

Info Lk
Abiturjahrgang 2011
Humboldt Gymnasium Eichwalde

Martin Lieser

18. April 2011

Inhaltsverzeichnis

I	Algorythmik	1
1	Strucktogramme	1
II	Pascal	2
2	Befehlslisten	2
2.1	Variablentypen	2
2.1.1	Zahlen	2
2.1.2	Zeichen	2
2.1.3	Zusammengesetzte Dateitypen	3
2.1.4	Dynamische Variablen	4
2.2	Allgemeine Befehle	4
2.3	Rechenoperationen	5
2.4	Schleifen und Bedingungen	5
2.5	Dateiarbeit	5
2.5.1	Textdateien	6
2.6	Grafik	6
3	Grundlagen	6
3.1	Aufbau eines Programms	6
3.2	Proceduren und Funktionen	7
3.3	Rekursion	7
4	Sortiervverfahren	7
4.1	Bubblesort	7
4.2	Selectionsort	8
4.3	Insertionsort	9
4.4	Merchesort	9
4.5	Quicksort	10
4.6	Heapsort	11
5	Suchverfahren	11
5.1	lineare Suche	11
5.2	Binäre Suche	12
5.3	Boyer-Moor-Suche	12
6	Dateiarbeit	13
6.1	Unypisierte Datei	13
6.2	Typisierte Datei	13
6.3	Textdatei	14
7	dynamische Listen	14
7.1	Aufbau einer Liste	15
7.2	Löschen in einer Liste	15
7.3	Aufbau einer Liste	16
7.4	doppelt verkettete Liste	16
7.5	Ringliste	16
7.6	Liste in Liste	16
8	Objektorientierte Programmierung	16
III	Datenstrukturen	16
9	Baumstrukturen	16
9.1	Binärer Baum	17
9.1.1	binärer Suchbaum	18
9.1.2	Huffmannbaum	18
9.1.3	Heap-geordneter Baum	19

10 Graphen	19
10.1 Graphentheorie	19
10.1.1 Begriffe	19
10.1.2 Darstellung	19
10.2 Spezialfälle	20
10.2.1 gerichteter Graph	20
10.2.2 gewichteter Graph	20
10.3 Algorithmus von Dijkstra	20
 IV Prolog	 21
11 Rätsel	21
11.1 Zahlenrätsel	21
11.2 Das Kohlkopf - Ziege - Wolf Problem	22
12 Datenbanken	23
12.1 Büchereri	24
12.2 Bundeskanzler	24
13 Rekursion	25
14 Listen	25
14.1 Grundlagen	25
14.2 Sortieren	25
 V Automatentheorie	 26
15 Kellerautomaten	26
16 erkennende Automaten	26
17 Turing Machine	27
17.1 universelle Turing-Maschine	27
 VI Schaltalgebra	 28
18 Grundsaltungen	28
19 Umstellen von Schaltgleichungen	28
19.1 Gesetzze	28
19.1.1 Disjunktive Normalform (DNF)	29
19.1.2 konjunktive Normalform (KNF)	29
19.2 Schaltungsentwicklung	29
19.2.1 Schaltwerttabelle	29
19.2.2 DNF	29
19.2.3 Kürzen	29
19.2.4 Schaltung	29
20 Flip-Flops	30
20.1 Grund Flip-Flop	30
20.2 Getacketer Flip-Flop	30
20.3 Master-Slave Flip-Flop	31
 VII Zahlensysteme	 31
21 Binär/Dualsystem	31
21.1 Von Binär zu Dezimal	31
21.2 Von Dezimal zu Binär	32
21.3 Rechnen mit Binärzahlen	32
21.3.1 Regeln	32
21.3.2 Multiplikation	32

<i>INHALTSVERZEICHNIS</i>	III
21.3.3 Subtraktion	32
21.3.4 Kommazahlen	32
VIII Bilderverzeichniss	32

Teil I

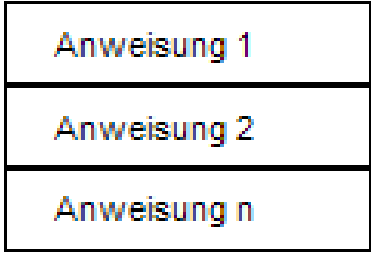
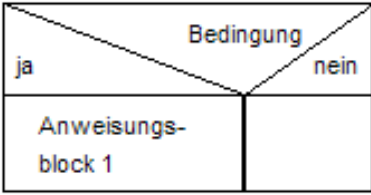
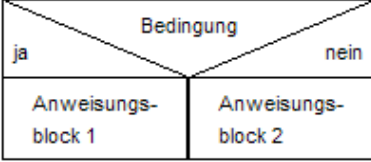

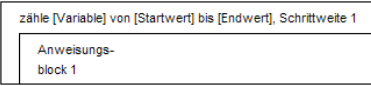

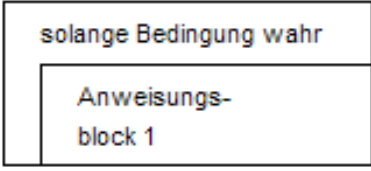
Algorithmen

Algorithmus: Ist eine Folge von eindeutigen und ausführbaren Anweisungen zur Bearbeitung bestimmter Ausgabedaten aus bestimmten Eingabedaten.

Bedingungen: Ein Algorithmus muss eindeutig sein. Zudem ist er Allgemeingültig und ausführbar, das heißt, er muss auch endlich sein. Er sollte möglichst effizient und sinnvoll sein.

Grundelemente: Sind Sequenzen, also die Reihenfolge der Anweisungen, Entscheidungen, hier unterscheiden sich Einfachentscheidungen (wenn...dann), Zweifachentscheidungen (wenn...dann...sonst) und Mehrfachentscheidungen (falls...dann), sowie Wiederholungen. Wiederholungen gibt es als Schleife mit Eintrittsbedingung, Schleife mit Austrittsbedingung und als gezählte Schleife.

1 Struktogramme

Element	Verbal	Struktogramm	Programmablaufplan
Sequenz	erst...,dann...,dann...		
Einfachentscheidung	wenn...dann...		
Zweifachentscheidung	wenn...,dann...,sonst...		
Mehrfachentscheidung	falls Bedingung 1 dann... falls Bedingung 2 dann... ...		
gezählte Wiederholung	mache ... für Zähler=... bis ...		
Schleife mit Austrittsbedingung	wiederhole ... bis Bedingung		
Schleife mit Eintrittsbedingung	solange Bedingung mache ...		

Teil II

Pascal

2 Befehlslisten

2.1 Variablentypen

Byte	ganze Zahlen von 0 bis 255
Word	ganze Zahlen von 0 bis 65535
Integer	ganze Zahlen von -32768 bis + 32767
longint	2147483648 bis - 2147483649
shortint	127 bis -128
Real	Kommazahlen
Char	einzelne Buchstaben
String	Zeichenketten
Feld	Zusammenfassung von Daten gleichen Typs

Die einzelnen Variablentypen werden im folgenden näher erläutert. Auch nur für bestimmte Typen verwendbare Befehle sind hier aufgelistet.

2.1.1 Zahlen

Byte Speicherplatz: 1 Byte(8 Bit)
 00000000 → 0
 11111111 → 255

Word Speicherplatz: 2 Byte (16 Bit)
 von 0 bis 65535

Integer Speicherplatz: 2 Byte (16 Bit)
 Das erste Bit gibt an Ob die Zahl positiv (0) oder negativ (1) ist.
 0111111111111111 → 32767
 1000000000000001 → -32767
 Bei Shortint ist der Speicherplatz 1 Byte, bei Longint 4 Byte. Der Aufbau ist wie bei Integer.

x:=round(y);	Rundet y (Real) auf, sodass es eine ganze Zahl ist.
x:=trunc(y)	Schneidet die Kommastellen von y (Real) ab, sodass es eine ganze Zahl ist.

Real Speicherplatz: 6 Byte
 größte: 2^{+127}
 kleinste: 2^{-128}

x:=round(y);	Rundet y auf, sodass es eine ganze Zahl (z.B. integer) ist.
x:=trunc(y)	Schneidet die Kommastellen von y ab, sodass es eine ganze Zahl ist(z.B. integer).
x:=int(y)	Entfernt die Kommastellen von y. Die Zahl ist immernoch im Real Format.

2.1.2 Zeichen

Char Speicherplatz 1 Byte
 Die Zeichen werden durch den ASCII Code mit den Zahlen von 0 bis 255 gespeichert.

y:=ord(x)	x ist im Typ Char. In y wird die ASCII-Code Zahl gespeichert.
y:=chr(x)	x ist eine Zahl. In y wird das im ASCII-Code der Zahl zugehörige Zeichen gespeichert.
x:=upcase(y);	Macht aus dem Char y einen Großbuchstaben und speichert diesen in x.
x:=readkey;	liest einen Wert in die Variable x ein

String Speicherplatz: 256 Byte

Das erste Byte ist für die Länge des Strings reserviert. In den restlichen Bytes sind die jeweiligen ASCII-Code Nummern gespeichert.

<code>y:=x[i]</code>	liest aus dem String x die Stelle i aus und speichert diese in y.
<code>x:=length(y);</code>	Ordnet der Variable x (integer) die Länge vom String y zu
<code>delete(x,i,y);</code>	Löscht aus x(String) an der Stelle i(integer) y(integer)-viele Buchstaben.
<code>insert(y,x,i);</code>	Fügt in x(String) an der Stelle i(integer) den String/Char y ein.
<code>y:=copy(x,a,b)</code>	Speichert in y String x so viele Zeichen wie b angibt ab der stelle a.
<code>z:=x + y</code>	Hängt an den String x den String y an und speichert dies in z.
<code>y:=pos(x,z)</code>	Sucht in dem String z den Teilstring x und speichert die Position in y. Wenn dieser nicht enthalten ist wird 0 ausgegeben. Wenn er mehrmal vorkommt wird nur der erste gefunden.
<code>val(x,y,s)</code>	Wandelt den String x in eine Zahl y um. s ist eine Kontrollvariable.
<code>str(x,y)</code>	Wandelt eine Zahl x in einen String y um.

2.1.3 Zusammengesetzte Datentypen

Feld Ein Feld ist eine Zusammenfassung von Daten gleichen Typs. Diese werden durch Indizes durchnummeriert.

Ein Feld wird folgendermaßen deklariert:

```
Var f:array[a..b,c..d,...] of Datentyp;
```

Es gibt 1 bis n dimensionale Felder. a und b geben die Indizes der ersten, c und d die der zweiten Dimension an. Wenn a=1 und b=5 ist, dann gibt es in der ersten Dimension 5 Datenplätze zur Verfügung. Genauso verhält es sich mit den anderen Dimensionen. Genau wie bei ForTo-Schleifen kann die Bezeichnung auch mit ASCII-Code Zeichen geschehen. Einzelne Datenplätze werden mit f[i,j,..] bezeichnet. Wobei i den Platz der ersten, j den der zweiten Dimension bezeichnet, ähnlich wie bei Strings. Belegt und ausgegeben werden diese meistens mithilfe einer ForTo-Schleife.

Mengen Bei einer Menge spielen die Anzahl verschiedener Elemente und deren Position keine Rolle.

Eine Menge wird folgendermaßen deklariert:

```
Var m:set of Datentyp;
```

Belegt wird es, indem ein Wert in eckigen Klammern mit einem + der Menge hinzuaddiert wird (`m:=m+[3]`). Es können so viel Zeichen wie nötig eingefügt werden. Genauso wird auch ein Element wieder gelöscht, dabei spielt es keine Rolle, wie oft das Element eingefügt wurde.

`x in m` | Überprüft ob das Element x in der Menge m enthalten ist.

Records Ist eine Zusammenfassung von Daten unterschiedlichen Typs. Besonders für Datenbanken ist das sehr praktisch. Für Listen werden auch Records verwendet.

Ein Record wird folgendermaßen deklariert:

```
Var name:Record
  x,y:Datentyp1;
  z :Datentyp2;
  ...
end;
```

Die einzelnen Werte werden nun mit `name.x` aufgerufen. Das heißt, der Name des Records mit einem "." und dann der Name der Variable im Record. Oder:

```
with name do begin
  readln(x);
  y:=x;
  writeln(z);
end;
```

2.1.4 Dynamische Variablen

Ein Zeiger(Pointer) beinhaltet keinen Wert, sonder eine Speicheradresse.

Deklaration: $z : \hat{integer}$;

$\hat{}$ Steht dabei für zeigt auf.

Zu beachten hierbei ist, das z die Speicheradresse ist und $z\hat{}$ der Inhalt der Speicheradresse. daher wird auch $z\hat{:=} 10$; geschrieben. Mit $New(z)$; wird ein Speicherplatz für die Variable freigehalten und die Speicheradresse in z gespeichert. $dispose(z)$ löscht diesen Zeiger und somit den freigehaltenen Platz.

2.2 Allgemeine Befehle

begin ... end;	Rahmt Anweisungen Ein. Z.B. bei Schleifen. Wird als eine Anweisung gesehen, kann mehrere enthalten. Auch als Anfang und Ende des Programms. Das Ende wird mit einem "end." festgelegt.
procedure ...(); var...; begin ...; end;	procedure (name) ((zu übernehmende Variablen)); Var (nur für die Procedure geltenden Variablen(optional)); begin(muss da sein, auch wenn es nur eine Anweisung ist) (Anweisungen); end; Die Procedures kommen vor dem Hauptprogram, jedoch nach dem Kopf
funktion ...():...; var...; begin ... (funktion):=...; end;	Definition im Program wie eine Procedure, nur muss sie einem Variablentyp zugeordnet werden, da sie sich wie eine verhält. In der Funktion wird ein Wert zugeordnet, beim Aufruf kann dieser durch (x:=(funktion)) in eine andere Variable gespeichert werden.
write(); writeln();	schreibt an die Aktuelle Position. schreibt an die Aktuelle Position und beginnt eine neue Zeile Wenn mehrere Variablen in einem Befehl ausgegeben werden, werden diese nicht durch ein Lehrzeichen getrennt. Dies geschieht dmit Hilfe eines Doppelpunktes und der Stellenangabe (write(a:4,b:4,c:4). Hier werden für Jede Zahl 4 Zeichen freigehalten. Bei Ausgabe einer Real Zahl können auch die Kommastellen festgelegt werden (write(a:4:1,b:4:1,c:4:1)).
read(); readln();	ließt eine Eingabe ließt eine Eingabe und beginnt eine neue Zeile
readkey;	wartet an dieser Stelle bis etwas eingegeben wird
clrscr;	löscht die gesamte Ausgabe
randomize; x:=random; x:=random(y);	bereitet für die Erzeugung von Zufallszahlen vor Ordnet x eine Zufallszahl zwischen einschließlich 0 und 1 zu. Ordnet x eine Zufallszahl zwischen einschließlich 0 und y zu. Es können stellenweise Häufungen auftreten, welche sich mit größeren Datenmengen ausgleichen, Somit sind keine Zahlenreihen erkennbar.
gettime(h,m,s,ms);	Speichert die Systemzeit in h=Stunde, m=Minute, s=Sekunde, ms=Milisekunde. Die Variablen sind im Typ Integer. Für diesen Befehl wird die Unit DOS benötigt
gotoxy(x,y); window(x1,y1,x2,y2)	Setzt den Cursor an die Stellen x und y im Bildschirm Legt einen Arbeitsbereich fest. x1,y1,x2,y2 sind die Begrenzungen. Der neue Arbeitsbereich verhält sich wie der Alte. mit window(1,1,80,49) kommt man wieder in den alten Arbeitsbereich.
textbackground(); textcolor();	Setzt für alle nachfolgenden Ausgaben die angegebene Farbe als Hintergrund, mit "clrscr"wird der ganze Bildschirm fabig(Zahlen 0 bis 15). Setzt für alle nachfolgenden Ausgaben die angegebene Farbe als Textfarbe
x:=round(y); x:=trunc(y)	Rundet y (z.B. Real) auf, sodass es eine ganze Zahl (z.B. für integer) ist. Schneidet die Kommastellen von y (z.B. Real) ab, sodass es eine ganze Zahl ist.
keypressed;	Für Bedingungen, ist solange false, bis eine Taste gedrückt wird, dann ist es true.

2.3 Rechenoperationen

<code>x:=y;</code>	Ordnet der Variable x den Wert y zu
<code>x=y;/x<>y;/ x<y;/x>=y;</code>	Vergleicht Variable x mit y, gibt false oder true zurück
<code>x div y x mod y</code>	Gibt das Ergebniss der Ganzzahligen Division von x durch y aus Gibt den Rest der Ganzzahligen Division von x durch y aus
<code>sqrt(y)</code>	Bildet die Wurzel von y
<code>succ(x) pred(x)</code>	Bildet den Nachfolger von x Bildet den Vorgänger von x

2.4 Schleifen und Bedingungen

<code>if ... then ... else ...;</code>	wenn (Bedingung) dann (Anweisung ¹) Sonnst (Anweisung)
<code>Case ... of ... : ...; ... : ...; end;</code>	Wenn (Variable) ist (Variante):(Anweisungen); (Variante):(Anweisung); end;
<code>for ... to ... do ...; for ... downto ... do ...;</code>	von (i=1) bis (9) mache (Anweisung); von (i=9) runter-bis (1) mache (Anweisung); Funktioniert auch mit Buchstaben. Hier wird der ASCCICODE benutzt. Gezählt wird also in dieser Reihenfolge.
<code>repeat ... Until ...;</code>	wiederhole (Anweisung); bis (Bedingung);
<code>while ... do ...;</code>	solange wie (Bedingung) mache (Anweisung);
<code>not() () and () () or ()</code>	nicht(Bedingung) (Bedingung) und (Bedingung) (Bedingung) und/oder (Bedingung)

2.5 Dateiarbeit

<code>assign(f,(Pfad²));</code>	Verknüpft den logischen Dateinamen mit dem Pfad zu der Datei.
<code>rewrite();</code>	legt die Datei neu an, wenn sie schon existiert wird sie überschrieben.
<code>reset();</code>	öffnet eine vorhandene Datei und setzt den Datenzeiger auf die erste Stelle ³ .
<code>rename(f,(Pfad));</code>	Gibt der Datei einen neuen Pfad und Speichert sie unter diesem ab. Die Datei muss dabei geschlossen sein (close(f))
<code>close();</code>	schließt eine Datei.
<code>write(f,x);</code>	schreibt Datensatz x in die Datei und setzt den Zeiger weiter ⁴ .
<code>read(f,x);</code>	ließt Datensatz x aus der Datei und setzt den zeiger weiter ⁵ .
<code>erase(f);</code>	löscht die Datei von der Platte ⁶
<code>eof(f);</code>	ergibt True, wenn der Zeiger am Ende der Datei steht.
<code>seek(f,x)</code>	Setzt den Dateinzeiger bei der Datei f auf die Position x.

¹Eine Anweisung besteht aus einem befehl mit anschließendem ";". Wenn man mehrere Befele nutzen möchte, Benutzt man "Begin" und "end", siehe oben

²Die Ordner im Pfad dürfen nicht länger als 8 Zeihen lang sein. Sind sie es doch, werden die ersten 6 Zeichen + ~ + das letzte Zeich verwendet.

³zur Funktion bei Textdatein siehe unten.

⁴zur Funktion bei Textdatein siehe unten.

⁵zur Funktion bei Textdatein siehe unten.

⁶Die Datei muss zuvor geschlossen werden.

<code>x:=filesize(f)</code>	Speichert die Länge der Datei f in die Variable x.
-----------------------------	--

2.5.1 Textdateien

<code>append();</code>	öffnet die Datei NUR zum anhängen von Datensätzen.
<code>reset(f);</code>	öffnet die Datei NUR zum lesen.
<code>writeln(f,x);</code>	schreibt String x in eine Zeile der Datei und setzt den Zeiger eine Zeile weiter.
<code>writelf,x;</code>	schreibt Char x an die Aktuelle Position und setzt den Zeiger eine Position weiter.
<code>readln(f,x);</code>	liest eine Zeile in einen String und setzt den Zeiger eine Zeile weiter.
<code>read(f,x);</code>	liest ein Zeichen in eine Char Variable und setzt den Zeiger eine Position weiter.
<code>eoln(f);</code>	gibt True wenn er am Ende einer Zeile steht, Analog eof(f).

2.6 Grafik

Um Grafikbefehle nutzen zu können wir die Unit graph benötigt. Außerdem braucht man die Integervariablen gd (Grafiktreiber) und gm (Grafikmodus).

Der Grafiktreiber wird mit `gd:=detect` bestimmt. Danach wird mit `initgraph(gd,gm,'')` die Grafische Oberfläche geöffnet. Diese arbeitet mit Pixeln.

<code>moveto(x,y);</code>	wie gotoxy(x,y) nur mit Pixeln.
<code>setcolor(f);</code>	Setzt die Farbe für folgende Befehle fest.
<code>setfillstyle(m,f);</code>	Setzt das Füllmuster (m von 1-12) und die Füllfarbe fest.
<code>floodfill(x,y,rf);</code>	Füllt mit dem angegebenen Füllstil eine Fläche aus. In dieser Fläche ist der Punkt x,y und sie ist durch eine Farbveränderung zu der angegebenen Farbe rf abgegrenzt.
<code>putpixel(x,y,f);</code>	Setzt an die Stelle x,y einen Pixel mit der Farbe f.
<code>line(x1,y1,x2,y1);</code>	Zeichnet eine Linie von Punkt x1,y1 zu Punkt x2,y2.
<code>lineto(x,y);</code>	Zeichnet eine Linie von der aktuellen Position zu Punkt x,y.
<code>circle(x,y,r);</code>	Zeichnet einen Kreis mit dem Mittelpunkt x,y und dem Radius r
<code>ellipse(xm,ym,aw,ew,xr,yr);</code>	Zeichnet eine Ellipse. Hierbei ist der Mittelpunkt xm,ym und die Radien xr und yr. Es können auch nur Teile der Ellipse gezeichnet werden mit dem Anfangswinkel aw und dem Endwinkel ew.
<code>rectangle(x1,y1,x2,y1);</code>	Erstellt ein Rechteck mit dem Unteren linken Punkt x1,y1 und dem oberen rechten Punkt x2,y2.

3 Grundlagen

3.1 Aufbau eines Programms

Kopf:	Name eingebundene Units (Hilfsprogramme)	program name; uses crt ¹ ;
Verinbarungsteil:	Typendeclaration Variablendeklaration	type Var Variable1:Datentyp; Variable2,3:Datentyp;
	Konstantendeklaration Prozeduren Funktionen	const x=1000;
Anweisungen	eigentliches Programm	Begin ... End.

¹Auch andere wie DOS oder GRAPH, können auch selber erstellt werden

3.2 Prozeduren und Funktionen

Eine Prozedure ist ein eigenständiges Teilprogramm, welches mit seinem Namen aufgerufen wird. Dahingegen sind Funktionen eigenständige Teilprogramme, welche einen Wert ermitteln und unter ihrem eigenen Namen zurückgeben. Sie werden deshalb immer in einer Anweisung aufgerufen und können nicht wie Prozeduren als einzelne Anweisung im Programm stehen. Prozeduren und Funktionen werden im Vereinbarungsteil deklariert.

Bei den Variablen sind einige Dinge zu beachten. Variablen, welche im Hauptprogramm deklariert werden, gelten im gesamten Programm (auch in den Prozeduren/Funktionen), sie werden globale Variablen genannt. Variablen die in der Procedure/Funktion deklariert werden gelten auch nur in diesen, sie werden lokale Variablen genannt. Wenn locale und globale Variablen die gleichen Namen haben, werden immer die lokalen Variablen verwendet.

Parameter werden beim Aufruf der Procedure/Funktion angegeben. Sie dienen dem Datenaustausch zwischen Haupt und Teilprogramm.

```
procedure test(x,y:integer; z:char)
```

Bei dem beispiel werden drei Variablen übergeben. x und y als Integer und z als Char. Beim Aufruf muss die Reihenfolge der Variablen berücksichtigt werden.

Man Unterscheidet zwischen Werte - und Variablenparametern. Bei den Werteparametern werden, falls die Parameter verändert werden, nicht zurückgegeben, bei den Variablenparametern werden veränderungen an das Hauptprogramm übermittelt. Um dies zu erreichen muss vor den Variablenparametern ein VAR stehen. Alles was nach einem VAR steht ist ein Variablenparameter, alles davor ein Werteparameter.

Aufbau einer Funktion add. Die Funktion addiert die Zahl x und die Zahl y, das Ergebniss gibt die Funktion mit sich selber zurück.

```
funktion add(x,y:integer):integer;
    Var z:integer;
    begin
        z:=x+y;
        add:=z;
    end;
```

3.3 Rekursion

Rekursion beschreibt die Lösung eines Problems durch Aufruf einer einfacheren Variante von sich selbst. Das Problem schachtelt sich sozusagen auf. Bei Pascal passiert das mit Funktionen und Prozeduren. Bei Prolog durch aufrufen einer Regel in sich selber. Nähere Details dazu in Abschnitt 13 Seite 25. Als Beispiel kann das rekursive Sortierverfahren Quicksort geannt werden (Siehe 4.5 Seite 10).

4 Sortierverfahren

Die erklärten Sortierverfahren sind auch auf Prolog anwendbar (siehe 14.2 Seite 25). Sie sind hier beispielhaft an Pascal Programmen erklärt.

Es gibt verschiedene Gütekriterien für Sortierverfahren. Die Geschwindigkeit wird anhand der Laufzeit festgestellt. Diese gibt an, wie viel mehr Zeit das Sortierverfahren bei einer größeren Datenmenge braucht. Auch der Speicherbedarf, den die Sortierverfahren brauchen ist ein wichtiges Kriterium. Außerdem gibt es Stabile und nicht Stabile Sortierverfahren. Stabil ist es, wenn bei gleichen Elementen das was im unsortierten Feld vorne stand auch im sortierten wieder vorne steht. Das ist wichtig, wenn man Daten hat, wie zum Beispiel Name und Vorname und diese nach den Vornamen Sortiert, und danach nach den Namen.

4.1 Bubblesort

Bubblesort geht das Feld (meist von vorne) durch. Nebeneinander stehende Elemente werden dabei verglichen, wenn das links stehende größer ist werden sie getauscht. Dies wird so oft wiederholt, bis kein Element mehr getauscht wird, jetzt ist das Feld sortiert.

Laufzeit:	Quadratisch
Stabilität:	gegeben
Speicherbedarf:	gering
vorsortiertes Feld	1 Durchlauf
rückwärts sortiertes Feld	lange Dauer

Aufruf mit bubble(n)

Das zu sortierende Feld ist a

```

procedure tauschen(var a,b:integer);
    Var y:integer;
    begin
        y:=a;
        a:=b;
        b:=y;
    end;

procedure bubble(n:integer);
    var j,i:integer;
    begin
        Repeat
            j:=0;
            for i:=1 to n do
                If a[i]>a[i+1] then
                    begin
                        tauschen(a[i],a[i+1]);
                        j:=1;
                    end;
            Until j=0;
    end;

```

4.2 Selectionsort

Selectionsort Sucht sich im Feld das kleinste (größte) Element und tauscht es mit dem ersten Element. Das neue kleine Element wird ohne das schon sortierte, vordere Feld gesucht. Der letzte getauschte Wert ist der mit n-1, dann ist das Feld sortiert.

Laufzeit:	Quadratisch, jedoch schneller als Bubblesort
Stabilität:	nicht gegeben
Speichebedarf:	gering
vorsortiertes Feld	1 Durchlauf
rückwärts sortiertes Feld	lange Dauer

Aufruf mit `selec(n)`

Das zu sortierende Feld ist `a`

```

procedure selec(n:integer);
  var i,i2,y,k,x:integer;
  begin
    for i:=1 to n-1 do
      begin
        y:=a[i];
        k:=a[i];
        x:=i;
        for i2:=i to n do
          if a[i2]<k then
            begin
              k:=a[i2];
              x:=i2;
            end;
          a[x]:=y;
          a[i]:=k;
        end;
      end;
    end;
  end;

```

4.3 Insertionsort

Insertionsort durchläuft das Feld vom 2ten bis zum nten Element. Das aktuelle Element wird gemerkt und die Position solange nach links verschoben, wie die Elemente größer als das aktuelle sind, oder bis das Ende des Feldes erreicht ist. Das gemerkte Element wird an die aktuelle Position eingefügt.

Laufzeitverhalten	Quadratisch
Stabilität	stabil
Speicherbedarf	gering
vorsortiertes Feld	nichts verschieben
rückwärts sortiertes Feld	lange Laufzeit

Aufruf mit `ins(n)`

Das zu sortierende Feld ist `a`

```

procedure ins(n:integer);
  var i,y,k:integer;
  begin
    for i:=2 to n do
      begin
        y:=1;
        k:=a[i];
        while (k<a[i-y]) and (i-y<>0) do
          begin
            a[i-y+1]:=a[i-y];
            a[i-y]:=k;
            y:=y+1;
          end;
        end;
      end;
    end;
  end;

```

4.4 Mergesort

Mergesort ist ein rekursives Sortierverfahren.

Es teilt das Feld in der Mitte und ruft sich nacheinander mit den entstehenden Teilfeldern wieder auf, um diese weiter zu zerteilen. Dies passiert so oft, bis einelementige Felder entstanden sind. Diese werden jetzt wieder zusammengefügt. Es werden die jeweils zuletzt getrennten Felder in ein Hilfsfeld geschrieben. Dabei werden von den (sortierten) Teilfeldern jeweils die ersten Elemente verglichen und das kleinere in das Feld geschrieben. Wenn ein Feld leer ist, wird der Rest vom anderen Feld auch zurückgeschrieben. Danach wird das Hilfsfeld in das Originalfeld zurückgeschrieben.

Laufzeitverhalten

Logarithmisch

Stabilität

Speicherbedarf

benötigt Hilfsfeld mit der gleichen Länge wie Originalfeld, große Stack

vorsortiertes Feld

rückwärts sortiertes Feld

Aufruf mit `merg(1,n)`

Das zu sortierende Feld ist `a`

```

procedure mischen(b,x,e:integer);
  var i,z,y,u:integer;
      c:array[1..n] of integer;
  begin
    z:=b;
    y:=x+1;
    i:=b;
    while (z<=x) and (y<=e) do
      If a[z]<=a[y] then
        begin
          c[i]:=a[z];
          i:=i+1;
          z:=z+1;
        end
      else
        begin
          c[i]:=a[y];
          i:=i+1;
          y:=y+1;
        end;
      If z<=x then for u:=z to x do
        begin
          c[i]:=a[u];
          i:=i+1;
        end;
      If y<=e then for u:=y to e do
        begin
          c[i]:=a[u];
          i:=i+1;
        end;
      for i:=b to e do
        a[i]:=c[i];
      end;
end;

procedure merg(b,e:integer);
  var x:integer;
  begin
    if b<e then
      begin
        x:=(b+e) div 2;
        merg(b,x);
        merg(x+1,e);
        mischen(b,x,e);
      end;
  end;
end;

```

4.5 Quicksort

Quicksort ist ein rekursives Sortierverfahren.

Es Nimmt sich in seinem Feld ein beliebiges Element als Piboelement. Er ruft sich nun mit zwei kleineren Feldern wieder auf (Rekursion). Dabei sind in dem linken alle kleinergleich dem Piboelement, in dem rechten

alle größergleich einsortiert. Dies passiert solange, bis nurnoch einelementige Listen vorhanden sind. Wenn man diese nun wieder zusammensetzt, ist das Feld sortiert.

Laufzeit:	logarithmisch
Stabilität:	variantenabhängig
Speicherbedarf:	stackabhängig
vorsortiertes Feld	
rückwärts sortiertes Feld	

Aufruf mit qui(1,n)

Das unsortierte Feld ist a.

```

procedure zerlegen(b,e:integer; var t:integer);
  var p,i,j,x:integer;
  begin
    p:=a[b];
    i:=b;
    for j:=i+1 to e do
      if a[j]<p then
        begin
          i:=i+1;
          x:=a[i];
          a[i]:=a[j];
          a[j]:=x;
        end;
    t:=i;
    x:=a[b];
    a[b]:=a[i];
    a[i]:=x;
  end;

procedure qui (b,e:integer);
  var y:integer;
  begin
    if e>b then
      begin
        zerlegen(b,e,t);
        y:=t;
        qui(b,t-1);
        qui(y+1,e);
      end;
  end;

```

4.6 Heapsort

Das Prinzip von Heapsort basiert auf dem Heap-Baum.(siehe 9.1.3 Seite 19). In diesem Baum wird die Heap-Ordnung mit dem entsprechendem Feld hergestellt. Jetzt wird das am weitesten unte-rechts stehende Element mit dem Kopf getauscht und die Heap-Ordnung ohne den vorherigen Kopf wiederholt. Wenn nurnoch ein Element im Baum enthalten ist, kann der ganze Baum Level-Order (siehe 9.1 Seite 17) ausgegeben werden und das so entstehende Feld ist geordnet.

5 Suchverfahren

5.1 lineare Suche

Die lineare Suche ist die einfachste Variante ein Element zu suchen. Das Feld wird von vorne bis hinten durchlaufen, die Elemente werden mit dem gesuchten verglichen. Wenn sie gleich sind, dann hat man das Element gefunden.

Aufruf mit linear(x,n).
 x ist das gesuchte Element, n die Anzahl der Elemente.
 Das zu durchsuchende Feld ist a.

```

procedure linear(x,n:integer);
  var i,k:integer;
  begin
    k:=0;
    for i:=1 to n do
      if a[i]=x then
        begin
          writeln('Gefunden an Stelle ',i);
          k:=1;
        end;
    If k=0 then
      writeln('Das Element ist nicht enthalten.');
```

```

  end;

```

5.2 Binäre Suche

Die Binäre Suche benötigt ein sortiertes Feld. Der Algorithmus nimmt die Mitte des Feldes und schaut, wenn das Element kleiner ist, im linken Teilfeld, wenn es größer ist, im rechten. Mit den Teilfeldern wird genauso verfahren (Rekursion) und abgebrochen wird, wenn das Vergleichselement das gesuchte ist, oder wenn das Feld nurnoch ein Element hat. Bei letzterem wurde das Element nicht gefunden.

Aufruf mit binar(x,1,n).
 x ist das gesuchte Element, n die Anzahl der Elemente.
 Das zu durchsuchende und sortierte Feld ist a.

```

procedure binar(x,a,n:integer);
  var i;
  begin
    i:= (a+n) div 2;
    If a[i]=x then write('Gefunden an Stelle 'i)
    else
      If a[i] > x then binar(x,a,i)
      else binar(x,i,n);
  end;

```

5.3 Boyer-Moor-Suche

Boyer-Moor-Suche ist ein Verfahren zum suchen von Teilstrings in einem größeren. Hier wird von dem gesuchten String das letzte Element mit dem dazugehörigen Element im durchsuchten String verglichen. Wenn n die länge des gesuchten Strings ist, dann bei dem ersten Versuch mit dem n-ten Element des durchsuchten Strings. Wenn Die Zeichen nicht übereinstimmen, dann wir gesucht, ob das Element des durchsuchten Strings im gesuchten String enthalten ist. Wenn ja, dann wird der gesuchte String so verschoben, dass die Elemente untereinander Stehen, und fängt wieder von vorne an. Wenn das Zeichen nicht im gesuchten String enthalten ist, wird der gesuchte String um n Stellen verschoben. Wenn die Elemente gleich sind, dann überprüft er, ob die anderen auch gleich sind, wenn ja ist der String gefunden, wenn nicht, dann wird so verfahren wie oben beschrieben. Wenn der gesuchte String über den durchsuchten hinaus verschoben wurde, dann ist der String nicht enthalten.

Dieses Programm sucht im String t den String sw und gibt diesen im String t in roter Schrift aus.

```

program boyermoor;
uses crt;
var f :array['a'..'z'] of integer;
    t,sw :string;
    i,lt,ls,swp,st,v :integer;
    i1 :char;
begin
    clrscr;
    readln(t);
    readln(sw);
    lt:=length(t);
    ls:=length(sw);
    for i1:='a' to 'z' do f[i1]:=0;
    for i:=1 to ls do f[sw[i]]:=i;
    swp:=ls;
    st:=ls;
    repeat
        i:=0;
        while t[st]=sw[swp] do
            begin
                st:=st-1;
                swp:=swp-1;
                i:=i+1;
            end;
        if i<>ls then
            begin
                swp:=swp+i;
                v:=ls-f[t[st]];
                st:=st+v+i;
            end;
    until (i=ls) or (st>lt);
    if i=ls then
        begin
            gotoxy(st+1,1);
            textcolor(red);
            write(sw);
        end
    else
        writeln('Nicht drinne!');
    readkey;
end.

```

6 Dateiarbeit

6.1 Unypisierte Datei

Deklaration: Var name: file;

Eine untypisierte Datei verhält sich wie eine typisierte. Jedoch ist zu beachten, dass die Datensätze unterschiedlich groß in der Datei sein können. Somit sollte man die Reihenfolge der Datensätze kennen.

6.2 Typisierte Datei

Deklaration

Var (logische Datei): file of (Datentyp);

Die logische Datei ist die im Program verwendete Variable.

Der Datentyp gibt an, welcher Variablentyp (string, char, integer) in die Datei gespeichert werden kann.

Um einen einzelnen Datensatz zu löschen gibt es keinen direkten Befehl. Hier muss die Datei ohne das zu löschende Element in eine andere geschrieben werden. Dazu legt man eine neue Datei an und durchläuft die Alte, um alle benötigten Datensätze in die neue Datei zu schreiben. Danach muss noch die neue Datei in die alte Datei umbenannt werden. Dazu müssen beide zuerst geschlossen werden und die alte Datei gelöscht.

Das Program löscht einen einzelnen Datensatz aus einer Datei. Hier wird die Zahl x aus der Datei f gelöscht. Die Datei f ist im Hauptprogram als integer Datei deklariert und ihr Pfad sei p und n der name.

```

procedure losch(x:integer);
  Var a,k:integer;
  Var g:file of integer;
  begin
    reset(f);
    assign(g,p+n+'2.doc');
    reset(g);
    rewrite(g);
    k:=0;
    while (not eof(f)) or (k=0); do
      begin
        read(f,a);
        Ifx=a then
          k:=1
        else
          write(g,a);
        end;
      while not eof(f) do
        begin
          read(f,a);
          write(g,a);
        end;
      close(f);
      close(g);
      erase(f);
      rename(g,p+n+'.txt');
    end;
  end;

```

6.3 Textdatei

Deklaration: Var name:text;

7 dynamische Listen

Bei einer Liste verweist ein Zeiger auf einen Datenverbund(Record) mit einem oder mehreren Werten und einem oder mehreren weiteren Zeigern.

Der Stack ist aufgebaut wie eine Liste nach dem LIFO-Prinzip (Last in, first out). Hier werden Elemente vorne in die Liste eingefügt und auch vorne wieder herausgenommen.

Eine Schlange ist das gegenteilige Prinzip, FIFO (First in, first out). Hinten wird angefügt, vorne herausgenommen.

Aufbau einer Liste mit n vielen Zufallszahlen. Die Deklaration geschieht wie oben beschrieben.

```

Type zeigertyp=^Knoten;
  knoten=Record
    wert:integer;
    next:zeigertyp;
  end;
Var a,n,start:zeigertyp;

```

7.1 Aufbau einer Liste

Aufbau einer Liste mit n vielen Zufallszahlen. Die Deklaration geschieht wie oben beschrieben.

```
new(start);
start.wert:=random(10);
start.next:=nil;
  a:=start;
for i:=2 to n do
  begin
    new(n);
    n.wert:=random(10);
    n.next:=nil;
    a.next:=n;
    a:=n;
  end;
```

7.2 Löschen in einer Liste

Hier brauch mann eine Fallunterscheidung. Wenn der erste Wert der gesuchte ist, dann wird der Startzeiger weitergesetzt. Wenn es nicht das erste Element ist, dann wird die Liste durchlaufen. Wenn nach dem aktuellen Zeiger der gesuchte Wert kommt, wird der next-Zeiger vom Aktuellen auf den Next Zeiger vom nächsten Zeicher gesetzt ($a.next := a.next.next$). Der So herausgenommene Zeiger muss vorher gemerkt werden, damit man ihn mit dispose() engültig löschen kann.

Aufgerufen wird die Procedure mit loeschen(a,start); a ist der zu löschende Wert und start der Startzeiger der Liste. Die Liste ist wie oben beschrieben deklariert.

```
procedure loeschen(a:integer; Var start:zeigertyp);
  var h,b: zeigertyp;
      k: char;
  begin
    clrscr;
    while start.wert=a do
      begin
        h:=start;
        start:=start.next;
        dispose(h);
      end;
    b:=start;
    repeat
      k:='0';
      while (not (b.next.wert=w)) and (not (b.next=nil)) do
        b:=b.next;
      if b.next.wert=w then
        begin
          h:=b.next;
          b.next:=b.next.next;
          dispose(h);
          k:='1';
        end;
    until k='0';
  end;
```

7.3 Aufbau einer Liste

Aufbau einer Liste mit n vielen Zufallszahlen. Die Deklaration geschieht wie oben beschrieben.

```
new(start);
start.wert:=random(10);
start.next:=nil;
  a:=start;
for i:=2 to n do
  begin
    new(n);
    n.wert:=random(10);
    n.next:=nil;
    a.next:=n;
    a:=n;
  end;
```

7.4 doppelt verkettete Liste

Hier wird nicht nur ein next, sondern auch ein pre Zeiger mit in den Record geschrieben. Der Pre-Zeiger zeigt auf den Vorgänger. Somit ist es einfacher, mit der Liste zu arbeiten.

7.5 Ringliste

Hier aht die Liste weder Anfang noch Ende. Das "letzte" Element hat in seinem next-Zeiger das "erste" Element.

7.6 Liste in Liste

Hier werden zwei Zeigertypen deklariert. Der eine Zeiger ist für die Hauptliste, von der wiederum gehen in jedem Zeiger weitere Listen ab.

Deklaration einer Liste aus Listen.

```
type
  zeiger2=knoten2;
  knoten2=record
    wert:string;
    next:zeiger2;
  end;
  zeiger=knoten;
  knoten=record
    wert:char;
    next:zeiger;
    list:zeiger2;
  end;
```

8 Objektorientierte Programmierung

Bei der Objektorientierten Programmierung steht die Datenstruktur im Mittelpunkt. Die einzelnen Objekte haben eigene Algorithmen zu Verfügung. Die Grundlegenden Prinzipien sind Abstraktion, Kapselung, Modularisierung und Hierarchie.

Teil III

Datenstrukturen

9 Baumstrukturen

Bäume sind hierarchische Datenstrukturen bei denen genau ein Element keinen Vorgänger hat (Wurzel) und jedes weitere Element genau einen Vorgänger (Vater) und endlich viele Nachfolger (Söhne) hat. Elemente ohne

Nachfolger werden Blätter genannt.

Höhe eines Baumes: Maximale Anzahl an Kanten von der Wurzel zum Blatt.

Tiefe eines Knotens: Die Anzahl der Kanten von der Wurzel zum Knoten.

In Pascal werden Bäume ähnlich wie Listen deklariert (Siehe 7 Seite 14). Hier muss nur anstelle von nur einem next- Zeiger je einer für jeden Sohn deklariert werden, bei Binären Bäumen also rechts und links.

9.1 Binärer Baum

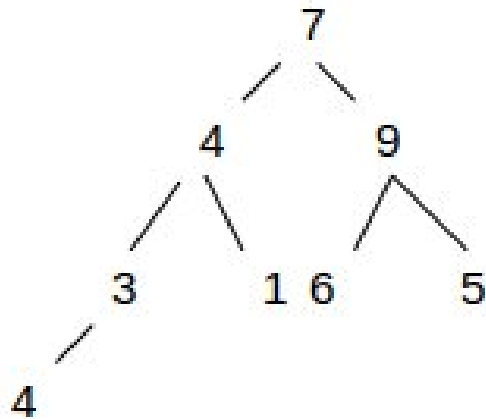
Der Binäre Baum hat an jedem Knoten maximal 2 Söhne.

vollständiger Baum: Hat an jedem Knoten die maximale Anzahl Söhne.

balancierter Baum: in jedem Knoten unterscheidet sich die Höhe der Teilbäume um maximal 1.

Es gibt verschiedene Möglichkeiten, Elemente in einen Binären Baum einzuordnen.

Level Order 7-4-9-3-1-6-5-4



Die Elemente werden von links nach recht, Ebene für Ebene eingeordnet

PRE-Order (WLR) 7-4-3-4-1-9-6-5

Die Procedure wird mit `aus(a);` aufgerufen. `a` ist der Zeiger, welcher auf die Wurzel des auszugebenden Baumes zeigt.
Zeiger ist wie oben beschrieben deklariert.

```

procedure aus(a:zeiger);
begin
  If not(a=nil) then
    begin
      write(a^.w:4);
      aus(a^.l);
      aus(a^.r);
    end;
  end;
end;
  
```

IN-Order (LWR) 4-3-4-1-7-6-9-5

Die Procedure wird mit `aus(a)`; aufgerufen. `a` ist der Zeiger, welcher auf die Wurzel des auszugebenden Baumes zeigt.
Zeiger ist wie oben beschrieben deklariert.

```
procedure aus(a:zeiger);
begin
  If not(a=nil) then
    begin
      aus(a^.l);
      write(a^.w:4);
      aus(a^.r);
    end;
end;
```

POST-Order (LRW) 4-3-1-4-6-5-9-7

Die Procedure wird mit `aus(a)`; aufgerufen. `a` ist der Zeiger, welcher auf die Wurzel des auszugebenden Baumes zeigt.
Zeiger ist wie oben beschrieben deklariert.

```
procedure aus(a:zeiger);
begin
  If not(a=nil) then
    begin
      aus(a^.l);
      aus(a^.r);
      write(a^.w:4);
    end;
end;
```

9.1.1 binärer Suchbaum

Bei einem Binären Suchbaum ist der linke Sohn kleiner als der rechte. somit ist der rechte Sohn größer oder gleich dem Vater. Wenn diese Struktur gegeben ist, können die Elemente In-Order ausgegeben werden und sie sind Sortiert. In dem Baum selber ist es sehr leicht, einzelne Elemente zu suchen.

Die Procedure `wib(w,z)`; schreibt das Element `w` in die Liste mit der Wurzel `z`. Dies geschieht rekursiv und der nach dem Prinzip des Suchbaums. Wenn man nun alle Elemente eines Baumes mit dieser Prozedure einließt, dann ist der Baum nach dem Suchbaum geordnet.

```
procedure wib(w: integer; var z: zeigertyp);
begin
  if z=nil then
    begin
      new(z);
      z^.wert:=w;
      z^.l:=nil;
      z^.r:=nil;
    end
  else
    if w<z^.wert then
      wib(w,z^.l)
    else wib(w,z^.r);
  end;
```

9.1.2 Huffmanbaum

Der Huffmanbaum ist ein Häufigkeitsbaum. Er wird für die Huffman Codierung genutzt. diese Komprimiert Texte. Alle im Text vorkommenden Zeichen werden aufgelistet und ihre Häufigkeit dazugeschrieben. Nun werden immer die Zwei Zeichen mit der wenigsten Häufigkeit zusammengefasst. Die Zeichenanzahl zusammengefasst und für den weiteren Verlauf mitbenutzt. Damit entsteht ein Baum, mit welchem man den Text komprimieren kann. Der Neue Code für die einzelnen Buchstaben besteht jetzt nichtmehr aus dem Assiccode, sondern wird aus

dem Baum gewonnen. Die Äste bekommen jetzt die Zahlen 0 und 1. Links immer 0, rechts 1. Die Pfade bis zu den Buchstaben sind dann die neuen Codes. Hier kann eine sehr große Komprimierung erreicht werden. Jedoch muss die Tabelle für die Codierung mitgeliefert werden, wodurch dies erst bei großen Datenmengen sinnvoll ist.

9.1.3 Heap-geordneter Baum

Bei der Heapordnung sind beide Söhne kleiner als der Vater. Hierzu wird der Baum ebenenweise von unten nach oben durchlaufen, in der jeweiligen Ebene von rechts nach links. der größere der beiden jeweiligen Söhne wird mit dem Vater verglichen und gegebenenfalls gelöscht. Das wird sooft wiederholt, bis die Heapordnung hergestellt ist.

Heapsort basiert auf den Heap-geordneten Baum (siehe 4.6 Seite 11).

10 Graphen

10.1 Graphentheorie

Def.: Ein Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, welche je zwei Knoten miteinander verbinden.

10.1.1 Begriffe

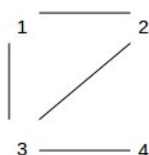
schlichter Graph	Besitzt keine Mehrfachkanten und keine Schlingen.
Weg:	Beschreibt eine Folge Knoten (1 - 2 - 3).
Kantenzug:	Beschreibt eine Folge von Kanten ([1;2] - [2;3]).
Kette	Beschreibt eine Folge von Knoten und Kanten (1 - [1;2] - 2 - [2;3] - 3).
Kreis	Beschreibt eine Folge von Kanten, bei der der Ausgangspunkt wieder erreicht wird und jede Kante nur 1 mal vorhanden ist.
Eulerkreis	Beschreibt einen Kreis, bei dem Jede Kante des Graphens genau 1 mal vorhanden ist.
Zyklus	Beschreibt eine Folge von Knoten, bei der der Ausgangspunkt wieder erreicht wird und jeder Knoten nur 1 mal vorhanden ist.
Hamilton-Zyklus	Beschreibt einen Zyklus bei dem Jeder Knoten genau 1 mal vorhanden ist.
zusammenhängender Graph	Besitzt keine Isolierten Knoten.
vollständiger Graph	Beschreibt einen Graphen bei dem jeder Knoten mit jedem anderen Knoten des Graphen verbunden ist.
spannender Graph	Beschreibt einen zusammenhängenden Graphen mit minimalen Kanten.
Grad eines Knoten	Beschreibt die Anzahl der an dem Knoten hängenden Kanten.
Schnittpunkt	Beschreibt einen Knoten bei dessen Entfernung der Graph in nicht zusammenhängende Teilgraphen zerfällt.
Brücke	Beschreibt eine Kante bei dessen Entfernung der Graph in nicht zusammenhängende Teilgraphen zerfällt.

10.1.2 Darstellung

mathematische Darstellung

$$\begin{aligned} \text{Knoten} &= \{1, 2, 3, 4\} \\ \text{Kanten} &= \{[1;2], [1;3], [2;3], [3;4]\} \end{aligned}$$

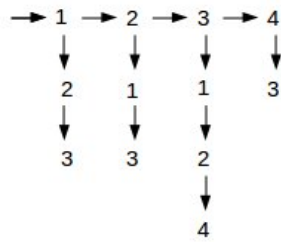
graphische Darstellung



Adjazenzmatrix

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0

Adjazenzliste



10.2 Spezialfälle

10.2.1 gerichteter Graph

Bei einem gerichteten Graphen besitzen die Kanten eine Ausrichtung, welche durch einen Pfeil gekennzeichnet wird. Beim Grad eines Knotens gibt es nun Außengrad, die von dem Knoten wegführende, und Innengrad, die zum Knoten hinführenden Kanten.

10.2.2 gewichteter Graph

Bei einem Gewichteten Graphen besitzen die Kanten eine Wichtung, diese werden an die Kante rangeschrieben. Bei dem Algorithmus von Dijkstra wird ein gewichteter Graph benutzt.

10.3 Algorithmus von Dijkstra

Hier wird der kürzeste Weg in einem gewichteten Graphen gesucht. Dafür werden alle möglichen Wege ausprobiert. Wenn ein Weg zu einem beliebigen Punkt länger ist als ein anderer Weg zu dem gleichen Punkt, wird dieser Weg gelöscht und mit den kürzeren weitergemacht. Somit wird der Aufwand verringert.

Die Procedure `pathfinder(i1,i2)`; sucht von einem Punkt in einer Matrix mit den Koordinaten `i1` und `i2` den kürzesten weg zu dem Punkt mit den Koordinaten `zu1`, `zu2`. Dieser Weg wird in eine Liste des Hauptprogramms gespeichert. Die Matrix steht in einem 3 Deimensionalen Feld. in der zweiten Ebene der dritten Dimension ist der jeweilige Weg vom Anfangspunkt zu diesem Gespeichert.

```

procedure pathfinder(i1,i2:integer);
var n:zeigertyp;
begin
  n:=start;
  if not ((i1=zu1) and (i2=zu2)) then
  begin
    while n<>nil do
    begin
      if n^.wert.von=a[i1,i2,1] then
      begin
        for ii1:=1 to 5 do
        for ii2:=1 to 5 do
        if a[ii1,ii2,1]=n^.wert.zu then
        begin
          i11:=ii1;
          i22:=ii2;
        end;
        if a[i1,i2,2]+n^.wert.l<a[i11,i22,2] then
        begin
          a[i11,i22,2]:=a[i1,i2,2]+n^.wert.l;
          pathfinder(i11,i22);
        end;
        end;
        n:=n^.next;
      end;
    end;
  end;
end;

```

Teil IV

Prolog

In Prolog werden hier nur einige Beispielprogramme aufgeführt.

11 Rätsel

11.1 Zahlenrätsel

Bei Zahlenrätseln oder ähnlichen Problemen werden alle Bedingungen in eine Regel geschrieben. Hier, welche Zahlen nicht gleich sein dürfen und wie sich die Zahlen zueinander verhalten. Prolog probiert alle Varianten aus und findet alle Möglichkeiten.

Dieses Programm löst das Zahlenrätsel

$$\begin{array}{rcccc}
 & & A & B & A & C \\
 + & & G & F & E & D \\
 \hline
 = & F & H & D & F & H
 \end{array}$$

```

z(0).
z(1).
z(2).
z(3).
z(4).
z(5).
z(6).
z(7).
z(8).
z(9).
loesung(A,B,C,D,E,F,G,H):-z(C),z(D),CD,H is (C+D) mod 10,U1 is (C+D) // 10,HC,HD,
                        z(A),AC,AD,AH,z(E),EC,ED,EH,EA,F is (A+E+U1) mod 10,
                        U2 is (A+E+U1) // 10, FC,FD,FH,FA,FE,
                        z(B),BC,BD,BH,BA,BE,BF,D is (B+F+U2) mod 10,U3 is (B+F+U2) // 10,
                        z(G),GC,GD,GH,GA,GE,GF,GB,H is (A+G+U3) mod 10, F is (A+G+U3) // 10.
?- loesung(A,B,C,D,E,F,G,H),write(" "),write(A),write(B),write(A),write(C),nl,
                        write(" +"),write(G),write(F),write(E),write(D),nl,
                        write(" ="),write(F),write(H),write(D),write(F),write(H).

```

11.2 Das Kohlkopf - Ziege - Wolf Problem

Das Problem sollte jedem bekannt sein. Prolog geht ähnlich vor wie bei den Zahlenrätseln. Er probiert alles aus, was nicht möglich ist, wird nicht weiter verfolgt. Da die Bedingungen hier jedoch komplexer sind, können diese jedoch nicht in eine Regel geschrieben werden.

```

illegal(z(B,W,Z,K)):-Z=K,gegenueber(B,Z).
illegal(z(B,W,Z,K)):-W=Z,gegenueber(B,W).

gegenueber(n,s).
gegenueber(s,n).

erlaubt(z(B,W,Z,K)):-not(illegal(z(B,W,Z,K))).

rudern(z(B1,W,Z,K),z(B2,W,Z,K)):-gegenueber(B1,B2),erlaubt(z(B1,W,Z,K)),
                                     erlaubt(z(B2,W,Z,K)).
rudern(z(B1,B1,Z,K),z(B2,B2,Z,K)):-gegenueber(B1,B2),erlaubt(z(B1,B1,Z,K)),
                                     erlaubt(z(B2,B2,Z,K)).
rudern(z(B1,W,B1,K),z(B2,W,B2,K)):-gegenueber(B1,B2),erlaubt(z(B1,W,B1,K)),
                                     erlaubt(z(B2,W,B2,K)).
rudern(z(B1,W,Z,B1),z(B2,W,Z,B2)):-gegenueber(B1,B2),erlaubt(z(B1,W,Z,B1)),
                                     erlaubt(z(B2,W,Z,B2)).

ausgabe([K]):-schreibt(K,z(s,s,s,s)).
ausgabe([K,K2|R]):-schreibt(K,K2),ausgabe([K2|R]).

schreibt(z(B,W,Z,K),z(B1,W,Z,K1)):-gegenueber(K,K1),write("Bauer rudert mit Kohlkopf von "),
                                                         write(B),write(" nach "),write(B1),nl.
schreibt(z(B,W,Z,K),z(B1,W,Z1,K)):-gegenueber(Z1,Z),write("Bauer rudert mit Ziege von "),
                                                         write(B),write(" nach "),write(B1),nl.
schreibt(z(B,W,Z,K),z(B1,W1,Z,K)):-gegenueber(W1,W),write("Bauer rudert mit Wolf von "),
                                                         write(B),write(" nach "),write(B1),nl.
schreibt(z(B,W,Z,K),z(B1,W,Z,K)):-write("Bauer rudert alleine von "),write(B),
                                                         write(" nach "),write(B1),nl.

anhang(Y,[],[Y]).
anhang(Y,[K|R],[K|X]):-anhang(Y,R,X).

element(X,[X|R]).
element(X,[K|R]):-element(X,R).

verlauf(z(s,s,s,s),Ws):-ausgabe(Ws).
verlauf(X,Ws):-anhang(X,Ws,Ws1),rudern(X,Z),not(element(Z,Ws1)),verlauf(Z,Ws1).

?-verlauf(z(n,n,n,n),[]).
```

12 Datenbanken

Hier werden die Daten als Regeln aufgeschrieben. Aus diesen lassen sich durch weitere Regeln alle relevanten Fakten herauslesen.

12.1 Büchereri

Die Regeln können nach beliben erweitert werden.

```
buch(1,autor(thilo,sarrazin),'Deutschland schafft sich ab').
buch(2,autor(adolf,hitler),'Mein Kampf - Sonderausgabe').
buch(3,autor(charlotte,roche),'Feuchtgebiete').
buch(4,autor(lambacher,schweizer),'Analytische Geometrie mit linearer Algebra - Leistungskurs').
buch(5,autor(vatsyayana,mallanaga),'Kamasutra').
buch(6,autor(dreifaltiger,gott),'Die Bibel').
buch(7,autor(marion,blisse),'Duden - Deutsche Rechtschreibung').
buch(8,autor(dreifaltiger,gott),'Die 10 Gebote').
buch(9,autor(bill,gates),'Notebook').
buch(10,autor(steve,jobs),'Mac Book Pro').
```

```
genre(1,'Politische Literatur').
genre(2,'Propagandaliteratur').
genre(3,'Erotik').
genre(4,'Fachliteratur').
genre(5,'Erotik').
genre(6,'Kirchliche Literatur').
genre(7,'Fachliteratur').
genre(8,'Kirchliche Literatur').
genre(9,'Computerliteratur').
```

```
ausgeliehen(1,7,278,292).
ausgeliehen(3,6,270,284).
ausgeliehen(5,8,270,277).
ausgeliehen(2,7,273,287).
ausgeliehen(10,3,260,284).
```

```
mitglied(1,name(paul,paulsen)).
mitglied(2,name(peter,pan)).
mitglied(3,name(dreifaltiger,gott)).
mitglied(4,name(heino,fertig-aus)).
mitglied(5,name(rock,fertig-aus)).
mitglied(6,name(michael,jackson)).
mitglied(7,name(dunkler,lord)).
mitglied(8,name(weihnachts,mann)).
mitglied(9,name(niko,laus)).
mitglied(10,name(rosa,schlüpfer)).
```

mahnung(*T*, *X*, *Y*) : -*ausgeliehen*(*M*, *A*), *T* > *A*, *mitglied*(*M*, *name*(*X*, *Y*)).

weg(*T*, *N*) : -*buch*(*X*, *N*), *ausgeliehen*(*X*, *Y*, *Z*), *Z* >= *T*, *Y* = < *T*, *write*('Buch ausgeliehen!').

?-*mahnung*(290,*X*,*Y*),*write*(*X*),*write*(' '),*write*(*Y*).

12.2 Bundeskanzler

Ein weiteres Beispiel für eine Datenbank.

```
kanzler(adenauer,1949,1963).
kanzler(erhard,1963,1966).
kanzler(kiesinger,1966,1969).
kanzler(brandt,1969,1974).
kanzler(schmidt,1974,1982).
kanzler(kohl,1982,1998).
kanzler(schroeder,1998,2005).
```

regiert(*X*, *Jahr*) : -*kanzler*(*X*, *A*, *B*), *A* = < *Jahr*, *Jahr* = < *B*.

dauer(*X*, *Zeit*) : -*kanzler*(*X*, *A*, *B*), *Zeit* = *B* - *A*.

?-*dauer*(*X*, *Y*), *write*(*X*), *write*(' regierte '), *write*(*Y*), *write*(' Jahre').

13 Rekursion

Rekursion ist bei Prolog genauso anwendbar wie bei Pascal (Siehe 3.3 Seite 7). Ein gutes Beispiel gibt dieses Programm.

Das Program berechnet Rekursiv die Fakultät von 5 und gibt diese aus.

```
fak(0,1).
fak(X,Y):-X>0,Z is X - 1,fak(Z,W),Y is X * W.
?-fak(5,Y),write(Y).
```

Hier kann man gut die einzelnen Teile der Rekursion erkennen. Die Regel fak(0,1). ist die Abbruchbedingung. Das heißt, wenn das Programm die Fakultät von 0 errechnen will, was die einfachste Variante ist, gibt es 0 zurück. In der zweiten Regel wird die Fakultät für alle Zahlen größer als 0 errechnet. Das geschieht, indem das Programm sich neu aufruft und die Fakultät von der eins kleineren Zahl ausrechnet, dies geschieht, bis der Einfachste Fall erreicht ist. Beim Aufstieg wird die jeweils neue Fakultät ausgerechnet und zurückgegeben.

14 Listen

Listen sind in Prolog deutlich einfacher zu generieren wie in Pascal.

14.1 Grundlagen

Hier stehen ein paar grundlegende Möglichkeiten, um mit Listen zu arbeiten. Sie sollen nur eine kleine Einführung in die Bedienung geben.

Die namen der Regeln geben die Verwendung an. Listen werden in Eckigen Klammern geschrieben und die Elemente mit “,” getrennt.

```
erstes([K|R],K).

zweites([K,X|R],X).

letztes([K],K).
letztes([K|R],X):-letztes(R,X).

vorletztes([K,X],K).
vorletztes([K|R],X):-vorletztes(R,X).

loesch(Z,[ ],[ ]).
loesch(Z,[Z|R],L):-loesch(Z,R,L).
loesch(Z,[K|R],[K|Q]):-Z\=K,loesch(Z,R,Q).

anhang([ ],Y,Y).
anhang([K|R],L,[K|X]):-anhang(R,L,X).

lang([ ],0).
lang([K|R],X):-lang(R,Y),X is Y+1.

max([X],X).
max([K|R],K):-max(R,Y),K>Y.
max([K|R],X):-max(R,X),X>K.
```

14.2 Sortieren

Hier ein paar sortierverfahren in Prolog umgesetzt. Die Grundlegenden Funktionsweisen sind bei Pascal beschrieben (Siehe 4 Seite 7).

Mit Listen.

```
quicksort([ ],[ ]).
quicksort([K|R],S):- zerlegen(R,K,KL,GL),quicksort(KL,KSL),quicksort(GL,GSL),
    anhang(KSL,[K|GSL],S).
zerlegen([ ],X,[ ],[ ]).
zerlegen([K|R],X,[K|KL],GL):-K<X,zerlegen(R,X,KL,GL).
zerlegen([K|R],X,KL,[K|GL]):-K>=X,zerlegen(R,X,KL,GL).

bubble(L,L):-sortiert(L),!.
bubble([K,Z|R],X):-K>Z,bubble([K|R],Y),bubble([Z|Y],X).
bubble([K,Z|R],X):-K<=Z,bubble([Z|R],Y),bubble([K|Y],X).
sortiert([X]).
sortiert([K,X|R]):-K<=X,sortiert([X|R]).

select([X],[X]).
select(L,[X|S]):-kleinstes(L,X,LOX),select(LOX,S).
kleinstes([X],X,[ ]).
kleinstes([K|R],K,R):-kleinstes(R,X,ROX),K<X.
kleinstes([K|R],X,[K|ROX]):-kleinstes(R,X,ROX),X<K.

insert([ ],[ ]).
insert([K|R],S):-insert(R,X),einfuegen(K,X,S).
einfuegen(K,[ ],[K]).
einfuegen(K,[Y|X],[K,Y|X]):-K<Y.
einfuegen(K,[Y|X],[Y|S]):-K>=Y,einfuegen(K,X,S).
```

Teil V

Automatentheorie

Ein Automat ist ein System, was auf eine Eingabe eine Ausgabe generiert, welche von der Eingabe und vom Momentanen Zustand abhängig ist. Ein Automat besitzt eine endliche Menge an Eingabezeichen, sowie Ausgabezeichen. Zudem besitzt er auch eine endliche Menge an Zuständen, z.B. Start- und Endzustand. Die Zustände, oder Übergangsfunktionen ermittelt einen neuen Zustand in Abhängigkeit von Eingabezeichen und aktueller Funktion. Die Ausgabefunktion ermittelt die dazugehörige Ausgabe.

15 Kellerautomaten

Er besitzt neben dem Eingabeband auch ein Kellerband. Dieses Arbeitet nach dem Lifo-Prinzip¹ (Stack). Es kann immer nur das oberste Element des Kellerbandes gelesen werden. Es gibt drei Grundoperationen für das Kellerband sind:

Push - Wert in Keller

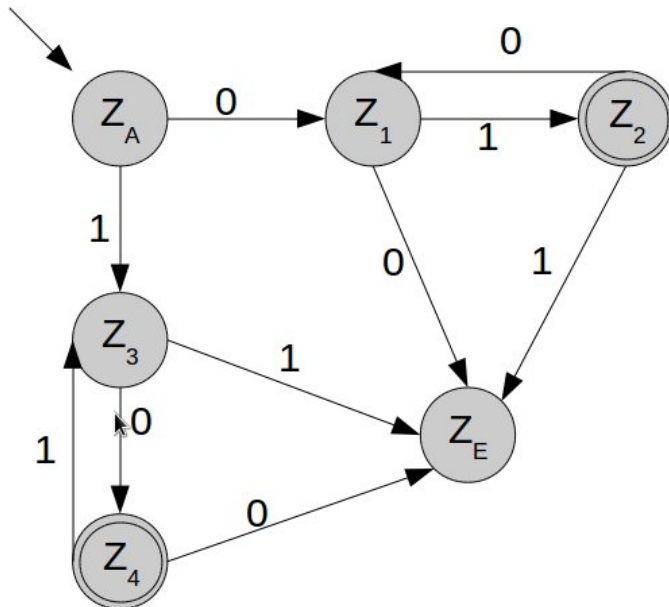
Pop - Wert aus Keller

Top - Überprüfen des obersten Wertes

16 erkennende Automaten

Erkennende Automaten haben keine Ausgabe. Sie können entweder eine Eingabe erkennen, oder nicht erkennen. Das signalisieren sie durch einen Erkennungszustand. Als Beispile ist ein erkennender Automat gegeben, welcher Reihen von 0 und 1 erkennt, jedoch nur, wenn von beiden die gleiche Anzahl in der Folge enthalten sind, und immer abwechselnt.

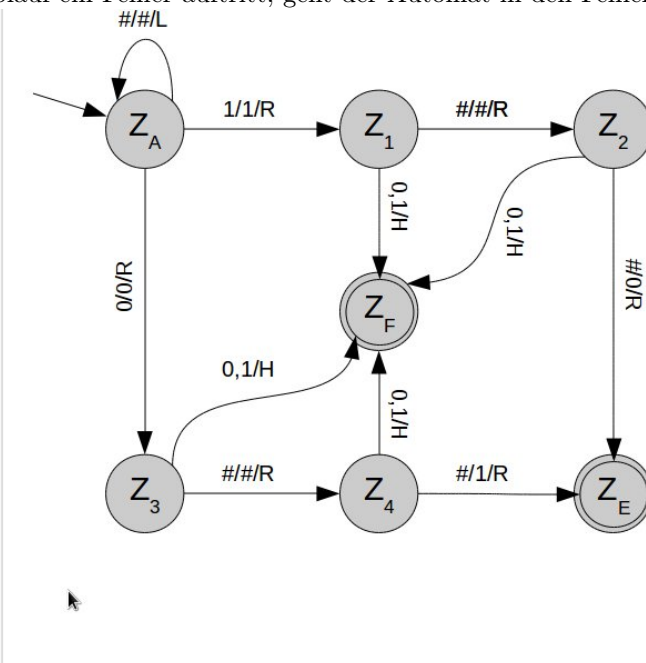
¹Last-In - first-out



17 Turing Machine

Die Turing Maschine wurde 1936 von Alan M. Turing erfunden. Eine Turingmaschine hat ein Eingabeband, auf dem sich der Lesekopf hin und her bewegen kann. Außerdem kann die Turingmaschine auf das Band schreiben.

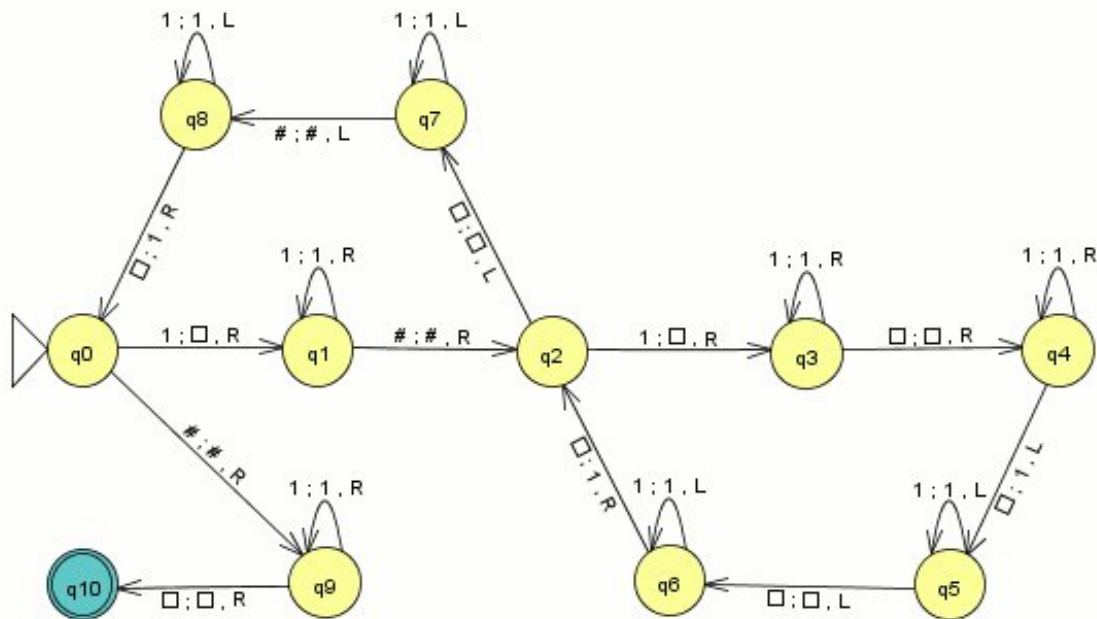
Auf dem Eingabeband steht eine Zahl in Binärschreibweise. Der Automat soll zwischen einer geraden und einer ungeraden unterscheiden. Wenn sie gerade ist, soll er eine 0 hinter die Zahl schreiben, wenn nicht eine 1. Die Eingabezeichen sind 0,1,# und die Ausgabezeichen ebenfalls. Die Startposition des Zeigers ist rechts neben der Zahl. Die Endposition bei dem Zustand Z_E ist rechts neben der neu geschriebenen 0 bzw 1. Wenn bei dem Ablauf ein Fehler auftritt, geht der Automat in den Fehlerzustand Z_F und hält den Zeiger.



17.1 universelle Turing-Maschine

Eine universelle Turing-Maschine hat auf dem Eingabeband nur die Zahlen 0 und 1. Dabei ist 0 das Trennzeichen. Die 1 verschlüsselt die Werte. So ist eine 0 eine 1, eine 1 eine 11. Dieses Zahlensystem wird auch Unär-system genannt.

Als Beispiel ist ein Unär-Multiplizierer gezeigt. Beim Anfangszustand sind beide Operatoren durch eine 0 getrennt auf dem Band. Der Lesekopf steht am Anfang des ersten Operanden. Beim Endzustand steht das Ergebniss hinter den Operatoren und der Kopf steht auf dem ersten Zeichen vom Ergebniss.

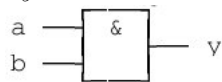


Teil VI

Schaltalgebra

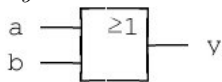
18 Grundsaltungen

Und ergibt WAHR, wenn alle Komponenten WAHR sind.
Schaltfunktion: $y = a \cdot b$



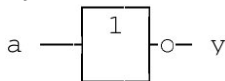
Schaltsymbol:

Oder ergibt WAHR, wenn mindestens eine Komponente WAHR ist.
Schaltfunktion: $y = a \vee b$



Schaltsymbol:

Nagation kehrt den Eingang um.
Schaltfunktion: $y = \bar{a}$



Schaltsymbol:

19 Umstellen von Schaltgleichungen

19.1 Gesetzte

Kommutativgesetz

$$\begin{aligned} a \cdot b &= b \cdot a \\ a \vee b &= b \vee a \end{aligned}$$

Assoziativgesetz

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$$

Distributivgesetz

$$a \cdot (b \vee c) = (a \cdot b) \vee (a \cdot c)$$

Absorbtionsgesetz

$$\begin{aligned} a \cdot (a \vee b) &= a \\ a \vee (a \cdot b) &= a \end{aligned}$$

neutrale Elemente

$$\begin{aligned} a \cdot 1 &= a \\ a \vee 0 &= a \end{aligned}$$

komplementäre Elemente	$\left \begin{array}{l} a\widehat{a} = 0 \\ a^{\vee}\overline{a} = 1 \end{array} \right.$
Gesetz von de Morgan	$\left \begin{array}{l} \overline{a\widehat{b}} = \overline{a}^{\vee}\overline{b} \\ \overline{a^{\vee}b} = \overline{a}\widehat{b} \end{array} \right.$

19.1.1 Disjunktive Normalform (DNF)

Wird benutzt, um aus Schalttabellen günstig gekürzte Schaltgleichungen zu erstellen. Als Faustregel gilt, wenn sich in zwei Klammern nur ein Element unterscheidet (a und \overline{a}), dann können diese zusammengefasst werden und das unterschiedliche Element ausgespart werden. Dabei kann jede Klammer öfter benutzt werden.

$$y = (... \widehat{...} ... \widehat{...} ...) \vee (... \widehat{...} ... \widehat{...} ...) \vee (... \widehat{...} ... \widehat{...} ...) \vee ...$$

19.1.2 konjunktive Normalform (KNF)

$$y = (... \vee .. \vee ... \vee ...) \widehat{(... \vee .. \vee ... \vee ...)} \widehat{(... \vee .. \vee ... \vee ...)} \widehat{(... \vee .. \vee ... \vee ...)} \widehat{...}$$

19.2 Schaltungsentwicklung

19.2.1 Schaltwerttabelle

a	b	c	y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

19.2.2 DNF

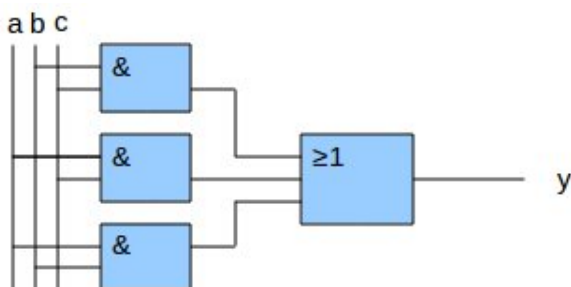
$$y = (\overline{a}\widehat{b}\widehat{c}) \vee (a\widehat{b}\widehat{c}) \vee (a\widehat{b}\widehat{c}) \vee (a\widehat{b}\widehat{c})$$

$$y = (\overline{a}bc) \vee (a\overline{b}c) \vee (ab\overline{c}) \vee (abc)$$

19.2.3 Kürzen

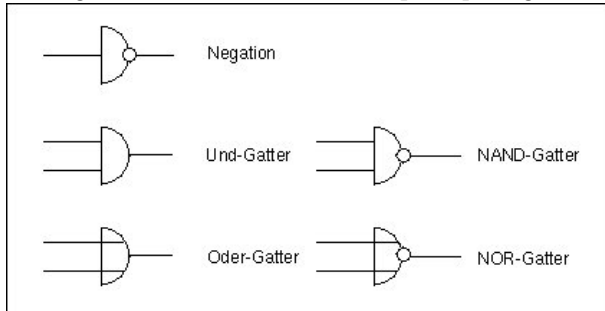
$$y = (bc) \vee (ac) \vee (ab)$$

19.2.4 Schaltung

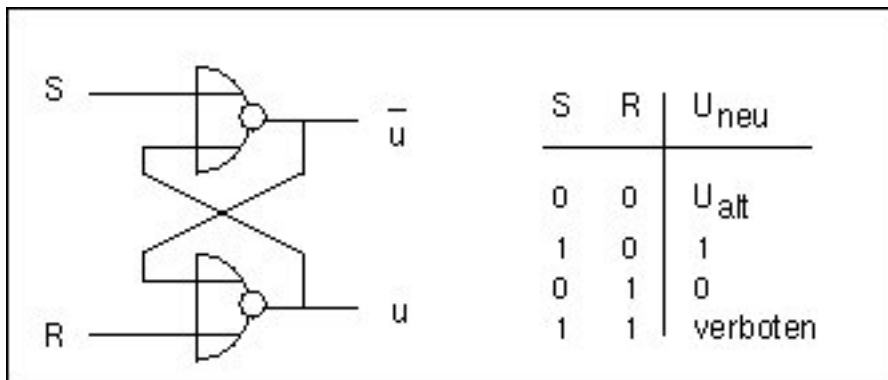


20 Flip-Flops

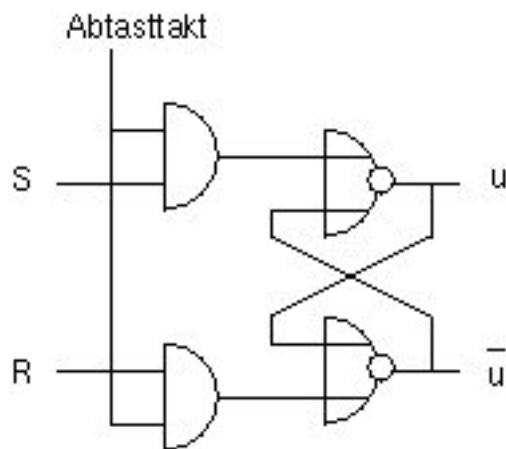
Im folgenden sind die einzelnen Flip-Flops abgebildet. Die Schaltsymbole richten sich nach folgenden Vorgaben:



20.1 Grund Flip-Flop



20.2 Getakteter Flip-Flop



21.2 Von Dezimal zu Binär

	Div2							
211	105	52	36	13	6	3	1	0
mod2	1	1	0	0	1	0	1	1

Die Zahl muss umgedreht werden, dann hat man die Dezimal in eine Binär Zahl umgewandelt.

21.3 Rechnen mit Binärzahlen

21.3.1 Regeln

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 0 \text{ Übertrag } 1$$

21.3.2 Multiplikation

1	0	1	1	*	1	1	0	1
		1	0	1	1			
			1	0	1	1		
				0	0	0	0	
					1	0	1	1
1	0	0	0	1	1	1	1	1

21.3.3 Subtraktion

Entsteht durch die Addition mit dem Zweierkomplement. Das Zweierkomplement ist die Darstellungsart von Negativen Zahlen.

Faustregel: Von rechts beginnend, alle Ziffern bis einschließlich der ersten 1 unverändert lassen und alle anderen Zahlen Kippen ($0 = 1$; $1 = 0$)

Originalregel: Alle Zahlen Kippen und 1 Addieren.

21.3.4 Kommazahlen

Kommazahlen werden auch mit einem Komma bgetrennt. Nach dem Komma sind die Potenzen ab -1 mit der Basis zwei. Also erste Kommastelle 2^{-1} zweite Kommastelle 2^{-2} , ...

Teil VIII

Bilderverzeichniss

1 Seite 1 <http://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm> (4.4.2011)

17.1 Seite 28 <http://www.hsg-kl.de/faecher/inf/theorie/berechenbar/modelle/turing/grund/multi.gif> (4.4.2011)