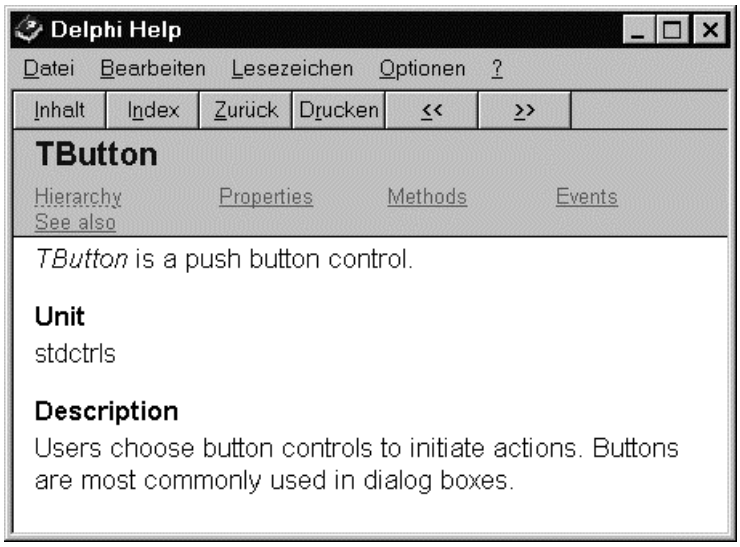
- Im *Implementierungs*-Teil werden die Details der Ausführung der Methode zur Behandlung des Ereignisses *Button1Click* beschrieben. Diese eine fett hervorgehobene Zeile ist die einzige von Hand erstellte Codezeile, alles andere ist vom System automatisch generiert worden

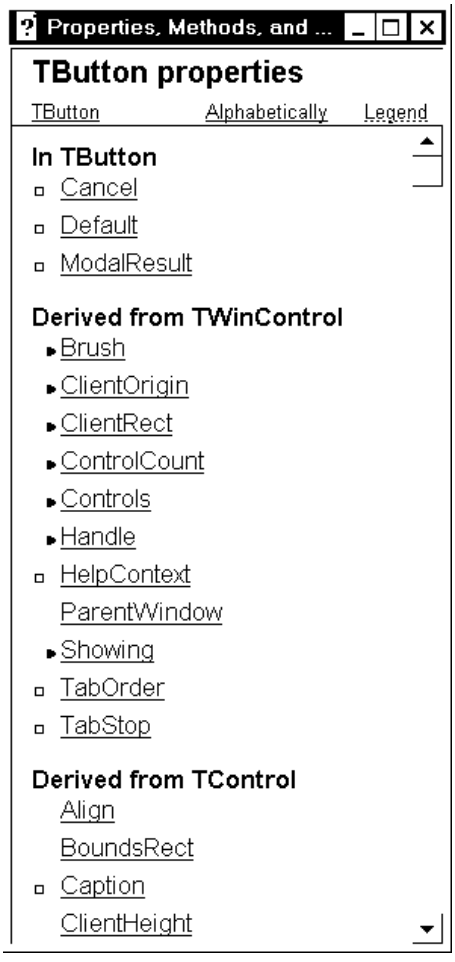
Die Klassenstruktur von Delphi

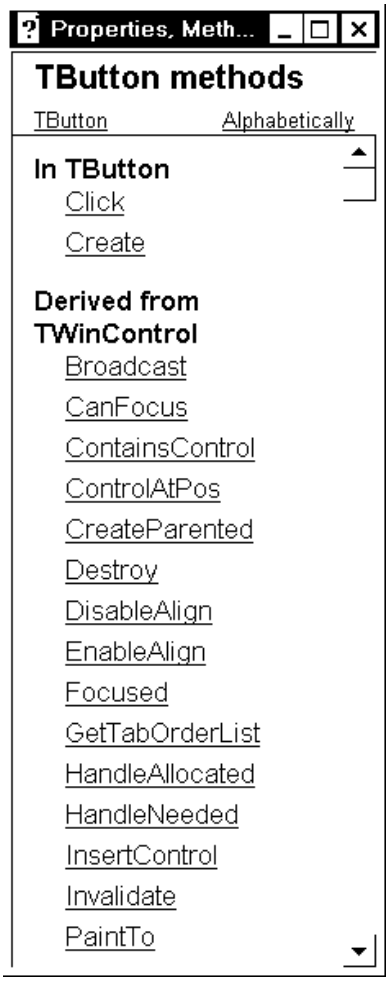
Das Hilfesystem von Delphi zeigt zu jeder Klasse von Objekten

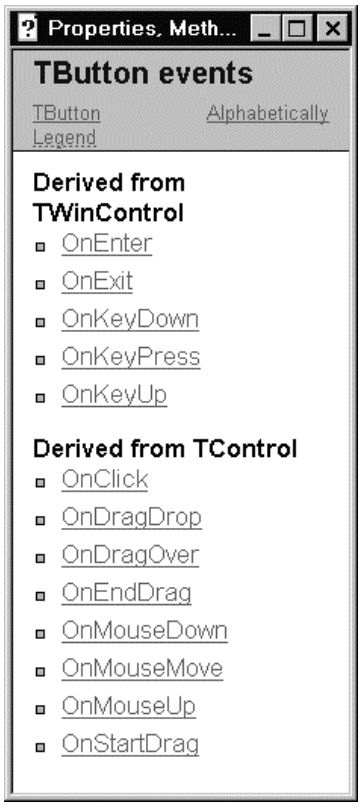
- die Stellung in der Klassen-Hierarchie
- die Eigenschaften (properties)
- die Methoden (methods)
- die Ereignisse (events), auf die diese Objekte reagieren können.

Dies ist hier am Beispiel der Schaltflächen-Klasse **TButton** gezeigt.









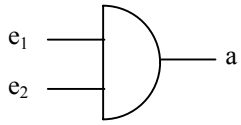
Die große Schwierigkeit des Erlernens der Programmiersprache Delphi besteht darin, die Struktur der Klassenhierarchie mit allen ihren vielen Objekten, Eigenschaften, Methoden und Ereignissen und deren Zusammenwirken zu erfassen.

4 Aufbau eines Computers: Von der Hardware zum Betriebssystem.

4.1 Schaltelemente, Schaltnetze

Schaltelemente (Gatter) sind Bauteile, die gewisse elektrische Eingangssignale e_1, e_2, e_3 ...in Ausgangssignale a_1, a_2, a_3, ... umwandeln. Die Ein- und Ausgangssignale werden mit 0 und 1 bezeichnet, wobei 0 und 1 z.B. für die Spannungswerte 0V und 5V stehen.
Die Grundbausteine der heutigen Schaltelemente sind Dioden, Transistoren und Widerstände, die mittlerweile soweit miniaturisiert sind, dass ca. 1 Million davon auf einem wenige cm^2großen Siliciumchip integriert werden können. Die Integrationsdichte nimmt dabei jedes Jahr dramatisch zu. Waren in der Anfangszeit der Computer einzelne Bauteile montiert worden, so wird heute der gesamte Entwurf einer Schaltung phototechnisch auf ein geeignetes Trägermaterial übertragen und chemisch weiter bearbeitet. Die Breite einer Leitung oder eines Transistors kann dabei im Bereich von einigen Nanometern (Größenordnung von Lichtwellenlängen) liegen. Dies erfordert hohe Anforderungen an die Genauigkeit und Reinheit während des Herstellungsprozesses.

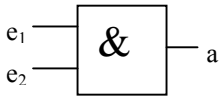
UND-Gatter (AND)



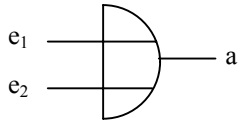
Verknüpfungstabelle:

e_1	e_2	$a = e_1 \wedge e_2$
0	0	0
0	1	0
1	0	0
1	1	1

Symbole nach der DIN-Norm

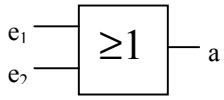


ODER-Gatter (OR)

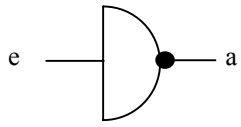


Verknüpfungstabelle:

e_1	e_2	$a = e_1 \vee e_2$
0	0	0
0	1	1
1	0	1
1	1	1

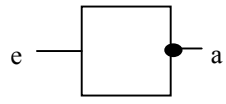


NICHT-Gatter (NOT)

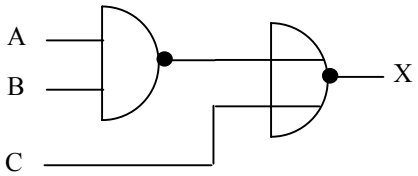


Verknüpfungstabelle:

e	$a = \neg e$
0	1
0	1
1	0
1	0



Solche Basis-Schaltelemente können zu Schaltnetzen zusammen geschaltet werden, für die man wiederum eine Verknüpfungstabelle aufstellen kann. Der Punkt an den Schaltelementen bedeutet dabei eine Negation.



$X = \neg (\neg (A \wedge B) \vee C)$

Füllen Sie die Verknüpfungstabelle aus.

Verknüpfungstabelle:

A	B	C	X
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Man kann zeigen, dass man alleine mit den Schaltelementen UND und NICHT alle möglichen Verknüpfungstabellen realisieren kann.
Weitere häufig benutzte Basis-Schaltelemente sind:

NOR	definiert durch	A NOR B	= NOT (A OR B)
NAND	definiert durch	A NAND B	= NOT (A AND B)
XOR	definiert durch	A XOR B	= (A OR B) AND NOT (A AND B) (e**X**clusives **OR**)

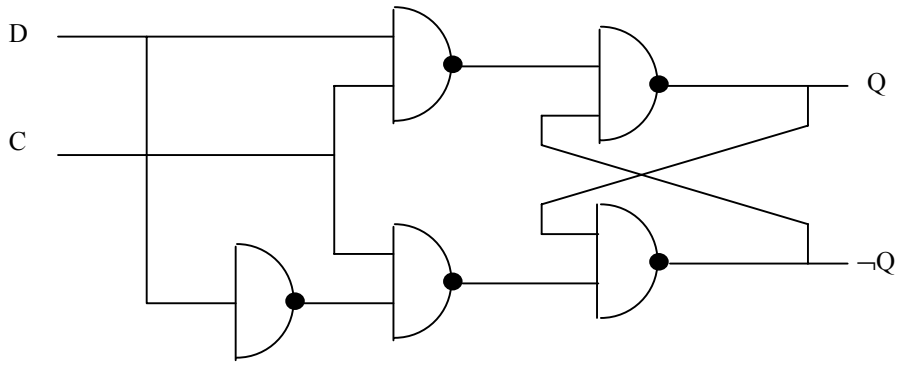
XOR entspricht der umgangssprachlichen Formulierung „*entweder ..... oder*“.

4.2 Speicherelemente, Schaltwerke

Mit Hilfe von einfachen Gattern lassen sich durch Rückkopplung Schaltelemente bauen, die zwei stabile Zustände annehmen können. Damit erhält man Schaltungen, die zwei Zustände 0 und 1 *speichern.* Speichern bedeutet hier, dass das Ausgangssignal nicht nur vom angelegten Eingangssignal abhängt, sondern auch vom *Zustand* der Schaltung, das heißt von der Vorgeschichte der Schaltung. Der Zustand eines Speicherelementes läßt sich durch ein Eingangssignal zu einem gewissen Zeitpunkt (Takt) ändern und bleibt solange stabil erhalten, bis er zu einem anderen Zeitpunkt (eventuell im nächsten Takt) durch ein neues Eingangssignal wieder verändert wird. Das Beispiel eines *D-Flip-Flops* soll dies erläutern.

Schaltungen, die sich aus Schaltnetzen und Speicherelementen zusammensetzen, bei denen also die Zeit eine Rolle spielt, nennt man *Schaltwerke*.

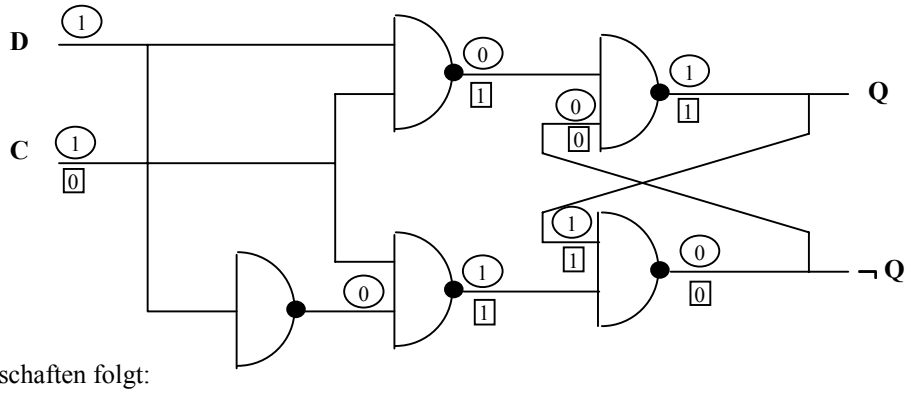
D-Flip-Flop



Eigenschaften

C=0 : Gleichgültig welches Signal an D anliegt, die Ausgänge Q und ¬Q behalten den Zustand bei, den sie auf Grund der Vorgeschichte hatten (entweder Q=1, ¬Q=0 oder Q=0, ¬Q=1).
In der folgenden Zeichnung wird dies nachgeprüft mit den Werten in Rechtecken [1], und zwar für Q=1. Der Fall Q=0 ist ähnlich.

C=1 : Wenn D den Wert 0 hat, nimmt Q den Wert 0 an, wenn D den Wert 1 hat, nimmt Q den Wert 1 an. Q nimmt also den Wert von D an.
In der folgenden Zeichnung wird dies nachgeprüft mit den Werten in Kreisen (1), und zwar für D=1. Der Fall D=0 ist ähnlich.



Aus den Eigenschaften folgt:

D kann als **D**ateneingang, C als Steuereingang (**C**ontrol) benutzt werden.
Wird während eines Taktes der Steuereingang auf den Wert 1 gesetzt, so kann über den Dateneingang D ein Wert 0 oder 1 gesetzt werden. Nimmt der Steuereingang dann wieder den Wert 0 an, so bleibt der gesetzte Wert erhalten (→ Speicherung).

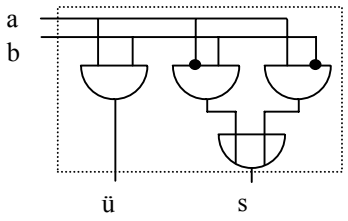
4.3 Arithmetische Operationen: Halbaddierer, Volladdierer

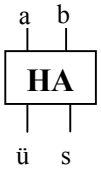
Eine wesentliche Aufgabe eines Prozessors ist die Ausführung arithmetischer Operationen. Hier soll beispielhaft ein Schaltnetz behandelt werden, mit dem zwei 8-Bit Binärzahlen addiert werden können (Akkumulator). Solche Addierschaltungen finden sich z.B. im Rechenwerk (ALU **A**rithmetic **L**ogic **U**nit) eines Mikroprozessors.

Addiert man zunächst nur zwei Binärziffern a und b, dann ergibt sich für die Summe s und den Übertrag ü die folgende Tabelle, die durch die daneben stehende Schaltung realisiert wird und mit einem neuen Symbol abgekürzt wird.

Verknüpfungstabelle:

a	b	s = a+b	ü
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

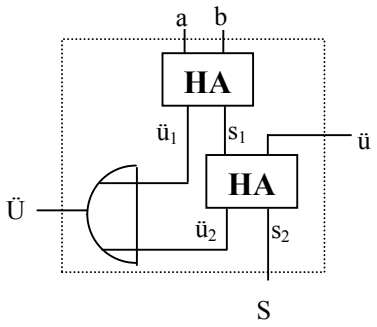


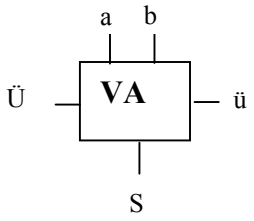


Sollen zwei Binärzahlen ziffernweise addiert werden, dann muß man ab der zweiten Ziffer den Übertrag der vorangehenden Addition berücksichtigen, also benötigt man ein Schaltglied, das drei Binärziffern a, b und ü addiert und deren Summe S sowie den Übertrag Ü als Ausgangssignal liefert. Dies geschieht durch Zusammenschalten von Halbaddierern. Die resultierende Schaltung heißt Volladdierer und wird durch ein neues Symbol VA repräsentiert.
Zum Verständnis der Schaltung ist zu beachten, dass die zwischendurch auftretenden Überträge $ü_1$ und $ü_2$ nie beide gleichzeitig 1 werden können, da $ü_1$ nur 1 wird, wenn gleichzeitig s_1 0 ist.

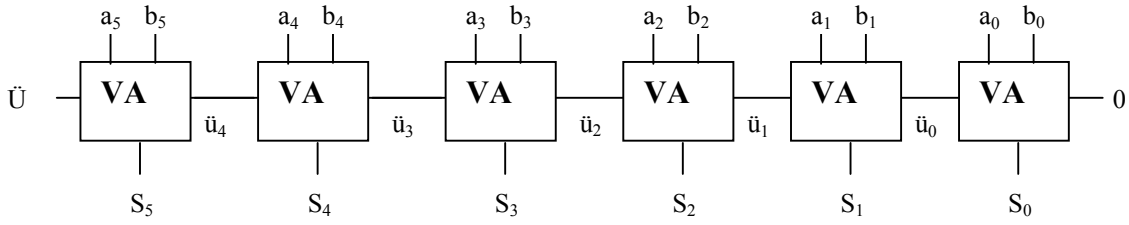
Verknüpfungstabelle Volladdierer:

a	b	ü	s	Ü
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1





Aus Volladdierern kann jetzt eine Addierschaltung für zwei 6-Bit Binärzahlen $a_5a_4a_3a_2a_1a_0$ und $b_5b_4b_3b_2b_1b_0$ aufgebaut werden.



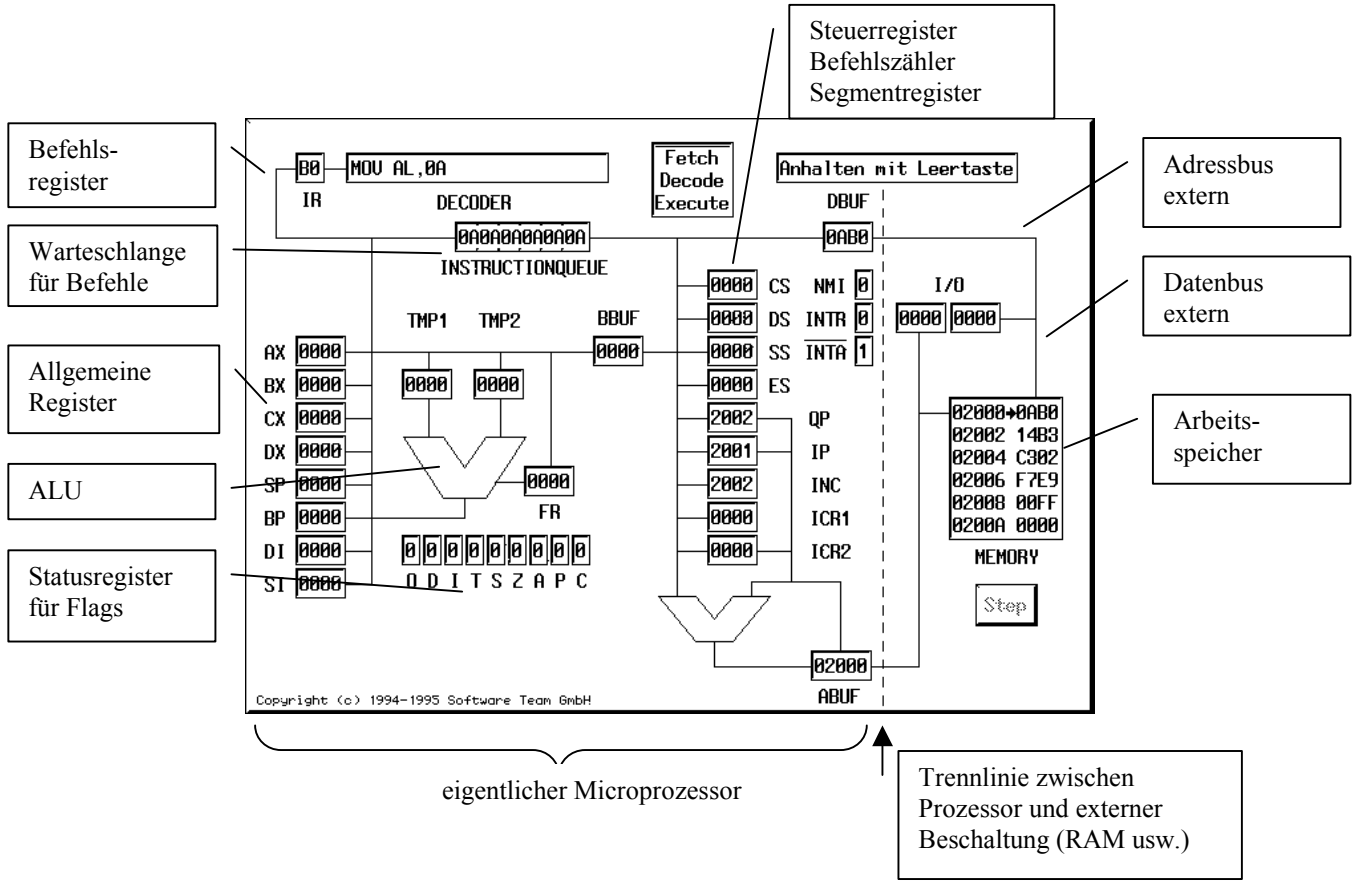
4.4 Prozessor und Arbeitsspeicher

Die zentrale Einheit, die alle Abläufe in einem Computer steuert ist ein **Prozessor**. Dieser enthält im wesentlichen einige Register (Speicher mit schnellem Zugriff) für Binärdaten und Adressen, ein Rechenwerk, in dem Binärzahlen verarbeitet werden können (ALU), eine Steuereinheit zur Steuerung aller Abläufe, sowie Leitungen, über die Daten, Adressen und Steuersignale transportiert werden können. Eine solche Leitung heißt **BUS** (**B**asic **U**tility **S**ystem). Die Anzahl der Bits, die in einem Register gespeichert werden können und somit auch parallel verarbeitet und transportiert werden, bestimmt (unter anderem) die Leistungsfähigkeit des Prozessors. Waren früher zunächst 8 Bit Prozessoren üblich (z.B. Z80 Prozessor; Motorola 6502 in den ersten Apple Home Computern) so werden heute 32 Bit und 64 Bit Prozessoren benutzt.

Neben dem Prozessor mit seinen wenigen internen Speicherstellen (Registern) benötigt man umfangreicheren **Arbeitsspeicher** außerhalb des Prozessors. Kann dieser Speicher sowohl beschrieben als auch gelesen werden, so spricht man von einem **RAM** Speicher (**R**andom **A**ccess **M**emory), wenn nur von diesem Speicher gelesen werden kann spricht man von einem **ROM** Speicher (**R**ead **O**nly **M**emory). Ein Computer besitzt beide Arten von Speichern. Im ROM Speicher hat der Computerhersteller grundlegende Programme abgelegt, die den Rechner erst befähigen, die Hardware zu nutzen (z.B. ROM-BIOS). Den RAM Speicher nutzt das Betriebssystem des Rechners für Daten, die von externen Datenträgern erst später geladen werden (Variable Betriebssystemteile, Anwendungsprogramme, Daten wie z.B. Bilder) und ein Programmierer kann diesen Speicher für seine Zwecke nutzen.
Der Prozessor wird heute häufig als CPU (**C**entral **P**rocessing **U**nit) bezeichnet. Manche Autoren bezeichnen aber als CPU die Einheit aus Prozessor *und* Arbeitsspeicher.

Beispiel für einen Microprozessor

Schematisierte Darstellung eines Microprozessors Intel 8086 (der Urahn aller Intel Microprozessoren) aus einem Simulationsprogramm:

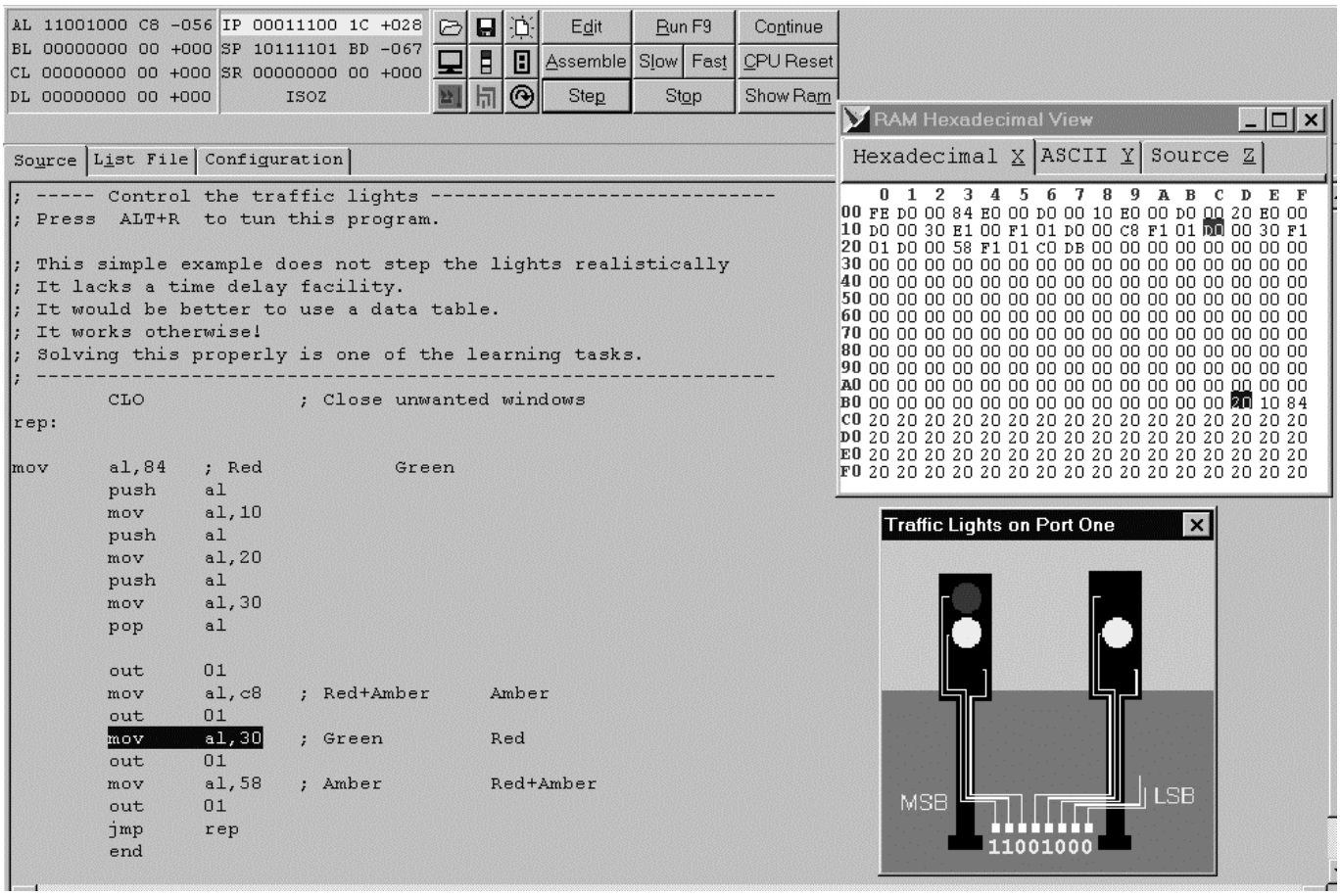


4.5 Assembler- und Maschinenprogrammierung mit dem Mikroprozessor Simulator SMS32

Das unten stehende Bild zeigt das Bild eines simulierten Microprozessors zusammen mit dem externen Speicher und einem Ausgang (Port) zur Steuerung einer Ampelanlage [2]. Mit diesem Simulationsprogramm kann man

- die Assemblerprogrammierung lernen
- die Steuerung von externen Geräten über Ports lernen
- den Ablauf von Maschinenprogrammen und die Veränderungen in den Registern und im Speicher verfolgen.

Der Simulator simuliert einen abgemagerten Intel-Prozessor (etwa Intel 8086)



Im Gegensatz zum zuvor gezeigten Schema eines Microprozessors werden hier viele Einzelheiten der Prozessorarchitektur weggelassen. So fehlen hier viele Register sowie sämtliche Busleitungen und Puffer, der Befehlsdecoder, die ALU und die Adressregister. Es werden nur die Register gezeigt, deren Funktion zum Verständnis der Maschinenprogrammierung unbedingt bekannt sein muß. Man sieht

- den Inhalt der allgemeinen Register AL bis DL in Binär-, Hex- und Dezimaldarstellung,
- den Inhalt des **Befehlszeigers** (**I**nstruction **P**ointer **IP**), des **Stapelzeigers** (**S**tack **P**ointer **SP**) und des **Statusregisters** (**S**tatus **R**egister **SR**)
- den Inhalt des Speichers zusammen mit einer Markierung des nächsten zur Ausführung gelangenden Maschinenbefehls und des Stapels (beginnend bei der Adresse Speicheradresse BF)
- eine Darstellung mehrerer 8-Bit Ports zum Anschluß externer Geräte (Ampelanlage, Thermostat, Aufzug usw.)
- das gerade ablaufende Assemblerprogramm mit Markierung des nächsten zur Ausführung gelangenden Maschinenbefehls in mnemonischer Form
- die Ausgabe von Zeichen auf einem simulierten Bildschirm, der auf die Speicheradressen C0 bis FF abgebildet wird, wobei ein Zeichen auf dem Bildschirm ausgegeben wird indem man an die entsprechende Speicherstelle den ASCII-Code des Zeichens schreibt.
- die Eingabe von Zeichen über die Tastatur über Port 00

[2] Dieses Simulationsprogramm ist innerhalb der PH Freiburg frei verfügbar und kann kopiert werden(PH-Lizenz). Bezugsquelle: http://www.samphire.demon.co.uk/

Beispiel für ein ganz einfaches Assembler-Programm zur Addition zweier Zahlen

Das Programm benutzt die Assembleranweisungen
clo, mov, add, sub, mul, div und end

Programmname 01first.asm

Um das Programm schrittweise ablaufen zu lassen, drückt man wiederholt den Knopf **Step,** solange bis das Programm endet. Die jeweils geänderten Registerinhalte werden gelb unterlegt angezeigt.

Was man über die Assemblersprache zum Verständnis des Programms wissen muß:

Kommentare

Jeder Text nach einem Semikolon ist nicht Bestandteil des Programms und wird ignoriert. Kommentare dienen der Erläuterung des Programmcodes und sollen gerade in der Assemblerprogrammierung ausgiebig verwandt werden, da sich der Code in keiner Weise selbst dokumentiert wie in höheren Programmiersprachen.

clo

Die *clo* Anweisung gibt es nur im Simulator. Sie schließt alle Fenster, die beim Programmlauf nicht benötigt werden. Dies erlaubt es, einfacher Demonstrationsprogramme zu schreiben.

mov

Die *mov* Anweisung ist eine Abkürzung für *Move*. In diesem Beispiel werden Zahlen in Register kopiert, in denen dann die arithmetischen Operationen vorgenommen werden können. Die Daten in der ursprünglichen Speicherzelle werden durch eine *mov* Anweisung *nicht* verändert. **mov AL,2**

Im allgemeinen können mit dem mov-Befehl Daten aus Registern in den Speicher (RAM) oder aus dem Speicher in Register kopiert werden. Die folgende kurze Übersicht zeigt alle Möglichkeiten der Adressierung. Grundsätzlich wird bei allen Befehlen mit zwei Adressen die Zieladresse als erste genannt. Daher bedeutet **mov AL,[15]** dass der Inhalt der Speicherzelle 15 in das Register AL kopiert wird.

```
mov AL,15     AL   = 15     Kopiere die Hex-Zahl 15 in AL
mov BL,[15]   BL   = [15]   Kopiere die Speicherzelle[15] in BL
mov [15],CL   [15] = CL     Kopiere CL in die Speicherzelle [15]
mov DL,[AL]   DL   = [AL]   Kopiere die Speicherzelle, deren Adresse in AL steht in DL
mov [CL],AL   [CL] = AL     Kopiere Al in die Speicherzelle, deren Adresse in CL steht
```

Beachten Sie die unterschiedlichen Bedeutungen der Adressen:

```
mov AL,15    unmittelbare Adressierung, 15 bedeutet eine Zahl
mov AL,[15]  direkte Adressierung, 15 bedeutet die Adresse einer Speicherzelle
mov AL,[CL]  indirekte Adressierung, CL ist das Register, in dem die Adresse einer
             Speicherstelle steht.
```

Register

Register sind Speicherzellen im Prozessor, wo 8 bit Binärzahlen gespeichert werden. Die CPU des Simulators hat vier allgemeine Register AL, BL, CL und DL. Diese Register können mit wenigen Ausnahmen für alle Zwecke benutzt werden.

Neuere Mikroprozessoren haben 16, 32 oder sogar 64 bit Register. Diese arbeiten prinzipiell in der selben Art, können aber mehr Daten gleichzeitig in einem Schritt bewegen und sind daher schneller. Breitere Register können auch größere Ganzzahlen speichern, was manche Programmierarbeit erleichtert. Die anderen Register SP, IP und SR werden später beschrieben. In AL steht L für Low, da in den Intel-Prozessoren das 16 Bit-Register A in die Teile AL und AH (High) aufgeteilt ist die getrennt angesprochen werden können.

Hexadezimalzahlen

In der Anweisung **mov AL,2** ist die 2 eine Hexadezimalzahl. Hexadezimalzahlen werden in der Maschinenprogrammierung durchgängig benutzt weil die Übersetzung von Hexadezimalzahlen in Binärzahlen sehr einfach ist

Arithmetische Anweisungen add, sub, mul, div

add wird benutzt um zwei Registerinhalte oder einen Registerinhalt und eine unmittelbar angegebene Zahl zu addieren.

Entsprechendes gilt für die Anweisungen *sub* (Subtraktion), *mul* (Multiplikation) und *div* (Division). Zu beachten ist hier, dass sich diese Operationen nur auf Ganzzahlen beziehen, was besonders bei der Division zunächst nicht einleuchten mag. Die arithmetischen Anweisungen verändern die Statusregister (Flags), so dass nach einer solchen Operation ein Sprungbefehl ausgeführt werden kann, dessen Ziel davon abhängt, ob das Ergebnis der Operation 0 war oder nicht (Zero-Flag abgefragt).

end

Die letzte Anweisung in einem Programm sollte *end* sein. Jeder Text nach *end* wird ignoriert.

Das Beispielprogramm „2 + 3“ (auf der Simulatordiskette Programm 01First.Asm)

Assemblercode:

```
; _____ Berechne 2 PLUS 3 ________________________________________

     CLO         ; Schließe nicht benötigte Fenster.

     MOV   AL,2  ; Kopiere eine 2 in das Register AL.
     MOV   BL,3  ; Kopiere eine 3 in das Register BL.
     ADD   AL,BL ; Addiere BL zu AL. Das Ergebnis geht nach AL, BL bleibt unverändert.
     END         ; Programmende

; _____ Ende des Programms _______________________________________
```

Dieses Assemblerprogramm, das statt der hexadezimal codierten Maschinenbefehle mnemonische Assemblerbefehle enthält, wird vom Assembler in hexadezimalen Maschinencode übersetzt (Assemblieren). Diese Programm wird im Gegensatz zum Quellprogramm üblicherweise als Objektprogramm bezeichnet.
Im Simulator wird das Maschinenprogramm unmittelbar im Speicher des Rechners abgelegt, wo es dann vom Mikroprozessor ausgeführt werden kann.
Unter dem Betriebssystem DOS (oder Windows) wird die Einbindung des Maschinencodes in das System durch Zuordnen zu absoluten Speicheradressen erst von einem Bindeprogramm (Binder, Linker) vorgenommen.

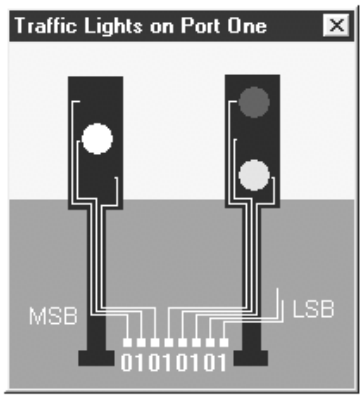
Nach dem Übersetzen in echten, hexadezimalen Maschinencode (Assemblieren) und Binden:

```
; LIST FILE : SUCCESS : No errors found.
;
; Dieses Listfile zeigt den Maschinencode, der zu den entsprechenden mnemonischen
; Assembleranweisungen erzeugt wurde.
; In eckigen Klammern erscheint die Speicheradresse, an der der Code abgelegt wurde.
; _____ WORK OUT 2 PLUS 3 ________________________________________
                  ;
     CLO          ; [00]  FE        ; Schließe nicht benötigte Fenster.
                  ;
     MOV   AL,2   ; [01]  D0 00 02  ; Kopiere eine 2 in das Register AL.
     MOV   BL,3   ; [04]  D0 01 03  ; Kopiere eine 3 in das Register AL.
     ADD   AL,BL  ; [07]  A0 00 01  ; Addiere BL zu AL, Ergebnis geht nach AL.
                  ;
     END          ; [0A]  00        ; Programmende
;
; SUCCESS : No errors found.
```

Aufgabe

Schreiben Sie Programme mit *sub, mul, div*, die Zahlen subtrahieren, multiplizieren und dividieren. Schreiben Sie auch ein Programm, das durch 0 dividiert und beobachten Sie den Effekt.

Das Assembler-Programm zur Steuerung der Ampelanlage



Das Programm benutzt *clo, mov, out, jmp end.*

Die Ampeln werden gesteuert indem man Daten zu einem input output (IO) port schickt. Sechs Lampen sind zu steuern: Rot, gelb und grün für jede Ampel. Dies erreicht man mit einem einzigen 8 Bit Zahl, wobei zwei Bits ungnutzt bleiben.
Setzt man die richtigen Bits auf 1, dann leuchten die richtigen Lampen auf.

Im Simulator hat das Programm den Namen 02tlight.asm

Neu bei diesem Programm sind **Labels** und der Sprungbefehl **JMP**.

Labels

Labels markieren Positionen die von den Sprungbefehlen benutzt werden um eine Einsprungstelle festzulegen. Labels müssen mit einem Buchstaben beginnen und einem Doppelpunkt enden.

JMP

Die Zeile JMP Start bewirkt einen Rücksprung des Programms und eine Wiederholung früherer Befehle.

OUT 01

Dieser Befehl kopiert den Inhalt des registers AL in den Output Port Nummer eins. Die Ampelanlage ist mit Port eins verbunden.. Eine 1 schaltet eine Lampe an, eine 0 schaltet sie aus.

```
; _____ Ampelanlage ____________________________________

      CLO         ; unnötige Fenster schließen.
Start:
                  ; Alle Lampen aus.
      MOV   AL,0  ; Kopiere 00000000 in das AL Register.
      OUT   01    ; Sende AL an Port Nummer eins (Ampelanlage).

                  ; Alle Lampen an.
      MOV   AL,FC ; Kopiere 11111100 in das AL Register.
      OUT   01    ; Sende AL an Port eins (Ampelanlage).

      JMP   Start ; Rücksprung nach start.

      END         ; Programmende.

; _____ Programmende ___________________________________
```

Aufgabe

Modifizieren Sie das Programm, so dass eine realistische Abfolge von Lampenschaltungen erfolgt.

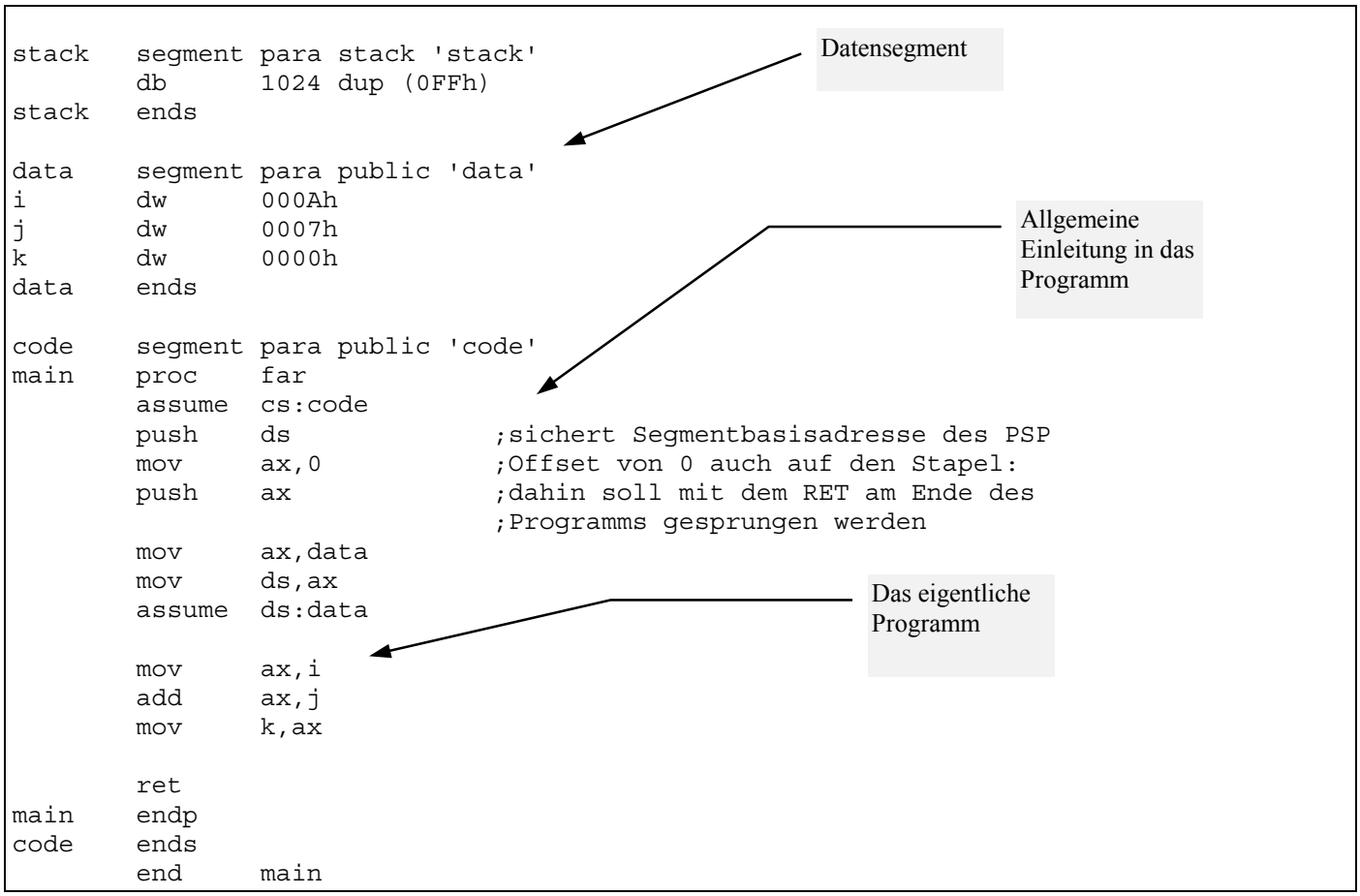
Eine detailliertere Beschreibung des Mikroprozessor-Simulators mit Aufgaben und Lösungen ist als Anhang angefügt.

4.6 Maschinensprache und Assembler für die Intel-Prozessoren 80x86

Die 80x86-Prozessoren sind wie der Mikroprozessor Simulator SMS32 aufgebaut. Dabei bestehen die folgenden wesentlichen Unterschiede:

- Es gibt mehr und breitere Register (Rechenregister sowie Register für spezielle Aufgaben).
- Der Speicher wird in Segmente aufgeteilt, deren Basisadresse in Registern CS, DS, ES, SS verwaltet wird.
- Es gibt eine Vielzahl von Schleifenbefehlen.
- zusätzliche Adressierungsmöglichkeiten (unmittelbar, direkt, registerbezogen, indirekt, indiziert-indirekt).
- Befehle zur Stringbearbeitung sind eingebaut.
- Es gibt mehr Interrupts (zur Unterbrechungsverarbeitung). Damit reagiert der Prozessor z.B. unmittelbar auf wichtige Ereignisse von außen. Dies geschieht durch Unterbrechung des laufenden Programms, Sicherung des aktuellen Prozessorzustandes, Sprung in ein spezielles Unterbrechungsprogramm zur Bearbeitung des Ereignisses und Rückkehr ins ursprüngliche Programm.
- Neuere Prozessoren unterstützen die fortgeschrittene Graphikausgabe (z.B. ab 1997 MMX Versionen von Pentiums), zusätzliche komplexe mathematische Operationen (die früher auf sogenannten mathematischen Coprozessoren ausgeführt wurden) sowie Multitasking (gleichzeitige Ausführung mehrerer Programme mit eigenen Speicherbereichen), ...

Die folgende Abbildung zeigt ein einfaches Assembler-Programm für einen 80x86-Prozessor, das die Zahlen i und j addiert und das Ergebnis als z speichert. Die Werte für i, j und z stehen dabei an gewissen Stellen im Speicher, und zwar im Datensegment an den Speicherstellen 0, 2 und 4. Der Rechner gibt diese Adressen in der Form DS:00, DS:02 und DS:04 an.




```
stack   segment para stack 'stack'
        db      1024 dup (0FFh)
stack   ends

data    segment para public 'data'
i       dw      000Ah
j       dw      0007h
k       dw      0000h
data    ends

code    segment para public 'code'
main    proc    far
        assume  cs:code
        push    ds              ;sichert Segmentbasisadresse des PSP
        mov     ax,0            ;Offset von 0 auch auf den Stapel:
        push    ax              ;dahin soll mit dem RET am Ende des
                                ;Programms gesprungen werden
        mov     ax,data
        mov     ds,ax
        assume  ds:data

        mov     ax,i
        add     ax,j
        mov     k,ax

        ret
main    endp
code    ends
        end     main
```

Das oben gezeigte Assembler-Programm wird vom Assembler in ein **Objekt-Programm** übersetzt, das zwar im wesentlichen aus Maschinenbefehlen besteht, aber noch nicht lauffähig ist, da es noch symbolische Adressen enthält. Erst das sogenannte **Link-Programm** (Linker, Binder) wandelt das Programm in ein ausführbares Programm (EXEcutable Program) um. Der Linker bindet auch mehrere getrennt übersetzte Objekt-Programme zu einem ausführbaren Programm zusammen.
Der Ablauf des ausführbaren Programms läßt sich (wie im Simulator) im **DEBUGGER** auf Maschinenebene verfolgen. Der Debugger zeigt die Zustände der Prozessorregister, den Speicherbereich mit dem Programm-Code in mnemonischer Form sowie den Speicher in hexadezimaler Form und läßt die schrittweise Ausführung und Überwachung der Maschinenbefehle zu. (Er bietet darüber hinaus viele weitere Möglichkeiten, ins Innere des Prozessors zu sehen).
In der folgenden Abbildung wird unser Programm im Debugger verfolgt.

```
-  File  Edit  View  Run  Breakpoints  Data  Options  Window  Help       READY
+-[_]-CPU 80486-------------------------------+-------1-[□][□]-+_________________
|  cs:0000 1E             push   ds           |  ax 000A   |c=0|_________________
|  cs:0001 B80000         mov    ax,0000      |  bx 0000   |z=0|_________________
|  cs:0004 50             push   ax           |  cx 0000   |s=0|_________________
|  cs:0005 B8B460         mov    ax,60B4      |  dx 0000   |o=0|_________________
|  cs:0008 8ED8           mov    ds,ax        |  si 0000   |p=0|_________________
|  cs:000A A10000         mov    ax,[0000]    |  di 0000   |a=0|_________________
|  cs:000D□03060200       add    ax,[0002]    |  bp 0000   |i=1|_________________
|  cs:0011 A30400         mov    [0004],ax    |  sp 13FC   |d=0|_________________
|  cs:0014 CB             retf                |  ds 60B4   |   |_________________
|  cs:0015 0000           add    [bx+si],al   |  es 5F64   |   |_________________
|  cs:0017 0000           add    [bx+si],al   |  ss 5F74   |   |_________________
|  cs:0019 0000           add    [bx+si],al   |  cs 60B9   |   |_________________
|  cs:001B 0000           add    [bx+si],al   |  ip 000D   |   |_________________
+-----------------------------------------+   |            |   |_________________
|  ds:0000 0A 00 07 00 00 00 00 00            |            |   |_________________
|  ds:0008 00 00 00 00 00 00 00 00            |  ---------------+_________________
|  ds:0010 0A 00 07 00 00 00 00 00            |  ss:13FE 5F64  |_________________
|  ds:0018 00 00 00 00 00 00 00 00            |  ss:13FC□0000  |_________________
_____________________________________________   ---------------+_________________
_____________________________________________________________________________
_____________________________________________________________________________
_____________________________________________________________________________
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

4.7 Prozessorarchitekturen und Prozessortypen

CISC Architektur (**C**omplex **I**nstruction **S**et **C**omputer)
Prozessor mit komplexen und leistungsfähigen Maschinenbefehlen, deren Ausführung mehrere Prozessortakte benötigt. Verschiedene Befehle benötigen je nach Komplexitätsgrad unterschiedlich viele Takte.

RISC Architektur (**R**educed **I**nstruction **S**et **C**omputer)
Komplexe Befehle kommen in Maschinenprogrammen relativ selten vor. Prozessor mit nur sehr einfachen Maschinenbefehlen, deren Ausführung nur einen Prozessortakt benötigt. Damit können Befehle im Fließbandverfahren bearbeitet werden. Komplexe Befehle kommen in Maschinenprogrammen relativ selten vor. In RISC Prozessoren werden daher die komplexen Befehle aus einfacheren zusammengesetzt, der Prozessor insgesamt und die Verarbeitungsschritte werden einfacher und somit schneller. Durch die Einsparung von sogenannten Mikroprogrammen für die komplexen Befehle gewinnt man Platz für eine größere Zahl von Registern mit sehr schnellem Zugriff.

Fließbandverarbeitung (Pipelining) von Maschinenbefehlen
Maschinenbefehle stehen in einer Schlange, bei jedem Prozessortakt wird ein Bearbeitungsschritt ausgeführt (Befehl holen, Befehl decodieren, Adresse der Operanden berechnen, Operanden holen, Befehl ausführen)..
Automatische Parallelisierung der Abarbeitung von Maschinenbefehlen im Prozessor, wenn das möglich ist (z.B. im Prozessor Intel Pentium II)

Cache Speicher
Kleiner Speicher mit schnellem Zugriff, in dem aktuell benötigte Daten und Befehle gespeichert werden. Teile des langsameren Arbeitsspeichers, die gerade benötigt werden, werden zusammen mit ihren Herkunftsadressen hier gespeichert um die Arbeitsgeschwindigkeit zu erhöhen. Zum Verwalten des Cache-Speichers wird Verwaltungsarbeit benötigt, die umso größer wird, je kleiner der Cache ist.

Primary Cache (First level Cache)
Kleiner schneller Cache Speicher, oft auf dem Prozessor Chip. Intel Pentium Prozessoren haben einen 16 KB primary cache auf dem Prozessor-Chip, je 8 KB für Daten und Befehle.

Secondary Cache (Second level Cache)
Größerer, langsamerer Cache-Speicher zwischen primary Cache und Hauptspeicher, meist über den externen Bus mit dem Prozessor verbunden.

Pentium I
Prozessor P6. 32 Bit-Prozessor. Interne RISC Architektur mit CISC-RISC Übersetzung, teilweise automatischer interner Parallelisierung von Maschinenbefehlen, Pipeline Verarbeitung, 16 KB Primary Cache und 256 oder 512 KB Secondary Cache auf dem Prozessorchip. 5.5 Millionen Transistoren auf einem Chip!! Taktraten bis 200 MHz.
Mehrere Konkurrenzfirmen von INTEL produzieren vergleichbare Prozessoren mit anderen Namen, die aber bezüglich der Software kompatibel sind und zum Teil bessere Leistungsdaten als die Intel Prozessoren aufweisen .

Pentium II
Nachfolger des Intel Pentium Pro, hat MMX eingebaut, Taktraten bis 333 MHz

Pentium III, Pentium IV
Nachfolger des Intel Pentium II, zur Zeit Taktraten bis 3 GHz. Billige Version des Pentium III „Celeron“, auch bis 1,3 GHz Takt.

Itanium
64 Bit Prozessoren von Intel. Zur Zeit Taktraten bis 1 GHz. Verschiedene Versionen verschiedener Leistungsstufen, „Merced“ als Testprozessor, z.B. zur Zeit aktuell „McKinley“ unter dem Namen Itanium 2. Einige Daten:
221 Millionen Transistoren auf ca. 5 cm^2, 3 MB Level-3-Cache auf dem Chip, 130 W Leistungsaufnahme. Itanium z.Zt. als Prozessor für Server vorgesehen.

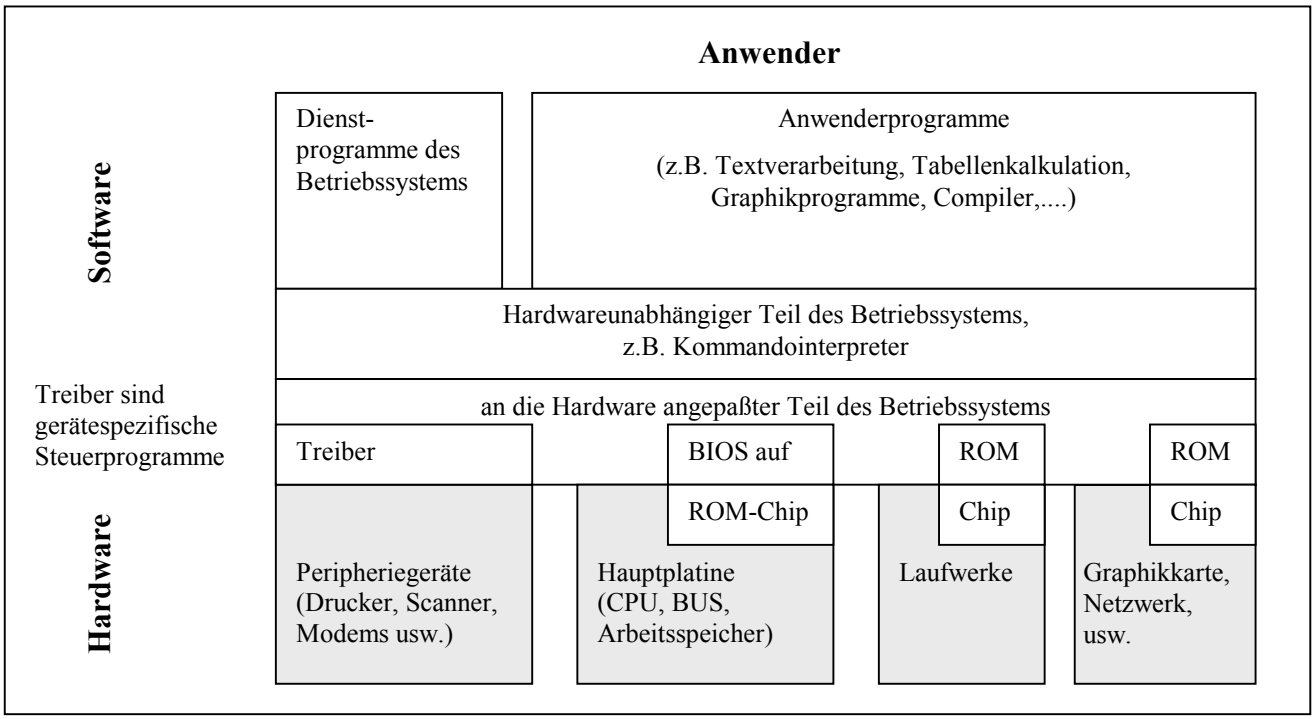
AMD (Advanced Micro Devices)
Konkurrent von Intel auf dem Prozessormarkt, weitgehend kompatible, billigere Prozessoren. Permanenter Wettlauf mit INTEL um die beste „Performance“. Einige Namen der Prozessoren: „K6“, „K7“, „Duron“, „Athlon“.

Parallelrechner (2 bis 64 000 Prozessoren parallel), verschiedene Architekturen
wenige Prozessoren parallel für Server (z.B. 1 bis 4 Intel Pentium Prozessoren)
viele Prozessoren parallel für Spezialaufgaben (Thinking Machines Rechner 64 000 Prozessoren, Cray-Rechner RZ Freiburg, RZ Stuttgart)

4.8 Betriebssystem

Aufgaben eines Betriebssystems

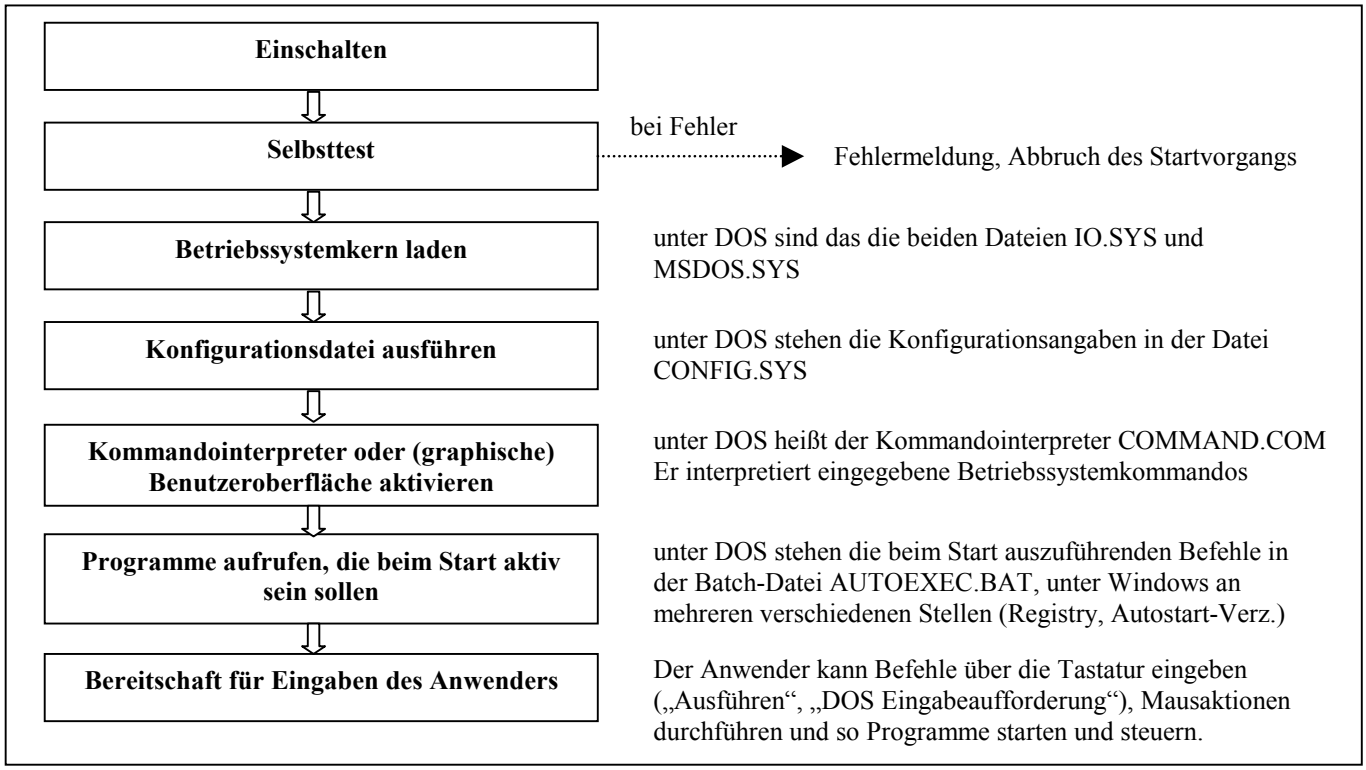
- Starten und Beenden des Rechnerbetriebs
- Organisation und Verwaltung des Arbeitsspeichers
- Steuerung der Hardwarekomponenten
- Organisation und Verwaltung der Speichermedien
- Organisation der Bildschirmanzeige
- Laden und Kontrollieren der Anwenderprogramme, Weitergabe von Benutzereingaben, Behandlung von Fehlern, Verwaltung von Benutzerrechten
- Verwaltung und Bedienung mehrerer Nutzer mit eigenen Zugriffsrechten und Nutzungsprofilen
- Bereitstellung von Dienstprogrammen für verschiedenste Zwecke: Datensicherung, Texteingabe, Telekommunikation, Spracheingabe, Zeichnen, Rechnen etc.
- Unterstützung der Vernetzung mehrerer Computer (Netzwerkbetriebssystem)



Start des Betriebs: Bootvorgang

Wenn ein PC eingeschaltet wird, erhält der Prozessor einen ersten Befehl, der ihn dazu veranlaßt, ein bestimmtes Programm auszuführen. Dieses Programm steht in einem ROM-Chip (*Read Only Memory*) und heißt BIOS (**B**asic **I**nput **O**utput **S**ystem) .

Das folgende Bild zeigt wichtige Schritte beim Starten eines Rechners. Meist erscheinen zu allen Aktionen Meldungen auf dem Bildschirm, die dem Anwender den Erfolg oder Mißerfolg der Startvorgänge anzeigen.



Selbsttest

Dieses Programm überprüft zunächst die Funktionsfähigkeit des Prozessors selbst. Wenn ein System diese Prüfung nicht besteht, bleibt dem Computer nur noch der Versuch, dieses unerfreuliche Ergebnis dem Anwender mitzuteilen. Einige Computer verwenden dazu verschiedene Folgen von Pieptönen, da der Monitor noch nicht ansprechbar ist. Andernfalls wird der Selbsttest fortgeführt, zunächst mit der Prüfung von Umfang und Funktionstüchtigkeit des Arbeitsspeichers. Nachdem die Funktionen der Hauptplatine des Computers erfolgreich getestet wurden, werden die vorhandenen Hardware-Erweiterungen (z. B. Laufwerke) überprüft bzw. dazu aufgefordert, einen Selbsttest durchzuführen.

Setup

Häufig werden Informationen über diese Erweiterungen und andere Einstellungen in einem batteriegepufferten RAM-Chip (CMOS) dauerhaft gespeichert. Daher kann der Computer vergleichen, ob die vorgefundenen Verhältnisse in seinem Inneren der gespeicherten Konfiguration entsprechen. Ist dies nicht der Fall, etwa wenn man eine Komponente des Rechners ausgetauscht hat, dann kann er den Anwender auffordern, ihm die notwendigen Informationen zu geben. Unter Umständen muß dies der Anwender auch ohne Aufforderung von Seiten des Rechners tun. In der Fachsprache bezeichnet man diese dem Computer vorgegebenen Grundeinstellungen häufig als **Setup.**

Password

Im batteriegepufferten RAM-Baustein befinden sich meist auch Vorgaben über frei definierbare Einstellungen des Rechners sowie ein **Password.** Wenn der Anwender dies so vorgibt, dann verlangt der Rechner zunächst die Eingabe dieses Passworts. Wird es falsch eingegeben, verweigert der Rechner weitere Dienste.

Betriebssystem laden

Der letzte Schritt, den das Programm auf dem ROM-Chip ausführt, ist das Laden eines bestimmten Teils des Betriebssystems von einem Datenträger (meist von der Festplatte) in den Arbeitsspeicher. Dieses Programm erhält nun die Kontrolle. Es organisiert den Arbeitsspeicher und den Bildschirm, kopiert weitere Teile des Betriebssystems von einem Datenträger in den Arbeitsspeicher und führt in Dateien gespeicherte Anweisungen zur Konfiguration des Betriebssystems aus. Der Ablauf vom Einschalten bis zur Bereitschaft des Betriebssystems wird als **Booten** bezeichnet.

Ende des Betriebs

Ein Computer kann nicht verhindern, dass man ihn einfach ausschaltet. Bei einem einfachen PC mit dem Betriebssystem *DOS* richtet man damit in der Regel keinen größeren Schaden an. Bei modernen Systemen werden jedoch häufig mehrere Programme im Hintergrund betrieben. Diese müssen erst kontrolliert abgebrochen werden, wenn man keinen Verlust von Daten in Kauf nehmen will. Zum anderen werden aktuelle Einstellungen in Dateien gespeichert, die man beim nächsten Starten wieder benötigt. Auch Plattenzugriffe erfolgen häufig gepuffert; Daten werden also nicht unbedingt sofort beim Anfordern eines Schreibvorgangs auf Platte gesichert, sondern im Hauptspeicher zwischengespeichert und dann in regelmäßigen Abständen durch das Betriebssystem auf die Festplatte geschrieben. Deshalb sehen moderne Systeme ein eigenes **Shutdown-Programm** vor das diese Aufgaben erledigt. Damit wird das System in einem ordentlichen und definierten Zustand beendet.

Multitasking

Können auf einem Computer mehrere Programme gleichzeitig ablaufen, dann spricht man von Multitasking. So können atwa ein Druckauftrag, ein Speichervorgang und eine Bildbearbeitung nebeneinander ausgeführt werden. Tatsächlich führt der Prozessor jedoch zu jedem Zeitpunkt nur einen Prozess aus, zwischen diesen Prozessen wird aber in so kurzen Abständen umgeschaltet, dass der Benutzer dies nicht wahrnimmt (Time Sharing Verfahren). Bei echten Multitasking Systemen (UNIX, Windows 95, Windows NT, OS/2) wird dabei jedem Prozess ein eigener Speicherbereich zugeordnet in dem dieser Prozess abläuft, und das Betriebssystem weist den Prozessen ihre „Zeitscheiben“ zu. Es behält dabei vollständig die Kontrolle und kann einzelne Prozesse getrennt beenden, wenn diese etwa „hängen“ bleiben oder zu lange laufen (Windows: Strg-Alt-Entf – Verfahren). In unechten Multitasking Systemen (Windows 3.x) übergibt das Betriebssystem die Kontrolle an den gerade ausgeführten Prozess, die anderen ruhen solange. Gibt ein Prozess die Kontrolle auf Grund eines Fehlers nicht mehr zurück, dann können die anderen Prozesse nie mehr weiter ausgeführt werden und bei einem Abbruch (Windows: Strg-Alt-Entf – Verfahren) wird das gesamte Betriebssystem heruntergefahren (Totalabsturz, oft mit Datenverlusten). Das Betriebssystem MS DOS läßt überhaupt kein Multitasking zu.

Ausführbare Dateien

In jedem Betriebssystem gibt es neben Dateien die Daten enthalten auch solche, die Programmcode enthalten. Diese Dateien enthalten Maschinencode, der vom Prozessor ausgeführt werden soll. Dazu muß der Maschinencode an eine geeignete Stelle in den Speicher geladen werden, die jedoch nicht absolut festgelegt werden kann, da ja oft mehrere Programme gleichzeitig im Speicher gehalten werden müssen oder Programmcode des Betriebssystems verschieden viel Platz benötigt (Treiberprogramme). Deshalb müssen während des Ladevorgangs noch relative Adressen des Maschinenprogramms in absolute Adressen umgewandelt werden. Der Vorgang, diese Adressenzuweisungen vorzunehmen heißt Binden (Linken). Solche ausführbaren Dateien haben unter MS DOS/Windows im Dateinamen die Erweiterung .EXE oder .COM.

Dynamic Link Libraries

Sehr umfangreichen Programme bestehen aus vielen, voneinander weitgehend unabhängigen Programmteilen, die auch getrennt entwickelt und compiliert werden (modulare Programmentwicklung). Diese getrennten Teile können bei der Entwicklung in eine große gemeinsame ausführbare Datei zusammengebunden werden (→ *linken*), die beim Laden dann als ganzes in den Arbeitsspeicher übertragen wird. Bei umfangreichen Programmen werden meist nicht alle Programmteile gleichzeitig benötigt. So wird ein Benutzer eines Textverarbeitungsprogramms nie alle der vielen gebotenen Bearbeitungsmöglichkeiten auf einmal verwenden. Daher kann man solche Programmteile getrennt übersetzen und auf einer Platte speichern und den Bindevorgang erst zur Laufzeit bei Bedarf durchführen (dynamisches Binden). Dieses Vorgehen hat zwei Vorteile gegenüber dem statischen Binden: Zum einen wird nur diejenige Programmcode in den Arbeitsspeicher geladen, der auch gerade benötigt wird. Dadurch wird Speicherplatz gespart. Zum anderen können solche dynamischen Programmdateien von mehreren verschiedenen Programmen benutzt werden, da sie ja nicht fest eingebunden werden. Da in dynamischen Programmdateien meist Code steht, der Standardaufgaben erledigt spricht man auch von Bibliotheken (Libraries), nennt sie **D**ynamic **L**ink **L**ibraries und gibt ihnen unter Windows die Dateierweiterung .DLL.

Kommandosprache und graphische Oberfläche

Jedes Betriebssystem besitzt eine Kommandosprache, die vom Kommandointerpreter interpretiert wird. Unter DOS heißt der Kommandointerpreter COMMAND.COM und ist selbst eine ausführbare Datei, im allgemeinen spricht man von einer Betriebssystem Shell (UNIX, LINUX). Beispiele für Betriebssystemkommandos sind unter MS DOS/Windows die Befehle *DIR* zur Anzeige eines Verzeichnisinhaltes, *COPY *.EXE A:/* zum Kopieren aller EXE-Dateien aus dem aktuellen Laufwerk auf die Diskette A:. Da der Umgang mit einem solchen Kommando-orientierten Betriebssystem viel Übung und dicke Handbücher erfordert sind alle modernen, für Nicht-Profis ausgelegten Systeme mit einer über der Kommandoebene liegenden graphischen Benutzeroberfläche ausgestattet. Fast alle Routineoperationen können auf dieser intuitiv verstehbaren Ebene viel einfacher ausgeführt werden, etwa Anzeigen eines Verzeichnisses durch Mausklick auf ein Verzeichnissymbol und Kopieren von Dateien durch Ziehen und Fallenlassen von Dateisymbolen („drag and drop“).

Stapelverarbeitungsdateien (Batch Dateien)

Muß man häufig immer gleichbleibende Anweisungsfolgen auf der Kommandoebene eines Betriebssystems ausführen, dann kann man diese Kommandos in eine Datei schreiben. Solche Dateien heißen unter MS DOS Stapeldateien (Batch Dateien), da

sie einen Stapel von Kommandos enthalten und erhalten die Dateierweiterung .BAT. Batch-Dateien können wie EXE-Dateien ausgeführt werden. Sie führen die Betriebssystemkommandos in der angegebenen Reihenfolge aus. Die bekannteste Batch-Datei unter MS DOS ist die Datei AUTOEXEC.BAT, die diejenigen Befehle enthält, die beim Booten des Computers zuerst ausgeführt werden. Unter UNIX spricht man statt von Batch Dateien von Shell-Scripts.

Eine typische Batch-Datei:

```
rem - By Windows Setup - MSCDEX.EXE /D:OEMCD001 /L:D
PATH C:\NC;%PATH%
SET MIDI=SYNTH:1 MAP:E
SET SOUND=C:\PROGRA~2\CREATIVE\CTSND
SET BLASTER=A220 I5 D1 H5 P330 T6
mode con codepage prepare=((437) C:\WINDOWS\COMMAND\ega.cpi)
mode con codepage select=437
keyb gr,,C:\WINDOWS\COMMAND\keyboard.sys
rem
PATH=C:\delphi\bin;C:\IBLOCAL\BIN;C:\IDAPI;%Path%PATH=C:\ZU_DELPH\IBLOCAL\BIN;C:\ZU_DE
LPH\IDAPI;%PATH%
SET CLASSPATH=E:\NETSCA~1\PALOMAR.ZIP
```

Literatur

Zur Einführung in die Informatik zu empfehlen

GOLDSCHLAGER/LISTER, Informatik
3.Auflage, Hanser 1990
Etwas älter (keine neuere Auflage erschienen), aber immer noch gut brauchbar, da die behandelten Konzepte sehr allgemein und daher immer noch gültig sind. Gut lesbar, kann mit einigen Auslassungen gut neben der Vorlesung benutzt werden. Objektorientierte Programmierung und Modellierungstechniken fehlen allerdings ganz.

RECHENBERG, P., Was ist Informatik
3.Aufl., Hanser , München, 2000.
Sehr gut lesbar, im Aufbau aber nicht ganz der Vorlesung entsprechend, da zuerst mit der technischen Informatik begonnen wird.
⇒ Gutes Begriffswörterbuch im Anhang.
⇒ Sehr lesenswerte kurze Kapitel über "Philosohie der Informatik".

RECHENBERG, P., POMBERGER, G. (HRSG.), Informatik-Handbuch
3.Auflage, 1192 Seiten, Hanser 2002.
Gut lesbares sehr umfangreiches *Nachschlagewerk* für alle Gebiete der Informatik.

PRECHT, M., MEIER, N. KLEINLEIN, J., EDV Grundwissen
6.Auflage, Addison-Wesley, Bonn 2001
Sehr **aktuelle**, aber etwas kurze Beschreibung von gegenwärtig verfügbaren Geräten, Anwenderprogrammen, Betriebssystemen und gängigen Theorien. Eignet sich sehr gut zum Nachschlagen, weniger als Text für die Informatik. Gute, klare Abbildungen.

GUMM, H.-P., SOMMER, M., Einführung in die Informatik
2.Auflage, Addison-Wesley, Bonn 1995
Gut lesbare klassische Einführung in die Informatik, die zwar auch neuere Entwicklungen noch berücksichtigt (Objektorientierte Programmierung, aktuelle Betriebssysteme, Netze), aber im Grunde prozedurale Aspekte in den Vordergrund stellt . Objektorientierte Modellierungstechniken fehlen. Nicht sehr theoretisch. Folgt im Aufbau im weitgehend der Vorlesung.

WHITE, RON, So funktionieren Computer. Ein universeller Streifzug durch den Computer
Markt und Technik, Haar bei München 1996
Kein Informatikbuch, aber sehr detaillierte Beschreibungen und Illustrationen zum Aufbau und zur Wirkungsweise von Computern sowie der Peripheriegeräte und Kommunikationseinrichtungen (Netzwerke, Modems,...).

Weitere Literatur zur Informatik: Ergänzend oder interessante Aspekte

BAUMANN, R., Informatik für die Sekundarstufe II, 2 Bände
Klett, Stuttgart , Band 1 1992, Band 2 1993
Band 1: Einfach lesbare Einführung in die Informatik für die Schule (Gymnasium) mit vielen Beispielen in Pascal. Auch als Einführung in die Pascal-Programmierung geeignet.
Band 2: Weiterführende Gebiete werden behandelt, z.B. Höhere Datenstrukturen, abstrakte Datentypen, Automatentheorie und formale Sprachen, Grenzen der Berechenbarkeit, Komplexitätstheorie. Dieser Band führt in sehr verständlicher Form in die o.g. Gebiete ein, ohne zu formal vorzugehen. Auch zum Nachlesen einzelner Kapitel geeignet, wenn einige Grundkenntnisse vorhanden sind.

WIRTH, N. , Algorithmen und Datenstrukturen
Teubner, Stuttgart 1983
Hervorragende klassische Einführung in das genannte Gebiet vom Vater der Sprache PASCAL, auch Jahre nach dem Erscheinen noch aktuell. Enthält z.B. sehr gute Darstellung und Analyse von Sortierverfahren. Nicht ganz einfach zu lesen.

WARNKE, M., Informatik (für das Nebenfachstudium)
Oldenbourg Verlag, München 1989
Klassische Einführung mit Pascal, nicht zu abstrakt.
Scheint mir von allen für das **Hochschulstudium** gedachten Büchern am besten geeignet.

KURZWEIL, R., Das Zeitalter der Künstlichen Intelligenz
Hanser, München.1992
Schöne popularwissenschaftliche Darstellung der Geschichte der Computerentwicklung mit Schwerpunkt auf der KI. Viele Bilder aus der Geschichte der EDV. Einige grundlegende Originalbeiträge. Sehr gut zu lesen, auch auszugsweise, interessant und anregend. Steht allerdings ganz in der Tradition der amerikanischen optimistisch-technikgläubigen Tradition der KI-Päpste vom MIT.

Zu Betriebssystemen

RIEDL. R., Vorlesung Betriebssysteme an der Universität Zürich, Sommer 2002
Einführung in Betriebssysteme. Für die Vorlesung im Wesentlichen die einführenden Kapitel geeignet, sonst zu speziell und detailliert.
http://www.ifi.unizh.ch/~riedl/lectures/BS02.html

HEIß, H.U., Vorlesung Betriebssysteme an der TU Berlin, WS 2002/03
Einführung in Betriebssysteme. Für die Vorlesung im Wesentlichen die einführenden Kapitel geeignet, sonst zu speziell und fortgeschritten.
http://kbs.cs.tu-berlin.de/teaching/ws2002/bs/bs_index.htm

STROEHL, Elektronisches Buch zum Betriebssystem Windows XP
Einführungskapitel geben ganz gute Beschreibung der Entwicklung der Windows-Betriebssysteme mit Endprodukt Windows XP. Ansonsten Beschreibung und der Bedienung der Oberfläche von Windows XP.
http://www.stroehl.info/html/EBOOK/DATA/START.HTM

Zur Objektorientierten Modellierung und Programmierung

BALZERT, H. , Objektorientierung in 7 Tagen
Vom UML-Modell zur fertigen Web-Anwendung
Sppektum Akad.Verl., Heidelberg 2000
Einführung in die Objektorientierung ohne Vorkenntnisse. Gut lesbar, nicht sehr technisch. Schwerpunkt auf der objektorientierten Modellierung, nicht auf irgend einer Programmiersprache.

GOLL, J., WEIß, C., ROTHLÄNDER, P., Java als erste Programmiersprache
2.Auflage, Teubner, Stuttgart 2000
Einführung die Java-Programmierung, ohne Kenntnisse anderer Programmiersprachen voraus zu setzen. Geht weit über die Vorlesung hinaus, Anfang aber brauchbar zur Einführung in die grundlegenden Konzepte.

KÜCHLIN, W., WEBER, A. , Einführung in die Informatik (Objektorientiert mit Java)
Springer, Berlin 2003
Einführung in die Informatik, mit Schwerpunkt auf Objektorientierung. Alle Beispiele in Java, eigentlich gleichzeitig eine Einführung in Java mit vielen technischen Details, geht daher weit über die Vorlesung hinaus.

Zur Didaktik der Informatik

BAUMANN, R., Didaktik der Informatik
Klett, Stuttgart 1997
Umfangreiche Sammlung von grundlegenden Konzepten der Informatik und ihren Randgebieten, einschließlich der unterrichtlichen Umsetzung bis hin zu Projektvorschlägen und Kriterien für die Leistungsbeurteilung.

GIERTH, U., Datenschutz im Unterricht
Dümmler 1988 Nicht mehr sehr aktuell, Probleme rund ums Internet spielen noch gar keine Rolle

HUBWIESER, P., Didaktik der Informatik
Springer, Berlin 2000
Versucht ein geschlossenes Konzept für eine Schulinformatik zu entwickeln. Insbesondere die Begriffe der Information und der Modellierung werden als allgemeinbildende Inhalte ausführlich dargestellt.
Enthält allgemeine didaktische Konzepte, aber auch viele (relativ) konkrete Unterrichtsvorschläge. Sicher ein wichtiges Buch zur neueren Entwicklung der Didaktik der Informatik.
Umfangreiches Literaturverzeichnis zu den neueren Entwicklungen in der Didaktik der Informatik. Geht einher mit der Entwicklung und Erprobung eines Informatik-Curriculums für alle bayerischen Schularten.

MODROW, E., Zur Didaktik des Informatikunterrichts
Band 1 und 2, Dümmler 1991
Band 2: Gesellschaftliche Auswirkungen)

CYRANEK, G., FORNECK,H., GOORHUIS,H. (HRSG.), Beiträge zur Didaktik der Informatik.
Frankfurt, Diesterweg 1990
Ganz gute Beiträge zur Bedeutung der Informatik in der Schule

Gesellschaftliche Auswirkungen, kritische Auseinandersetzung mit der Informatik

COY, W., Sichtweisen der Informatik
Vieweg, Braunschweig 1992
Behandelt auch Gefahren und Sozialverträglichkeit der Informatik

GMEHLICH, R., RUST, H., Mehr als nur Programmieren
Vieweg, Braunschweig 1993
Einführung in die Informatik mit Betrachtung kritischer Situationen.

LANGENHEDER, W., MÜLLER, G. (HRSG.), Informatik-cui bono
Springer Berlin 1992
Tagungsband einer Tagung am Institut für Informatik und Gesellschaft in Freiburg 1992. Beiträge verschiedener Autoren zur Auswirkung der Informatik in der Gesellschaft.

WEIZENBAUM, J., Kurs auf den Eisberg
Pendo Verlag, Zürich 1984
Interview mit J.Weizenbaum über die Auswirkungen der Informationstechnologie. Kurz und einfach zu lesen.

FRIEDRICH, J., HERRMANN, TH., PESCHEK, M.,.ROLF, A. (Hrsg.), Informatik und Gesellschaft
Spektrum Akad.Verlag, Heidelberg 1995
Behandelt die Auswirkungen, Gefahren und Chancen der Informatik in Bezug auf die Gesellschaft und Arbeitswelt, Themen, über die in den klassischen Informatikbüchern nur im Vorspann oder im Anhang einige Seiten zu finden sind.

Glossar zu Begriffen der Informatik

http://www.software-technik.de/microsoft/glossar/Glossar.htm