

## Inhaltsverzeichnis

### Operatoren

1-2

### Hinweise zur Prüfung u. zu theoretischen Schwerpunkten

3-4

### Anlagen

5-6

## I. Funktionsprinzipien von Hard- und Softwaresystemen

7

### 1. Zahlensysteme

7

### 2. Grundlagen der Schaltalgebra

9

#### 2.1 Logische Grundschaltungen

9

#### 2.2 Schaltfunktionen

10

#### 2.3 Gesetze der Schaltalgebra

10

### 3. Schaltungsanalyse

11

#### 3.1 Wertetabelle

11

#### 3.2 Funktionsgleichung aus Schaltungsdigramm

11

#### 3.3 Funktionsgleichung aus Wertetabelle

12

##### 3.3.1 Disjunktive Normalform

12

##### 3.3.2 Konjunktive Normalform

12

#### 3.4 Systematisches Minimieren von Schaltternnen

13

##### 3.4.1 Vereinfachen von Schaltternnen mit Redezetzen

13

##### 3.4.2 Karnaugh-Veitch-Diagramme

14

### 4. Elektronische Grundschaltungen

15

#### 4.1 Flipflops

15

##### 4.1.1 RS-Flipflop

15

##### 4.1.1.1 Tafelgetriebenes RS-Flipflop

15

##### 4.1.2 D-Flipflop

16

#### 4.2 Anwendungen

16

##### 4.2.1 Addierwerk

16

##### 4.2.2 Entziffern von Schaltern

16

##### 4.2.3 Schieberegister

16

### 5. Endliche Automaten

17

#### 5.1 Anwendungen

18

#### 5.2 Umsetzung eines endlichen Automaten in ein Schaltwerk

19

## II. Werkzeuge und Methoden der Informatik

21

### 1. Lazarus - Delphi - Pascal

21

#### 1.1 Anweisungen und Schlüsselwörter

21

#### 1.2 Bedingungen

21

#### 1.3 Variablen

22

#### 1.4 Konstanten

23

#### 1.5 Datentypen

24

##### 1.5.1 Arrays

24

##### 1.5.2 Records

25

##### 1.5.3 Eigene Typdeklarationen

26

##### 1.5.4 Zeiger

26

#### 1.6 Operatoren

27

#### 1.7 Units

28

#### 1.8 Prozeduren

29

#### 1.9 Funktionen

30

#### 1.10 Integrierte Prozeduren und Funktionen

31

2. Objektorientierte Programmierung	31
2.1 Kapselung	32
2.2 Vererbung	32
2.3 Properties	32
2.4 Implementierung	32
2.5 Klassendiagramme	33
3. Algorithmen	35
3.1 Analyse mit Transitionstabellen	35
3.2 Rekursive Algorithmen	35
3.3 Dynamische Datentypen	36
3.3.1 Verkettete Listen	36
3.3.2 Sortierte Liste	37
3.3.3 Stapel	37
3.3.4 Schleife	37
<b>III. Anwendung von Hard- und Softwaresystemen</b>	<b>38</b>
1. Codierung	38
2. Komprimierung	38
2.1 Graustufencodierung	38
2.2 Huffman-Codierung	38
3. Verschlüsselung	40
3.1 Cäsar-Verschlüsselung	40
3.2 Szytale-Verschlüsselung	40
3.3 Vigenère-Verschlüsselung	41
3.4 Asymmetrische Verschlüsselung	41

# Operatoren

## Liste der Operatoren für das Fach Informatik

S. 1

Operator/Anforderungsbereich	Definition	Beispiele
<b>abschätzen</b> II	durch begründete Überlegungen Größenordnungen angeben	Schätzen Sie das Zeitverhalten des Verfahrens ab, wenn sich die Anzahl der zu bearbeitenden Daten verdoppelt
<b>analysieren / untersuchen</b> II	unter einer gegebenen Fragestellung wichtige Bestandteile oder Eigenschaften nach fachlich üblichen Kriterien herausarbeiten	Analysieren Sie die Funktionsweise des Algorithmus Untersuchen Sie den Algorithmus für die folgenden Beispiele
<b>begründen / zeigen</b> II-III	einen Sachverhalt auf Gesetzmäßigkeiten bzw. kausale Zusammenhänge zurückführen	Begründen Sie die folgenden Aussagen
<b>berechnen</b> I	Ergebnisse durch Rechenoperationen gewinnen	Berechnen Sie die Größe
<b>beschreiben</b> I	Sachverhalte oder Verfahren in Textform unter Verwendung der Fachsprache in vollständigen Sätzen in eigenen Worten wiedergeben (hier sind auch Einschränkungen möglich: Beschreiben Sie in Stichworten...)	Beschreiben Sie das RSA-Verfahren Beschreiben Sie die Syntax ...
<b>bestimmen</b> I-II (s. berechnen)	einen Lösungsweg darstellen und das Ergebnis formulieren	Bestimmen Sie die Anzahl der rekursiven Aufrufe bis zu dem Ergebnis ...
<b>beurteilen</b> II-III (Jährlid)	zu einem Sachverhalt ein selbstständiges Urteil unter Verwendung von Fachwissen und Fachmethoden formulieren und begründen	Beurteilen Sie die gesetzlichen Einschränkungen bei der Verwendung kryptologischer Verfahren Beurteilen Sie die These ...
<b>bewerten</b> II	Sachverhalte, Methoden, Ergebnisse etc. an Beurteilungskriterien messen	Bewerten Sie symmetrische und asymmetrische Verschlüsselung hinsichtlich ... Bewerten Sie die Angemessenheit der Lösung
<b>darstellen</b> I-II	Sachverhalte, Zusammenhänge etc. strukturiert wiedergeben	Stellen Sie Ihr Ergebnis in einer Tabelle dar
<b>diskutieren / erörtern</b> II-III	Argumente zu einer Aussage oder These einander gegenüberstellen und abwägen	Diskutieren Sie beide Sachverhalte aus rechtlicher Sicht. Diskutieren Sie Vor- und Nachteile aus der Sicht des Benutzers
<b>entscheiden</b> II-III	bei Alternativen sich begründet und eindeutig auf eine Möglichkeit festlegen	Entscheiden Sie sich für ein Modell
<b>entwerfen / entwickeln</b> II-III	Nach vorgegebenen Bedingungen eine sinnvolles Konzept selbstständig planen / erarbeiten	Entwerfen Sie eine Datenstruktur... Entwerfen Sie ein ER-Modell Entwickeln Sie einen Algorithmus
<b>ergänzen / vervollständigen / verändern</b> II	eine vorgegebene Problemlösung erweitern	Ergänzen Sie das ER-Modell so, dass ... Ergänzen Sie die Syntaxdiagramme Ergänzen Sie den ADT geeignet
<b>erklären</b> II (erklären + zus. Infos)	einen Sachverhalt durch zusätzliche Informationen veranschaulichen und verständlich machen	Erklären Sie die Funktionsweise von Backtracking
<b>erläutern</b> II	einen Sachverhalt nachvollziehbar und verständlich machen	Erläutern Sie den Nutzen der Methode/Prozedur ... Erläutern Sie die Syntaxdiagramme
<b>erstellen / konstruieren</b> II	bekannte Verfahren zur Lösung eines neuen Problems aus einem bekannten Problemkreis anwenden	Erstellen Sie ein Klassendiagramm Konstruieren Sie einen endlichen Automaten
<b>formulieren / schreiben</b> I-II	einen Sachverhalt / eine Problemlösung in einer fachspezifischen Form darstellen	Formulieren Sie eine SQL-Abfrage Schreiben Sie eine Prozedur / Funktion
<b>implementieren</b> I-II	codieren einer vorliegenden Datenstruktur oder eines vorliegenden Verfahrens	Implementieren Sie diesen Algorithmus
<b>modellieren</b> II-III	Kenntnisse grundlegender Modellierungstechniken zur Problemlösung verwenden	Modellieren Sie das vorgestellte Problem mit Hilfe einer Datenbank

vertauscht?  
s. Chemie-  
Operatoren/  
Mathe/  
Erdkunde

	Definition	Beispiele
<b>nennen / angeben</b> I	ohne nähere Erläuterungen und Begründungen aufzählen	Nennen Sie vier Operationen für den ADT ... Nennen Sie drei weitere Beispiele zu ... Geben Sie den Typ der Grammatik an.
<b>protokollieren</b> I-II	Beobachtungen detailgenau fachsprachlich richtig wiedergeben	Protokollieren Sie den Programmablauf ... (z.B. Trace-Tabelle)
<b>überprüfen</b> II	Sachverhalte an Fakten oder innerer Logik messen und eventuelle Widersprüche oder Lücken aufdecken	Überprüfen Sie die Funktionsweise des Algorithmus
<b>vereinfachen / reduzieren</b> I-II	die Komplexität eines Sachverhalts nach bekannten Regeln verringern	Vereinfachen Sie diesen booleschen Term Reduzieren Sie den endlichen Automaten
<b>verfeinern</b> II	eine vorhandene Struktur präzisieren / ergänzen / erweitern	Verfeinern Sie den Grobentwurf / das Modell
<b>vergleichen</b> II-III	nach vorgegebenen oder selbst gewählten Gesichtspunkten Gemeinsamkeiten, Ähnlichkeiten und Unterschiede ermitteln und darstellen	Vergleichen Sie symmetrische und asymmetrische Verschlüsselung Vergleichen Sie diese Implementation mit ...
<b>zeichnen / graphisch darstellen</b> I-II	die wesentlichen Eigenschaften eines Objektes möglichst übersichtlich in einer Zeichnung darstellen	Zeichnen Sie den Anfang eines Suchbaums Stellen Sie die Kommunikation graphisch dar

# Hinweise zur Abiturprüfung zu thematischen Schwerpunkten

Niedersächsisches Kultusministerium

Juli 2012

## 18. Informatik

### A. Fachbezogene Hinweise

Die Rahmenrichtlinien Informatik sind so offen formuliert, dass sie Raum für die Gestaltung eines zeitgemäßen Informatikunterrichts lassen.

Neue Inhalte der Informatik lassen sich unter die vorgegebenen Unterrichtsinhalte subsumieren. So findet sich in den Rahmenrichtlinien (RRL) z.B. zwar nicht der Begriff „Internet“. Ein Informatikunterricht, in dem das Internet nicht an geeigneten Stellen thematisch Niederschlag findet, ist heute jedoch kaum vorstellbar.

Für die Thematischen Schwerpunkte des Zentralabiturs ergeben sich deshalb die folgenden Konsequenzen:

- Die für die Abiturprüfung verpflichtenden Kerninhalte der RRL und der Einheitlichen Prüfungsanforderungen in der Abiturprüfung Informatik (EPA) bilden die Grundlage für die Aufgabenstellungen des Zentralabiturs.
- Zeitgemäße Abituraufgaben können sich nicht auf in den RRL explizit genannte Inhalte beschränken (vgl. „Internet“).
- Die vorliegenden Thematischen Schwerpunkte beschreiben den stofflichen Umfang der Aufgaben des Zentralabiturs 2015. Sie sollen die Inhalte eines zeitgemäßen Informatikunterrichts widerspiegeln, sind aber nicht so angelegt, dass dadurch die in der Qualifikationsphase zur Verfügung stehende Unterrichtszeit vollständig ausgefüllt wird.

Für Unterricht auf erhöhtem Anforderungsniveau werden in den jeweiligen Themenbereichen Ergänzungen angegeben, die zusätzlich zu den genannten Themen zu behandeln sind.

#### Reihenfolge der Thematischen Schwerpunkte:

Die beiden ersten Thematischen Schwerpunkte sind im ersten Schuljahrgang der Qualifikationsphase zu unterrichten. Der Thematische Schwerpunkt 3 ist anschließend zu unterrichten. Er wird für die Abiturprüfung 2016 als Thematischer Schwerpunkt 1 übernommen.

### B. Thematische Schwerpunkte

#### Thematischer Schwerpunkt 1: Funktionsprinzipien von Hard- und Softwaresystemen einschließlich theoretischer bzw. technischer Modellvorstellungen

##### Schaltnetze

- Entwicklung eines Schaltnetzes mit vorgegebenen Eigenschaften (Schaltwerttabelle, Schaltfunktionen, Gatterdarstellung)
- Analyse einer vorgegebenen Gatterdarstellung
- Entwicklung einer Schaltung mit vorgegebenen Eigenschaften aus gegebenen Komponenten
- Systematische Vereinfachung von Schalttermen

##### Endliche Automaten

- Entwicklung eines Zustandsgraphen für ein gegebenes Problem
- Analyse eines gegebenen Zustandsgraphen
- Erweiterung eines gegebenen Zustandsgraphen
- Umsetzung eines endlichen Automaten in ein Schaltwerk

#### Ergänzung für Kurse auf erhöhtem Anforderungsniveau

##### Turingmaschinen

- Entwicklung einer Turingmaschine für ein gegebenes Problem
- Analyse einer gegebenen Turingmaschine
- Erweiterung einer gegebenen Turingmaschine

## Thematischer Schwerpunkt 2: Werkzeuge und Methoden der Informatik

### Algorithmen (auch rekursive)

- Erstellung eines Algorithmus zu einem gegebenen Problem in schriftlich verbalisierter Form oder als Struktogramm
- Bearbeitung eines Algorithmus, gegeben durch ein Struktogramm, Pseudocode oder Code in Wortform
  - Analyse, z. B. mit einer Tracetabelle oder durch Auswahl geeigneter Testdaten
  - Vervollständigung
  - Präzisierung
  - Korrektur
- Implementierung eines Algorithmus in Java oder einer vergleichbaren Programmiersprache

### Ergänzung für Kurse auf erhöhtem Anforderungsniveau

- Abschätzen der Komplexität eines Algorithmus

### Objektorientierte Modellierung

- Klassendiagramme (Vererbung, Aggregation, Assoziation)
- Anwendung der Klassen (ADTs) „Liste“, „Schlange“ und „Stapel“

### Ergänzung für Kurse auf erhöhtem Anforderungsniveau

- Implementierung der oben genannten Klassen
- Anwendung der Klasse (ADT) „Binärbaum“

## Thematischer Schwerpunkt 3: Anwendung von Hard- und Softwaresystemen sowie deren gesellschaftliche Auswirkungen

### Chiffrieren und Codieren

- Kryptografische Verfahren
  - monoalphabetische Verfahren (u.a. Caesar), polyalphabetische Verfahren (u.a. Vigenère)
  - Analyse monoalphabetischer Verfahren (Häufigkeitsanalyse)
  - Implementierung klassischer Verschlüsselungsverfahren
- Codierung
  - Anwenden und Analysieren eines gegebenen verlustfreien Codierungsverfahrens
  - komprimierende Codes (Lauflängencodierung, Huffman-Codierung (ohne Implementierung))

### Ergänzung für Kurse auf erhöhtem Anforderungsniveau

- fehlererkennende und fehlerkorrigierende Codes (u.a. Paritätsbit, (7,4)-Hamming-Code)

### Datenschutz und Datensicherheit

- Einsatz von asymmetrischen Verfahren zur Authentifikation (allgemeines Prinzip, digitale Signatur)
- Erläuterung grundlegender Begriffe im Kontext der informationellen Selbstbestimmung

## C. Sonstige Hinweise

- Die Aufgabentexte selber enthalten keinen Code in einer konkreten Programmiersprache. Diejenigen Aufgabenteile, die die Implementierung in einer konkreten Programmiersprache erfordern, sind von den Schülerinnen und Schülern in Java oder einer vergleichbaren objektorientierten Programmiersprache zu bearbeiten.
- Anstelle der unterschiedlichen, sprachspezifischen Bezeichnungen „Prozedur“, „Funktion“ bzw. „Methode“ wird in den Aufgabenstellungen der Begriff „Operation“ verwendet.
- Aufgaben, die am Rechner zu bearbeiten sind, werden nicht gestellt.
- Das Lehrermaterial wird weiterhin ausführliche Lösungsskizzen enthalten. Die Implementierungen in einer Programmiersprache werden nur in Java vorgelegt.

Die Anlagen (Operationen der Klassen Liste, Stapel, Schlange und Binärbaum, Notation der Turingmaschinen) sind als Hilfsmittel in der Abiturprüfung zugelassen.

# Anlagen

Niedersächsisches Kultusministerium

Juli 2012

## Anlagen : Als Hilfsmittel zugelassen!

### **1. Operationen der Klassen Liste, Stapel, Schlange und Binärbaum**

Die Klassen benutzen Inhaltsklassen bzw. -typen, die jeweils der aktuellen Aufgabenstellung angepasst werden. Die Klassen werden in den Aufgabenstellungen gegebenenfalls um Attribute und weitere Operationen ergänzt.

Die im Folgenden benutzte Notation entspricht der UML-Notation in Klassendiagrammen.

Mögliche Laufzeitfehler bei der Anwendung der Operationen, z. B. „Entnehmen“ bei einem leeren Stapel, müssen bei der Bearbeitung entsprechender Aufgaben explizit abgefangen werden.

### **Liste**

`Liste()`

Eine leere Liste wird angelegt. Der interne Positionszeiger, der das aktuelle Element markiert, wird auf `null` gesetzt.

`istLeer(): Wahrheitswert`

Wenn die Liste kein Element enthält, wird der Wert `wahr` zurückgegeben, sonst der Wert `falsch`.

`inhaltGeben(): Inhalt`

Der Inhalt des aktuellen Listenelements wird zurückgegeben.

`einfuegen(Inhalt inhalt)`

Ein neues Listenelement mit dem angegebenen Inhalt wird angelegt und hinter der aktuellen Position in die Liste eingefügt, alle weiteren Elemente werden nach hinten verschoben. Der interne Positionszeiger steht auf dem neu eingefügten Element.

`einfuegen(Ganzzahl position, Inhalt inhalt)`

Ein neues Listenelement mit dem angegebenen Inhalt wird angelegt und an der angegebenen Position in die Liste eingefügt. Das vorher an dieser Position befindliche Element und alle weiteren Elemente werden nach hinten verschoben. Der interne Positionszeiger steht auf dem neu eingefügten Element. Falls die angegebene Position nicht existiert, hat die Operation keine Wirkung.

`loeschen()`

Das aktuelle Listenelement wird gelöscht. Der interne Positionszeiger steht anschließend auf dem Nachfolger des gelöschten Elements. Falls kein Nachfolger existiert, zeigt er auf den Vorgänger.

`positionGeben(): Ganzzahl`

Der Wert des internen Positionszeigers wird zurückgegeben.

`positionSetzen(Ganzzahl position)`

Der interne Positionszeiger wird auf den angegebenen Wert gesetzt. Falls die angegebene Position nicht existiert, wird der Positionszeiger nicht verändert.

`laengeGeben(): Ganzzahl`

Die Anzahl der Elemente in der Liste wird zurückgegeben.

### **Stapel**

`Stapel()`

Ein leerer Stapel wird angelegt.

`istLeer(): Wahrheitswert`

Wenn der Stapel kein Element enthält, wird der Wert `wahr` zurückgegeben, sonst der Wert `falsch`.

`inhaltGeben(): Inhalt`

Der Inhalt des obersten Elements des Stapsels wird zurückgegeben, das Element aber nicht entfernt.

`ablegen(Inhalt inhalt)`

Ein neues Element mit dem angegebenen Inhalt wird auf den Stapel gelegt.

`entnehmen(): Inhalt`

Der Inhalt des obersten Elements wird zurückgegeben und das Element wird entfernt.

### **Schlange**

`Schlange()`

Eine leere Schlange wird angelegt.

`istLeer(): Wahrheitswert`

Wenn die Schlange kein Element enthält, wird der Wert *wahr* zurückgegeben, sonst der Wert *falsch*.

`inhaltGeben(): Inhalt`

Der Inhalt des ersten Elements der Schlange wird zurückgegeben, das Element aber nicht entfernt.

`anhaengen(Inhalt inhalt)`

Ein neues Element mit dem angegebenen Inhalt wird angelegt und am Ende an die Schlange angehängt.

`entnehmen(): Inhalt`

Der Inhalt des ersten Elements wird zurückgegeben und das Element wird entfernt.

### **Binärbaum**

`Baum()`

Ein leerer Baum wird erzeugt.

`Baum(Inhalt inhalt)`

Ein Baum wird erzeugt. Die Wurzel erhält den übergebenen Inhalt als Wert.

`istLeer(): Wahrheitswert`

Die Anfrage liefert den Wert *wahr*, wenn der Baum leer ist, sonst liefert sie den Wert *falsch*.

`istBlatt(): Wahrheitswert`

Die Anfrage liefert den Wert *wahr*, wenn der Baum keine Nachfolger hat, sonst liefert sie den Wert *falsch*.

`linkerTeilbaum(): Baum`

Die Operation gibt den linken Teilbaum zurück. Existiert kein linker Nachfolger, so ist das Ergebnis *null*.

`rechterTeilbaum(): Baum`

Die Operation gibt den rechten Teilbaum zurück. Existiert kein rechter Nachfolger, so ist das Ergebnis *null*.

`linkenTeilbaumSetzen(Baum b)`

Der übergebene Baum wird als linker Teilbaum gesetzt.

`rechtenTeilbaumSetzen(Baum b)`

Der übergebene Baum wird als rechter Teilbaum gesetzt.

`inhaltGeben(): Inhalt`

Die Operation gibt den Inhaltswert der Wurzel des aktuellen Baumes zurück.

`inhaltSetzen(Inhalt inhalt)`

Die Operation setzt den Inhaltswert der Wurzel des aktuellen Baumes.

## **2. Notation der Turingmaschinen**

`Bandalphabet = { *, 1 }`

Ein leeres Band enthält nur Sterne.

Die Position des Kopfes der Turingmaschine wird durch einen Unterstrich gekennzeichnet, z.B. symbolisiert \*\*\*111\*\*\*, dass der Kopf unter der rechten 1 des Eingabewortes steht.

Die Maschine wird durch einen Zustandsgraphen beschrieben. Der Anfangszustand wird durch einen Pfeil gekennzeichnet, der Endzustand durch eine doppelte Umrundung.

An den Kanten stehen die Informationen in der folgenden Reihenfolge:  
gelesenes Zeichen, geschriebenes Zeichen, Kopfbewegung.

Kopfbewegung: L: nach links, R: nach rechts, N: keine Bewegung.

Im Zustandsdiagramm nicht aufgeführte Übergänge führen zu einem Fehlerzustand.

# I. Funktionsprinzipien von Hard- und Software systemen

## 1. Zahlensysteme

Zahlen können auf unterschiedliche Art dargestellt werden. Die Basis gibt dabei an, wie viele Werte eine einzelne Stelle annehmen kann:

Basis  $b = 10$ : Dezimalsystem (Alltag)

= 2: Binärsystem (Elektrotechnik) (auch: Dualsystem)

= 16: Hexadezimalsystem (verkürztes Binärsystem)

= 8: Oktalsystem (Dateiberechtigungen)

= 12: Duodezimalsystem (Stunden, Monate)

Die Wertigkeit einer Stelle gibt an, wie sehr diese für die gesamte Zahl „zählt“:

$$\begin{array}{r}
 \text{2} \cdot 10^3 + \\
 1 \cdot 10^2 + \\
 5 \cdot 10^1 + \\
 7 \cdot 10^0 = 
 \end{array}
 \begin{array}{l}
 \xrightarrow{\quad\quad\quad} \text{Wertigkeit der Stellen} \\
 \text{Basis} \\
 \xrightarrow{\quad\quad\quad} \text{Werte der Stellen}
 \end{array}$$

Bsp.:  $1578_{10} = 1 \cdot 10^3 + 5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0 = 1000 + 500 + 70 + 8$

$$11001011_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

$$= 128 + 64 + 8 + 2 + 1 = 203_{10}$$

$$EA_{16} = E_{16} \cdot 16^3 + A_{16} \cdot 16^2 = 14_{10} \cdot 16 + 10_{10} \cdot 1 = 224 + 10 = 234_{10}$$

Im Binärsystem kann jede Stelle einen der Werte 0 oder 1 annehmen.

Diese stellen oft für angelegte bzw. nicht angelegte Spannung. Typischerweise gruppiert man Binärzahlen in 2 große Abschnitte (z.B. 4 oder 8 Stellen.) Die Stellen heißen Bits.

Das Hexadezimalsystem besteht aus den 16 Ziffern 0-9 und anschließend A-F.

Eine 4er-Gruppe im Binärsystem kann als eine Hexzahl interpretiert werden.

## Umrechnungen mit dem Taschenrechner

Dec  $\rightarrow$  Bin: 240  $\blacktriangleright$  Bin

Bin  $\rightarrow$  Dec: 0b11110000

Dec  $\rightarrow$  Hex: 240  $\blacktriangleright$  Hex

Hex  $\rightarrow$  Dec: 0xF0

Hex  $\rightarrow$  Bin: 0xF0  $\blacktriangleright$  Bin

Bin  $\rightarrow$  Hex: 0b11110000  $\blacktriangleright$  Hex

## Umrechnungen durch den Taschenrechner

... → Dezimalsystem:

$$1101_8 = 1 \cdot 8^3 + 1 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = \dots_{10}$$

⇒ Die Werte, multipliziert mit ihrer Wertigkeit, addieren (s.o.)

Hexadezimal → Binärsystem:

$$\begin{aligned} AFD_{16} &\Rightarrow A \stackrel{1}{=} 10 \stackrel{1}{=} 1010 \\ F &\stackrel{1}{=} 15 \stackrel{1}{=} 1111 \stackrel{1}{=} 2 \Rightarrow AFD_{16} = \frac{1010}{1} \frac{1111}{2} \frac{1101}{3} \stackrel{1}{=} 2 \\ D &\stackrel{1}{=} 13 \stackrel{1}{=} 1101 \stackrel{1}{=} 3 \end{aligned}$$

⇒ 4er-Gruppen bilden und aneinanderlängen

Binär - → Hexadezimalsystem:

$$\begin{aligned} 10101111101_2 &\Rightarrow \begin{array}{c} 1010 \stackrel{1}{=} 10 \stackrel{1}{=} A \\ 1111 \stackrel{1}{=} 15 \stackrel{1}{=} F \\ 1101 \stackrel{1}{=} 13 \stackrel{1}{=} D \end{array} \\ &10101111101_2 = AFD_{16} \end{aligned}$$

⇒ in 4er-Gruppen teilen, Herzahl zuordnen und aneinanderlängen:

Binär - → Binärsystem:

$$\begin{array}{r} 20_{10} : 2 = 10 \text{ R } 0 \\ 10 : 2 = 5 \text{ R } 0 \\ 5 : 2 = 2 \text{ R } 1 \\ 2 : 2 = 1 \text{ R } 0 \\ 1 : 2 = 0 \text{ R } 1 \end{array} \Rightarrow 20_{10} = 10100_2$$

Restwertalgorithmus: ganzzahlig dividiert, Reste rückwärts ablesen

Dezimal - → Hexadezimalsystem:

$$\begin{array}{r} 20_{10} : 16 = 1 \text{ R } 4 \\ 1 : 16 = 0 \text{ R } 1 \end{array} \Rightarrow 20_{10} = 14_{16} \text{ s.o.}$$

## 2. Grundlagen der Schaltalgebra

2.1

### 2.1 Gagische Grundschatungen

AND / Konjunktion /  $x_1$  und  $x_2$ :

$x_1$	$x_2$	$y$
0	0	0
0	1	0
1	0	0
1	1	1

$$y = x_1 * x_2 = x_1 x_2$$

OR / Disjunktion /  $x_1$  oder  $x_2$ :

$x_1$	$x_2$	$y$
0	0	0
0	1	1
1	0	1
1	1	1

$$y = x_1 + x_2$$

Identität:



NOT / Negation / Inverter:



NAW / Sheffer-Funktion / nicht Reduzibel:

$x_1$	$x_2$	$y$
0	0	1
0	1	1
1	0	1
1	1	0

$$y = \overline{x_1 x_2}$$

NOOR / Peirce-Funktion / weder  $x_1$  noch  $x_2$ :

$x_1$	$x_2$	$y$
0	0	1
0	1	0
1	0	0
1	1	0

$$y = \overline{x_1 + x_2}$$

EXOR / XOR / Exklusiv-oder / Antivalenz / entweder  $x_1$  oder  $x_2$ :

$x_1$	$x_2$	$y$
0	0	0
0	1	1
1	0	1
1	1	0

$$y = \overline{x_1 x_2} + x_1 x_2 = \overline{x_1 + x_2} (x_1 + x_2) = \overline{x_1 x_2} (x_1 + x_2)$$

Äquivalenz:



$x_1$	$x_2$	$y$
0	0	1
0	1	0
1	0	0
1	1	1

$$y = \overline{x_1 x_2} + x_1 x_2 = \overline{x_1 + x_2} + x_1 x_2$$

Implikation / aus  $x_1$  folgt  $x_2$ :



$x_1$	$x_2$	$y$
0	0	1
0	1	1
1	0	1
1	1	1

$$y = \overline{x_1 x_2} = \overline{x_1} + \overline{x_2} = \overline{x_1} + x_2$$

$x_1$  gilt nicht.  $x_2$  muss nicht gelten.

$x_1$  gilt nicht.  $x_2$  kann gelten.

$x_1$  gilt.  $x_2$  gilt nicht.  $\Rightarrow$  f.A.!!

$x_1$  gilt.  $x_2$  gilt.

Wenn  $\overline{x_1}$ ,  $x_1$   $x_2$   $y$ )

dann ist  $x_2$  egal

Wenn  $x_1$ ,  $x_1$   $x_2$   $y$ )

dann muss  $x_2$  0 0

auch  $x_2$ : 1 1 1

betrachten-  
de Be-  
dingung

## 2.2 Schaltfunktionen

Eine Variable, die mit zwei unterschiedlichen Werten belegt werden kann,

heißt (binäre) Eingangsvariable / Schaltvariable  $E_i$ :

Eine binäre Funktion  $f(E_1 \dots E_n)$  von  $n$  Schaltvariablen  $E_1 \dots E_n$

heißt  $n$ -stellige Schaltfunktion.

Sie kann für  $2^n$  unterschiedliche Kombinationen von Eingangsvariablen

$2^n$  verschiedene Werte annehmen.

Es existieren  $2^{2^n}$   $n$ -stellige Schaltfunktionen für  $n$  Eingangsvariablen.

Alle  $n$ -stelligen Schaltfunktionen sind reduzierbar auf die Grundfunktionen

AND, OR und NOT bzw. Konjunktion, Disjunktion und Negation, es

reichen demnach auch die Sheffer- und Pierce-Funktionen NAND und NR.

Bsp.:  $n = 2$

Jede 2-stellige Schaltfunktion kann  $2^2 = 4$  Werte annehmen.

Es existieren  $2^{2^2} = 16$  2-stellige Schaltfunktionen.

Dazu gehören die Grundfunktionen AND, OR, NOT, EQ, NAND,

IMP, NMPC, AND, EVER und NEVER.

## 2.3 Gesetze der Schaltalgebra

### Konjunktion AND

Kommutativgesetz  $A \cdot B = B \cdot A$

Assoziativgesetz  $(A \cdot B) \cdot C = A \cdot (B \cdot C)$

Distributivgesetz  $A \cdot (B + C) = A \cdot B + A \cdot C$

Idempotenz  $A \cdot A = A$

Absorptionsgesetz  $A \cdot (A + B) = A$

Negation  $A \cdot \bar{A} = 0$

Neutrales Element  $A \cdot 1 = A$   
 $A \cdot 0 = 0$

de Morgan

$$\begin{aligned}\bar{A} \cdot \bar{B} &= \bar{A} + \bar{B} \\ A \cdot B \cdot C &= \bar{\bar{A}} + \bar{B} + \bar{C}\end{aligned}$$

$\Rightarrow$  Einzelaussage vereinen und Verknüpfung austauschen

### Disjunktion OR

$A + B = B + A$

$(A + B) + C = A + (B + C)$

$A + (B \cdot C) = (A + B) \cdot (A + C)$

$A + A = A$

$A + (A \cdot B) = A$

$A + (\bar{A} \cdot B) = A + B$

$A + \bar{A} = 1$

$A + 1 = 1$

$A + 0 = A$

$\bar{A} + \bar{B} = \bar{A} \cdot \bar{B}$

$A + B + C = \bar{\bar{A}} \cdot \bar{\bar{B}} \cdot \bar{\bar{C}}$

$$\begin{aligned}A \cdot A + A \cdot B &= A + A \cdot B = \\ A \cdot (1 + B) &= A \cdot 1 = A\end{aligned}$$

$$\begin{aligned}A \cdot \bar{A} + A \cdot B &= 0 + A \cdot B = \\ A \cdot \bar{A} &= A \cdot 1 = A\end{aligned}$$

### 3. Schaltungsanalyse

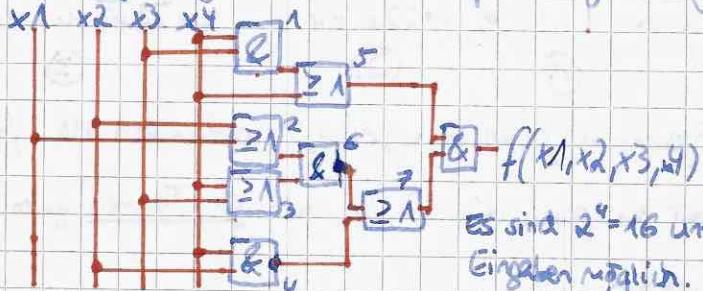
Ist eine schematische Zeichnung einer Schaltfunktion  $f$  gegeben, so kann daraus auf mehreren Wegen der Funktionsterm der Schaltfunktion konstruiert werden.

### 3.1 Wertetabelle

Zunächst bietet es sich an, eine Tabelle aufzustellen, die jedem möglichen Eingangsvektor einen Ausgangsvektor zuordnet.

$x_1 \dots x_n | f(x_1, \dots, x_n)$  Optionale Spalten können helfen, den Überblick zu bewahren.

Bsp.: Schaltungsdiagramm der 4-stelligen Schaltfunktion  $f$ :



Es sind  $2^4 = 16$  unterschiedliche Eingaben möglich.

<u><math>x_1</math></u>	<u><math>x_2</math></u>	<u><math>x_3</math></u>	<u><math>x_4</math></u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>f</u>	<u>Eingänge</u>	<u>Zwischenschritte</u>	<u>Ausgänge</u>
0	0	0	0	0	0	0	1	0	1	1	0	0000	000	1011
0	0	0	1	0	0	1	1	1	1	1	1	0001	001	1111
0	0	1	0	0	0	1	0	1	1	1	0	0010	001	1011
0	0	1	1	1	0	1	1	1	1	1	1	0011	101	1111
0	1	0	0	0	1	0	1	0	1	1	0	0100	010	1011
0	1	0	1	0	1	1	0	1	0	0	0	0101	011	0100
0	1	1	0	0	0	1	1	0	0	1	0	0110	011	1001
0	1	1	1	1	1	1	1	1	0	0	0	0111	111	1010
1	0	0	0	0	0	1	0	1	0	1	0	1000	010	1011
1	0	0	1	0	0	1	1	0	1	1	1	1001	011	1101
1	0	1	0	0	0	1	1	0	0	1	0	1010	011	1001
1	0	1	1	1	1	1	1	1	0	1	1	1011	111	1101
1	1	0	0	0	0	1	0	1	1	1	0	1100	010	1011
1	1	0	1	0	0	1	1	0	0	0	0	1101	011	0110
1	1	1	0	0	0	1	1	1	0	1	0	1110	011	1100
1	1	1	1	1	1	1	1	0	0	0	0	1111	111	1010

### 3.2 Funktionsgleichung aus Schaltungsschigramm

Die Schaltung kann schrittweise zu einem - in der Regel nicht optimalen - Schaltwerk „zusammengebaut“ werden.

$$\text{Bsp.: } f(x_1, x_2, x_3, x_4) = \underbrace{(x_3 x_4 + x_4)}_{\substack{1 \\ 5}} \left( \underbrace{(x_1 + x_2)}_{\substack{2 \\ 6}} \underbrace{(x_3 + x_4)}_{\substack{3 \\ 7}} + \underbrace{x_2 x_4}_{\substack{4 \\ 8}} \right)$$

$f$

### 3.3 Funktionsgleichung aus Wertetabelle

#### 3.3.1 Disjunktive Normalform

Um zu einer Schaltung die disjunktive Normalform (DNF), bestehend aus „1-Termen“, aufzustellen, sind folgende Schritte nötig:

- ① Alle Zeilen in Wertetabelle mit  $f(x_1, x_2, x_3, x_4) = 1$  suchen.
- ② Jeweils die Eingangsvariablen (UND-verknüpfen, jede Variable muss vorkommen).
- ③ Diese sog. Konjunktionsterme ODER-verknüpfen, Ergebnis ist die DNF.

Bsp.:	$x_1 \ x_2 \ x_3 \ x_4$	$f$	Konjunktionsterme	DNF
	0 0 0 1	1	$\Rightarrow \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4$	
	0 0 1 1	1	$\Rightarrow \bar{x}_1 \bar{x}_2 x_3 x_4$	$f(x_1, x_2, x_3, x_4) =$
	1 0 0 1	1	$\Rightarrow x_1 \bar{x}_2 \bar{x}_3 x_4$	$\bar{x}_1 \bar{x}_2 + \bar{x}_1 \bar{x}_3 + x_1 \bar{x}_2 + x_1 \bar{x}_3 + x_1 x_2 + x_1 x_3 + x_1 x_4$
	1 0 1 1	1	$\Rightarrow x_1 \bar{x}_2 x_3 x_4$	

(1)     .. (2)     ..     ..     ..     .. (3)

Sind mehr Ergebnisse  $f(x_1, x_2, x_3, x_4) = 0$  vorhanden als  $f=1$ , kann, um Arbeit zu sparen, die DNF mit „0-Termen“ gebildet werden.

Bsp: AND-Funktion

$$\begin{array}{l} 0 \ 0 \rightarrow 0 \\ 0 \ 1 \rightarrow 1 \\ 1 \ 0 \rightarrow 0 \\ 1 \ 1 \rightarrow 1 \end{array} \quad \left. \begin{array}{l} \Rightarrow \bar{x}_1 \bar{x}_2 \\ \dots \\ \Rightarrow x_1 x_2 \end{array} \right\}$$

(1)     (2)     (3)

$$\text{DNF: } f = \bar{x}_1 \bar{x}_2 + x_1 x_2$$

mit 0-Termen kommt die vereinfachte Funktion zustande!

#### 3.3.2 Konjunktive Normalform

Eine Konjunktive Normalform (KNF) ist eine Konjunktion von Disjunktions-terms - also das Gegenstück zur DNF - folgender Form:  $(x_1 + x_2)(\bar{x}_1 + x_2) \dots$

Sie kommt zum Einsatz, wenn zu einem gegerten Funktionsterm f die Funktion f ermittelt werden soll, also dann, wenn man 0-Terme verwendet.

Bsp.: AND-Funktion: DNF:  $f = \bar{x}_1 \bar{x}_2 + x_1 x_2$

$$\text{KNF: } f = \bar{x}_1 \bar{x}_2 + x_1 x_2 = \bar{x}_1 \bar{x}_2 \cdot x_1 x_2$$

$$= (x_1 + x_2)(\bar{x}_1 + x_2)$$

grann ausmultipliziert werden:

$$= x_1 \bar{x}_1 + x_1 \bar{x}_2 + x_1 x_2 + x_2 \bar{x}_2$$

$$= 1 + x_1 \bar{x}_2 + \bar{x}_1 x_2 + 1$$

$$= \bar{x}_1 x_2 + x_1 \bar{x}_2, \quad (\text{DNF})$$

entspricht der DNF bei Verwendung von 1-Termen

Sowohl die DNF als auch die

KNF sind in der Regel keine op-  
timalen Funktionsterme.

### 3.4 Systematisches Minimieren von Schalttermenen

Hat man anhand eines Schaltungsdiaagramms oder mithilfe einer Wahrtafel einen Schaltterm ermittelt (z.B. in Konjunktiver oder disjunktiver Normalform), ist dieser i. d. R. nicht optimiert/minimiert.

Optimierung von Schalttermen bedeutet, die Anzahl der nötigen logischen Gatter auf ein Minimum zu reduzieren. Dazu gibt es zwei Vorgehensweisen.

#### 3.4.1 Vereinfachen von Schalttermen mit Rechengesetzen

Insbesondere die Negation und das Absorptionsgesetz eignen sich zur Vereinfachung von Schalttermen. Faustregeln sind z.B.:

$$\bar{A}\bar{B}\bar{C}D + \bar{A}\bar{B}\bar{C}\bar{D} = \bar{A}\bar{B}\bar{C}(D + \bar{D}) = \bar{A}\bar{B}\bar{C}(1) = \bar{A}\bar{B}\bar{C}$$

zwei Konjunktorsterme unterscheiden sich nur durch eine Variable

$$\bar{A}\bar{B}\bar{C} + A\bar{B}\bar{C}\bar{D} = \bar{B}\bar{C}(\bar{A} + A\bar{D}) = \bar{B}\bar{C}(\bar{A} + \bar{D}) = \bar{A}\bar{B}\bar{C} + \bar{B}\bar{C}\bar{D}$$

unbeschöpftbare Variablen  
verschwinden, wenn konjunktivische mit disjunktiven Varianten

$$\bar{A}\bar{B}\bar{C} \cdot \bar{A}\bar{B}\bar{C}\bar{D} = \bar{A}\bar{B}\bar{C}(1 + \bar{D}) = \bar{A}\bar{B}\bar{C}(1) = \bar{A}\bar{B}\bar{C}$$

" "

Hat man eine neue DNF z.B.  $\bar{A}\bar{B}\bar{C} + \bar{B}\bar{C}\bar{D} + \dots$  aufgestellt, muss man noch

so viel wie möglich ausklammern, um die optimale Variante zu erhalten, z.B.

$$\bar{B}\bar{C}(\bar{A} + \bar{D}) + \dots$$

Bsp.:  $f(x_1, x_2, x_3, x_4) = \cancel{\bar{x}_1 \bar{x}_2 \bar{x}_3 x_4} + \cancel{\bar{x}_1 \bar{x}_2 x_3 x_4} + \cancel{x_1 \bar{x}_2 \bar{x}_3 x_4} + \cancel{x_1 \bar{x}_2 x_3 x_4}$

$$= \cancel{\bar{x}_2 \bar{x}_3 x_4} (\cancel{x_1 + \bar{x}_1}) + \cancel{\bar{x}_2 + x_3 x_4} (\cancel{x_1 + \bar{x}_1})$$

$$= \cancel{\bar{x}_2 \bar{x}_3 x_4} + \cancel{\bar{x}_2 x_3 x_4}$$

$$= \cancel{\bar{x}_2 x_4} (\cancel{x_3 + \bar{x}_3})$$

$$= \cancel{\bar{x}_2 x_4} \Rightarrow \text{kein Punktammern möglich}$$

Es ist sinnvoll, das Ergebnis mit einer Wahrtafel zu überprüfen:

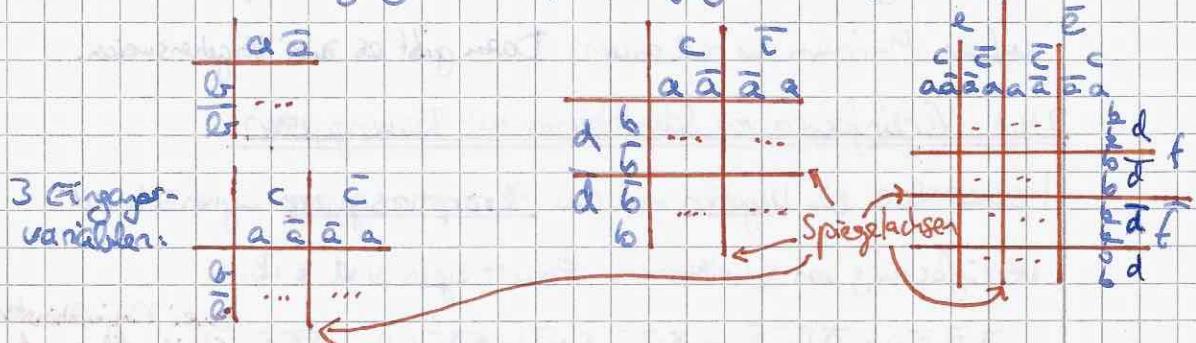
$x_1 x_2 x_3 x_4$	$\bar{x}_2 x_4$	$f$
0 0 0 0	1 0	0
0 0 0 1	1 1	1
0 0 1 0	1 0	0
0 0 1 1	1 1	1
0 1 0 0	0 0	0
0 1 0 1	0 1	0
0 1 1 0	0 0	0
0 1 1 1	0 1	0
1 0 0 0	1 0	0
1 0 0 1	1 1	1
1 0 1 0	1 0	0
1 0 1 1	1 1	1
1 1 0 0	0 0	0
1 1 0 1	0 1	0
1 1 1 0	0 0	0
1 1 1 1	0 1	0

$f$  stimmt mit  $f$  aus der ursprünglichen Wahrtafel überein, das Ergebnis ist korrekt.

### 3.4.2 Karnaugh-Veitch-Diagramme

Karnaugh-Veitch-Diagramme oder auch KV-Diagramme können das Minimieren von Schalterschaltern erheblich vereinfachen. Dazu erstellt man eine Art „Vierfeldertafel“, die bei weiteren Eingangsvariablen durch Spiegelungswert fest wird:

2 Eingangsvariablen; 4 Eingangsvariablen; 6 Eingangsvariablen



Dieses Diagramm füllt man dann gemäß der Wertetabelle aus. Dann gruppiert man überstehende Ergebnisse zu  $2^n$  großen Gruppen, die möglichst groß sind. Je nach Anzahl kann man Einsen oder Nullen gruppieren, um die Terme  $f$  bzw.  $\bar{f}$  zu erhalten. Auch leer-Gruppen sind möglich.

(Hat man größtmögliche Gruppen gebildet, kann man aus diesen den optimalen Funktionsterm ableiten.)

Bsp.:	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	$f = \bar{x}_3 x_4$	Bsp.:	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	$f = x_3 x_4 + \bar{x}_3 \bar{x}_4$
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$			$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$	$+ x_1 x_2$
	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$			$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$			$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$	
	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$			$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$			$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$	
	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$			$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$			$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	

zu beachten sind  
nicht relevante verbindungen!  
(wie ein Regel)

Bsp.:	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	$f = \bar{x}_1 \bar{x}_3 x_4 + \bar{x}_2 \bar{x}_3 \bar{x}_4 + \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 x_3 \bar{x}_4$	man kann auch Nullen gruppieren
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$	$f = \bar{x}_1 \bar{x}_3 x_4 + \bar{x}_2 \bar{x}_3 \bar{x}_4 + \bar{x}_2 x_3 \bar{x}_4 + \bar{x}_1 x_3 \bar{x}_4$	und dann nur DUF umformen.
	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	$= (x_1 + x_3 + \bar{x}_4)(x_2 + \bar{x}_3 + \bar{x}_4)(x_2 + \bar{x}_3 + x_4)(x_1 + x_3 + x_4)$	
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$	$= x_1 + x_3 + x_4 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4 + x_2 + x_3 + x_4 + x_1 + x_3 + x_4$	
	$x_3 \quad \bar{x}_3$	$x_4 \quad \bar{x}_4$	$= 12 + 13 + 14 + 32 + 33 + 34 + \bar{5}2 + \bar{5}3 + \bar{5}4 + \dots$	
	$x_1 \quad \bar{x}_1$	$x_2 \quad \bar{x}_2$		Das kann u. U. unötig komplex werden...

Manchmal können verschiedene optimale Lösungen gefunden werden.

Bsp.:	$f$ -Segment einer Siebensegmentanzeige	$E^F$	NOT AND NOR
	$c \quad \bar{c}$	$a \quad \bar{a} \quad \bar{a} \quad a$	als 0 oder 1 behandelt werden, wenn Gruppen gebildet werden!
	$d \quad \bar{d}$	$b \quad \bar{b} \quad \bar{d} \quad d$	
	$e \quad \bar{e}$	$c \quad \bar{c} \quad \bar{c} \quad c$	
	$f \quad \bar{f}$	$a \quad \bar{a} \quad \bar{a} \quad a$	
	$g \quad \bar{g}$	$b \quad \bar{b} \quad \bar{d} \quad d$	
		$\Sigma 1$	
		$\Sigma 4$	
		$\Sigma 10$	
		$\Sigma 11$	
		$f$	

$dc = \text{don't care}$

## 4. Elektronische Grundschaltungen

Schaltnetze sind Schaltungen, die keine Rückkopplung enthalten. Im Gegensatz dazu sind in Schaltwerken ein oder mehrere Rückkopplungen vorhanden, wodurch die Schaltung einen speziellen Charakter erhält.

### 4.1 Flipflops

Flipflops sind Schaltwerke, die Zustände speichern können, wofür sie einen „Haltet/Speichernzustand“ haben. Es gibt verschiedene Arten:

Nicht taktgesteuerte und taktgesteuerte (taktzustands- und taktfallenden- gesteuerte (einfallengesteuerte und zweifallengesteuerte)) Flipflops.

#### 4.1.1 RS - Flipflop

Das Reset/Set-Flipflop oder

RS-FF kann mit NAND oder

NOR-Gattern gebildet werden (v.a.

NOR). Man kann aktiv

den Ausgang Q setzen oder löschen:

R S Q Q̄

0 0 ? ?

0 1 1 0

1 0 0 1

( 1 1 1 1 )

Halten/Speichern: Gibt den für Q gestellten Wert zurück.

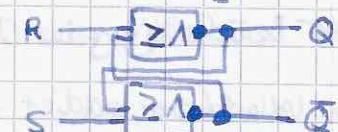
Set / setzen: Setzt Q auf 1.

Reset / Löschen: Setzt Q auf 0.

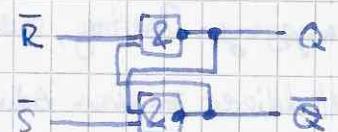
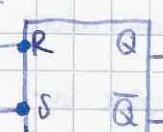
potentielle Race condition

Schaltsymbol

Stellung



} RS-FF  
(NOR)



} RS-FF  
(NAND)

#### 4.1.1.1 Taktgesteuertes RS-Flipflop

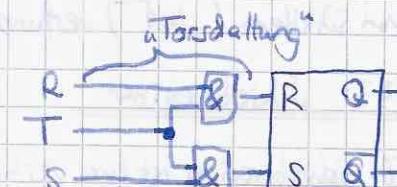
Nur wenn die angelegte Taktleitung Strom liefert, werden R und S übernommen.

Es handelt sich um ein taktzustandsgesteuertes FF.

Daneben sind einfallengesteuerte ...

... und zweifallengesteuerte FFs clackbar:  
(s. DDR Speicher („double data rate“))

Fügt man eine Negation für den R-Eingang hinzu, erhält man ein Flipflop mit der



} Taktzustandsgesteuertes  
RS-FF

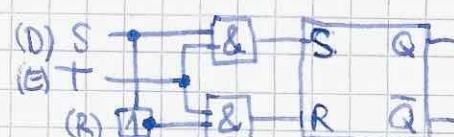
Übernahme bei  
 $T=1$



„rising edge“



„rising & falling edge“



Funktionalität eines D-Flipflops und vermeidet den unzulässigen Zustand.

## 4.1.2 D-Flipflop

Das D-Flipflop hat eine integrierte Torsdrehung zur Vermeidung des verbotenen Zustands. Es wird i.d.R. an einer Taktleitung verwendet.

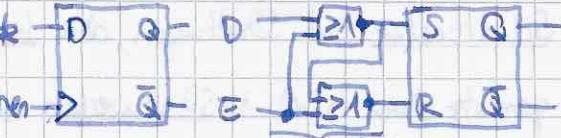
D	E	Q	$\bar{Q}$
0	0	?	?
0	1	0	1
1	0	?	?
1	1	1	0

Speichern  
setzen  
speichern  
setzen

Bei inaktiver Takt persistiert nichts (halten)  
Bei aktiver Takt wird D übernommen:  $Q = D$

Schaltsymbol

Schaltung

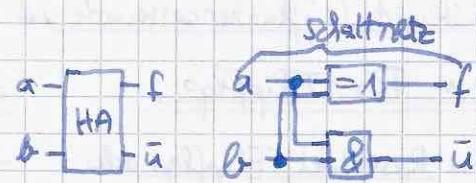


Torsdrehung

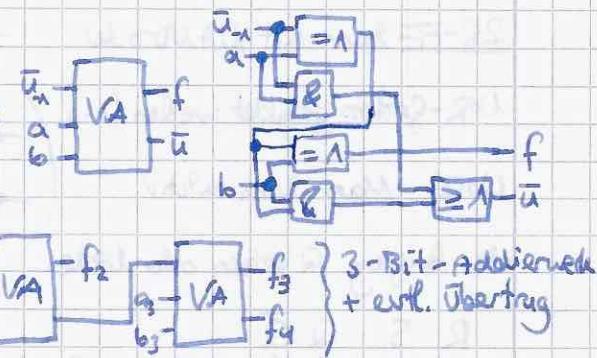
## 4.2 Anwendungen

### 4.2.1 Addierwerke

Ein Halbaddierer addiert zwei Bits unter Berücksichtigung des Übertrags.

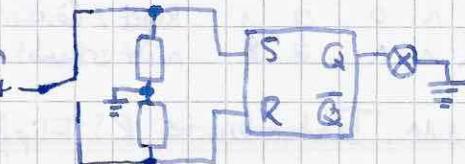


Ein Volladdierer addiert zudem einen vorangegangen Übertrag, sodass man rechte Volladdierer zu einem Addierwerk zusammenstellen kann:



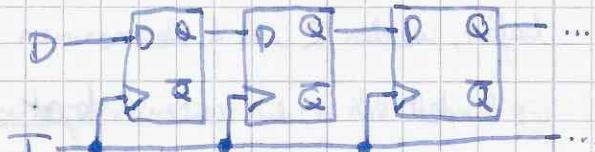
### 4.2.2 Entprellen von Schaltern

Mit einem RS-Flipflop kann man das Prellen von Schaltern (MW) verhindern.



### 4.2.3 Schieberegister

Ein Schieberegister ist eine Verknüpfung von mehreren D-Flipflops. Bei jedem neuen Takt signal wird eine neue Information abgepeistet und „verschoben“.

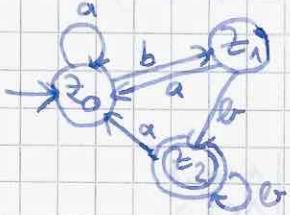


## 5. Endliche Automaten

Ein endlicher Automat ist ein spezielles Zustandsdiagramm mit endlich vielen Zuständen. Für reguläre Sprachen (spezielle formale Sprachen) kann man mit einem endlichen Automaten prüfen, ob ein Wort zu dieser Sprache gehört.

Der Startzustand ist mit einem Pfeil gekennzeichnet, der die Endzustände erhalten eine doppelte Umrandung. Der Automat arbeitet alle Buchstaben eines Worts ab und wechselt abhängig von Buchstaben den Zustand. Ist der Automat am Ende im Endzustand angelangt, gehört das Wort zu der Sprache.

Bsp:



$$\begin{aligned} Z &= \{z_0, z_1, z_2\} \\ A &= \{a, b\} \\ S &= z_0 \\ E &= \{z_1, z_2\} \\ f(z_0, a) &= z_0 \\ f(z_0, b) &= z_1 \\ f(z_1, a) &= z_0 \\ f(z_1, b) &= z_2 \\ f(z_2, a) &= z_0 \\ f(z_2, b) &= z_1 \end{aligned}$$

Zustände  
Eingabealphabet  
Startzustand ( $S \in Z$ )  
Endzustände ( $E \subseteq Z$ )

} Übergangsfunction:  
 $f(z_{alt}, zeichen) = z_{neu}$

Existiert in einem Zustand kein Übergang für ein eingegebenes Zeichen, führt dies zur Nichtakzeptanz des Worts - man kann dafür einen separaten Fehlerzustand einführen.

Gibt es für jedes Zeichen des Eingabealphabets in jedem Zustand einen Übergang, heißt der Automat vollständig. Gibt es jeweils höchstens einen Übergang, heißt der Automat deterministisch; nichtdeterministische Automaten (DEA = Deterministischer Endlicher Automat) können in gleichwertige deterministische Automaten umgewandelt werden.

Aus einem Zustandsdiagramm kann man eine Grammatik ermitteln, deren

Produktionsregeln die folgende Form haben:  $\Rightarrow$   $\langle z_0 \rangle \rightarrow a \langle z_0 \rangle$  (j. Automat den)

Der Übergang  $f(z_{alt}, zeichen) = z_{neu}$  wird zur

Produktionsregel  $\langle z_{alt} \rangle \rightarrow zeichen \langle z_{neu} \rangle$ ; jeder

Endzustand wird zur Regel  $\langle z_{alt} \rangle \rightarrow E$

(E steht für ein leerer Zeichen, also keine Eingabe).

Sprachen, die durch eine solche Grammatik beschrieben und damit von

$$\begin{aligned} \langle z_0 \rangle &\rightarrow b \langle z_1 \rangle \\ \langle z_1 \rangle &\rightarrow a \langle z_0 \rangle \\ \langle z_1 \rangle &\rightarrow b \langle z_2 \rangle \\ \langle z_2 \rangle &\rightarrow a \langle z_0 \rangle \\ \langle z_2 \rangle &\rightarrow b \langle z_1 \rangle \\ \langle z_2 \rangle &\rightarrow E \end{aligned}$$

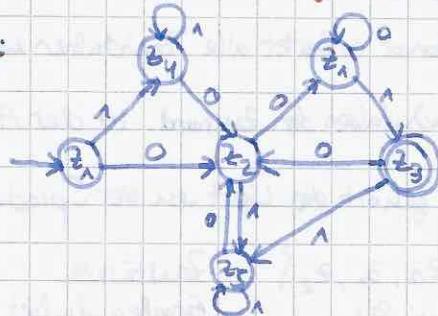
einem endlichen Automaten erkannt werden können, nennt man reguläre Sprachen (spezielle formale Sprachen). Es kann also nicht jede formale Sprache durch einen endlichen Automaten beschrieben werden.

### Formale Beschreibung eines Automaten und seiner Grammatik

Automat:  $M = (\{ \text{Zustände} \}, \{ \text{Eingabetyp} \}, \text{Übergangsfunktion}, \text{Startzustand}, \{ \text{Endzustände} \})$

Grammatik:  $G = (\{ \text{Zustände} \}, \{ \text{Eingabetyp} \}, \{ \text{Produktionsregeln} \}, \text{Startzustand})$

Bsp.:



$$\begin{aligned} Z &= \{z_1, z_2, z_3, z_4, z_5\} \\ A &= \{0, 1\} \\ S &= z_1 \\ E &= \{z_4\} \\ M &= (Z, A, S, E) \end{aligned}$$

Übergangstabelle:

S	0	1
$z_1$	$z_2$	$z_4$
$z_2$	$z_1$	$z_5$
$z_3$	$z_2$	$z_5$
$z_4$	$z_2$	$z_4$
$z_5$	$z_2$	$z_5$

Produktionsregeln:

$$\begin{aligned} P &= \{ \langle z_1 \rangle \rightarrow 0 \langle z_2 \rangle | 1 \langle z_4 \rangle \\ &\quad \langle z_2 \rangle \rightarrow 0 \langle z_1 \rangle | \wedge \langle z_5 \rangle \\ &\quad \langle z_3 \rangle \rightarrow 0 \langle z_2 \rangle | 1 \langle z_5 \rangle | \varepsilon \\ &\quad \langle z_4 \rangle \rightarrow 0 \langle z_2 \rangle | 1 \langle z_4 \rangle \\ &\quad \langle z_5 \rangle \rightarrow 0 \langle z_2 \rangle | \wedge \langle z_5 \rangle \} \end{aligned}$$

Grammatik

$$G = (Z, A, P, S)$$

Syntaktische Diagramme stellen endliche Automaten vereinfacht dar.

Bsp.: ganze Zahl:

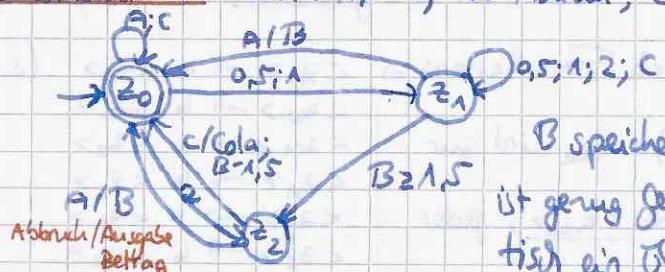


Schleife: - [REPEAT] - [Anweisung] - [UNTIL] - [Bedingung] ->

### 5.1 Anwendungen

Bei angewandten Automaten kann man auch noch die Ausgabe ergänzen sowie zusätzliche Speicher. Ausgabe erfolgt dann bspw.: Eingabe / Ausgabe.

Getränkeautomat: Cola 1,50€; A: Abbruch; C: Cola ausgeben; Erkauft:  $\frac{1}{2} \text{€}$



T3: Geldspeicher

B speichert den aktuellen Geldbetrag, ist genug Geld eingeworfen, erfolgt automatisch ein Übergang nach  $z_2$ .

Bei Colaausgabe wird ggf. ein Betrag zurückgegeben, falls zu viel Geld eingegeben wurde.

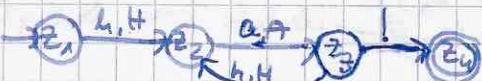
$z_0$ : kein Geld eingeworfen

$z_1$ : zu wenig Geld eingeworfen

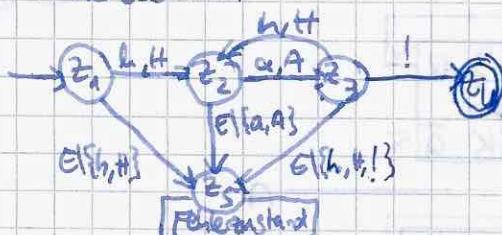
$z_2$ : genug Geld eingeworfen

Gachautomat: erkennt beliebig viele 'a's, gefolgt von einem '!'

$$A = \{a \dots z; A \dots Z; !\}$$



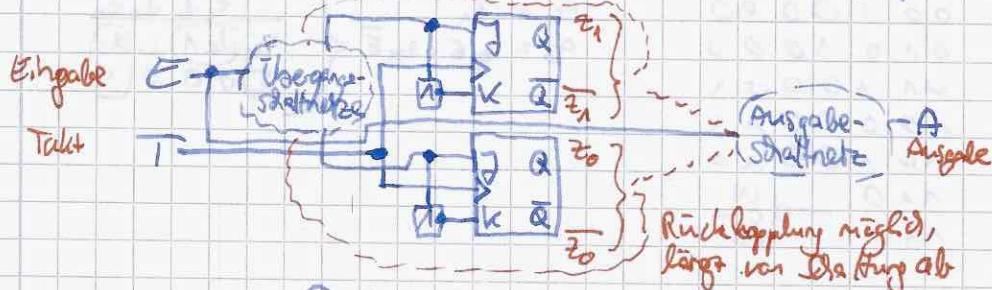
Mit Fehlerzustand:



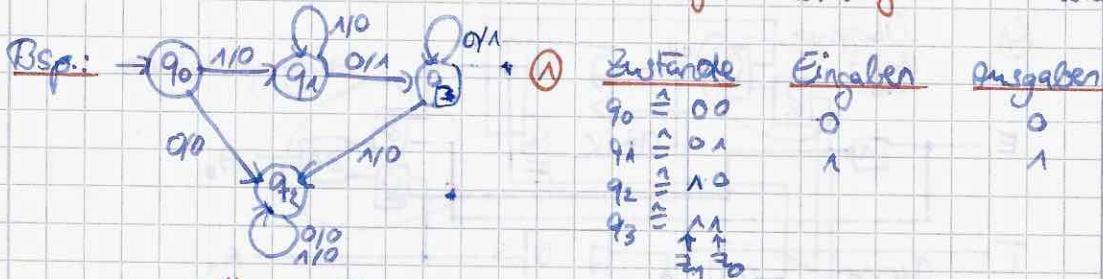
## 5.2 Umsetzung eines endlichen Automaten in ein Schaltwerk

Um einen Automaten in ein Schaltwerk umzusetzen, verwendet man JK-Flipflops, welche ähnlich wie D-Flipflops verwendet werden:  $\begin{matrix} J & Q \\ K & \bar{Q} \end{matrix} \quad (J = \bar{K})$

- ① Zunächst kodiert man mögliche Zustände, Eingaben und Ausgaben binär.
- ② Man stellt eine Schaltwerttafel auf, die die Übergänge beschreibt.
- ③ Man liest die Terme für die neuen Zustände und Ausgaben ab und minimiert diese.
- ④ Man zeichnet das Schaltwerk nach folgendem Muster: (für jedes Zustandsbit ein JK-FF)



Hier für 3 oder 4 Zustände, da 2 Zustandsbits  $z_0$  und  $z_1$  berücksichtigt werden, nicht auf die Zustände  $z_0$  und  $z_1$ !

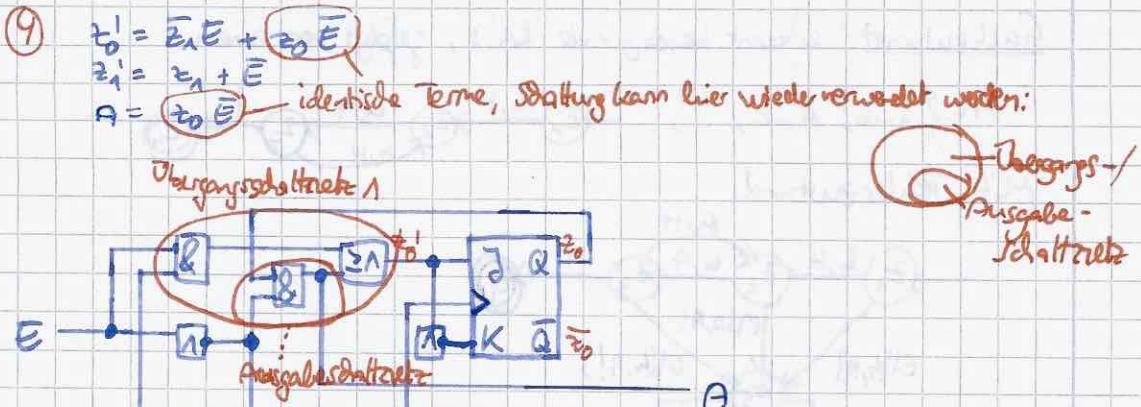


	alt	neu
$z_1 z_0$	E	$z_1' z_0' A$
q <sub>0</sub>	00	0100
q <sub>1</sub>	00	010
q <sub>2</sub>	01	111
q <sub>3</sub>	01	100
q <sub>0</sub>	10	100
q <sub>2</sub>	10	100
q <sub>3</sub>	11	111
q <sub>0</sub>	11	100

2 Zustandsbits, also 2 JK-Flipflops.

$$\begin{aligned} z_1' &= z_1 + \bar{E} \\ z_0' &= z_1 E + z_0 \bar{E} \\ A &= z_0 \bar{E} \end{aligned}$$

	E	$\bar{E}$
$\bar{z}_1$	00	11
$\bar{z}_0$	00	00



Übergangs-/  
Ausgabe-  
Schaltkreis

Bsp:

Kaffee = 1,  $\bar{R} = \text{Rückgat}$   
 $N = \text{Niedrig}$   
 $K = \text{Kaffee}$   
 $T = \text{S0ct}$

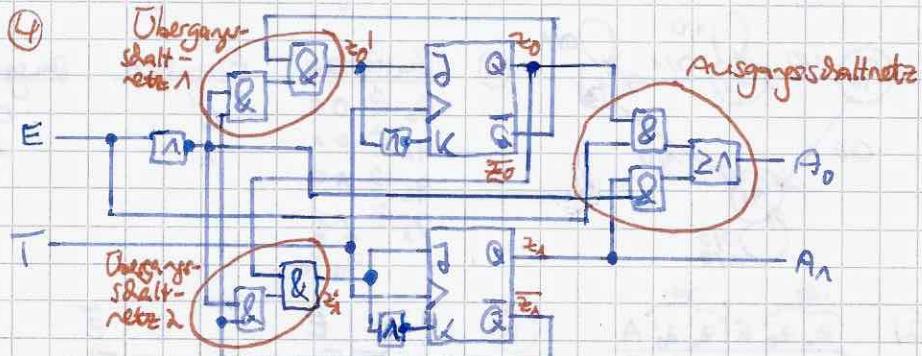
(1)  $z_0 = q_0 \hat{=} 00 \quad A = N \hat{=} 00$   
 $q_1 \hat{=} 01 \quad N \hat{=} 01$   
 $q_2 \hat{=} 10 \quad \wedge \hat{=} 10$   
 $q_3 \hat{=} 11 \quad K \hat{=} 11$

E:  $q_2 \hat{=} 0 \quad R = 1$

(2)  $\begin{array}{c|ccccc} z_1 & z_0 & E & z_1' & z_0' & A_0 \\ \hline 00 & 0 & 0 & 01 & 10 & 00 \\ 00 & 1 & 0 & 00 & 11 & 00 \\ 01 & 0 & 1 & 00 & 01 & 00 \\ 01 & 1 & 0 & 00 & 11 & 00 \\ 10 & 0 & 0 & 01 & 00 & 11 \\ 10 & 1 & 0 & 00 & 10 & 11 \\ 11 & 0 & - & -dc- & -dc- & 11 \\ 11 & 1 & - & -dc- & -dc- & 11 \end{array}$

(3)  $z_1' = \bar{z}_1 z_0 \bar{E}$   
 $z_0' = \bar{z}_1 \bar{z}_0 \bar{E}$   
 $A_1 = z_1$   
 $A_0 = z_0 E + z_1 \bar{E}$

E	$\bar{E}$
$\bar{z}_1$	$\bar{z}_1$
$\bar{z}_0$	$\bar{z}_0$
$\sum$	0001



## II. Verzwege und Methoden der Informatik

### 1. Lazarus - Delphi - Pascal

#### 1.1 Anweisungen und Schlüsselwörter

##### Anweisung

###### Zuweisung

$a := b;$

Bedingte Verzweigung (wenn... dann...) (bzw. wenn... dann... sonst...)

$\text{if } a \text{ then } <\text{Anweisung}>;$

*(ein Semikolon!)*

$\text{if } a \text{ then } <\text{Anweisung}>;$   
 $\text{else } <\text{Anweisung}>;$

###### Kontrollstrukturen

Schleife (true ... solange...)

$\text{while } a \text{ do } <\text{Anweisung}>;$

###### Fußgesteuerte Schleife

(true ... bis...)

$\text{repeat } <\text{Anweisung}>$   
 $\text{until } a;$

###### Zählgeruste

Schleife (für... von... bis... true...)

$\text{for } i := a \text{ to } b \text{ do } <\text{Anweisung}>;$

$\text{for } i := b \text{ downto } a \text{ do } <\text{Anweisung}>;$

###### Funktionsaufruf

$\text{func}(a);$

##### Struktogramm

$a := b;$

$a?$   
ja      nein

$<\text{Anweisung}>$

$a?$

ja      nein

$<\text{Anweisung}>$

$<\text{Anweisung}>$

$\text{Solange } a \text{ wahr}$

$<\text{Anweisung}>$

$\text{bis } a \text{ wahr}$

$\text{Zähle } i \text{ von } a \text{ bis } b,$   
 $\text{Schrittweite } 1$

$<\text{Anweisung}>$

$\text{Zähle } i \text{ von } b \text{ bis } a,$   
 $\text{Schrittweite } -1$

$<\text{Anweisung}>$

$\text{func}(a)$

Jede <Anweisung> kann eine der oben angegebenen oder ein Block

zur:

$\begin{array}{l} \text{begin} \\ \quad <\text{Anweisung}> \\ \quad [ <\text{Anweisung}> ] \\ \text{end}; \end{array}$

Bspv.:  $\text{if } a \text{ then begin } <\text{Anweisung}>;$

$\quad <\text{Anweisung}>;$

$\quad \text{end} \text{ else begin } <\text{Anweisung}>;$

$\quad <\text{Anweisung}>;$

$\quad \text{end};$

#### 1.2 Bedingungen

Bedingungen ( $\text{if } <\text{Bedingung}> \text{ then } ...;$ ) sind Wahrheitswerte (Boolean).

Man verwendet Variablen vom Typ Boolean oder Vergleichsausdrücke.

Dabei sind die folgenden Operatoren erlaubt:  $=, >, <, \geq, \leq, \neq$

gleich, größer, kleiner/gleich, ungleich

Man kann Bedingungen verknüpfen: (<Bedingung>) OPERATOR (<Bedingung>)

Möglich sind die Operatoren and, or, xor, not. Klammer nicht vergessen!

Bsp.:  $\text{if true } \dots \text{ if } 1=2 \dots \text{ if } 5>3 \dots \text{ if not (false)} \dots$   
 $\text{if false } \dots \text{ if } a>b \dots \text{ if } a<b \dots \text{ if } (1<2) \text{ and } (a>b) \dots$

### 1.3 Variablen

Variablen sind Speicherplätze, die mit Daten besetzt werden können. Jede Variable hat einen Datentyp, der eindeutig festlegt, wie mit den Daten umzugehen ist. Jeder Variable ist eine eindeutige Position im Speicher zugeordnet, die Adresse. Die Deklaration (= Bekanntmachung) einer Variable erfolgt mit dem Schlüsselwort var:

	Name	Datentyp	Adresse (Bsp)
var a: integer;	a	integer	0x098E2
var a, b: integer; s: string;	a	integer	0x098E2
	b	integer	0x09FE6
	s	string	0x0A13CD

Die Position einer Variablen-deklaration entscheidet über ihren Lebensbereich. Es gibt fünf Möglichkeiten, Variablen anzulegen:

- im implementation-Bereich einer Unit  $\Rightarrow$  global:  
Die Variable kann in der gesamten Unit verwendet werden, nicht jedoch in anderen Units.
- im interface-Bereich einer Unit  $\Rightarrow$  global:  
Die Variable kann in der gesamten Unit verwendet werden sowie in allen anderen Units, die diese Unit verwenden (uses).
- im Deklarationsteil einer Funktion oder Prozedur als lokale Variable:  
Die Variable kann nur innerhalb dieser Funktion/Prozedur verwendet werden.
- im Deklarationsteil einer Funktion oder Prozedur als Parameter:  
Die Variable / der Parameter kann nur innerhalb dieser Funktion/Prozedur verwendet werden. Der Wert des Parameters wird durch den Aufruf festgelegt.
- implizit bei der Deklaration einer Funktion als Variable result:  
Die Variable result enthält den Wert, den eine Funktion zurückgeben soll.

Man unterscheidet zwischen globalen und lokalen Variablen:

- Globalen Variablen können in der gesamten Unit und ggf. weiteren Units verwendet werden. Beim Programmstart wird Speicher reserviert, beim Beenden wieder freigegeben.
- Lokale Variablen gelten nur innerhalb einer Prozedur/Funktion. Beim Aufruf der Prozedur/Funktion wird Speicher reserviert und der Inhalt der Variable auf dem Stack (Stack) abgelegt, beim Beenden des Aufrufs wieder freigegeben. Da auf diese Weise Speicher wiederverwendet wird, muss jede lokale Variable initialisiert werden (einen definierten Wert haben), um Sideeffekte zu verhindern: var i: integer;

```
begin
  if i = 5 then ...; // i kann alles mögliche sein,
                     // das Verhalten ist unvorhersehbar
end;
```

globale Variablen

lokale Variablen

Ist eine Variable deklariert und im Sichtungsbereich vorhanden, kann sowohl lesend als auch schreibend auf sie zugegriffen werden.

Das Schreiben einer Variablen nennt man Zuweisung. Die erste Zuweisung eines Wertes zu einer Variablen nennt man Initialisierung. Diese ist bei lokalen Variablen unbedingt nötig. Nur Werte mit dem Datentypen der Variablen können dieser zugewiesen werden:

```
procedure myproc;  
var i: integer;  
begin  
  i := 10;  
  if i = 10 then ...;  
end;
```

✓ korrekt

```
procedure myproc;  
var i: integer;  
begin  
  if i = 10 then ...;  
end;
```

X verursacht Seiteneffekte

```
procedure myproc;  
var i: integer;  
begin  
  i := true;  
  if i = true then ...;  
end;
```

X Zuweisung nicht erlaubt

Dies gilt insbesondere für Zeichenketten, die Zahlen enthalten:

```
procedure myproc;  
var i: integer;  
  s: string;  
begin  
  i := 10; // ✓ korrekte Initialisierung  
  s := '10'; // ✓ korrekte Initialisierung  
  i := '10'; // X '10' ist keine Zahl!  
  s := 10; // X 10 ist keine Zeichenkette!  
           // besser:  
  i := strToInt('10'); // ✓ i enthält jetzt 10 (strToInt wandelt '10' in 10 um)  
  s := intToStr(10); // ✓ s enthält jetzt '10' (intToStr wandelt 10 in '10' um)  
  s := Form1.MyEdit.Text; // } korrekte Methode, um eine  
  i := strToInt(s);      // } eingegebene Zahl einzulesen  
end;
```

## 1.4 Konstanten

Will man Schreibzugriffe unterbinden und konstante Werte als per Definitionem Konstante hoffen, verwendet man das Schlüsselwort const:

```
const c = 300000;           // Konstante vom Typ integer  
  Schule = 'Langweilig';    // Konstante vom Typ string
```

Der Typ einer Konstante wird i.d.R. automatisch erkannt.

Auch komplexere Definitionen sind möglich, dann aber mit Typangabe:

```
const Grottozahlen: array[1..6] of integer = (1, 2, 3, 4, 5, 6);
```

(Diese Art der Initialisierung ist bei globalen Variablen auch möglich.)

Konstanten können überall verwendet werden, wo auch Variablen deklariert werden können.

## 1.5 Datentypen

Jede Variable hat einen Datentypen, der festgelegt, welche Operationen mit der Variable durchgeführt werden können. So kann man bspw. Zahlen addieren (+), Wahrheitswerte jedoch nicht.

Pascal kennt einige grundlegende Datentypen, die durch eigene Zusammensetze-Datentypen ergänzt werden können.

<u>Datentyp</u>	<u>Inhalt</u>	<u>Beispiel</u>
<u>integer</u>	ganze Zahlen (4 Byte) (weitere Zahltypen: shortint, smallint, longint, int64, byte, word, longword)	i := 666; i := 1 + 2 + 3; //=> 6 i := 678 div 100; //=> 6 i := 678 mod 100; //=> 78
<u>single / double / Extended</u>	Fließkommazahlen (4/8/10 Byte) (auch: real, currency)	f := 1.2345; f := 1.25 * 2; //=> 2.5 f := 1 / 2; //=> 0.5
<u>boolean</u>	Wahrheitswert (wahr oder falsch)	b := true; b := false;
<u>char</u>	einzelles ASCII-Zeichen	c := 'a'; c := #97; //=> 'a'
<u>string</u>	Zeichenkette mit variabler Länge (intern gespeichert als:  Länge der Zeichenkette Zeichenkette)	s := 'Dies ist ein Testtext!'; s := s + '!', das Texte testet!'; s := s + #10; // Zeilenumbroch s := s + #820; // Leerzeichen (het)

Es existieren weitere Datentypen, die man auch Datenstrukturen nennt.

### 1.5.1 Arrays

Anstatt eine Sammlung vieler Variablen wie Person1, Person2, ..., Person10 zu deklarieren, verwendet man leicht handzuladende Arrays (= nummerierte Listen von Werten derselben Datentypen): var Persons: array[1..10] of string;

Allgemein schreibt man ① array [<Start>..<Ende>] of <Datentyp> oder  
② array of <Datentyp>.

Auf den Wert eines Arrays an Position i greift man dann mit myArray[i] zu.

Bsp.: procedure Gottoziehen;  
var Zahlen: array[1..6] of integer; i: integer;  
begin  
randomize;  
for i := 1 to 6 do  
Zahlen[i] := random(49) + 1;  
end;

0..48  
1..49

Bei ① handelt es sich um ein Array mit statischer Länge. Beim Array

Persons würde der Begriff Persons[11] fehlverlagen.

Ist die Länge eines Arrays erst zur Gültigkeit bekannt, dann man dynamische Arrays ② verwenden. Bei diesen ist der erste Inhalt immer 0 und die Länge kann dynamisch durch SetLength gegeben werden.

Beispiel:

```
procedure myproc;
var name: string;
    Persons: array of string;
begin
repeat
    readln(name);
    if name <> "" then begin
        SetLength(Persons, Length(Persons) + 1);
        Persons[Length(Persons)] := name;
    end;
until name = ".";
end;
```

Mit SetLength(Persons, 0) wird der gesamte Speicher wieder freigegeben.

Auch  mehrdimensionale Arrays sind möglich.

```
var a1: array[1..10, 0..5] of integer; // 2-dimensional
a2: array[1..10, 1..10, 1..10] of integer; // 3-dimensional
a3: array of array of integer; // 2-dimensional (dynamisch)
```

Zugriff: a1[1,0] oder a1[1][0]  
a2[2,4,8] oder a2[2][4][8]

### 1.5.2 Records

Records dienen (ähnlich wie Arrays) der Gruppierung mehrerer Variablen.

Allerdings handelt es sich nicht um eine Liste von Daten gleichen Typs, sondern eine Gruppierung möglicherweise unterschiedlich typisieter Variablen:

```
var Person: record
    Vorname, Nachname: string;
    Alter: integer;
end;
```

Mit Person.Vorname,  
Person.Nachname und  
Person.Alter können die  
einzelnen Felder öfter an-  
gesprochen werden.

Man kann so auch Listen von Personen anlegen:

```
var Persons: array of record
    Vorname, Nachname: string;
    Alter: integer;
end;
```

Persons[0].Vorname ist dann der Vorname der ersten Person.

Mit der with-Anweisung kann man sich Tipparbeit ersparen:

```
with Persons[0] do begin  
  Vorname := 'Jan';  
  Nachname := 'Snoeck';  
  Alter := 14;  
end;
```

### 1.5.3 Eigene Typdeklarationen

Verwendet man eigene Array- oder Recordtypen und benutzt diese mehrfach, kann man eine Typdeklaration erstellen, die Tipparbeit spart und Typsicherheit schafft. Außerdem können so auch Arrays und Records als Parameter einer Prozedur/Funktion übergeben werden.

```
type <NeuerTyp> = <AltTyp>;
```

Bsp.: type TInt = integer;  
TZeile = 0..9;  
TSchulze = 1..6;  
TWochentag = (Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag);  
TKlausur = record  
 Name: string;  
 Note: TZeile;  
end;  
TKlausuren: array of TKlausur;  
TWoche: array [TWochentag] of TKlausuren;  
var Woche: TWoche;  
procedure myproc;  
begin  
 Form1.MyLabel.Caption := Woche[Montag][0].Name;  
end;

### 1.5.4 Zeiger

Zeiger/Pointer sind Variablen, die auf einen Speicherbereich des Computers verweisen. Jede Variable hat eine Adresse im Speicher. Diese Adresse kann man in einer Zeigervariable speichern. Das hat den Vorteil, dass ein Zeiger sehr klein ist, während das Objekt, auf das er zeigt u. U. groß sein kann, was Speicherplatz spart (ähnlich wie Verknüpfungen auf dem Desktop).

Die Deklaration von Zeigen erfolgt durch das vorgestellte Zirkumflex ^.

Die Adresse einer Variable erhält man mit dem Operator @ oder der Funktion Addr. Den Inhalt eines Zeigers (die Daten, auf die er verweist) erhält man durch ein vorgestelltes Zirkumflex ^.

```

Bsp.: procedure myproc;
var zahl: integer; // zahl enthält einen integer
    zeiger: ^integer; // zeiger enthält einen Zeiger / Verweis / Referenz / Adresse
begin
    // Zahl = ?, @zahl (Adresse von Zahl) = 0xAFEE
    // zeiger = ?, @zeiger (Adresse von Zeiger) = 0xDEAD
    zeiger := @zahl; // zeiger = 0xAFEE
    // zeiger^ (Wert, auf den Zeiger zeigt) = ?
    zahl := 1234; // zeiger^ = 1234
    zeiger := 666; // Zahl = 666
    // Zahl und zeiger^ sind gleich bedeutend, da Zeiger auf Zahl zeigt.
end;

```

In Beispiel wurde bereits vorhandener Speicherplatz (Zahl) verwendet. Man kann allerdings auch neuen Speicherplatz anfordern (allozieren), man spricht von dynamischer Speicherverwaltung.

Zum Anfordern neuen Speichers nutzt man New, zum Freigeben Dispose.

```

Bsp.: procedure myproc;
var zeiger: ^integer;
begin
    New(zeiger); // Speicherplatz für einen Integer wird reserviert und dessen Adresse zeiger zugewiesen
    zeiger^. := 1337; // der neue Speicherplatz kann jetzt genutzt werden
    Dispose(zeiger); // zeiger^ wird freigegeben man könnte jetzt mit New wieder neuen Speicherplatz allozieren

```

Die Funktionen SetMem und FreeMem erfüllen eine ähnliche Funktion für untypisierte Zeiger vom Typ Pointer.

In Kombination mit Typedefinitionen kann man so sehr einfach eine verhoffte Größe deklarieren:

```

type TData = integer;
TRecord = record
    Data: TData;
    Next: PRecord;
    end;
PRecord = ^TRecord;

```

## 1.6 Operatoren

Operatoren in Pascal führen Operationen aus, vergleichen Werte usw.

### Allgemeine Operatoren

:= Zuweisungsoperator variable := Wert;

@ Adressoperator zeiger := @variable;

Arithmetische Operatoren: + - \* /      div      mod  
 für Kommazahlen      für feste Zahlen      Divisionsrest

Vergleichsoperatoren: = <> < > <= >=

Logikoperatoren: not and or xor

Zeichenkettenoperatoren: + { Konkatenation

Bitweise logische Operatoren: not and or xor shr shr

Zeigeroperatoren: ^ { Dereferenz < >

## 1.7 Units

Units sind ein Organisationskonzept von Pascal, mit dem Zusammengehörige Funktionalität in separaten Dateien, genannt Units, hinterlegt wird.

Bei Lazarus bietet es sich bspw. an, das Standardformular in einer Unit UForm und das eigentliche Programm in weiteren Units zu spezifizieren.

Jede Unit besteht aus einem interface- und einem implementation-Bereich.

- interface: Hier wird deklariert, welche Typen, Konstanten, Variablen und Funktionen/Prozeduren die Unit anderen Units bereitstellt. Diese Daten stehen zur Verfügung, wenn eine andere Unit diese mit uses anfordert.
- implementation: Die veröffentlichten Funktionen und Prozeduren (Interface) werden hier implementiert / definiert. Es können auch zusätzliche, nur für diese Unit verfügbare Funktionen, Prozeduren, Typen, Konstanten und Variablen deklariert werden.

Exemplarisches Aufbau einer Unit:

unit Unit1;

interface

```

uses Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, ...;
type ...;
const ...;
var ...;
procedure ....;
function ...;
  
```

} veröffentlichte Informationen,  
} stellt für andere Units bereit

implementation

```

uses ...; // kann auch hier stehen, um cyclische Abhängigkeiten zu vermeiden
type ...;
const ...;
var ...;
  
```

} steht nur für diese Unit zur Verfügung

```

procedure ...;
function ...;
  
```

} Implementation der oben deklarierten Operationen  
(und weiterer, privater Operationen)

end.

## 1.8 Prozeduren

Prozeduren sind Abläufe in einem Programm, die meist mehrere Anweisungen enthalten und wiederholbar werden können. Ihre Deklaration im interface-Bereich einer Unit hat die Form:

```
procedure <Name> (<Parameterliste>); // oder:  
procedure <Name>;
```

Die Definition erfolgt im implementation-Bereich der Unit:

```
procedure <Name> [<Parameterliste>]; // Parameter  
[var <Variablendeclarations>;] ① // Sowie Variablen sind optional  
begin ②  
  <Anweisung>;  
end;
```

Der Aufruf einer Prozedur lautet: <Name> [<Parameterliste>];

Die lokalen Variablen ② sind als Hilfsvariablen zu sehen, die nur innerhalb der Prozedur gültig sind und am Ende entfernt werden.

Parameter ① sind Informationen, die einer Prozedur beim Aufruf mitgegeben werden, um eine bestimmte Operation durchzuführen. Prozeduren sind also z.B. "Schüsse", Anleitungen, Vorlagen, die erst auf konkrete Werte angewendet werden können, z.B. für Berechnungen. Die Deklaration der <Parameterliste> enthält den <Variablendeclarations>. Parameter sind lokale Variablen, die beim Aufruf mit Daten befüllt werden.

Bsp.: Definition einer Prozedur:

```
procedure add(a, b : integer);  
begin  
  Form1.MyLabel.Caption := IntToStr(a + b);  
end;
```

Oder mit Hilfsvariable sum:

```
procedure add(a, b : integer);  
var sum : integer;  
begin  
  sum := a + b;  
  Form1.MyLabel.Caption := IntToStr(sum);  
end;
```

Aufruf:  
add(1, 2);  
add(a, b);  
add(333, 333);

Es gibt zwei unterschiedliche Möglichkeiten, Parameter zu übergeben:

- Wertparameter (call-by-value): Der Wert, der an die Prozedur übergeben werden soll, wird auf den Parameter (die lokale Variable) kopiert. Änderungen des Parameters / der lokalen Variablen wirken sich nicht auf die aufrufende Umgebung aus.
- Variablenparameter (call-by-reference): Die Adresse der aufgerufenen Variablen wird auf den Parameter (die lokale Variable) kopiert, sodass die aufgerufene Variable unter zwei verschiedenen Namen erreichbar ist. Ände-  
rungen wirken also auf die aufgerufene Variable.

Bsp.: Wertparameter

```
procedure p (i: integer);
begin
  i := 42;
end;
:
var i: integer;
begin
  i := 7;
  p(i);
  // i = 7 keine Veränderung
end;
```

Variablenparameter

```
procedure p (var i: integer);
begin
  i := 42;
:
var i: integer;
begin
  i := 7;
  p(i);
  // i = 42 Veränderung
end;
```

Wenn man alle Daten, von denen eine Prozedur abhängig ist, als Parameter übergibt und keine globalen Variablen mehr verwendet werden, spricht man von einer ganzfunktionalen Prozedur. Man vermeidet das versehentliche Verändern globaler Variablen, außerdem schafft man Durchsichtigkeit und Wiederverwendung und kann Prozeduren einfach in anderen Projekten wiederverwenden.

### 1.3 Funktionen

Funktionen liefern, im Gegensatz zu Prozeduren immer (genau) ein Ergebnis.

Dieses wird an der Stelle des Funktionsaufrufs eingesetzt, so dass direkt weitergedeutet werden kann. Das Rückgabewert wird durch die Pseudovariable result festgelegt.

Bsp.: funktion add (a, b: integer): integer;  
begin  
 result := a + b;  
end;

Aufruf: Form1. MyLabel. Caption := inttostr (sum(1,1));

Im Prinzip sind Funktionen ähnlich wie Prozeduren, bei denen ein Variablenparameter als Rückgabewert genutzt wird (s.o.), allerdings sind Funktionen viel einfacher zu benutzen.

## 1.10 Integrierte Prozeduren und Funktionen

`random(): extended`  $\Rightarrow$  gibt eine Zufallszahl (float) im Intervall  $[0; 1[$  zurück.

`random(int): int`  $\Rightarrow$  gibt eine Zufallszahl (int) im Intervall  $[0; \text{int}[$  zurück.

`tmp := a1;`  $\Rightarrow$  Transoperation mit einer Hilfsvariable  
`a1 := a2;`  
`a2 := tmp;`

`abs, cos, sin, sqrt, sqrt, pi(..)`  $\Rightarrow$  arithmetische Routinen

`inc / dec(..)`  $\Rightarrow$  incrementieren / dekrementieren

`min / max(.., ..)`  $\Rightarrow$  Minimum, Maximum zweier Werte bestimmen

`ord(char): int`  $\Rightarrow$  liefert den ASCII-Code eines Zeichens

`chr(int): char`  $\Rightarrow$  liefert das Zeichen zu einem ASCII-Code

`length(String): int`  $\Rightarrow$  gibt die Länge eines Strings oder Arrays zurück:

`for i:=1 to length(s) do s[i] ...;`

`s[0]`  $\Rightarrow$  Länge eines Strings (besser `length` verwenden)

`s[1, ...]`  $\Rightarrow$  n-tes Zeichen des Strings (char)

## 2. Objektorientierte Programmierung

In der OOP modelliert man Programme und Abläufe, indem man sich an Objekten

des realen Lebens orientiert. Jedes Objekt hat einen Zustand, ein Verhalten } quasi mix  
und eine Identität. Attributte Methoden } aus Record-  
und Funktionen

Eine Klasse ist der Bauplan / die Vorlage / Schablone / der „Blueprint“ für ein Objekt, genannt Instanz. Zur Laufzeit werden Objekte / Instanzen aus Klassen instanziert. Klassen beschreiben die Attribute (Eigenschaften) und Methoden (Verhaltensweisen) eines Objektes.

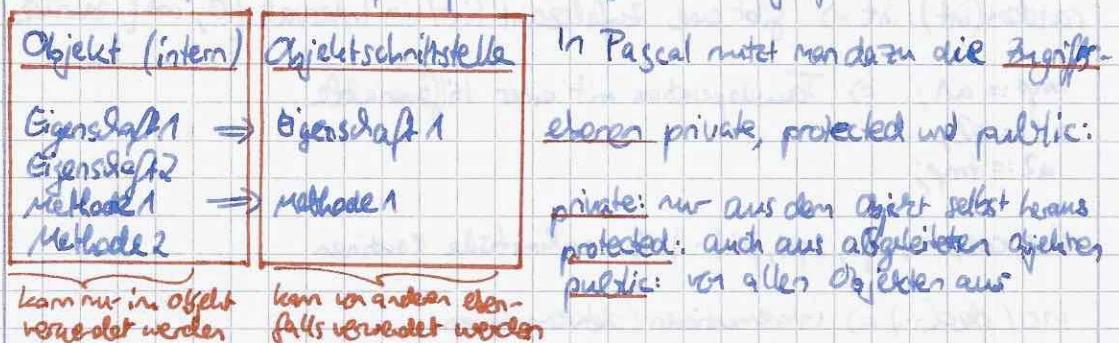
Eigenschaften beschreiben den Zustand eines Objektes und seine Beziehungen zu anderen Objekten.

Methoden beschreiben das Verhalten eines Objektes; sie werden durch Funktionen und Prozeduren implementiert.

Dadurch hat jedes Objekt eine definierte Schnittstelle (API), auf die andere Objekte zugreifen können.

## 2.1 Kapselung

Kapselung bezeichnet das Verbergen von Implementierungsdetails, so dass man von außen nicht auf Implementierungsdetails zugreifen kann.



## 2.2 Vererbung

Klassen können in hierarchischen Beziehungen stehen, in Form einer Basisklasse („Elternkategorie“) und einer abgeleiteten Klasse („Kindklasse“).

↳ Spezialisierung

Bei der Vererbung erbt (als Besitzt) die Subklasse alle public/protected Methoden und Attribute von der Basisklasse. Neue Bestandteile können hinzugefügt und vorhandene überlagert werden.

## 2.3 Properties

Properties ermöglichen den flexiblen Zugriff (lesen, schreiben, berechnen) auf Werte von privaten Eigenschaften. Dadurch kann man Schutzmechanismen einbauen, um nur den kontrollierten Zugriff zu zulassen (z.B. keine negativen Zahlen).

## 2.4 Implementierung

Eine Klasse ist wie folgt zu deklarieren (als Datentyp):

```

type TFlaeche = class
    private
        Color: string;
    protected
        function SetArea: double; virtual; abstract;
        function GetPerimeter: double; virtual; abstract;
    public
        property Area: double read SetArea; { readonly }
        property Perimeter: double read GetPerimeter; { Properties }
        property Color: string read Color write Color; //readwrite
        constructor Create(Color: string);
    end;
var MeineFlaeche: TFlaeche;
    
```

Kapselung ↗ class ist automatisch mit object gefügt wenn die Klasse aus der Hierarchie aus

Virtual: überschreibbar  
Abstract: wird hier nur benannt

property Name: Typ [read <Attribut/Methode>]  
[write <Attribut/Methode>];

## Implementation

```

constructor TFlaeche.Create (Color : string);
begin
  inherited Create; // wichtig: den Objekt-Konstruktor aufrufen
  self.Color := Color; // (eff: das Objekt selbst
end; // (inherited: Methoden der Elternklasse aufrufen)

```

constructor: verantwortlich  
für Instantiierung eines  
Objektes

```

procedure TTabletButton.Click (Sender: TObject);
begin
  MeineFlaeche := TFlaeche.Create ('blue'); // Instantiierung
end; // Zugriff mit MeineFlaeche. Color := ... etc.

```

Ergänzung durch Überbildung:

```

type TRechteck = class (TFlaeche) Basisklasse
  private _Breite, _Laenge: double;
  protected function SetArea: double; eine als virtual gekennzeichnete  
Methode überschreiben override;
  function GetPerimeter: double; override;
  public property Breite: double read _Breite write _Breite;
  property Laenge: double read _Laenge write _Laenge;
  constructor Create(Breite, Laenge: double; Color: string);
end;
var MeinRechteck: TRechteck;

```

## Implementation

```

constructor TRechteck.Create (Breite, Laenge: double; Color: string);
begin
  inherited Create(Color);
  self.Breite := Breite;
  self.Laenge := Laenge;
end;

```

} neuer Konstruktor, der den  
Konstruktor der Basisklasse  
verwendet

```

function TRechteck.SetArea: double;
begin
  result := Breite * Laenge;
end;

```

} Implementierung der  
Properties der Elternklasse

```

function TRechteck.GetPerimeter: double;
begin
  result := 2 * Breite + 2 * Laenge;
end;

```

## 2.5 Klassendiagramme

Ein UML-Diagramm stellt Klassen und deren Beziehungen untereinander dar.

Klassenname
+ an: integer + n2: integer # n3: integer
+! MA(i: integer) +? MB(i: integer)

+: public  
-: private  
#: protected  
!: Prozedur  
?: Funktion  
(...): Parameter  
.....: Rückgabewert

Bsp.:

### TFläche

```
- -Color: string  
#? SetArea: double; virtual; abstract;  
#? SetPerimeter: double; virtual; abstract;  
+? SetColor: string {Property}  
+! SetColor(string) {Konstruktor}  
+! Create(Color: String) {Konstruktor}
```

↑ (... Vererbung)

### TRechtecke

```
- -Breite: double  
- -Länge: double  
#? SetArea: double; override;  
#? SetPerimeter: double; override;  
+? SetBreite: double  
+! SetBreite(b: double) {Properties}  
+? SetLänge: double  
+! SetLänge(l: double) {Konstruktor}  
+! Create(Breite, Länge: double; Color: string)
```

### Assoziationen

#### Angestellter



Multiplizitäten: 0

Lektion od. Lek. 0..\*

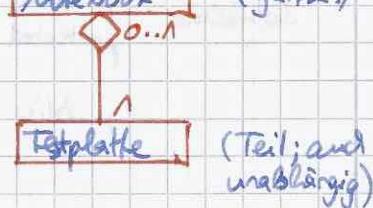
Lek. \*

1..3

1..5..7

### Aggregation

#### Notebook



### Komposition

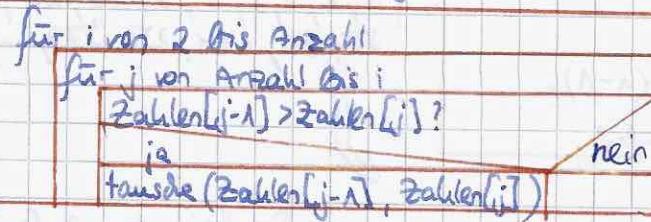
#### Kunde



## 3. Algorithmen

### 3.1 Analyse mit Tracezellen

#### Analyse des BubbleSort-Algorithmus



Tracedurchlauf (Anzahl = 6)

Zeit →

Zahlen[1]	17	9				9
Zahlen[2]	12	9 17	9			9
Zahlen[3]	22	9 12	9 17	12		12
Zahlen[4]	3	22	9 12	17	13	13
Zahlen[5]	13	9	22	13	17	17
Zahlen[6]	9 13		22		22	22
i	2	3	4	5	6	
j	6 5 4 3 2 6 5 4 3 6 5 4 6 5 6					
	unsortiert					sortiert

Man protokolliert jede Zuweisungsoperation.  
Mehrere Operationen können zusammengefasst werden.  
(Hier jeder Schleifendurchlauf.)

#### Bewertungskriterien von Sortierverfahren

- Stabilität (gleiche Werte werden nicht getauscht)
- Zeitverhalten / Speicherbedarf (Anzahl Vergleiche / Tauschvorgänge)
- Zahlsmenge (sortierte / ungleich sortierte / zufällige Zahlen)

Unterscheidung nach:

- Sortierung im Speicher
- Sortierung auf externen Medien (Festplatten, USB-Sticks etc.)

Wann sortiert man? ⇒ Damit man besser suchen kann.

- sequentiell (sehr aufwändig)
- binäres Suchen (Annäherung durch Intervalle):
  - ⇒ in der Mitte beginnen
  - wenn x gleich: festig
  - wenn x kleiner: in linken Hälfte suchen
  - wenn x größer: in rechten Hälfte suchen
  - ⇒ nur möglich, wenn die Datensätze bereits sortiert sind

### 3.2 Rekursive Algorithmen

Definition „rekursiv“: siehe „rekursiv“ („schreibt sich selbst definiert“)

Bsp.: Fakultät  $n!$  ⇒  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

$$f(n) = \begin{cases} 1; n=1 \\ n \cdot f(n-1); n>1 \end{cases}$$

Jeder rekursiven Algorithmus kann auch „linear“/iterativ gelöst werden.

Das hat z.B. den Vorteil, dass der Stack nicht belastet wird.

## Implementierung

```

function fakultaet(n: integer): integer;
begin
  if n=1 then
    result := ① Basisfall
  else
    result := n · fakultaet(n-1);
  end;
  ↴ recursive Aufruf

```

```

function fibonacci(n: integer): integer;
begin
  if (n=0) or (n=1) then
    result := ① Basisfall
  else
    result := fibonacci(n-1) + fibonacci(n-2);
  end;
  ↴ recursive Aufruf

```

## 3.3 Dynamische Datenstrukturen

Dynamische Datenstrukturen sind in ihrer Größe variabel, da sie durch New / Dispose verwaltet werden. Dadurch kann man speichersparende dynamische Arrays ("linked lists") erstellen, die allerdings auch mehr Verwaltungsmechanismen benötigen.

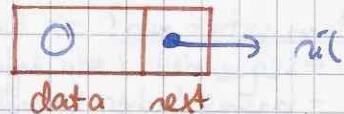
### 3.3.1 Verkettete Liste

Ein Hinweis der verketteten Liste ist wie folgt definiert:

```

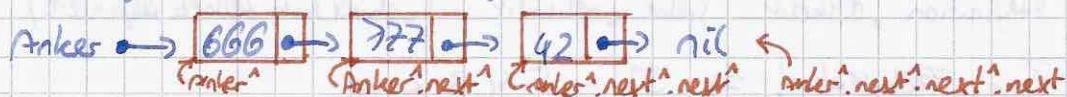
type TElement = record
  data: integer;
  next: PElement;
end;
PElement = ^TElement;

```



Eine solde Liste hat ein Kopf- oder Wurzelement, den Anker, welcher zunächst ins Leere zeigt: Anker → nil

Eine gefüllte Liste kann wie folgt aussehen:



Folgende Operationen sind mit so einer Liste möglich:

- Einfügen (am Kopf / Ende / willkürlich)
- Löschen eines Elements
- Löschen der gesamten Liste

## Fibonacci-Folge

$$fib_n = \begin{cases} 1 & \text{falls } n=0 \text{ oder } n=1 \\ fib_{n-1} + fib_{n-2} & \text{sonst} \end{cases}$$

(1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...)



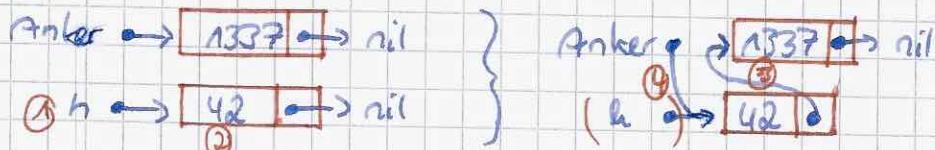
Da bei jedem Aufruf alle vorigen Fibonacci-Zahlen neu berechnet werden, sind solche rekursiven Lösungen sehr unperfekt.

## Einfügen am Kopf:

Hilfselment erstellen: ① `New(h);` //  $h \neq \text{PElement}$

②  $h^1.\text{data} := 42;$

(Falls Liste schon vorliegtens ③  $h^1.\text{next} := \text{Anker};$ ) - nicht, falls Liste noch leer  
Anker aktualisieren: ④  $\text{Anker} := h;$



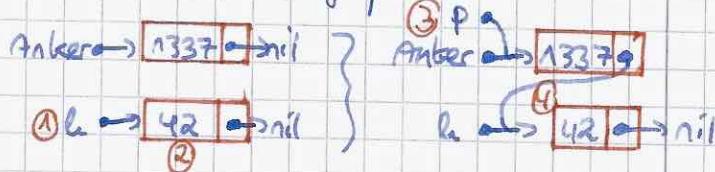
## Einfügen am Ende: Hilfselment (s.o.) ; Hilfszeiger p

$p := \text{Anker};$

while  $p^1.\text{next} \neq \text{nil}$  do

③  $p := p^1.\text{next};$

④  $p^1.\text{next} := h;$



Selante Liste löschen: ~~Anker = nil;~~ Der Speicherplatz für die Elemente  
gibt weiterhin als Belegt.~~!!!~~ !!!

```

h = Anker;
while h^1.next <= nil do begin
    tmp := h^1.next;
    Dispose(h);
    h := tmp;
end;
Anker := nil;
    
```

besser: Liste nach und nach durchgehen,  
alle Elemente freigeben und  
am Ende den Anker zurück-  
setzen.

Das Entfernen eines einzelnen Elements erfolgt ähnlich wie oben für das Einfügen besrieben.

## 3.3.2 Sortierte Liste

In einer sortierten Liste muss für ein neues Element erst die richtige Einfügeposition bestimmt werden. Der Einfügemechanismus gleicht dem der verstreuten Liste.

## 3.3.3 Stapel

Bei einem Stapel oder Stack handelt es sich um eine verketzte Liste, in der immer am Anfang eingefügt und entfernt wird (LIFO-Prinzip - last in, first out).

## 3.3.4 Schlange

Bei der Schlange handelt es sich um einen weiteren Listentyp, bei dem am Ende eingefügt sowie am Anfang entfernt wird (FIFO-Prinzip - first in, first out).

## III. Anwendung von Hard- und Softwaresystemen

### 1. Codierung

Unter Codierung versteht man Methoden zur Abbildung und Übermittlung von Nachrichten. Auch das Alphabet oder Mosikalphabet sind Codierungen.

Unterklassen sind Verschlüsselung (Codierung mit Schlüsseln) und Komprimierung (Codierung mit Platzersparnis).

Der Computer verwendet u.a. den ASCII-Code (American Standard Code for Information Interchange); Übertragungen erfolgen meistens nach zuvor vereinbarten seriellen Protokollen wie RS232: (+ Bandrate, z.B. 9600 bps)

- 8 Datenbits
- N Parität
  - E - Even: Ergänzung der Anzahl 1en zu einer geraden Anzahl
  - O - Odd: Ergänzung der Anzahl 1en zu einer ungeraden Anzahl
- 1 Stopplbit

### 2. Komprimierung

Unter verlustfreien Komprimierungsmethoden versteht man Codierungstechniken, die die Länge einer Nachricht reduzieren und wiederum die Originalnachricht durch Decodieren (verlustfrei) wiederhergestellt werden kann.

#### 2.1 Graustufen-Codierung

0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 1 1 1 0 0 0
0 1 1 0 1 1 0 0
0 1 1 1 1 1 0 0
0 1 1 1 1 1 0 0
0 1 1 0 1 1 0 0

Bei RLE (= Run Length Encoding) notiert man, wie viele 0en und 1en auftreten. Dabei definiert man, ob man mit einer 0 oder 1 anfängt.

Ohne RLE: 0 0 0 . 0 0 ^ ^ ^  
mit RLE: 5 · 0 + 3 · 1 - - - - - Erspart

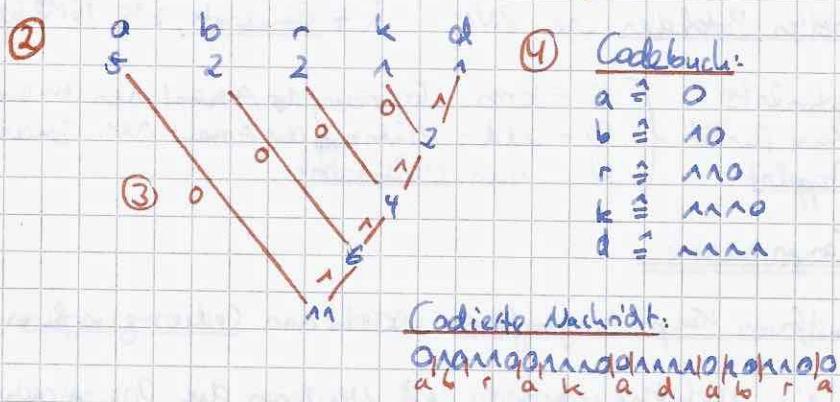
Bsp.: 8x8 Bild; beginnt mit 0: 0, 1, 6, 3, 4, 2, 1, 2, 3, 5, 3, 5, 3, 2, 1, 2, 2

#### 2.2 Huffmann-Codierung

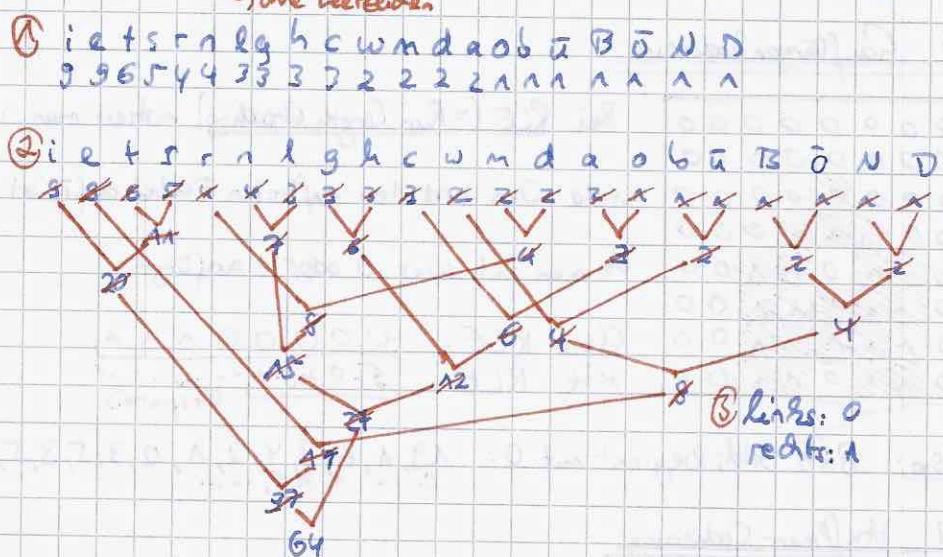
Die Idee der Huffmann-Codierung ist, dass man sich wiederholende Muster durch kürzere Muster ersetzt. Dabei werden häufige Muster durch kurze ersetzt und seltene durch längere, was insgesamt eine Komprimierung erzielt. (Die Länge der Codewörter entspricht also dem Informationsgehalt.) Für das eindeutige Dekodieren müssen die Codes präfixfrei sein (ein Codewort darf nicht im Anfang eines anderen sein), dazu verwendet man einen Huffmann-Baum.

- ① Auszählen der Häufigkeiten jedes Quellsymbols (Zeidens) und Sortieren
  - ② Für jedes Zeidens einen Knoten mit Häufigkeit notieren und wiederholen, bis nur noch ein Baum übrig ist:
    - a) Wähle die 2 Teilbäume mit der geringsten Häufigkeit.
    - b) Fasse diese beiden Bäume zu einem neuen zusammen.
    - c) Notiere die Summe der Häufigkeiten in der Wurzel.
  - ③ An jede linke Abzweigung eine 0, jede rechte eine 1 schreiben (oder andersrum).
  - ④ Für jedes Blätter das Codewort von der Wurzel aus ablesen.

Bsp.: Macht: a b r a c h a d a b r a ① a b r k d  
5 2 3 ↑ 1



Bsp.: Nachricht: Dies ist eine Nachricht die mit möglichst wenig Bits übertragen werden soll  
→ ohne Fehler.



<u>Codebrud:</u>		
i 000	c 1110	ö 011101
e 010	w 01100	n 011110
t 0010	m 10110	d 011111
s 0011	p 10111	
r 1010	a 11110	
v 1000	o 11111	
l 1001	b 011010	
g 1100	ü 011011	
h 1101	ß 011100	

### 3. Verschlüsselung

Bei Verschlüsselungen codiert man eine Nachricht mit einem Schlüssel. Zum Decodieren der Nachricht ist dieser oder ein weiterer Schlüssel nötig.  
 ↗symmetrisch      ↗asymmetrisch

#### 3.1 Cäsar-Verschlüsselung

Man verschiebt/rotiert das Alphabet um  $n$  Buchstaben.

Bsp.: Nachricht: DIESISTEINENACHRICHT ;  $n = 10$ ;  $A = K$   
 Scheintext: NSOCSODOSXO XKMRTBSMRD

##### Implementation:

```
function caesar(s: string; n: integer): string; var i, code, newCode: integer;
begin
  for i := 1 to length(s) do begin
    code := ord(s[i]);
    if code + n > ord('z') then
      newCode := code + n - 26
    else
      newCode := code + n;
    result := result + chr(newCode);
  end;
end;
```

// jedes Zeichen durchgehen  
 // Zeichencode ablegen  
 } Zeichen um  $n$  verschieben  
 }  
 // verschlüsseltes Zeichen anhängen.

#### 3.2 Skytale-Verschlüsselung

Man trägt die zu verschlüsselnde Nachricht in eine Tabelle ein mit einer Spaltenbreite  $n$  (der Schlüssel). Dann liest man die Nachricht vertikal ab.

Bsp.: Nachricht: DIESISTENACHRICHT ;  $n = 5$

1 DIESI	2 STEIN	3 ENACH	4 RICHT	5
^ ^ ^ ^ ^				

Scheintext: D S E R I T N I G E E A T S I C H L N A T

##### Implementation:

```
function skytale(s: string; n: integer): string;
var i, j, rows, currentPos: integer;
  matrix: array[1..10, 1..10] of integer; // Tabelle (10, 10 => Maximale
                                                // Zeilen-/Spaltenanzahl)
begin
  rows := length(s) div n;
  if length(s) mod n <> 0 then inc(rows);
  for i := 1 to rows do begin
    for j := 1 to columns do begin
      currentPos := (i * columns) - columns + j;
      if currentPos <= length(s) then
        matrix[i, j] := ord(s[currentPos]);
    end;
  end; for i := 1 to columns do
    for j := 1 to rows do
      result := result + chr(matrix[i, j]);
end;
```

} Anzahl Tabellezeilen berechnen  
 } Tabelle zeilenweise befüllen  
 } senkrecht ablesen

### 3.3 Vigenère - Verschlüsselung

Beim Vigenère-Vorfahren setzt man einen Schlüssel ein, um dessen Buchstaben die Buchstaben der Nachricht verschlüsseln werden (s. Caesar). Der Schlüssel wird oft wiederholt. Es handelt sich um ein polyalphabetisches Verfahren.

Bsp.: Nachricht: INFORMATIK  
 Schlüssel: VIGENÈRE (Schlüssel = VIGENÈRE)  
 Scheintext: DVLSEQ RXDS

(Urn.) ASCII: 73 78 90 73 82 77 65 84 73 75 }  
 (Schlüssel) Verschiebung: 22 8 6 4 13 4 17 4 21 8 } + (ggf. -26)  
 (real.) ASCII: 68 86 76 83 69 74 82 88 68 83

#### Implementation:

```
function vigenere(s: string; k: string): string; var i, keyPos: integer;
begin
  for i:=1 to Length(s) do begin
    keyPos := i mod Length(k);
    result := result +
      chr(
        ord('a') +
        ((ord(s[i]) + ord(k[keyPos])) - 2 * ord('a')) mod 26)
  );
end;
end;
```

### 3.4 Asymmetrische Verschlüsselung

In gegensatz zu symmetrischen Verschlüsselungsverfahren (Sender und Empfänger haben den gleichen Schlüssel) hat bei asymmetrischen Verfahren der Sender einen anderen Schlüssel (public key des Empfängers) als der Empfänger (private key).

Der Empfänger gibt offene Schlosser aus (public key).

Der Sender verschlüsselt die Nachricht damit. Mit seinem private key

kann der Empfänger die Nachricht dann entschlüsseln; der private key

bleibt geheim!  $\Rightarrow$  RSA-Verschlüsselung (Prinzipalzerlegung / ggT)

$\hookrightarrow$  Euklidischer Algorithmus

