

Informatik 1

Skript und Folien zur Vorlesung

Wintersemester 2000/2001

Prof. Dr. Karlheinz Hug

Fachhochschule Reutlingen
Hochschule für Technik und Wirtschaft
Fachbereich Elektrotechnik und Maschinenbau
Studiengang Elektronik
Federnseestraße 4
72764 Reutlingen

Inhaltsverzeichnis

Kapitel 1 Einführung

1.1	Geschichtliche Entwicklung	1
1.1.1	Vorgeschichte.....	1
1.1.2	Geschichte	4
1.2	Was bedeutet Informatik?	6
1.2.1	Merkmale von Information.....	7
1.2.2	Merkmale von Informatik.....	8
1.2.3	Beziehungen der Informatik zu anderen Fachgebieten	9
1.2.4	Teilgebiete der Informatik	10

Kapitel 2 Zeichen

2.1	Zeichen, Alphabete und Symbole.....	1
2.2	Codes und Codierungen.....	3
2.3	ASCII-Code	4
2.4	Unicode.....	5
2.5	Darstellungen von Größen.....	5

Kapitel 3 Rechensysteme

3.1	Was ist ein Rechensystem?	1
3.1.1	Hauptbestandteile	2
3.1.2	Charakteristika heutiger Rechner	4
3.1.3	Arten von Rechensystemen	5
3.1.3.1	Anwendungsbereiche.....	5
3.1.3.2	Größen	5
3.2	Hardware.....	6
3.2.1	Bestandteile	6
3.2.2	Peripheriegeräte	7
3.2.2.1	Ein-/Ausgabegeräte	7
3.2.2.2	Hintergrundspeicher	8
3.2.2.3	Geräte zur Interaktion mit Umgebung.....	9
3.2.2.4	Anschlüsse an Rechnernetze	9
3.2.3	Zentraleinheiten	10
3.2.3.1	Hauptspeicher	11
3.2.3.2	Speichereinheiten.....	12
3.2.3.3	Speicherkapazitäten und Zugriffszeiten bei PCs	13
3.2.3.4	Hauptspeicher dargestellt in Component-Pascal-Notation... ..	14
3.2.3.5	Prozessor.....	15
3.2.3.6	Befehlszyklus in Component-Pascal-Pseudocode.....	16
3.2.4	Das von Neumann'sche Konzept der Rechnerarchitektur	17
3.3	Software.....	19

3.3.1	Klassifizierung.....	19
3.3.2	Betriebssysteme	20
3.3.2.1	Microsoft MS-DOS	21
3.3.2.2	Microsoft Windows 3.1	22
3.3.2.3	Microsoft Windows 95, 98	22
3.3.2.4	Microsoft Windows NT, 2000	22
3.3.2.5	Das lokale PC-Netz des SG Elektronik	23

Kapitel 4 Programmiersprachen

4.1	Was ist eine Programmiersprache?	1
4.2	Abstraktionsebenen.....	2
4.3	Werkzeuge	4
4.4	Syntax, Semantik, Pragmatik.....	6
4.5	Fehlerarten	9
4.6	Denkmodelle und Konzepte	10
4.7	Skizze der geschichtlichen Entwicklung	11

Kapitel 5 Ein erster Blick auf BlackBox und Component Pascal

5.1	Etwas Geschichte	1
5.2	Was ist.....	3
5.2.1	...Component Pascal?	3
5.2.2	...BlackBox Component Builder?	3
5.3	Merkmale von Component Pascal	4
5.4	Merkmale von BlackBox	5
5.5	Programmstruktur von Component Pascal	6
5.5.1	Was ist ein Modul?	6
5.5.2	Woraus besteht ein Modul?	6
5.5.3	Einfaches Beispiel	7
5.5.4	Konzept der Datenabstraktion	8
5.5.5	Modulkonstrukt	9
5.5.5.1	Syntax	9
5.5.5.2	Muster	10
5.6	BlackBox-Verzeichnisstruktur	11
5.6.1	Subsysteme	11
5.6.1.1	Standard-Subsysteme.....	12
5.6.1.2	FB-EM-Subsysteme.....	13
5.6.2	Module und Dateien	14
5.6.3	Unterverzeichnisse und Dateien	15
5.6.4	Namengebung.....	16
5.6.4.1	Beispiel	17
5.7	BlackBox-Werkzeuge	18
5.8	Oberon-Entwicklungsumgebungen	21

5.9	Informationen im Internet.....	22
-----	--------------------------------	----

Kapitel 6 Grundelemente der Lehrsprache

6.1	Erweiterte Backus-Naur-Form.....	1
6.2	Terminale Symbole	3
6.2.1	Namen.....	4
6.2.2	Zahlen	5
6.2.3	Zeichen	6
6.2.4	Zeichenketten	7
6.2.5	Operatoren und Begrenzer	8
6.2.6	Kommentare	9
6.2.7	Vordefinierte Namen.....	10
6.3	Grundtypen	11
6.4	Ablaufstrukturblöcke	13
6.5	Elementare Ein-/Ausgabe	15

Kapitel 7 Algorithmen

7.1	Was ist ein Algorithmus?.....	1
7.1.1	Beispiele	1
7.1.2	Merkmale von Algorithmen	1
7.2	Darstellung und Entwicklung von Algorithmen.....	2
7.2.1	Anforderungen.....	2
7.2.2	Was ist ein Programm?	3
7.2.3	Modell des Wegs vom Problem zur Lösung.....	4
7.2.3.1	Tätigkeiten	4
7.2.3.2	Vorgegebenes und Ergebnisse	5
7.2.3.3	Akteure	5
7.2.3.4	Darstellungen.....	5
7.3	Entwurf von Algorithmen.....	5
7.3.1	Prinzipien.....	6
7.3.1.1	Abstraktion	6
7.3.1.2	Zerlegung.....	6
7.3.1.3	Strukturierung.....	6
7.3.2	Konzepte und Methoden.....	6
7.3.2.1	Top-down-Entwurf	6
7.3.2.1.1	Beispiel: Sieb des Eratosthenes	6
7.3.2.2	Schrittweise Verfeinerung.....	7
7.3.2.3	Strukturierte Programmierung.....	7
7.3.2.4	Zerlegung in Kommandos und Prozeduren	7
7.3.3	Graphische Darstellungsmittel	8
7.3.3.1	Flußdiagramme	8
7.3.3.2	Struktogramme	8

7.3.3.2.1	Hinweise zu Struktogrammen.....	8
7.3.3.2.2	Regeln zum Programmieren	9
7.3.3.3	Zustandsübergangsdiagramme	9
7.3.3.4	Petri-Netze	10
7.4	Korrektheit von Algorithmen	10
7.5	Komplexität von Algorithmen	10

Kapitel 8 Daten

8.1	Exemplare und Typen	1
8.2	Werte und Namen	2
8.3	Vereinbarung und Benutzung	3
8.4	Konstanten und Variablen	4
8.5	Vordefinierte und definierbare Datentypen.....	5
8.6	Einfache Datentypen.....	6
8.6.1	Zeichen	6
8.6.2	Zahlen und Arithmetik	7
8.6.2.1	Ganze Zahlen	8
8.6.2.2	Reelle Zahlen und Gleitpunktzahlen	9
8.6.2.2.1	Darstellungen von Gleitpunktzahlen	10
8.6.2.2.2	Maschinendarstellung reeller Zahlen.....	12
8.6.2.2.3	Fehler bei Gleitpunktzahlen.....	13
8.6.2.2.4	Gleitpunktzahlenarithmetik	14
8.6.3	Logische Größen und Ausdrücke	16
8.7	Strukturierte Datentypen.....	16
8.7.1	Reihungen	16
8.7.2	Zeichenketten	17
8.7.3	Verbunde	17
8.7.4	Mengen	17
8.7.5	Zeiger.....	18
8.7.6	Dateien.....	18
8.7.7	Datenbanken	18

Kapitel 9 Vereinbarungen

9.1	Arten von Vereinbarungen	1
9.1.1	Syntax	1
9.2	Konstantenvereinbarungen	2
9.2.1	Beispiele	2
9.2.2	Syntax	2
9.2.3	Semantik	3
9.2.4	Wozu und wie sind Konstanten zu benutzen?	4
9.3	Typvereinbarungen	5
9.3.1	Beispiele	5

9.3.2	Syntax	6
9.3.3	Semantik	7
9.4	Variablenvereinbarungen	8
9.4.1	Beispiele	8
9.4.2	Syntax	8
9.4.3	Semantik	9
9.4.4	Initialisierung von Variablen	10
9.5	Prozedurvereinbarungen	12
9.5.1	Beispiele	12
9.6	Namengebung	13
9.7	Reihenfolge von Vereinbarungen	14

Kapitel 10 Ausdrücke

10.1	Syntax	2
10.2	Operatoren	3
10.2.1	Arten	3
10.2.2	Vorrangregeln	3
10.2.3	Reihenfolge der Operandenauswertung	3
10.2.4	Überladungen	4
10.3	Typregeln	4
10.3.1	Typbindung	4
10.3.1.1	Statische Typbindung	5
10.3.2	Typeinschluß	6
10.3.3	Typanpassungen	7
10.3.3.1	Implizite Typanpassung	8
10.3.3.2	Explizite Typanpassung	8
10.3.4	Ausdruckskompatibilität	9
10.4	Arithmetische Ausdrücke	10
10.5	Relationale Ausdrücke	11
10.6	Boole'sche Ausdrücke	11
10.7	Zeichenkettenausdrücke	13
10.8	Mengenausdrücke	13
10.9	Vordeclarierte Funktionen	14

Kapitel 11 Anweisungen

11.1	Arten von Anweisungen	1
11.1.1	Syntax	2
11.1.2	Formatierung	2
11.2	Einfache Anweisungen	3
11.2.1	Wertzuweisungen	3
11.2.1.1	Beispiele	3
11.2.1.2	Syntax	3

11.2.1.3	Semantik	4
11.2.1.4	Typregel und Zuweisungskompatibilität	4
11.2.2	Prozeduranweisungen	5
11.2.2.1	Beispiele	5
11.2.3	Sprunganweisungen.....	5
11.3	Strukturierte Anweisungen	6
11.3.1	Auswahanweisungen	6
11.3.1.1	Ein-/zwei-/mehrfache Auswahl	6
11.3.1.2	Mehrfache Auswahl.....	10
11.3.1.3	With-Anweisung	13
11.3.2	Wiederholungsanweisungen	14
11.3.2.1	Syntax	14
11.3.2.2	Allgemeine Semantik	14
11.3.2.3	Bedingungsschleifen.....	15
11.3.2.4	Bedingungslose Schleifen.....	22
11.3.2.5	Zählschleifen	25
11.4	Vordeclarierte Prozeduren	28
11.4.1	Alphabetisch sortierte Liste	28
11.4.2	Programmabbruch	29
11.4.2.1	Beispiel	29
11.4.3	Zusicherungen	30
11.4.3.1	Allgemeine Definition	30
11.4.3.2	Bedeutung.....	30
11.4.3.3	Realisierung	31
11.4.3.4	Beispiel	31
11.4.3.5	Wozu sind Zusicherungen nütze?	32
11.4.3.6	Programmierrichtlinie.....	32

Kapitel 12 Prozeduren

12.1	Allgemeines	2
12.1.1	Kennen wir schon Prozeduren?	2
12.1.2	Was kommt Neues hinzu?	3
12.1.3	Wozu brauchen wir Prozeduren?	4
12.1.4	Merkmale von Prozeduren	5
12.2	Gewöhnliche Prozeduren.....	6
12.2.1	Beispiel	7
12.3	Funktionen	8
12.3.1	Beispiel	9
12.4	Syntax	10
12.5	Semantik	11
12.5.1	Die Return-Anweisung.....	12
12.5.1.1	Programmierregel	12

12.6	Empfehlung zur Namengebung	13
12.6.1	Gewöhnliche Prozedur	13
12.6.2	Funktion.....	14
12.6.3	Boole'sche Funktion	15
12.7	Datenübergabekonzepte.....	16
12.7.1	Formale und aktuelle Parameter	17
12.7.2	Wertübergabe	21
12.7.3	Referenzübergabe	22
12.7.4	Resultatübergabe	24
12.7.5	Spezielle Regeln bei Funktionen	25
12.7.6	Programmierrichtlinien.....	26
12.8	Existenz und Sichtbarkeit vereinbarter Größen.....	28
12.8.1	Lokale und globale Größen	28
12.8.2	Existenzdauern von Variablen	28
12.8.3	Sichtbarkeitsbereiche.....	29
12.9	Aliasing.....	30
12.9.1	Programmierregeln	30
12.10	Seiteneffekte	31
12.10.1	Programmierregeln	32
12.11	Spezifikation von Prozeduren.....	33
12.11.1	Merkmale von Spezifikationen.....	34
12.11.2	Das Kunden-Lieferanten-Modell.....	35
12.11.3	Spezifikation mit Vor- und Nachbedingungen	36

Kapitel 13 Definierbare Datentypen

13.1	Typkonstruktoren und Typangaben	2
13.1.1	Beispiele für Konstruktionen.....	2
13.1.2	Arten von Typangaben.....	3
13.1.3	Anonyme Typen	4
13.1.3.1	Programmierregel	4
13.2	Typnamen und Typvereinbarungen	5
13.2.1	Beispiel für Namengebung	5
13.3	Strukturierte Datentypen.....	6
13.3.1	Allgemeine Eigenschaften.....	6
13.3.2	Reihungen.....	7
13.3.2.1	Beispiele für Typangaben	7
13.3.2.2	Definition.....	8
13.3.2.3	Typangabe.....	9
13.3.2.4	Syntax	9
13.3.2.5	Eigenschaften von Reihungen als Ganzes	10
13.3.2.6	Eigenschaften von Elementen von Reihungen	12
13.3.2.7	Eindimensionale Reihungen	15

13.3.2.8	Zweidimensionale Reihungen	16
13.3.2.9	Mehrdimensionale Reihungen.....	18
13.3.2.10	Offene Reihungen.....	19
13.3.3	Zeichenketten	20
13.3.3.1	Definition.....	20
13.3.3.2	Realisierung	20
13.3.3.3	Unterschiede zwischen	21
13.3.3.4	Zeichenkettenvariablen.....	22
13.3.3.5	Bearbeitung von Zeichenketten	23
13.3.4	Verbunde	26
13.3.4.1	Beispiele für Typangaben	26
13.3.4.2	Definition.....	27
13.3.4.3	Typangabe.....	27
13.3.4.4	Syntax	28
13.3.4.5	Eigenschaften von Verbunden als Ganzes	28
13.3.4.6	Eigenschaften von Verbundfeldern.....	30
13.3.4.7	Sichtbarkeitsregeln für Feldnamen	32
13.4	Typgleichheit	34
13.4.1	Lasche Definition	34
13.4.2	Wozu ist Typgleichheit nütze?.....	35
13.4.3	Beispiele	35

Kapitel 14 Sortiervverfahren

14.1	Einfache Verfahren	2
14.2	Höhere Verfahren.....	2
14.3	Aufwandsabschätzung	3

Kapitel 15 Modularisierung und Datenabstraktion

15.1	Konzept der Modularisierung	2
15.2	Konzept der Datenabstraktion	3
15.3	Modulare Programmierung.....	4
15.4	Getrennte Übersetzbarkeit	4

Literaturverzeichnis

Nachspann-1

Kapitel 1

Einführung

„When I’m drivin’ in my car
And that man comes on the radio
And he’s tellin’ me more and more
About some useless information
Supposed to fire my imagination“

Mick Jagger, Keith Richards,
(*I Can’t Get No*) Satisfaction, 1965.

1.1 Geschichtliche Entwicklung

1.1.1 Vorgeschichte

4000 v.u.Z. Ägypten: Zahlzeichen.

2000 v.u.Z. Babylon: Zahlensystem mit Stellenwerten.

1700 v.u.Z. Ägypten: Rechenbretter; Papyrus Rhind, älteste schriftliche Rechenaufgaben.

1100 v.u.Z. China: Suan-pan, chinesische Form des Rechenbretts.

300 v.u.Z. Euklid, griechischer Mathematiker, Euklidischer Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen.

um 60 Heron von Alexandrien, Mechaniker, beschreibt zahlreiche Automaten und z.B. mechanische Zählräder zur Wegmessung.

5 Jh. Indien: Dezimalsystem wird erfunden, ab 600 voll ausgebildet.

um 820 Mohammed Ibn-Musa Al-Chowarizmi (ca. 780 - 850), arabisch-persischer Mathematiker und Astronom, schreibt ein einflußreiches Buch über Algebra. Von ihm bzw. seinem Namen stammen die Bezeichnungen Algebra und Algorithmus ab.

um 1200 Arabische Ziffern werden in Europa eingeführt.

1202 Leonardo von Pisa, genannt Fibonacci (ca. 1180 - 1240), italienischer Mathematiker, verfaßt die erste systematische Einführung in das dezimale Zahlenrechnen.

1524 Adam Riese (1492 - 1559), deutscher Rechenmeister, veröffentlicht ein Rechenbuch, in dem er die Rechengesetze des Dezimalsystems darstellt. Das Rechnen mit der Feder verdrängt das Rechnen auf Linien. Das Dezimalsystem setzt sich in Europa durch.

- 1588** Jost Bürgi (1552 - 1632) stellt erste Logarithmentafeln zusammen.
- 1614** Lord John Napier of Merchiston (1550 - 1617), schottischer Gelehrter, der die Logarithmen unabhängig von Bürgi entdeckt hat, führt logarithmische Rechenstäbe ein, die auch Multiplikationen und Divisionen ermöglichen und so das Rechnen erleichtern.
- 1622** William Oughtred (1574 - 1660) entwickelt den Rechenschieber mit zwei logarithmischen Skalen.
- 1623** Wilhelm Schickard (1592 - 1635), Theologe, Mathematiker und Astronom in Tübingen, beschreibt eine Maschine, die die vier Grundrechenarten ausführen kann. Sie bleibt unbeachtet, wird vergessen und erst 1958 - 1960 rekonstruiert.
- 1641** Blaise Pascal (1623 - 1662), französischer Mathematiker und Philosoph, konstruiert eine Maschine, die sechstellige Zahlen addieren kann.
- 1673** Gottfried Wilhelm Leibniz (1646 - 1716), deutscher Philosoph, Mathematiker, Physiker und Jurist, demonstriert eine Rechenmaschine für die vier Grundrechenarten mit Staffelwalzen und dezimaler Stellenverschiebung.
- Die ersten Rechenmaschinen funktionieren wegen unzulänglicher Feinmechanik nicht zuverlässig.
- 1703** Leibniz begründet das duale Zahlensystem *Arithmetica Dyatica*.
- 1774** Philipp Matthäus Hahn (1739 - 1790), württembergischer Pfarrer und Uhrmacher, entwickelt die erste zuverlässig arbeitende mechanische Rechenmaschine.
- 1805** Der Franzose Joseph-Marie Jacquard (1752 - 1834) fertigt einen Musterwebstuhl, der durch gelochte Holzplättchen automatisch gesteuert wird.
- ab 1818** Charles Xavier Thomas (1785 - 1870), französischer Versicherungsunternehmer, organisiert die serienmäßige Herstellung mechanischer Rechenmaschinen nach dem Vorbild der Leibnizschen Maschine, um den gewachsenen Bedarf an Rechenerleichterung bei Versicherungen, Handel, Banken und der Industrie zu befriedigen.
- 1822** Charles Babbage (1792 - 1871), englischer Mathematiker, Ingenieur und Ökonom, stellt ein Anschauungsmodell seiner *Difference Engine* her. Diese nach Jacquards Lochkartenprinzip programmgesteuerte mechanische Rechenmaschine soll der Erstellung astronomischer und nautischer Tabellen dienen.
- 1833** Babbage entwirft seine *Analytical Engine*. Dieser universelle Rechenautomat weist schon alle Prinzipien und Baugruppen heutiger Rechner auf: Rechenwerk, digitaler Zahlenspeicher, Ein-/Ausgabewerke und ein Steuerwerk, das Programmsprünge - bedingte Verzweigungen und Wiederholungen - erlaubt. Babbages Versuche scheitern an Mängeln der technischen Ausführung und werden vergessen.
- 1847** George Boole (1815 - 1864), englischer Logiker und Mathematiker, behandelt logische Probleme algebraisch und begründet die Boole'sche Algebra.

- 1886** Hermann Hollerith (1860 - 1929), Ingenieur und Beamter im Statistischen Bundesamt der USA, erfindet die Lochkarte. 1889 erhält er ein Patent auf ein Lochkartensystem zum Zählen und Sortieren von Daten, das bei Volkszählungen in den USA und Österreich eingesetzt wird.
- 1896** Hollerith gründet die Tabulating Machine Company, Vorläufergesellschaft der IBM.
- 1914** Leonardo Torres y Quevedo (1852 - 1936) führt die Gleitpunktzahlendarstellung in den Rechenmaschinenbau ein.
- 1928** Lochkartenmaschinen werden für wissenschaftliche Berechnungen verwendet.
- 1934** Konrad Zuse (1910 - 1995), deutscher Bauingenieur, plant eine programmgesteuerte Rechenmaschine mit Dual- und Gleitpunktzahlendarstellung für baustatische Berechnungen.
- 1936** Alan Turing (1912 - 1954), britischer Mathematiker, veröffentlicht eine Diskussion theoretisch unlösbarer Probleme und beschreibt ein mathematisches Modell einer Rechenmaschine, die als einfachst möglicher, universeller Rechenautomat endliche mathematische und logische Probleme lösen kann.
- 1938** Zuse fertigt die mechanische Z1.
- 1939** John Atanasoff (geb. 1903) und Clifford Berry (1919 - 1963) bauen eine binäre Rechenmaschine, die lineare Gleichungssysteme löst.
- 1941** Zuses elektromechanische Z3 ist der erste funktionsfähige frei programmierbare Rechner (2600 Relais, Multiplikationszeit 4 s, Programmierung mit Lochstreifen). Ihre Entwicklung wird durch Rüstungsaufträge finanziert, die sich auf aerodynamische Berechnungen an Militärflugzeugen und ferngesteuerten Bomben beziehen.
- 1944** M. H. A. Newman leitet in Bletchley Park, Großbritannien, ein streng geheim gehaltenes militärisches Projekt, an dem Turing mitarbeitet und bei dem eine Reihe elektronischer Röhrenrechner mit der Bezeichnung *Colossus* entwickelt und gebaut werden. Sie werden erfolgreich bei der Entschlüsselung von Nachrichten der deutschen Wehrmacht eingesetzt.
- Howard H. Aiken (1900 - 1973), Mathematiker an der Harvard Universität, Boston, USA, baut den elektromechanischen Rechner *Mark I* (Gewicht 35 t, Multiplikationszeit 6 s, Dezimalzahlendarstellung) für ballistische Berechnungen.
- 1945** Zuse entwickelt mit dem *Plankalkül* die erste Programmiersprache und nimmt den Z4 in Betrieb.
- 1946** J. Prosper Eckert (geb. 1919) und John W. Mauchly (1907 - 1980), Universität von Pennsylvania, USA, vollenden den *ENIAC* (*Electronic Numerical Integrator And Computer*), den ersten voll elektronischen Rechner (Grundfläche 140 m², 18000 Elektronenröhren, Multiplikationszeit 3 ms, Programmierung mit Steckbrettern) für Berechnungen zur Weiterentwicklung der Atombombe.

John von Neumann (1902 - 1957), aus Ungarn stammender Mathematiker, schlägt die interne Speicherung des Programms vor und konzipiert den Rechenautomaten *EDVAC* (*Electronic Discrete Variable Automatic Computer*).

1948 Norbert Wiener (1894 - 1964) begründet die Kybernetik, Claude E. Shannon (geb. 1916) die Informationstheorie.

John Bardeen, William Shockley und Walter Brattain erfinden in den Bell Laboratories den Transistor.

1949 Maurice V. Wilkes, Mathematiker an der Universität Cambridge, Großbritannien, baut *EDSAC* (*Electronic Delay Storage Automatic Calculator*), den ersten Digitalrechner mit intern gespeichertem Programm.

ab 1950 Industrielle Rechnerproduktion.

1.1.2 Geschichte

Die Entwicklung der Rechensysteme in den letzten vier Jahrzehnten weist allgemein folgende Merkmale auf.

■ Die Hardware ist durch Miniaturisierung und Massenproduktion der Komponenten gekennzeichnet.

- Verarbeitungsgeschwindigkeiten und Speicherkapazitäten wachsen.
- Platzbedarf und Energieverbrauch sinken.
- Preise fallen.
- Zuverlässigkeit nimmt zu.

■ Software:

- Die Komplexität der bearbeiteten Probleme wächst.
- Methoden und Werkzeuge zur Software-Entwicklung werden besser und mächtiger.
- Gegenüber der rasanten Entwicklung neuer Hardware-Technologien und -Möglichkeiten verläuft die Entwicklung im Software-Bereich weniger dynamisch.
- Der Anteil der Kosten für Software an den Kosten von Rechensystemen wächst.

■ Anwendungen:

- Es werden immer mehr Anwendungsbereiche erschlossen.
- Die gestellten Anforderungen werden immer anspruchsvoller.

Die folgende Skizze der Entwicklung hält sich an die Einteilung der Rechner in Generationen, die sich an der zugrundeliegenden Technik orientiert.

bis 1958 Rechner der 1. Generation

Hardware: Elektronenröhren als Schaltelemente; Befehlszeiten 10 ms - 1 s; Hauptspeicherkapazitäten bis 100 Zahlen; Probleme mit ungenügender Zuverlässigkeit.

Software: embryonaler Zustand; maschinenabhängige Assembler- und Makro-Assemblersprachen; erste Betriebssysteme; Ein-/Ausgabehilfen.

Organisation: Betrieb, Programmierung und Nutzung eines Rechners in einer Hand.

Anwendungen: militär-technische Berechnungen; kommerzielle Datenverarbeitung; Lohnabrechnungen; Finanz- und Rechnungswesen.

1954 - 1956 John Backus entwickelt mit einem Team bei IBM *Fortran*, eine der ersten höheren, maschinenunabhängigen Programmiersprachen, die weite Verbreitung findet.

1958 - 1966 Rechner der 2. Generation

Hardware: Transistoren als Schaltelemente; Befehlszeiten 1 - 10 ms; Ferritkernspeicher als Hauptspeicher, Kapazitäten bis einige 1000 Zahlen; Magnetband-, -trommel- und -plattenspeicher als Hintergrundspeicher; Lochkartenleser als Eingabemedien; Fernschreiber als Ausgabemedien; erste Schritte zur Datenfernverarbeitung.

Software: Programmiersprachen *COBOL*, *Algol 60*, *PL/1*; Übersetzer; Betriebssysteme; Stapel-, Mehrprogramm- und Mehrprozeßbetrieb.

Organisation: Rechenzentren; Trennung zwischen OperateurInnen und ProgrammiererInnen.

Anwendungen: wissenschaftlich-technische Berechnungen; Echtzeitprobleme; Militär; Raumfahrt; Platzreservierung im Luftverkehr; Betriebsdatenverarbeitung in der Produktion.

1964 IBM kündigt die erste Rechnerfamilie *IBM System/360* an.

1968 - 1971 Niklaus Wirth, ETH Zürich, entwickelt die Programmiersprache *Pascal*.

1966 - 1974 Rechner der 3. Generation

Hardware: Integrierte Schaltkreise (Mikromodule); Befehlszeiten 100 μ s - 1 ms; Halbleiterspeicher als Hauptspeicher, Kapazitäten bis einige 100 KByte; Tastaturen als Eingabemedien; erste Bildschirme als Ausgabemedien; Großrechner; Mehrzwecksysteme; hierarchische Rechensysteme mit zentralem Gastrechner und untergeordneten Kleinrechnern.

Software: Software-Lücke, Software-Krise; Software Engineering als eigene Disziplin; Software-Entwicklungsmethoden; Datenstrukturen, strukturierte Programmierung; Programmiersprache *C*; Betriebssysteme *IBM OS/360*, *Unix*; Echtzeitprogrammiersprachen und -betriebssysteme; Kommandosprachen; interaktiver Teilnehmerbetrieb; graphische Datenverarbeitung.

Organisation: IBM beginnt, Hardware und Software getrennt zu verkaufen; Berufsbilder und Studiengänge für InformatikerInnen entstehen.

Anwendungen: Echtzeitsteuerungen in Fabrikationsanlagen mit Kleinrechnern; Datenerfassungsanlagen; automatische Meß- und Prüfstände; Prozeßsteuerungen.

1974 - 1982 Rechner der 4. Generation

Hardware: Hochintegrierte Schaltkreise; 10000e von Transistorfunktionen auf einem Chip; Mikroprozessoren: ein Prozessor auf einem Chip; 8- und 16-Bit-Architekturen; Befehlszeiten 10 - 100 μ s; mehrere Kilobit auf einem Chip; Hauptspeicherkapazitäten bis einige MByte; erste Arbeitsplatzrechner; Maus als Eingabegerät; Rechnernetze; lokale Netzwerke.

Software: Modularisierung; Objektorientierung; Programmiersprachen *Modula*, *Ada*, *Smalltalk*; Fenstertechnik für Benutzungsoberflächen; Datenbanken.

Organisation: Dezentralisierung von Datenverarbeitungsaufgaben in Fabriken, Banken, Behörden.

Anwendungen: CAD (Computer Aided Design); CAM (Computer Aided Manufacturing); Integration von kommerzieller Datenverarbeitung mit rechnergestützten Fertigungssystemen.

1981 IBM bringt PCs (Personal Computer) mit dem Betriebssystem *Microsoft MS-DOS* auf den Markt.

ab 1982 Rechner der 5. Generation

Hardware: Höchstintegrierte Schaltkreise; 100000e von Transistorfunktionen, mehrere Prozessoren, mehrere hundert Kilobit bis mehrere Megabit auf einem Chip; 32- und 64-Bit-Architekturen; Befehlszeiten 100 ns - 10 μ s; Hauptspeicherkapazitäten bis einige Dutzend MByte; Heimrechner; Mikrorechner-Netzwerke; Superrechner; parallele Rechnerarchitekturen; verteilte Rechensysteme.

Software: Programmiersprachen *C++*, *Eiffel*, *Oberon*, *Java*; CASE (Computer Aided Software Engineering).

Anwendungen: CIM (Computer Integrated Manufacturing).

Zukunft Hardware: MIMD-Rechner (Multiple Instruction Multiple Data) mit tausenden von Prozessoren, Befehlszeiten im Nanosekunden-Bereich; Hauptspeicher im Gigabyte-Bereich, wachsende Vernetzung der Rechner.

Software: Sprachen der „5. Generation“, z.B. für Expertensysteme, neuronale Netze.

Anwendungen: Flexible Fertigungssysteme.

1.2 Was bedeutet Informatik?

Das Wort Informatik wurde in den 60er Jahren geprägt, etwa nach der Formel

Informatik := *Information* + *Automatik*.

Informatik wird oft als Wissenschaft der automatischen Verarbeitung von Informationen bezeichnet. Der Begriff der **Information** ist jedoch sehr komplex und umfassend; wir versuchen daher nicht, ihn zu definieren, sondern nennen unten nur einige Merkmale.

Vom Begriff „Information“ ist der Begriff „Daten“ zu unterscheiden: **Daten** (*data*) sind Informationen in maschinell verarbeitbarer, speicherbarer und übermittelbarer Form.¹ Die Norm DIN 44300 bestimmt den Begriff so [Kle89], S.25:

„**Daten:** Gebilde aus Zeichen oder kontinuierliche Funktionen, die aufgrund bekannter oder unterstellter Abmachungen Information darstellen, vorrangig zum Zweck der Verarbeitung oder als deren Ergebnis.“

Damit Informationen durch Maschinen verarbeitet werden können, müssen sie in formalisierter Darstellung vorliegen. Die *Formalisierung* von Informationen ist immer mit einer Reduktion, einem Informationsverlust verbunden; manche Informationen lassen sich gar nicht formalisieren. Der Begriff „Daten“ ist also viel enger als der Begriff „Information“.

Beispiel: „Wenn Blicke töten könnten...“ - Wie sollte man die Informationen, die sich Menschen mit Blicken mitteilen können, formalisieren?

Es ist also angebracht, unter Informatik die Wissenschaft von der maschinellen, insbesondere automatischen Verarbeitung von Daten zu verstehen. Freilich müssen maschinell verarbeitete Daten von Menschen als nützliche Informationen interpretierbar sein, andernfalls ist Datenverarbeitung sinnlos.

Die datenverarbeitenden Maschinen heißen **Rechner** (*computer*) oder **Datenverarbeitungsanlagen** (DVA). In dem gebräuchlichen Begriff **elektronische Datenverarbeitung** (EDV) wird zum Ausdruck gebracht, daß die heutigen Rechner mit der Basistechnologie Elektronik realisiert sind.²

Im englischen Sprachraum ist der Begriff **computer science** (Rechnerwissenschaft) verbreitet. Dieser Begriff ist allerdings noch enger, da er nur an den Rechner gekoppelt, also *maschinenorientiert* ist.

Demgegenüber bietet der Begriff Informatik die Möglichkeit, ihn *arbeitsorientiert* oder *menschenorientiert* aufzufassen. Dazu gehört, die Zwecke, zu denen der Mensch die maschinelle Datenverarbeitung als Mittel braucht, nicht außer Acht zu lassen, sondern in den Gegenstandsbereich der Wissenschaft aufzunehmen. Der wichtigste Zweck, dem Rechensysteme dienen, ist die Organisation von Arbeitsprozessen.

1.2.1 Merkmale von Information

- Information ist *immateriell*. Information ist zwar stets an einen *materiellen Träger* gebunden, die Art dieses Trägers ist jedoch für die Information „an sich“ unwesentlich. Der Informationsträger kann stofflicher Art - z.B. bei der Speicherung - oder energetischer Art - z.B. bei der Übermittlung - sein.
- Information ist ein *Grundbegriff*, der gleichberechtigt neben die physikalischen Grundbegriffe Materie und Energie gestellt werden kann.

¹ Das Wort *Daten* ist nur im Plural gebräuchlich. Da man den Singular *Datum* leicht mit dem Tagesdatum verwechselt, kann man sich mit *Datenelement* behelfen.

² Für viele Gebiete der Informatik ist diese Tatsache unwesentlich, da sie Probleme behandeln, die von der Basistechnologie abstrahieren, von ihr unabhängig sind. Es sind auch Rechner denkbar, die auf anderen Technologien basieren.

- Information hat mit Wissen über Tatsachen und Erscheinungen zu tun, mit ihrer Bedeutung und Bewertung. Bedeutungen und Werte werden von Menschen festgelegt. Also hat Information *subjektiven* Charakter, sie tritt nur in und als Beziehung zu Subjekten auf.
- Der Mensch hat sich viele Mittel geschaffen, um Informationen darzustellen: Laute, gesprochene Sprache, Zeichen, Bilder, Texte usw. Mit solchen *Darstellungen* kann er Informationen an andere Menschen weitervermitteln und außerhalb seines Gedächtnisses speichern.
- Der Austausch von Informationen zwischen Menschen ist möglich aufgrund eines sozialen Prozesses, in dem sie einen *Konsens* über die Bedeutung der Darstellungen erzielen.
- Die Begriffe Information und *Nachricht* hängen zusammen. Mögliche Bestimmungen ihrer Bedeutung:
 - Synonyme für Mitteilung, Übermittlung von Wissen.
 - Nachricht := übermitteltes Wissen; Information := übermitteltes neues Wissen, Wissenszuwachs.
 - Nachricht := objektive Form, Träger übermittelter Informationen, konkrete physische Darstellung; Information := subjektiver Inhalt, Deutung einer Nachricht, abstrakte ideelle Interpretation.

Die folgenden Aussagen sollen zur Klärung der Begriffe anregen.

„Ist das Ihre persönliche Meinung oder verlässliche Information?“

„Der Vortrag war sehr informativ.“ - „Mir hat er keinerlei Information geboten.“

„Die gleiche Information hätte er in fünf Minuten statt zwei Stunden rüberbringen können.“ - „Dieses Buch ließe sich ohne Informationsverlust auf die Hälfte zusammenstreichen.“

„Zu diesem Problem muß ich mir mehr Informationen beschaffen.“

„Ich wurde nicht informiert.“ - „Ich wurde nicht benachrichtigt.“

„Diese Nachricht ist ein alter Hut (kalter Kaffee, Schnee von gestern), sie enthält keine Information.“

„Diese Daten sind schwer zu interpretieren.“ - „Berge von Daten, aber kaum brauchbare Informationen.“

1.2.2 Merkmale von Informatik

- Der *Gegenstandsbereich* der Informatik umfaßt Arbeitsprozesse, geistige und körperliche Tätigkeiten, organisatorische und technische Vorgänge einerseits und Daten, Datenverarbeitung und datenverarbeitende Maschinen andererseits.
- Die *Methoden* der Informatik umfassen Analyse, formale Beschreibung, Konstruktion und Veränderung von Objekten ihres Gegenstandsbereichs.
- Zu den *Aufgaben* der Informatik gehört es

- Arbeitsprozesse in verschiedenen Anwendungsbereichen zu analysieren;
- Prinzipien, Methoden und Techniken der Datenverarbeitung zu erforschen und zu entwickeln;
- Arbeitsprozesse durch Anwendungen der Datenverarbeitung konstruktiv maschinell zu unterstützen, zu gestalten und zu organisieren;
- die bei den zu lösenden Problemen auftretende Komplexität zu beherrschen und mit den daraus entstehenden Risiken verantwortungsvoll umzugehen.
- Informatik ist - ähnlich wie die Mathematik - eine *Strukturwissenschaft*: sie befaßt sich mit Merkmalen, Eigenschaften, Beziehungen, Ordnungen usw. - kurz: *Strukturen* - von Gegenständen, Sachverhalten, Vorgängen usw. - kurz: *Systemen*.
- Vergleich mit Naturwissenschaften:
 - + Gemeinsamkeit: Analyse und formale Beschreibung von Systemen;
 - Unterschied: Systeme nicht naturgegeben, sondern von Menschen geschaffen.
- Vergleich mit Ingenieurwissenschaften:
 - + Gemeinsamkeit: Formale Beschreibung, Entwurf und Konstruktion von Systemen;
 - Unterschied: Systeme meist nicht materieller, sondern informationeller Art.
- Vergleich mit Geisteswissenschaften:
 - + Gemeinsamkeit: Analyse und Beschreibung von Systemen;
 - Unterschied: nicht auf Erkenntnisgewinn beschränkt, sondern auf praktische Anwendung zielend.
- Vergleich mit Human- und Sozialwissenschaften:
 - + Gemeinsamkeit: Analyse, Beschreibung und Veränderung;
 - Unterschied: nicht der Mensch ist Gegenstand der Erforschung, sondern ein Ausschnitt seiner Lebens- und Arbeitswelt.

1.2.3 Beziehungen der Informatik zu anderen Fachgebieten

Zu den wissenschaftlichen und technischen Gebieten, die an die Informatik angrenzen, gehören

- Mathematik,
- Linguistik,
- Physik, insbesondere Halbleiterphysik,
- Elektrotechnik, Elektronik und Mikroelektronik,
- Nachrichten- und Kommunikationstechnik,
- Meß- und Regelungstechnik,
- Automatisierungs- und Robotertechnik,
- Wirtschaftsinformatik.

1.2.4 Teilgebiete der Informatik

Die Informatik entwickelt sich sehr dynamisch weiter, ihre Teilgebiete stehen in vielfältigen engen Beziehungen zueinander. Die folgende Einteilung ist daher als eine von vielen möglichen zu betrachten.

■ Theoretische Informatik

- Automatentheorie und Theorie der formalen Sprachen
- Theorie der Berechenbarkeit und Komplexitätstheorie
- Algorithmenanalyse
- Theorie der Programmierung
- Automatische Programmsynthese
- Formale Semantik
- Verifikation von Programmen
- Formale Modellierung

■ Technische Informatik

- Daten- und Digitaltechnik: Schaltnetze, Schaltwerke
- Hardware-Komponenten, insbesondere Prozessoren, Mikroprozessoren
- Mikroprogrammierung
- Rechnerorganisation und -architektur
- Schnittstellentechnik, Rechnernetze

■ Praktische Informatik

- Algorithmen, Datenstrukturen und Programmiermethoden
- Programmiersprachen³ und Übersetzer
- Software-Technologie
- Betriebssysteme
- Echtzeitverarbeitung
- Datenbanken
- Graphische Systeme
- Benutzungsoberflächen⁴, Mensch-Maschine-Interaktion⁵

■ Angewandte Informatik

- Informationssysteme

³ Bezeichnungen wie „formale Sprache“ und „Programmiersprache“ sind problematisch, da mit ihnen wesentliche Unterschiede zu menschlichen Sprachen verwischt werden. Im Sinne menschlicher sprachlicher Kommunikation kann ein Rechner weder (sich aus-)sprechen, noch zuhören, noch verstehen. Eine bessere Bezeichnung für Programmiersprache wäre daher *Programmierkalkül*.

- Künstliche Intelligenz⁶
- Digitale Signalverarbeitung
- Modellierung und Simulation (CAE)
- Rechnerunterstützter Entwurf, rechnerunterstützte Konstruktion (CAD)
- Rechnerunterstützte und rechnerintegrierte Fertigung (CAM, CIM)
- Textverarbeitung und Büroautomatisierung
- Spezifische Anwendungen in
 - ◆ Wirtschaft: Industrie, Verkehr, Telekommunikation, Handel, Banken, Dienstleistungsbereich
 - ◆ öffentliche und private Verwaltung: Behörden, Versicherungen
 - ◆ Bildungsbereich und häusliche Umgebung
 - ◆ Ingenieur- und Naturwissenschaften: Maschinenbau, Meteorologie, Umwelttechnik, Weltraumtechnik
 - ◆ Human-, Sozial- und Geisteswissenschaften: Medizin, Kunst
- Datenschutz

⁴ Begriffe wie „Mensch-Maschine-Schnittstelle“, „Benutzerschnittstelle“, „Mensch-Maschine-Kommunikation“ und „Dialogsysteme“ sind in Gebrauch, aber irreführend. Weder besitzt ein Mensch eine Schnittstelle zu einer Maschine, noch kann er mit einer Maschine kommunizieren, schon gar nicht einen Dialog führen. Kommunikation setzt gleichartige, autonome Akteure voraus, die übermittelte Informationen gleichartig interpretieren. Einem Rechensystem darf man kein echtes Verständnis für menschliche Intentionen unterstellen. Ein Dialog setzt zwei gleichberechtigte Subjekte voraus - dies ist bei Mensch und Maschine nicht der Fall. Der Mensch *benutzt* die Maschine! Ein Rechensystem ist für einen Menschen lediglich ein *Werkzeug* zur Verarbeitung von Daten und ein *Medium* zur Übermittlung von Daten an andere Menschen. Werkzeuge und Medien werden vom Menschen nicht „bedient“, sondern benutzt. Damit sie benutzbar sind, müssen sie eine menschengerechte Oberfläche aufweisen. Wir wollen die Unterschiede zwischen Menschen und Maschinen, zwischen menschlicher und technischer Kommunikation nicht verwischen.

⁵ Begriffe wie „Mensch-Maschine-Interaktion“ und „interaktive Systeme“ sind nicht unproblematisch, da der Mensch ein autonom und bewußt handelndes Subjekt, das Rechensystem hingegen eine reaktiv operierende Maschine ist. In Ermangelung besserer Begriffe verwenden wir jedoch diese Bezeichnungen.

⁶ In den 50er Jahren eingeführter Oberbegriff für eine Reihe von Teilgebieten der Informatik. Der Begriff ist problematisch, da er eine Analogie zwischen maschineller Operation und menschlicher Intelligenz suggeriert, die äußerst fragwürdig ist.

Kapitel 2

Zeichen

2.1 Zeichen, Alphabete und Symbole

Ein **Zeichen** (*character*) ist ein Element aus einer endlichen Menge, die zur Darstellung von Information vereinbart ist. Die Menge heißt **Zeichenvorrat** (*Zeichensatz*; *character set*) oder **Alphabet**. Die Norm DIN 44300 für Begriffe der Informationsverarbeitung fordert von einem Alphabet zusätzlich, daß die Zeichen in vereinbarter Reihenfolge geordnet sind.

Beispiele:

$\{a, b, c, \dots, z, A, B, C, \dots, Z\}$	das Alphabet der Buchstaben;
$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$	das Alphabet der Dezimalziffern;
$\{., ;, :, !, ?, (,), [,], \&, \$, \%, \&, +, -, *, /\}$	ein Alphabet von Sonderzeichen;
$\{ , , , \}$	ein Alphabet nichtdruckbarer Steuerzeichen;
$\{0, 1\}$	das Alphabet der Dualziffern, ein Alphabet von Binärzeichen.

Ein **numerisches** (*numeric*) Zeichen gehört zu einem Alphabet, das Ziffern und Sonderzeichen zur Darstellung von Zahlen enthält. Ein **alphanumerisches** (*alphanumeric*) Zeichen gehört zu einem Alphabet, das mindestens die Buchstaben und Dezimalziffern enthält. Ein **Binärzeichen** (*binary element*) oder **Bit** (*binary digit*) ist ein Zeichen aus einem Zeichenvorrat von zwei Zeichen, **Binäralphabet** genannt. Als Binärzeichen können beliebige Zeichen verwendet werden.

Beispiele: $\{0, 1\}$, $\{O, L\}$, $\{\text{nein, ja}\}$, $\{\text{falsch, wahr}\}$.

Man muß klar zwischen etwas *Bezeichnendem* und dem dadurch *Bezeichneten* unterscheiden. Durch Zeichen werden Informationen dargestellt. Was bestimmte Zeichen bedeuten, ist eine Frage der Interpretation. In den folgenden Beispielen kennzeichnen wir die Bedeutung der Zeichen durch Klammerung mit „“.

Beispiele:

Darstellung der Information → *Interpretation* → *Bedeutung des Zeichens*

a	„Buchstabe a“ „Hexadezimalzahl Zehn“
m	„Buchstabe m“ „Name einer natürlichen Zahl“ „Meter“

1	„Ziffer Eins“ „Zahl Eins“ „Note Eins oder sehr gut“
<u>1</u> 23	„Einhundert“

Die Bedeutung eines Zeichens ist i.a. *kontextabhängig*, d.h. sie hängt von der Umgebung, in der das Zeichen steht, ab.

Man kann Zeichen aneinanderreihen, z.B.

abc
xyz
012

Dadurch erhält man **Zeichenfolgen** oder **Zeichenketten** (*character string*). Ein **Wort** (*word*) ist eine als Einheit betrachtete Zeichenfolge. Da die Zeichen der Folge zu einem bestimmten Alphabet gehören, spricht man von einem **Wort über dem Alphabet**. Die **Länge** eines Worts ist die Anzahl der Zeichen des Worts.

Beispiele:

Wort	Länge
B	1
Oberon	6
01100010	8

Ein Wort über einem Binäralphabet heißt **Binärwort**. Für $n = 1, 2, \dots$ gibt es genau 2^n unterscheidbare Binärwörter der Länge n . Ein **Byte** ist ein Binärwort der Länge 8 (8 Bit = 1 Byte). Es gibt also 256 verschiedene Werte eines Bytes.

Aus Wörtern über einem Alphabet kann man neue Alphabete bilden.

Beispiele:

{falsch, wahr}	das Alphabet der Wahrheitswerte;
{rot, gelb, grün}	das Alphabet der Verkehrsampelsignale;
{sehr gut, gut, befriedigend, ausreichend}	ein Alphabet von Noten.

Auch bei Wörtern ist zwischen Darstellung und Bedeutung, zwischen Form und Sinn zu unterscheiden. Die Bedeutung eines Worts ist wieder i.a. *kontextabhängig*.

Beispiele:

Darstellung der Information → *Interpretation* → *Bedeutung des Worts*

Birne	„Obstsorte“ „Glühbirne“ „...“
rot	„Farbe rot“ „vor der Ampel anhalten“
h\$wß+*p@k--	„“

Einem beliebigen Wort über einem Alphabet müssen wir i.a. nicht unbedingt eine Bedeutung zuordnen können. Ein Zeichen oder Wort, dem eine Bedeutung beigemessen wird, heißt **Symbol**.

Wie Zeichen kann man Wörter aneinanderreihen - man erhält Wortfolgen, die man **Text** nennt.

Verkettet man Wörter (über einem Alphabet) miteinander, erhält man wieder Wörter (über demselben Alphabet).

Beispiel:

einzel stück — Verkettung → einzelstück

Gehört das (nichtdruckbare) **Leerzeichen** (*Zwischenraum; blank, space*) zu dem Alphabet, dann verschwindet formal der Unterschied zwischen den Begriffen Wort und Text.

Ist A ein Alphabet (A ist hier der Name einer Menge von Zeichen), dann bezeichne $W(A)$ die Menge aller Wörter über A .

Die **Verkettung** oder **Konkatenation** (*concatenation*) ist eine Verknüpfung in $W(A)$, d.h. eine Abbildung von $W(A) \times W(A)$ in $W(A)$, die je zwei Wörtern $x, y \in W(A)$ das verkettete Wort $xy \in W(A)$ zuordnet.

Eine Teilmenge S von Wörtern über einem Alphabet A heißt **formale Sprache** (*formal language*).

Welche Wörter $x \in W(A)$ zu einer Sprache $S \subseteq W(A)$ gehören und welche nicht, wird meist durch bestimmte Regeln festgelegt. Es gibt dafür viele Möglichkeiten.

2.2 Codes und Codierungen

Ein **Code** ist eine Abbildung zwischen zwei Alphabeten. Man beachte, daß die Alphabete selbst Sprachen sein können! Sind also A, B Alphabete, dann ist ein Code $c : A \rightarrow B$ eine eindeutige Vorschrift, die jedem Zeichen $a \in A$ genau ein Zeichen $c(a) = b \in B$ zuordnet. Die Zuordnung braucht nicht umkehrbar eindeutig zu sein. Die Abbildung wird meist in Form einer **Codetabelle** angegeben. Ein **Binärcode** ist ein Code, bei dem jedes Zeichen der Bildmenge ein Binärwort ist.

Beispiel: Ein Binärcode für Dezimalziffern; $A = \{0, \dots, 9\}$, $B = \{0, 1\}^4$.

$a \in A$	$b \in B$
0	0000
1	0001
2	0010
...	...
9	1001
	1010
	...
	1111

Ist $S \subseteq W(A)$ eine Sprache, so heißt die Abbildung $c : S \rightarrow W(B)$, die man erhält, indem man auf jedes einzelne Zeichen eines Worts $s \in S$ den Code c anwendet, **Codierung**.

Beispiel: Binärcodierung von Dezimalzahlen.

$x \in W(A)$ wird durch c zugeordnet	$y \in W(B)$
1995	0001 1001 1001 0101

Zu einem Alphabet kann es selbstverständlich unterschiedliche Codes geben.

Beispiel:

Zeichen	codiert als	Code
a	97	ASCII dezimal
	61	ASCII hexadezimal
	0110 0001	ASCII dual
	· _	Morse
	● .	Braille
	· ·	
	· ·	

Von zentraler Bedeutung ist die folgende Aussage.

Jedes Alphabet kann binär codiert werden.

Näheres zum Thema Codes hören Sie in der Vorlesung „Digitaltechnik“.

2.3 ASCII-Code

Der im Rahmen dieser Vorlesung maßgebliche Zeichenvorrat ist der *ASCII-Zeichensatz*. Dazu gehört der *ASCII-Code* (*American Standard Code for Information Interchange*). Er ist der am weitesten verbreitete Code für Datenübertragung und wird auch in der PC-Welt überwiegend eingesetzt. Der ASCII-Code wurde von *ISO* (*International Organization for Standardization*) genormt; die korrekte Bezeichnung lautet eigentlich *ISO-646-Code*.

Der ASCII-Code ist ein 8-Bit-Code, wobei aber nur die 7 rechten Bits genutzt werden, so daß $2^7 = 128$ verschiedene Zeichen dargestellt werden können. Der ASCII-Zeichensatz enthält alphanumerische Zeichen, Sonderzeichen und eine Reihe von Steuerzeichen. Die wichtigsten Steuerzeichen sind **CR** (*carriage return; Wagenrücklauf*) und **LF** (*line feed; Zeilenvorschub*).

In verschiedenen Ländern werden z.T. Varianten des ASCII-Codes verwendet, um landesspezifische Zeichen unterzubringen. Dies kann zu interessanten Effekten beim Ausdrucken von Texten führen. So benutzt z.B. die deutsche Referenzversion Umlautzeichen anstelle der eckigen und geschweiften Klammern der USA-Version.

Außerdem gibt es eine abgewandelte und erweiterte Version des ASCII-Codes, die für IBM PCs eingeführt wurde (freilich in verschiedenen Varianten), bei der auch das 8. Bit genutzt wird.

Eine wichtige Erweiterung des ASCII-Codes ist der durch die *ISO-Norm 8859-1* festgelegte Zeichensatz mit der Bezeichnung *Latin1*. Er enthält alle von ASCII nicht erfaßten europäischen Zeichen. Latin1 wird zunehmend in Programmiersprachen und Betriebssystemen

stemen eingesetzt. Als internationaler Standard ist er nationalen oder firmenspezifischen ASCII-Varianten vorzuziehen.

2.4 Unicode

Der *Unicode* ist eine Weiterentwicklung des ASCII-Codes. Er verwendet zur Zeichencodierung jedoch 16 Bits, so daß er $2^{16} = 65536$ verschiedene Zeichen darstellen kann. Damit umfaßt er praktisch alle Schriftarten der Erde, d.h. neben den in europäischen Sprachen verwendeten Schriftzeichen u.a. auch Katakana, Hiragana, chinesische Zeichen, aber auch alte Sprachen wie Sanskrit und ägyptische Hieroglyphen, sowie Satzzeichen, mathematische und graphische Zeichen.

Der Unicode wird in modernen Programmiersprachen wie Component Pascal und Java und Betriebssystemen wie Windows NT verwendet.

2.5 Darstellungen von Größen

Die Unterscheidung zwischen

- zählen und
- messen

läßt sich bis in die Urgeschichte des Rechnens zurückverfolgen. Offenbar gibt es in der Realität Größen, die sich aus Einheiten zusammensetzen, die man zählen kann: 1, 2, 3,... Bei anderen Größen läßt sich jede mögliche Zerlegungseinheit weiter zerlegen, so daß es keine augenscheinliche Einheit für das Zählen gibt - man muß eine künstliche Einheit schaffen und die Größe daran messen.

Eine **diskrete** (*discrete*) Größe kann nur endlich viele unterscheidbare Werte annehmen. Eine **stetige** (*continuous*) Größe kann unendlich viele Werte annehmen, die sich beliebig wenig unterscheiden können.

Diskrete und stetige Größen können auf unterschiedliche Arten dargestellt werden.

Eine Darstellung einer Größe heißt **digital**, wenn die Größe durch endlich viele Zeichen oder Zahlen dargestellt wird.¹ Eine digitale Darstellung ist eine *symbolische* Beschreibung der dargestellten Größe. Eine Darstellung einer Größe heißt **analog** (*analogous*), wenn die Größe *direkt* auf eine andere (geometrische oder physikalische) Größe abgebildet wird.

Diskrete Größen lassen sich leicht digital darstellen. Bei stetigen Größen liegt meist eine analoge Darstellung am Nächsten.

Unter **Diskretisierung** oder **Rasterung** verstehen wir eine Abbildung eines unendlichen Wertebereichs auf einen endlichen Wertebereich. Durch Diskretisierung erhält man aus einer analogen Darstellung einer (stetigen) Größe eine **diskretisierte analoge Darstellung**.

¹ Digital = ziffernhafte; lateinisch digitus = Finger; englisch digit = Ziffer.

Unter **Digitalisierung** verstehen wir eine Abbildung eines endlichen Wertebereichs auf einen endlichen Zahlenbereich. Durch Digitalisierung erhält man aus einer diskretisierten analogen Darstellung einer (stetigen) Größe eine **digitale Darstellung**.

Von zentraler Bedeutung ist die folgende Aussage.

Jede zählbare oder meßbare Größe kann

- **diskretisiert,**
- **digitalisiert und**
- **binärcodiert dargestellt**

werden und damit

- **maschinell verarbeitet**
- werden.**

Kapitel 3

Rechensysteme

3.1 Was ist ein Rechensystem?

Rechensystem, Rechner, Rechenmaschine, Maschine; *computer, computing machine* :=

- programmgesteuerte Maschine zur Datenverarbeitung;
- kann
 - Eingabedaten aufnehmen,
 - Programme & Daten speichern &
 - transformieren &
 - Ausgabedaten erzeugen.

Sichtweisen

- Rechner = abstrakte funktionale Eigenschaften der Maschine;
- Hardware = konkrete technische Realisierung.

3.1.1 Hauptbestandteile

Hardware (harte Ware, HW) :=

Gesamtheit der

- physikalisch-technischen,
- stofflichen &
- energetischen

Komponenten eines Rechensystems.

Software (weiche Ware, SW) :=

Gesamtheit der

- Programme, die auf einem Rechner ablaufen können & die Tätigkeit der HW steuern,
- sowie der dazu gehörenden Daten,
- Einsatzregeln &
- Dokumentationen,

also der informationellen Komponenten eines Rechensystems.

Schnittstelle (*interface*)

Eine Schnittstelle der HW zur SW bildet der **Befehlsvorrat** (*instruction set*) des Rechners.

Bei mikroprogrammierbarer HW tritt Firmware als Schicht zwischen HW & SW.

Firmware (feste Ware) :=

besteht aus einer Anzahl von permanent gespeicherten Mikroprogrammen, die die Maschinenbefehle auf die Ebene von Mikrobefehlen abbilden.

Rechensystem :=

Hardware + Software + Firmware + Orgware.

3.1.2 Charakteristika heutiger Rechner

- **Digital:**
Benutzung von diskreten Zuständen anstelle von stetigen.
- **Binär:**
zweiwertige Zustände anstelle von zehnwertigen.
- **Elektronisch:**
Elektronik anstelle von Mechanik oder Elektromechanik.
- **Frei programmierbar:**
gespeichertes, veränderbares Programm anstelle von fest eingebauten Mechanismen.
- **Universell einsetzbar:**
Massenprodukte anstelle von Spezialanfertigungen.
- **Konfigurierbar:**
Rechnerfamilien durch Baukastenkonzept.

3.1.3 Arten von Rechensystemen

3.1.3.1 Anwendungsbereiche

- Kommerzielle Anwendungen in Wirtschaft & Industrie.
- Technisch-wissenschaftliche Berechnungen in Forschung & Entwicklung.
- Echtzeitsteuerungen physischer Prozesse.

3.1.3.2 Größen

- Portable Rechner (*notebook, laptop*).
- Heimrechner (*home computer*).
- Arbeitsplatzrechner (*personal computer, PC*).
- Mikrorechner (*micro computer*).
- Arbeitsstationen (*workstation*).
- Minirechner (*mini computer*).
- Großrechner (*main frame*).
- Superrechner (*super computer*).

3.2 Hardware

3.2.1 Bestandteile

- **Zentraleinheiten.**
- **Peripheriegeräte,
periphere Geräte,
Peripherie;
*peripheral devices.***

3.2.2 Peripheriegeräte

3.2.2.1 Ein-/Ausgabegeräte

E/A-Gerät; *input/output device, I/O device*

zur Mensch-Maschine-Interaktion:

- Tastatur (*keyboard*),
- Bildschirm (*display, monitor, screen*),
- Drucker (*printer*),
- Maus (*mouse*),
- Plotter,
- Klarschriftleser, Scanner.

3.2.2.2 Hintergrundspeicher

Extern-, Sekundärspeicher; *backing storage*

zur mittel- und langfristigen Datenhaltung,
Speichermedien,

Unterscheidung zwischen Datenträger & Gerät:

- Magnetplattengeräte,
 - Fest- & Wechselplattenlaufwerk,
 - Diskettenlaufwerk,
 - ZIP-Laufwerk,
- Magnetbandgeräte,
 - große Geräte zur Datenarchivierung,
 - kleine Kassettengeräte: *streamer*,
 - DAT-Laufwerk,
- optische Speichergeräte,
 - CD-ROM-Geräte (*compact disk read only memory*),
 - CD-Brenner-Geräte.

3.2.2.3 Geräte zur Interaktion mit Umgebung

Physische Prozesse außerhalb des Rechensystems.

Beispiele:

- Meßgeräte,
- Steuerungen für Werkzeugmaschinen,
- Sensoren & Aktoren von Robotern.

3.2.2.4 Anschlüsse an Rechnernetze

Beispiele:

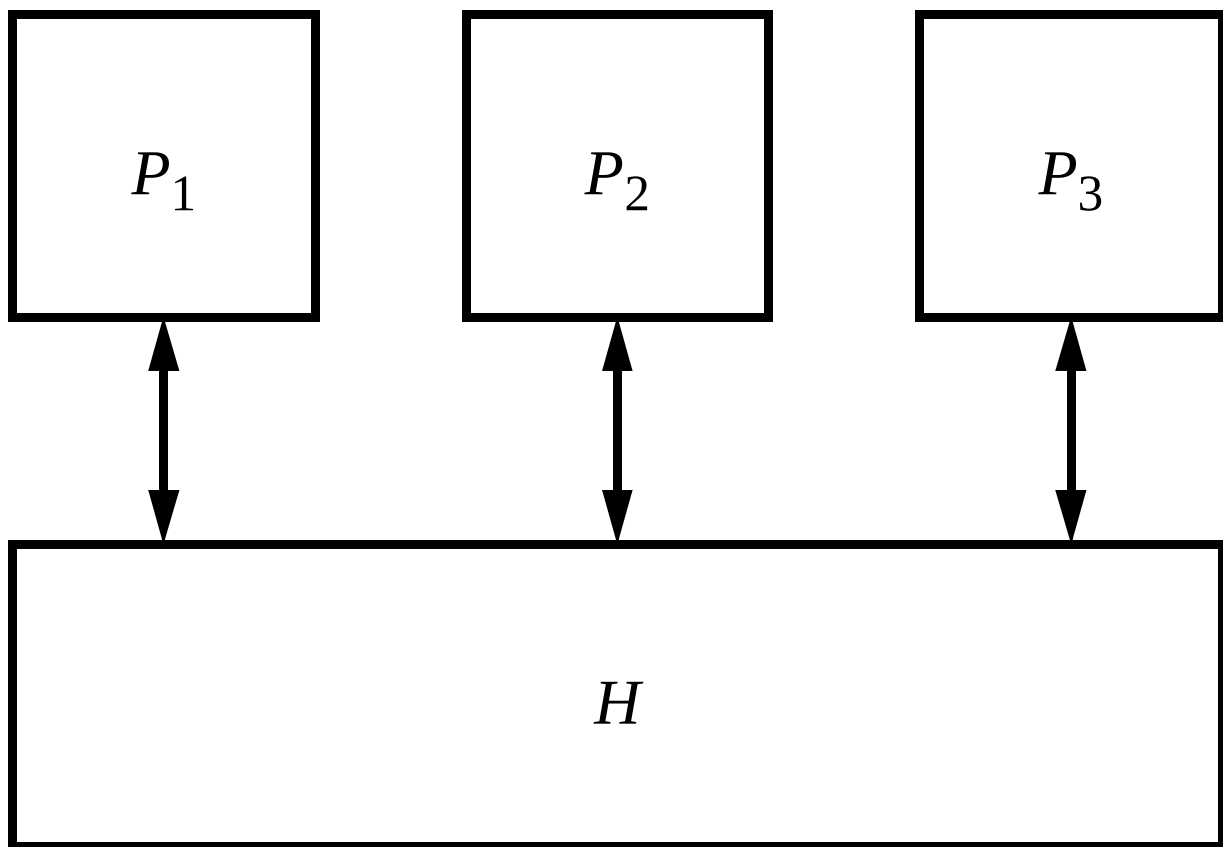
- lokale Rechnernetze,
- Telefonleitungen, ISDN,
- Satellitenverbindungen.

Siehe Vorlesung „Kommunikationsnetze“, 5. Sem.

Da Peripheriegeräte aus Systemsicht wie E/A-Geräte angeschlossen sind & funktionieren, verwenden wir den Begriff **Ein-/Ausgabe** für den Betrieb der Peripherie.

3.2.3 Zentraleinheiten

Grobstruktur einer Zentraleinheit



P_i : **Prozessor(en)**,
Zentralprozessor, *Central Processing Unit*,
aktive Komponente,
führt Befehle aus.

Technologie: Halbleiter, ein Chip.

H : **Hauptspeicher**,
Arbeitsspeicher, *main memory*,
passive Komponente,
enthält Befehle und Daten.

Technologie: Halbleiter, mehrere Chips.

3.2.3.1 Hauptspeicher

- Hauptspeicher = Menge Z gleichartiger **Zellen**.
- Jede Zelle kann **Werte** speichern.
- **Inhalt** einer Zelle = Wort über $\{0, 1\}$.
- Jede Zelle hat eine eindeutige **Adresse**.
- Adressen sind natürliche Zahlen.
Adressenmenge $A = \{0, \dots, k - 1\}$.
- Ein **Adreßraum** ist eine Abbildung
$$s : A \rightarrow Z, i \rightarrow s(i).$$
- Z ist linear geordnet.
- Rechnen mit Adressen ist möglich.
- Der Zugriff (Lesen, Schreiben) auf eine Zelle erfolgt über ihre Adresse.

Adressen	Zellen								
0									
1									
2	0	1	1	0	1	0	0	1	
...									
...									
47	11								Inhalt
48	48								Wert
...									
...									
i									
...									
$k - 2$									
$k - 1$									

3.2.3.2 Speichereinheiten

Zelle = kleinste adressierbare Einheit,
heute meist

1 Zelle = 1 Byte = 8 Bit.

1 Wort = 1 **Maschinenwort** = n Byte,
wobei rechnerabhängig $n = 1, 2, 4, \dots$
oft $n = 2$, also

1 Wort = 2 Byte = 16 Bit.

Meist wird bei den Begriffen *Byte* und *Wort* nicht genau zwischen den Bedeutungen *Zelle* und *Inhalt*, *Wert* unterschieden.

1 KByte = 1 **Kilobyte** = 2^{10} Byte
= 1024 Byte $\sim 10^3$ Byte.

1 MByte = 1 **Megabyte** = 2^{10} KByte = 2^{20} Byte
= 1048576 Byte $\sim 10^6$ Byte.

1 GByte = 1 **Gigabyte** = 2^{10} MByte = 2^{20} KByte
= 2^{30} Byte = 1073741824 Byte $\sim 10^9$ Byte.

3.2.3.3 Speicherkapazitäten und Zugriffszeiten bei PCs

Hauptspeicher

1 MByte - 1 GByte

einige 10 ns

Diskette

1,4 MByte

im ms-Bereich

Festplatte

40 MByte - 40 GByte

ca. 8 - 20 ms

ZIP-Diskette

100, 250 MByte

im ms-Bereich

CD-ROM

< 1 GByte

einige 100 ms

Magnetbandkassette

500 MByte - 50 GByte

kein Direktzugriff

3.2.3.4 Hauptspeicher dargestellt in Component-Pascal-Notation

(Vereinbarungsteil: *)*

CONST

Kapazität = 4 * 1024 * 1024;
(4 MByte *)*

VAR

Speicher :
 ARRAY Kapazität **OF** BYTE;
Adresse : INTEGER;
Register : BYTE;

(Anweisungsteil: *)*

Register := 11;
Adresse := 47;

(Schreiben: *)*

Speicher [Adresse] := Register;

(Lesen: *)*

Register :=
 Speicher [Adresse + 1];

3.2.3.5 Prozessor

Der Prozessor bearbeitet **Befehle** (*instruction*)

- sequentiell, d.h. einen nach dem anderen,
- linear, d.h. geordnet nach aufsteigenden Adressen.

Prozessorregister

Akkumulator

--

Befehlsregister

--

Befehlszähler

--

Hauptspeicher

...	...
...	...
$i - 1$	LOAD #10
i	ADD #1
$i + 1$	STORE \$47
...	...
...	...
...	...

- **Maschinensprache** (*machine language*) := Menge der Befehle, die der Prozessor ausführen kann.
- **Befehlsausführungszeit**: einige 10 ns - μ s.
- **Prozessorleistung**: bis einige 100 MIPS (*million instructions per second*).

3.2.3.6 Befehlszyklus in Component-Pascal-Pseudocode

```
WHILE Maschinenzustand = aktiv DO  
    (* hole nächsten Befehl: *)  
    Befehlsregister :=  
        Speicher [Befehlszähler];  
    (* setze Befehlszähler weiter: *)  
    Befehlszähler :=  
        Befehlszähler + 1;  
    (* führe Befehl aus: *)  
    entschlüssele Inhalt des Befehlsregisters,  
    hole Operanden aus dem Speicher,  
    führe Operation aus  
END;
```

- Abweichen von der linearen Befehlsausführung muß möglich sein (sonst wäre bald das Ende des Speichers erreicht).

3.2.4 Das von Neumann'sche Konzept der Rechnerarchitektur

- Der Rechner ist ein **Digitalrechner** (im Unterschied zu Analogrechnern).
- **Binärcodierung**: Alle Programme und Daten werden durch zweiwertige Zustände dargestellt (anstelle von z.B. zehnwertigen).
- Zum Rechnen mit Zahlen wird das **Dualsystem** verwendet (anstelle des Dezimalsystems).
- Der Rechner ist logisch und physisch gegliedert in
 - ein **Rechenwerk**, in dem Rechenoperationen und logische Verknüpfungen durchgeführt werden,
 - ein **Steuerwerk**, das den Programmablauf steuert,
 - ein **Speicherwerk**, in dem Programme und Daten gespeichert sind,
 - ein **Ein-/Ausgabewerk**, über das Programme und Daten ein- und ausgegeben werden.
- **Universelle, freie Programmierbarkeit**: Der Rechner ist strukturell unabhängig von den zu bearbeitenden Problemen; er wird erst arbeitsfä-

hig durch ein **Programm** (im Unterschied zu Rechenmaschinen mit fest eingebauten Mechanismen). Das Programm ist eine Folge von **Befehlen** und stellt einen durch den Rechner ausführbaren Algorithmus zur Lösung eines Problems dar.

- **Programmspeicherung:** Das Programm wird von außen in den Rechner eingegeben und im Speicher abgelegt.
- **Einheitlicher Speicher:** Programme und Daten werden in demselben Speicher abgelegt.
- **Speicheradressierung:** Der Speicher besteht aus gleichartigen Zellen, die durchnummeriert sind und deren Inhalt über ihre Nummer, die **Adresse** heißt, zugreifbar ist.
- **Lineare und sequentielle Befehlsausführung:** Die Befehle eines Programms werden i.d.R. aus aufeinanderfolgenden Speicherzellen geholt und nacheinander ausgeführt.
- Es gibt **Sprungbefehle**, d.h. nach der Ausführung des Befehls mit der Adresse i wird ein Befehl mit der Adresse $j \neq i + 1$ geholt.
- Es gibt **bedingte Sprungbefehle**, d.h. nach der Ausführung des Befehls mit der Adresse i wird ein Befehl mit der Adresse $j \neq i + 1$ geholt, falls eine Bedingung b erfüllt ist. Ist b nicht erfüllt, so wird bei $i + 1$ fortgesetzt.

3.3 Software

3.3.1 Klassifizierung

- **Anwendungs-Software** (*application software*):
Programme, die von Benutzenden eines Rechnersystems zur automatischen Lösung von Aufgaben oder zur Unterstützung bei der Lösung von Problemen in einem Anwendungsbereich eingesetzt werden.

- **Die System-Software**
stellt eine funktionale Verbindung zwischen der HW und der Anwendungs-SW her. Sie realisiert eine Menge von Diensten, die den Benutzenden bzw. der Anwendungs-SW zur Verfügung stehen.
 - Das **Betriebssystem** (BS, *operating system*) ermöglicht die anwendungsunabhängige Nutzung des Rechners.
 - **Dienstleistungs-Software** (*utilities*) bietet Unterstützung & Komfort bei der Entwicklung von Anwendungs-SW.
Beispiel: Programmierumgebung.

3.3.2 Betriebssysteme

Aufgaben

- Benutzungsoberfläche schaffen, die HW-Details verbirgt.
 - **Kommandosprache** (*command language*).
 - **Graphische Benutzungsoberfläche** (*graphical user interface, GUI*).
- **Betriebsabläufe** organisieren & steuern.
 - Programmabläufe überwachen.
 - Betriebsmittel verwalten & zuteilen.
 - Abläufe protokollieren.
- Langfristige **Datenhaltung** ermöglichen.
 - Hintergrundspeicherverwaltung.
 - Dateiverwaltung.
 - Schutz & Sicherung der Daten.
- **Peripheriegeräte** nutzen, steuern, überwachen.
- **Netzwerkbetrieb** unterstützen.

3.3.2.1 Microsoft MS-DOS

- Weit verbreitetes BS für IBM-PCs & kompatible PCs mit Intel-Prozessoren.
- Ursprung: QDOS (*quick & dirty operating system*), 1980 auf Intel 8086-Prozessor.
- Veraltet, aber Erblasten übrig in
 - MS-Windows 95, 98:
Kommandosprache, Dateisystem;
 - MS-Windows NT: Kommandosprache.

Grundbegriffe:

- **Kommandosprache**
- **Datei**
- **Verzeichnis**
- **Laufwerk**
- **Namengebung**
- **Editor**
- **Kommandodatei**

3.3.2.2 Microsoft Windows 3.1

- GUI, die auf MS-DOS läuft.

3.3.2.3 Microsoft Windows 95, 98

- Nachfolger von Windows 3.1.
 - integriert BS & GUI,
 - bietet **Mehrprozeßbetrieb** (*multitasking*).

3.3.2.4 Microsoft Windows NT, 2000

- NT = New Technology.
- Neues, objektorientiertes BS mit GUI.
- Mehrprozeßbetrieb.
- Neues Dateisystem NTFS.
- Workstation- & Server-Varianten.
- Für Netze, High-End-PCs & größere Rechner.

3.3.2.5 Das lokale PC-Netz des SG Elektronik

- Dienstrechner (*server*) Phoenix mit Windows NT Server:
 - Benutzerverwaltung,
 - Dateiverwaltung.
- Arbeitsplatzrechner (*PC workstation*) mit Windows NT Workstation im Rechnerraum F-308 & im Labor für Mikrorechnerntechnik, Raum F-205.

Sie erwerben folgende Grundkenntnisse in den ersten Praktikumsstunden:

- **Sitzungen starten & beenden**
(Ein- & Ausloggen)
- **Dateimanager Explorer benutzen**

Kapitel 4

Programmiersprachen

4.1 Was ist eine Programmiersprache?

Programmiersprache, algorithmische Sprache;
programming language, algorithmic language :=

formale Sprache zur Darstellung von Daten & Algorithmen als Programme, die auf Rechnern ausführbar sind.

Ein solcher Algorithmus beschreibt eine *berechenbare Funktion*, er transformiert eine Folge von Eingabezeichen mit endlich vielen Schritten in eine Folge von Ausgabezeichen.

- *Natürliche Sprachen* sind nicht natürlich, sondern Ergebnis einer sozialen & kulturellen Entwicklung.
- *Programmiersprachen* sind eigentlich keine Sprachen, sondern Kalküle, Regeln, Rechenvorschriften, Codiersysteme.

4.2 Abstraktionsebenen

■ **Maschinensprache** (*machine language*).

Maschinencode, Objektcode (*object code*):

Interne (ausführbare) Darstellung eines Maschinenprogramms als Bitmuster.

■ **Assemblersprache, maschinenorientierte Sprache** (*assembly language*):

Symbolische, textorientierte Darstellung einer Maschinensprache.

Merkmale:

- Symbolische Namen der Befehle ($1 \leftrightarrow 1$).
- Dezimalzahlen, symbolische Adressen.
- Pseudobefehle zur Definition von Konstanten, Variablen, Daten, Makroanweisungen (Makros).
- Formatierte Notation.
- Kommentare.

Assemblerprogramm, Assemblercode (*assembly program, code*):

Programm in Assemblersprache.

■ **Höhere, problemorientierte Programmiersprache, Hochsprache** (*high level language*):

Formale Sprache zur textuellen Darstellung von Programmen, deren Konstrukte

- mächtiger als einzelne Maschinenbefehle sind,
- Details der von-Neumann-Architektur verbergen,
- die Formulierung von Algorithmen unabhängig von einem bestimmten Rechensystem ermöglichen,
- sich an den Bedürfnissen eines Anwendungsbereichs orientieren.

Quellprogramm, Quellcode (*source program, code*):

Programm in Hochsprache.

4.3 Werkzeuge

■ Auf Assemblerebene.

- **Assemblierer** (*assembler*) übersetzt Assemblerprogramm in Maschinencode.
- **Binder** (*linker*) fügt assemblierte Programmteile zu ausführbarem Ganzen zusammen.
- **Lader** (*loader*) lädt ausführbaren Objektcode in den Hauptspeicher.

■ Auf Hochsprachenebene.

Zwei Arten von **Übersetzern** (*translator*):

- **Kompilierer** (*compiler*) transformiert Quellprogramm geschlossen in semantisch äquivalenten Assembler- oder Maschinencode.
- **Interpreter** (*interpreter*) untersucht die Konstrukte des Quellprogramms & führt sie direkt der Reihe nach auf dem Rechner aus.

- Zum Erstellen von Programmen.
 - **Editor** ermöglicht interaktive Eingabe des Quellprogrammtextes.
- Zum Ablauf von Programmen.
 - **Laufzeitumgebung** (*run time system*) stellt eine Bibliothek von Hilfsroutinen zur Verfügung & behandelt **Laufzeitfehler** von Programmen, d.h. Fehler, die erst beim Ablauf des Programms auftreten.
- Zum Testen von Programmen.
 - **Debugger** (*Entwanzer, Testmonitor*) hilft bei der Fehlersuche, u.a. mittels
 - ◆ **Ablauf- & Einzelschrittverfolgung** (*tracing, single stepping*),
 - ◆ **Haltepunkten** (*breakpoints*).
 - **Post-Mortem-Debugger** erlaubt
 - ◆ **Hauptspeicherabzug** (*memory dump*),
 - ◆ ggf. mit Bezug zum Quellcode.

4.4 Syntax, Semantik, Pragmatik

Syntax einer Programmiersprache

- behandelt Beziehungen der Zeichen untereinander, ihre Kombinierbarkeit
- ohne Rücksicht auf ihre spezielle Bedeutung & ihre Beziehung zur Umgebung.

Syntax legt fest,

- welche Sprachelemente & -konstrukte es gibt &
- wie diese sich zusammensetzen.

Syntax = Menge von Regeln, die die Struktur von Programmen bestimmen.

Syntax weitestgehend/vollständig formalisiert.

Darstellungsarten:

- Grammatiken.
- **(erweiterte) Backus-Naur-Formen.**
- Syntaxdiagramme.

Semantik einer Programmiersprache

- behandelt die Bedeutung der Zeichen,
- ihre Beziehungen zu den Objekten, auf die sie anwendbar sind.

Semantik legt fest,

- was welches Sprachelement oder -konstrukt bedeutet &
- welche Wirkung es im Programmablauf hervorruft.

Semantik = Menge von Verhaltensregeln, die die Funktionsweise von Programmen bestimmen.

Semantik nicht/weitgehend formalisiert.

Darstellungsarten:

- **Verbale Beschreibung.**
- Operationelle Methode.
- Denotationelle Methode.
- Axiomatische Methode.

Abgrenzung Syntax/Semantik unscharf.

Pragmatik einer Programmiersprache behandelt

- Herkunft, Gebrauch & Wirkung der Zeichen innerhalb der Umgebung, in der sie auftreten,
- ihr Verhältnis zu denen, die sie „verstehen“ sollen, d.h. zu Rechnern & Menschen.

Mechanische Pragmatik untersucht Fragen wie

- Übersetzbarkeit der Sprache,
- Anforderungen an das Betriebssystem,
- Abhängigkeiten von Rechnerarchitekturen.

Menschliche Pragmatik untersucht Fragen wie

- Lesbarkeit, Verständlichkeit,
- Lehrbarkeit & Erlernbarkeit,
- Anwendbarkeit, Nutzen zur Lösung praktischer Probleme.

Pragmatik ist Gegenstand der Diskussion, Entwicklung, Forschung.

4.5 Fehlerarten

- Ein syntaktisch fehlerhaftes Programm gehört nicht zur Sprache & ist nicht ablauffähig.
- Ein syntaktisch korrektes Programm kann semantische Fehler enthalten.
- Übersetzer erkennt **syntaktische & semantische Fehler (Übersetzungszeitfehler)**.
- Ein semantisch korrektes Programm kann **Laufzeitfehler** enthalten.
- Laufzeitfehler werden von der Laufzeitumgebung, vom Betriebssystem oder nicht erkannt.
- Ein semantisch korrektes, fehlerfrei laufendes Programm kann praktisch sinnlos sein.
- Eine Sprache A ist **besser** als eine Sprache B, wenn A Programmierfehler, die B akzeptiert, als syntaktische oder semantische Fehler zurückweist.

4.6 Denkmodelle und Konzepte

■ **Prozedurale, imperative Sprachen.**

Grundkonzepte: Variablen, Wertzuweisungen, Programm = Folge von Anweisungen, die eine Zustandsänderung bewirken.

- Frühzeitliche Sprachen.
- **Strukturierte** Sprachen.
- **Modulare** Sprachen.
- **Objektorientierte** Sprachen.
- Hybride Sprachen.

■ **Funktionale, applikative Sprachen.**

Grundkonzepte: Funktion i.S.d. Mathematik, Programm = Ausdruck, der einen Wert als Ergebnis liefert.

■ **Logische Sprachen.**

Grundkonzepte: Formale, mathematische Logik.

4.7 Skizze der geschichtlichen Entwicklung

- 1954 - 57 **Flowmatic** von G. M. Hopper, Univac.
Fortran (*Formula Translation*) von J. W. Backus, IBM.
Erste Hochsprachen.
- 1956 - 62 **Lisp** (*List Processing Language*) von J. McCarthy.
Funktional, Hauptsprache der „Künstlichen Intelligenz“ (KI).
- 1958 - 60 **Algol 60** (*Algorithmic Language*) von P. Naur u.a.
Einflußreich, Syntaxdefinition in Backus-Naur-Form, Blockschachtelung, Kellerkonzept, für naturwissenschaftlich-technische Anwendungen.
- 1959 - 61 **Cobol** (*Common Business Oriented Language*), Teamarbeit.
Noch heute weitestverbreitete Sprache für kommerzielle Anwendungen.
- 1967 **Simula 67** von Dahl/Nygaard.
Klassen, Koroutinen, erste objektorientierte Sprache für Simulationsanwendungen.
- 1968 - 71 **Pascal** von N. Wirth.
Einflußreich, einfach, strukturierte Program-

mierung, strenges Datentypkonzept, für Systemprogrammierung & Ausbildung.

1970 - 72 **C** von D. Ritchie.

Maschinennah, effizient, mit Unix verbunden, für Betriebssystemprogrammierung.

1970 - 80 **Smalltalk** von Kay/Goldberg/Ingalls.
Rein objektorientiert.

1975 - 80 **Ada** von J. Ichbiah/DoD.

Modular, Prozesse, Ausnahmebehandlung, komplex, für militärische Anwendungen.

1975 - 82 **Prolog** (*Programming in Logic*) von Colmerauer/Warren.

Modelliert logisches Schließen, KI-Sprache.

1977 - 84 **CHILL** (CCITT *High Level Language*).

Modular, für Telekommunikationsanwendungen.

1980 **Modula-2** von N. Wirth.

Modular, für Systemprogrammierung.

1980 - 86 **C++** von B. Stroustrup.

Objektorientierte Erweiterung von C.

1985 - 86 **Oberon** von N. Wirth.

Objektorientiert, für Systemprogrammierung.

1985 - 88 **Eiffel** von B. Meyer.

Objektorientiert, sw-technisch konzipiert.

Kapitel 5

Ein erster Blick auf BlackBox und Component Pascal

5.1 Etwas Geschichte

- **Oberon:** Der Herrscher der Geister, Gatte der Feenkönigin Titania in William Shakespeares „Ein Sommernachtstraum“.
- **Oberon:** Ein Mond des Uranus, in den 80er Jahren von einer Voyager-Sonde passiert.
- **Oberon:** Ein Projekt, von Niklaus Wirth & Jürg Gutknecht 1985 begonnen, bestehend aus
 - dem **Oberon-System**, einem objektorientiert-modularen Betriebssystem mit einer dokumentorientierten Entwicklungsumgebung, &
 - der **Oberon-Sprache**, einer objektorientiert-modularen Programmiersprache, in der das Oberon-System selbst vollständig geschrieben ist.

■ **Oberon:** Ursprüngliche Oberon-Sprache.

- Weiterentwicklung der von Wirth entworfenen Programmiersprachen Pascal & Modula-2;
- übernimmt von Pascal Grundkonzepte der strukturierten Programmierung,
- von Modula-2 Grundkonzepte der modularen Programmierung &
- bietet gegenüber seinen Vorgängersprachen zusätzlich objekt- & komponentenorientierte Konzepte.

■ **Oberon-2:** Aufwärtskompatible Erweiterung von Oberon, von Wirth und Hanspeter Mössenböck 1991 entwickelt.

5.2 Was ist...

5.2.1 ...Component Pascal?

- Eine kommerzielle nichtkompatible Erweiterung von Oberon-2,
- von der Firma Oberon microsystems Inc. 1997 entwickelt.

5.2.2 ...BlackBox Component Builder?

- Eine Component-Pascal-Entwicklungsumgebung von Oberon microsystems Inc.,
- für MS-Windows, Apple Macintosh OS & andere Plattformen.

5.3 Merkmale von Component Pascal

- Universelle, wohlfundierte & trotzdem einfache höhere Programmiersprache;
- gleichermaßen gut für Systemprogrammierung & Ausbildung geeignet;
- unterstützt
 - strukturierte, modulare, objekt- & komponentenorientierte Programmierung;
 - Datenabstraktion, Programmieren durch Vertrag, Flexibilität & Wiederverwendbarkeit;
- Blockstruktur;
- definierbare Datentypen & Prozeduren;
- Modulkonzept mit getrennter Übersetzung;
- Klassenkonzept durch Erweiterung von Verbundtypen & typgebundene Prozeduren;
- Polymorphie & dynamische Bindung;
- Sicherheit durch
 - strenge Typprüfungen zur Übersetzungszeit auch über Modulgrenzen hinweg;
 - Zusicherungen.

5.4 Merkmale von BlackBox

- Komfortable interaktive Entwicklungsumgebung;
- integriert verschiedene Werkzeuge wie Editor, Browser, Übersetzer, Binder, Lader, Laufzeitsystem, Online-Hilfe;
- erlaubt sehr kurze Editieren-Übersetzen-Ausführen-Zyklen;
- unterstützt graphische Benutzungsoberflächen wie MS-Windows, MacOS & Entwicklung portierbarer, plattformunabhängiger Anwendungen;
- hebt tendenziell Unterscheidung auf zwischen
 - verschiedenen Modi von Text,
 - Kommando- & Programmiersprache,
 - Kommando- & Menüoberfläche;
- dokumentorientiert:
 - Objekte wie Graphiken, Hyperlinks in Dokumente einbettbar;
 - interaktive Lerndokumente gestaltbar;
- uneingeschränkter Undo/Redo-Mechanismus;
- dynamisches Binden & Laden von Modulen;
- automatische Speicherbereinigung (*garbage collection*).

5.5 Programmstruktur von Component Pascal

- **Component-Pascal-Programm = Menge von Modulen.**

5.5.1 Was ist ein Modul?

- Eine **logische Programmeinheit**, eine Menge zusammengehöriger Daten & Algorithmen.
- Eine **Übersetzungseinheit**.
- Eine **Ladeeinheit**.

5.5.2 Woraus besteht ein Modul?

Drei Grundbestandteile eines Moduls:

- **Modulkopf,**
- **Vereinbarungsteil,**
- **Initialisierungsteil.**

5.5.3 Einfaches Beispiel

```
MODULE I1Count;
```

```
    IMPORT
```

```
        Out;
```

```
    VAR
```

```
        i : INTEGER;
```

```
    PROCEDURE Do*;
```

```
    BEGIN
```

```
        INC (i);
```

```
        Out.Open;
```

```
        Out.String ("You called me ");
```

```
        Out.Int (i, 0);
```

```
        Out.String(" times!");
```

```
        Out.Ln;
```

```
    END Do;
```

```
BEGIN
```

```
    i := 0;
```

```
END I1Count.
```

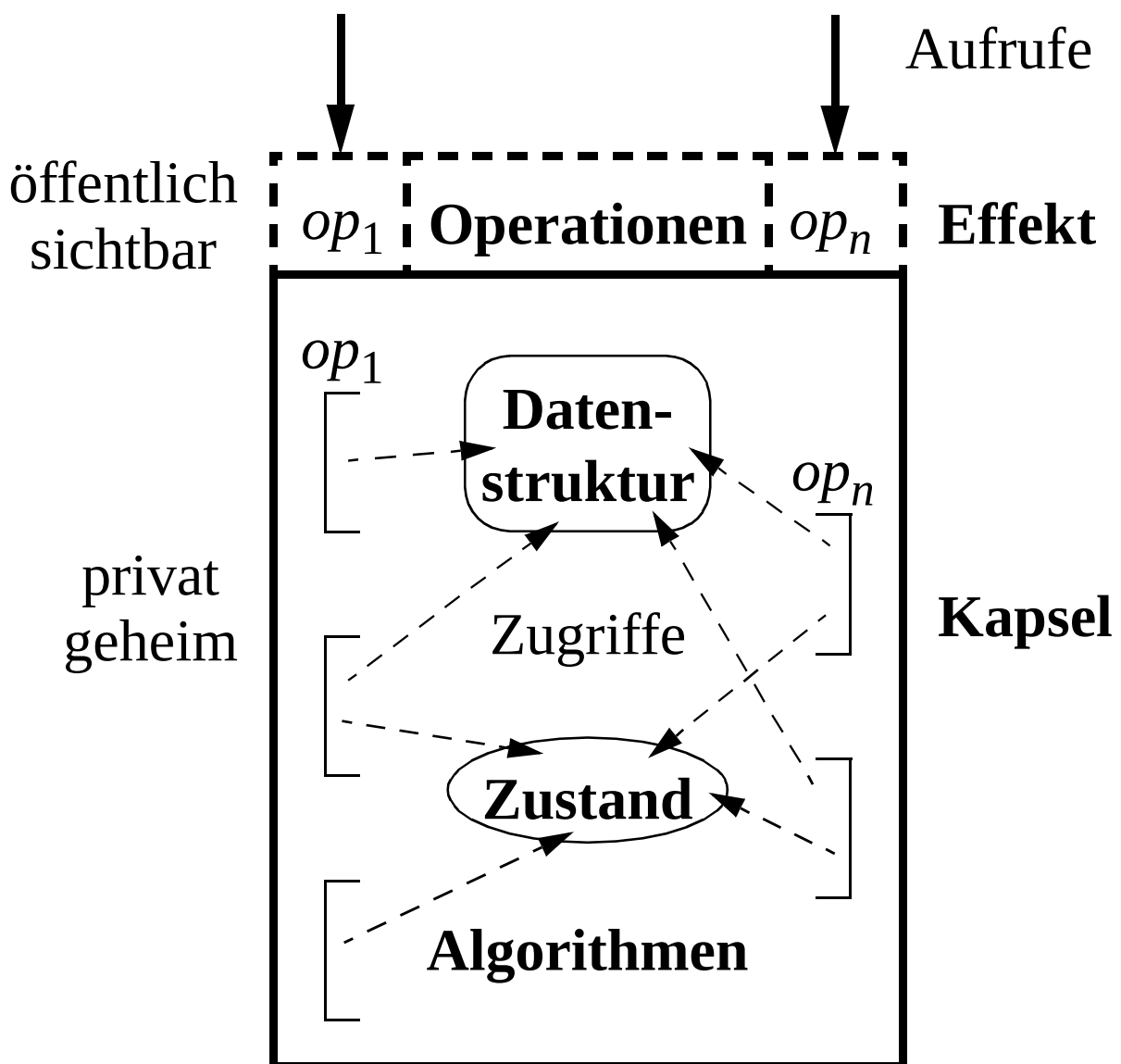
● I1Count.Do

5.5.4 Konzept der Datenabstraktion

Datenstrukturen & darauf zugreifende Operationen als modulare Einheit betrachten!

Was?

Schnittstelle



Wie?

Implementierung

5.5.5 Modulkonstrukt

5.5.5.1 Syntax

Module =

```
MODULE ident „;“ [ ImportList ]  
DeclSeq  
[ BEGIN StatementSeq ]  
[ CLOSE StatementSeq ]  
END ident „.“.
```

ImportList =

```
IMPORT [ ident „:=“ ] ident  
{ „,“ [ ident „:=“ ] ident } „;“.
```

DeclSeq =

```
{ CONST { ConstDecl „;“ } |  
TYPE { TypeDecl „;“ } |  
VAR { VarDecl „;“ } }  
{ ProcDecl „;“ | ForwardDecl „;“ }.
```

StatementSeq =

```
Statement { „;“ Statement }.
```

ConstDecl = ...

5.5.5.2 Muster

MODULE *SsnameMdname*;

IMPORT

Sub1Mod1, Sub2Mod2;

CONST

constName = ConstExpr;

TYPE

TypeName = Type;

VAR

varName : Type;

PROCEDURE *ProcName** (*par : Type*);

DeclSeq

BEGIN

StatementSeq

END *ProcName*;

BEGIN

StatementSeq

END *SsnameMdname*.

5.6 BlackBox-Verzeichnisstruktur

5.6.1 Subsysteme

- **BlackBox = Menge von Subsystemen.**
- **Subsystem =**
Menge zusammengehöriger Module.
- Zu jedem Subsystem gibt es ein Verzeichnis.

5.6.1.1 Standard-Subsysteme

- **Comm** (Communication, Schnittstelle zu Rechnernetzen),
- **Ctl** (Werkzeuge für OLE automation controllers),
- **Dev** (Development, Entwicklungswerkzeuge),
- **Docu** (Allgemeine Online-Dokumentation),
- **Dtf** (dF-Treiber für Sql),
- **Form** (GUI-Elemente, *visual designer*),
- **Host** (Schnittstelle zum BS, hier: MS-Windows),

- **Obx** (Oberon Examples, Beispielmodule),

- **Ole** (MS-Standard für Dokumente),
- **Sql** (Standard Query Language, Schnittstelle zu Datenbanken),
- **Std** (Standard-Kommando-Module),

- **System** (BlackBox Kern),

- **Text** (Standard-Dokument-/Programm-Editor),
- **Win** (Schnittstelle zu MS-Windows).

5.6.1.2 FB-EM-Subsysteme

- `Basis` (Allgemein verwendbare Module),
- `Containers` (Behälter-Module & -Klassen),
- `Electrics` (Beispiele aus der E-Technik),
- `Generals` (Allgemeines, keine Programme),
- `Graph` (Graphik-Module),
- `Guide` (Hypertextbasierter Sprachführer)
- `I0, I1, I2`
(Beispiele aus Vorlesung,
Lösungen zu Praktikumsübungen),
- `Math` (Mathematische Module),
- `Test` (Module zum Testen anderer Module),
- `Utilities` (Speziellere Dienst-Module).

5.6.2 Module und Dateien

Zu jedem Modul gibt es

- eine **Quelltextdatei** (*source*),
* .odc,
oberon document,
von Hand erstellt;
- eine **Objektcodedatei** (*code*),
* .ocf,
oberon code file,
vom Übersetzer erzeugt;
- eine **Schnittstellendatei** (*interface*),
* .osf,
oberon symbol file,
vom Übersetzer erzeugte binäre Darstellung der
Modulschnittstelle, enthält Definition der exportierten
Symbole des Moduls;
- optional eine **Dokumentationsdatei**,
* .odc,
von Hand erstellt.

5.6.3 Unterverzeichnisse und Dateien

Jedes Subsystemverzeichnis enthält folgende Unterverzeichnisse mit bestimmten Dateien:

- **Code**

enthält Objektcodedateien der Module `*.ocf`,
wird automatisch angelegt.

- **Docu**

enthält Dokumentationsdateien zu Modulen
`*.odc`,
muß von Hand angelegt werden.

- **Mod**

enthält Quelltextdateien der Module `*.odc`,
muß von Hand angelegt werden.

- **Rsrc**

enthält Ressourcen, Daten für Module, z.B. Zeichenketten & Dialoge `*.cur`, `*.ico`,
`*.odc`,
muß von Hand angelegt werden.

- **Sym**

enthält Schnittstellendefinitionen der Module
`*.osf`,
wird automatisch angelegt.

5.6.4 Namengebung

- **Subsystemname** = Verzeichnisname.
- **Modulname** =
Subsystemname + Moduldateiname.
- **Moduldateiname** =
Name der Quelltext-, Objektcode- & Schnittstellendatei (ohne Erweiterung).
- Subsystemname =
Präfix des Modulnamens
bis zum 1. Wechsel zu einem Großbuchstaben.
- Hat der Modulname keinen Wechsel von klein nach groß, so handelt es sich um ein **globales Modul**, das zum Subsystem **System** gehört.
- Auf manchen Plattformen:
Länge eines Datei- oder Verzeichnisnamens
max. 8 Zeichen.

5.6.4.1 Beispiel

Modulname: TextCmds

Subsystem: Text

Quelltextdatei: Text\Mod\Cmds.odc

Inhalt der Quelltextdatei:

```
MODULE TextCmds;
```

```
...
```

Objektcodedatei: Text\Code\Cmds.ocf

Schnittstellendatei: Text\Sym\Cmds.osf

Modulname: I1MyModule

Subsystem: I1

Quelltextdatei: I1\Mod\MyModule.odc

Inhalt der Quelltextdatei:

```
MODULE I1MyModule;
```

```
IMPORT TextCmds;
```

```
...
```

Objektcodedatei: I1\Code\MyModule.ocf

Schnittstellendatei: I1\Sym\MyModule.osf

5.7 BlackBox-Werkzeuge

■ Online-Hilfe.

Als Hypertextsystem organisiert.

Menü `Help->Contents` aufrufen,
blauen Text solange anklicken, bis gewünschte
Information gefunden.

■ Kommandos.

Kommando = exportierte Prozedur eines
Moduls mit eingeschränkter Signatur.

Aufruf:

ModuleName.Kommandoname selektieren &
Menü `Dev->Execute` aufrufen.

Aufruf mit Commander:

● *ModuleName.Kommandoname* anklicken.

■ Editor.

Quelltext von `I1MyModule` editieren,
unter `I1\Mod\MyModule.odc` speichern.

■ BlackBox-Werkzeuge kennen keine Suchpfade. Sie suchen Dateien nicht, sondern erwarten sie an Orten, die durch Namenskonventionen bestimmt sind.

■ Übersetzer.

Fenster mit Quelltext aktivieren oder
Modulnamen `I1MyModule` selektieren,
Menü `Dev->Compile` bzw.
`Dev->Compile Selection` aufrufen.

Übersetzer konsumiert Quelltext- & Schnittstellendateien der vom Quellmodul importierten Module, z.B.

```
I1\Mod\MyModule.odc  
Text\Sym\Cmds.osf
```

Übersetzer produziert Objektcode- & Schnittstellendateien, z.B.

```
I1\Code\MyModule.ocf  
I1\Sym\MyModule.osf
```

■ Lader.

Kommandonamen `I1MyModule.Do` selektieren,

Menü `Dev->Execute` aufrufen.

Lader bindet & lädt Objektcodedatei

```
I1\Code\MyModule.ocf
```

dynamisch & führt Prozedur `Do` aus.

- Module müssen explizit mit `Dev->Unload` entladen werden, bevor eine neue Version dynamisch geladen wird.

- Informationen über geladene Module liefern
Info->Loaded Modules und
Info->Global Variables.

■ **Browser.**

Modulnamen TextCmds selektieren, dann

Menü Info->Source aufrufen.

Browser zeigt Quelldatei

Text\Mod\Cmds.odc

Menü Info->Interface aufrufen.

Browser zeigt Schnittstellendatei

Text\Sym\Cmds.osf

Menü Info->Documentation aufrufen.

Browser zeigt Dokumentationsdatei

Text\Docu\Cmds.odc

■ **Component Pascal Language Report.**

Wichtiges Dokument, vollständige & exakte,
aber knappe Beschreibung der Sprache.

Online-Hilfe Menü Help->Contents aufrufen, Component Pascal Language Report anklicken.

5.8 Oberon-Entwicklungsumgebungen

- **Oberon V4 und Oberon System 3**
von der ETH Zürich.
- **Oberon-2 Pow!**
von der Universität Linz.
- **BlackBox**
von Oberon microsystems Inc.
- Einige andere, z.B. Uni Vologda, Rußland.

Unterschiede

- Sprache:
 - Oberon-2, Oberon-3, Component Pascal
- Interaktionsform:
 - 2- bzw. 3-Tasten-Maus,
Fenster bzw. Kacheln, Form der Rolleisten.
- Funktionsumfang:
 - Editor, Browser, Laufzeitsystem,
Online-Hilfe, Debugger.
- Unterstützung der Komponentenorientierung.

Gemeinsamkeit

Portable Varianten für verbreitete Plattformen.

5.9 Informationen im Internet

Alles über Oberon bietet *The Oberon Webring*:

- <http://www.factorial.com/hosted/webrings/oberon>

Einzelne Quellen:

- <http://www.oberon.ethz.ch>
- <http://oberon.ssw.uni-linz.ac.at/Oberon.html>
- <http://www.oberon.ch>
- <http://www.uni-vologda.ac.ru/JOB/Index.htm>
- <ftp://ftp.inf.ethz.ch/pub/software/Oberon>
- <ftp://ftp.ssw.uni-linz.ac.at/pub/Oberon>

Oberon Newsgroup:

- <comp.lang.oberon>

Kapitel 6

Grundelemente der Lehrsprache

6.1 Erweiterte Backus-Naur-Form

- **Erweiterter Backus-Naur-Formalismus:**
 - Notation zur Beschreibung der Syntax von Programmiersprachen,
 - **Grammatik:** Menge von Regeln zur Bildung von Sätzen, hier: Programmen.
- **Syntaktische Einheiten**, Konstrukte der Programmiersprache:
 - **Terminale Symbole:**
a 1 "yes!" + # MODULE ident ...
 - **Nichtterminale Symbole:**
A B Module ImportList DeclSeq ...
 - **Startsymbol** $\in$ Nichtterminale Symbole:
Module
- **Meta-Symbole** zur Beschreibung der Syntax:
= | () [] { } . „“

■ **Regel:**

$$A = B.$$

Konstrukt A ist definiert durch Konstrukt B .

■ **Aneinanderreihung, Sequenz:**

$$A = B C.$$

Konstrukt A ist definiert durch Konstrukt B gefolgt von Konstrukt C .

■ **Auswahl, Alternative:**

$$A = B \mid C.$$

Konstrukt A ist definiert durch Konstrukt B oder durch Konstrukt C .

■ **Option:**

$$A = [B].$$

Konstrukt A ist definiert durch Konstrukt B oder durch Nichts.

■ **Wiederholung, Iteration:**

$$A = \{ B \}.$$

Konstrukt A ist definiert durch eine beliebige Anzahl von B s, einschließlich keinem.

6.2 Terminale Symbole

- Alphabet der (terminalen) Symbole:
Latin1-Erweiterung des ASCII-Zeichensatzes (ISO 8859-1).
- Arten von Symbolen:
 - Namen (*identifier*),
 - Zahlen (*number*),
 - Zeichen (*character*) & -ketten (*string*),
 - Operatoren & Begrenzer (*operator*, *delimiter*).
- Groß- & Kleinbuchstaben sind verschieden!
`real Real REAL`
- Leerzeichen (*blank*), Zeilenumbruch usw. dürfen **nicht** in Symbolen auftreten, außer in Zeichenketten & Kommentaren.
Sie werden ignoriert, sofern sie nicht Symbole trennen.
`IF zero<one THEN`
`IFzero < one T H E N`

6.2.1 Namen

Name = Folge von Buchstaben & Dezimalziffern, beginnend mit Buchstabe.

Es gibt

- **vordefinierte &**
- **benutzerdefinierte** Namen.

Syntax

letter = „A“ .. „Z“ | „a“ .. „z“.

digit =

„0“ | „1“ | „2“ | „3“ | „4“ |
„5“ | „6“ | „7“ | „8“ | „9“.

ident = *letter* { *letter* | *digit* }.

Beispiele

X

a1

I1MyMod2

In

Out

Do

TRUE

INTEGER

6.2.2 Zahlen

Zahl = vorzeichenlose konstante ganze Dezimal- oder Hexadezimalzahl oder reelle Zahl.

„E“ bedeutet bei reellen Zahlen „mal 10 hoch“.

Syntax

hexDigit = *digit* | „A“ | „B“ | „C“ | „D“ | „E“ | „F“.

integer = *digit* { *digit* } | *digit* { *hexDigit* } „H“.

ScaleFactor = „E“ [„+“ | „-“] *digit* { *digit* }.

real = *digit* { *digit* } „.“ { *digit* } [*ScaleFactor*].

number = *integer* | *real*.

Beispiele

1998 ganze Dezimalzahl,
 Typ INTEGER, Wert 1998.

0AFFE0000CAFE0000H
 ganze Hexadezimalzahl,
 Typ LONGINT, Wert ?.

12.34 REAL-Zahl, Wert 12.34

5.67E8 REAL-Zahl, Wert 567000000.

0.59712566E-6 REAL-Zahl,
 Wert 0.00000059712566.

6.2.3 Zeichen

Zeichen = Ordinalzahl (Codezahl) des Zeichens in hexadezimaler Darstellung, gefolgt von einem X.

Syntax

character = *digit* { *hexDigit* } „X“.

Beispiel

061X Zeichen 'a'.

6.2.4 Zeichenketten

Zeichenkette = Folge von Zeichen, in einfache oder doppelte Hochkommata eingeschlossen.

Länge der Zeichenkette = Anzahl der Zeichen.

Zeichenkette der Länge 1 = Zeichen.

Syntax

string = „'“ { *char* } „'“ | ’"’ { *char* } ’"’.

Beispiele

'This is a string.'

"Don't worry!"

'He said: "Alright now!"'

6.2.5 Operatoren und Begrenzer

Operatoren & Begrenzer =

- Sonderzeichen, Paare von Sonderzeichen oder
- reservierte Wörter (Schlüsselwörter).

Reservierte Wörter bestehen aus Großbuchstaben & dürfen nicht für Namen verwendet werden.

	Operatoren	Begrenzer
Sonder- zeichen	+ - * / ~ & = # < > <= >=	. , ; () [] { } := ^ .. : \$
reser- vierte Wörter	DIV IN IS OR MOD	ABSTRACT ARRAY BEGIN BY CASE CLOSE CONST DO ELSE ELSIF EMPTY END EXIT EXTENSIBLE FOR IF IMPORT LIMITED LOOP MODULE NIL OF OUT POINTER PROCEDURE RECORD REPEAT RETURN THEN TO TYPE UNTIL VAR WHILE WITH

6.2.6 Kommentare

Kommentar = beliebiger Text, durch (* und *) begrenzt, zwischen zwei Symbolen.

Kommentare sind schachtelbar & beeinflussen das Programm nicht.

Syntax

Übungsaufgabe!

Beispiel

(* *This is a comment.* *)

6.2.7 Vordefinierte Namen

- Werte:

FALSE TRUE

- Typnamen:

BOOLEAN	BYTE	CHAR	INTEGER
LONGINT	REAL	SET	SHORTCHAR
SHORTINT	SHORTREAL		

- Standardprozeduren:

ABS	ASH	ASSERT	BITS	CAP	CHR
DEC	ENTIER	EXCL	HALT		INC
INCL	LEN	LONG	MAX	MIN	NEW
ODD	ORD	SHORT	SIZE		

6.3 Grundtypen

Bedeutung	Typname Format	Werte bzw. Wertebereich Genauigkeit
Boole'sch	BOOLEAN 1 Byte	FALSE TRUE
Zeichen	SHORTCHAR 1 Byte	Latin1-ASCII-Zeichen 0X .. 0FFX
	CHAR 2 Byte	Unicode-Zeichen 0X .. 0FFFFX
Ganze Zahlen, Integer- typen	BYTE 1 Byte	-128 .. 127
	SHORTINT 2 Byte	-32768 .. 32767
	INTEGER 4 Byte	-2^{31} .. $2^{31}-1$
	LONGINT 8 Byte	-2^{63} .. $2^{63}-1$
Reelle Zahlen, Real- typen	SHORTREAL 32-Bit-IEEE	-3.4E38 .. 3.4E38, INF 7-8 Dezimalstellen
	REAL 64-Bit-IEEE	-1.8E308..1.8E308, INF 15-16 Dezimalstellen
Mengen	SET 4 Byte	Teilmenge von 0 .. 31

■ Vordeklarierte Standardfunktionen:

- MIN (Grundtyp)
liefert den kleinsten,
- MAX (Grundtyp)
den größten Wert des Grundtyps.

6.4 Ablaufstrukturblöcke

■ Zuweisung - Wertzuweisung:

`variableName := Expr`

`y := Sin (2 * Pi * omega) + 17`

■ Bezugnahme - Prozeduranweisung:

`ProcName (ParameterList)`

`ObxHello.Do`

`Convert (string, int, error)`

`StuMod.list.Insert (element)`

■ Aneinanderreihung - Anweisungsfolge:

`Statement;`

`Statement;`

`Statement;`

`...`

■ Auswahl - **bedingte Anweisung:**

```
IF Condition_1 THEN  
    StatementSeq  
ELSIF Condition_2 THEN  
    StatementSeq  
ELSE  
    StatementSeq  
END
```

```
IF  $x < 0$  THEN  
    Out.String("negative")  
ELSE  
    Out.String("nonnegative")  
END
```

■ Wiederholung - **Wiederholungsanweisung:**

```
WHILE Condition DO  
    StatementSeq  
END
```

```
WHILE  $x > 0$  DO  
     $x := x / 2;$   
END
```

6.5 Elementare Ein-/Ausgabe

- Zum Lesen von einem Eingabefenster:
Modul `In`.
- Zum Schreiben in das Log-Fenster:
Modul `Out`.
- Für graphische Ausgabe in ein Fenster, Eingabe von der Tastatur:
Modul `XYplane`.
- Siehe Online-Auskunft
mit Menü `Info->Documentation`.

Kapitel 7

Algorithmen

7.1 Was ist ein Algorithmus?

Am Anfang steht immer ein **Problem**. Bei vielen Problemen wissen wir nicht einmal, wie eine Lösung aussehen könnte. Im folgenden verlangen wir aber, daß zu einer Beschreibung des Problems auch eine Beschreibung einer **Lösung** des Problems gehört. Das Ziel ist also schon bekannt, der Weg dorthin jedoch noch unbekannt. Als Nächstes müssen wir einen geeigneten **Lösungsweg** finden. Ein Problem, bei dem Lösung und Lösungsweg bekannt sind, ist eine **Aufgabe**. Viele Probleme und Aufgaben des täglichen Lebens erledigen wir intuitiv, durch spontane Einfälle, mit Fingerspitzengefühl, nach Lust und Laune oder nach Versuch und Irrtum. Demgegenüber betrachten wir hier nur Lösungswege, bei denen alle Schritte geregelt, strukturiert, geordnet sind und die man systematisch durchlaufen kann.

Ein **Algorithmus** (*algorithm*) ist eine Vorschrift, nach der eine Aufgabe schematisch durch bestimmte Tätigkeiten gelöst werden kann. Diese Definition ist noch ungenau, aber bevor wir uns genauer mit Merkmalen von Algorithmen befassen, betrachten wir einige Beispiele.

7.1.1 Beispiele

Nicht jede Vorschrift ist ein Algorithmus!

- „Sei anständig!“,
- „Seid nett zueinander!“

sind allgemeine Problemlösungsleitgedanken, keine konkreten Handlungsanweisungen.

- Montieranleitungen,
- Kochrezepte

sind Handlungsanweisungen, denen aber einige Merkmale von Algorithmen fehlen.

+ Euklidischer Algorithmus.

7.1.2 Merkmale von Algorithmen

- Ein Algorithmus ist ein Verfahren zur Ermittlung *gesuchter* Größen aus *vorgegebenen* Größen.
- Die Beschreibung eines Algorithmus besteht aus *endlich vielen Regeln* für einzelne Schritte, Aktionen, Tätigkeiten, Operationen, Anweisungen usw., die in einem gewissen Sinne elementar sind.

- Die Regeln eines Algorithmus werden durch *Reihenfolgen*, *Verzweigungen* und *Wiederholungen* zu einem Verfahren kombiniert.
- Die Ausführung eines Algorithmus bricht nach *endlich vielen Schritten* ab (auch wenn Regeln wiederholt ausgeführt werden).
- Die Regeln stellen nur eine Hälfte eines Algorithmus dar, sie beschreiben, *was* gemacht, welche Operation ausgeführt werden soll. Das ist aber nur sinnvoll, wenn gleichzeitig gesagt wird, *woran* etwas gemacht, welche Dinge durch die Operation verändert werden sollen. Im allgemeinen Fall wird Material verarbeitet, bei der Datenverarbeitung sind es selbstverständlich Daten.
- Es stellt sich auch die Frage, *womit* etwas gemacht werden soll. Die allgemeine Antwort lautet: mit Werkzeugen. Bei der Datenverarbeitung bestehen Werkzeuge wiederum aus Algorithmen und Daten.
- Ein Algorithmus löst i.a. nicht nur eine einzelne Aufgabe, sondern eine ganze *Klasse* von Aufgaben.

Algorithmen und Daten gehören zusammen. Der Streit, was von beiden wichtiger sei, erinnert an die Frage, was zuerst da war - die Henne oder das Ei. Wenn wir den Stoff in die Kapitel „Algorithmen“ und „Daten“ gliedern, dann hat das lediglich Darstellungsgründe. In Component Pascal werden Algorithmen und Daten, die zusammenwirken, um eine abgegrenzte Teilaufgabe zu lösen, in Modulen zusammengefaßt.

7.2 Darstellung und Entwicklung von Algorithmen

7.2.1 Anforderungen

- Algorithmen fallen nicht vom Himmel, sie müssen ausgedacht, gefunden, entworfen, entwickelt werden. Wir brauchen geeignete Darstellungsformen, die uns als Hilfsmittel beim Entwerfen von Algorithmen dienen.
- Ein Algorithmus soll für den Menschen verständlich dargestellt sein. Das bedeutet, wir brauchen
 - eine informale Darstellung,
 - die dem Problem angepaßt, d.h. in der Fachsprache des Anwendungsbereichs formuliert ist,
 - in der das Wesentliche des Problems und der Lösung hervorgehoben und
 - und von unwesentlichen Details der technischen Realisierung abstrahiert wird.
- Ein Algorithmus soll durch eine universelle Rechenmaschine ausführbar sein. Das bedeutet, wir brauchen
 - eine formale Darstellung,
 - die der Maschine angepaßt, d.h. mit ihren Befehlen formuliert ist,
 - wobei diese Befehle ganz elementare, primitive Operationen sind und
 - die Beschreibung ganz genau, eindeutig und vollständig sein muß.

Diese Anforderungen widersprechen sich zumindest teilweise! Die Lösung dieses Konflikts besteht darin, daß wir nebeneinander unterschiedliche Mittel zur Darstellung von Algorithmen einsetzen:

- Halbformale Darstellungen für den Entwurf, günstigerweise in graphischer Form, denn ein Bild sagt mehr als tausend Worte, Beispiel: Struktogramme.
- Formale Darstellungen zur Programmierung, textuell formuliert in einer höheren Programmiersprache mit den Merkmalen
 - formal, somit in Maschinensprache übersetzbar;
 - mit problemorientierten Sprachelementen, somit für geschulte Menschen verständlich und problemorientiertes Programmieren ermöglichend.

7.2.2 Was ist ein Programm?

Ein **Programm** (*program*) ist ein Algorithmus in einer maschinell ausführbaren Form. Ein Programm ist ein Objekt, etwas Statisches. Davon zu unterscheiden ist die **Ausführung** (*execution*) oder der **Ablauf** (*run*) eines Programms, das ist ein Vorgang, etwas Dynamisches. Zur Ausführung eines Programms wird ein ausführendes Organ benötigt - ein Rechner, genauer gesagt eine Komponente eines Rechners, die **Prozessor** (*processor*) genannt wird.

Wir haben schon darauf hingewiesen, daß Algorithmen und Daten zusammenhängen. Von Niklaus Wirth stammt die etwas plakative Formel:

Programm = Algorithmus + Daten.

Dabei ist der Algorithmus das Statische und die Daten sind das Dynamische: Die Daten werden durch die Ausführung des Algorithmus verändert. Die obige Formel ist ganz allgemein gehalten, sie gilt für verschiedene *Programmierungsansätze*, die die Informatik in ihrer kurzen Geschichte hervorgebracht hat.

Der Sicht der sog. *strukturierten Programmierung*, die Ende der 60er Jahre entwickelt wurde und für die Programmiersprachen wie Pascal und C stehen, entspricht die Formel

Programm = Menge von Prozeduren = Algorithmen + Daten.

Eine Weiterentwicklung der 70er Jahre ist die *modulare Programmierung*, die wir mit Component Pascal kennenlernen; aus ihrer Sicht gilt die Formel

Programm = Menge von Modulen = Algorithmen + Daten.

Der nächste Schritt ist die *objektorientierte Programmierung* (OOP), die sich in den 80er Jahren verbreitete und die wir später auch mit Component Pascal kennenlernen werden; die Formel wird dann abgewandelt zu

Programm = Menge von Klassen = Algorithmen + Daten.

Bei der *komponentenorientierten Programmierung* der 90er Jahre, für die bisher Oberon, Component Pascal und Java die einzigen Sprachen sind, gilt

Programm = Menge von Komponenten = Algorithmen + Daten.

In diesem Kapitel unterscheiden wir nicht, ob wir es nur mit einem Algorithmus oder mit mehreren zusammenwirkenden Algorithmen zu tun haben.

Ein Programm kann mehrere Male - beliebig oft - ausgeführt werden. Es handelt sich dann um verschiedene Programmabläufe desselben Programms. Die Abläufe können zeitlich nacheinander auf demselben Prozessor oder räumlich getrennt auf verschiedenen Prozessoren erfolgen. Programmabläufe können sich wesentlich in ihrem Verhalten, in den Ergebnissen, die sie produzieren, unterscheiden, obwohl das zugrundeliegende Programm immer dasselbe ist und sich nicht verändert.¹ Unterschiedliches Verhalten von Programmabläufen rührt natürlich in erster Linie von unterschiedlichen Eingabedaten her, es können aber auch andere Einflüsse mitwirken.

Ein Prozessor kann zu einem Zeitpunkt nur ein Programm ausführen, aber mehrere Programme nacheinander. Was auf einem Prozessor abläuft, wird unter dem Begriff **Prozeß** (*process*) gefaßt, das ist also eine dynamische Einheit, die mehrere aufeinanderfolgende Programmabläufe umfassen kann.

Der Begriff des Programms ist noch recht allgemein, er umfaßt verschiedene Formen. Ein Programm heißt **direkt ausführbar**, wenn es in einer Form vorliegt, die unmittelbar auf einem Prozessor ablaufen kann. Ein Programm heißt **indirekt ausführbar**, wenn es in einer Form vorliegt, von der es automatisch in ein direkt ausführbares Programm transformiert werden kann. Diese Transformation wird durch ein anderes Programm - genauer: durch die Ausführung eines anderen Programms - bewerkstelligt und bleibt der BenutzerIn oft verborgen.

7.2.3 Modell des Wegs vom Problem zur Lösung

Den Dualismus Daten/Algorithmen haben wir bereits kennengelernt, ebenso die Unterscheidung zwischen statischen und dynamischen Aspekten. Solche Dualismen treten häufig auf - so können wir z.B. unterscheiden zwischen

- *Tätigkeiten* und *Dingen*, die durch die Tätigkeiten verändert werden,
- *Operationen* und *Operanden*, auf denen die Operationen arbeiten,
- *Ereignissen* und *Zuständen*, die durch die Ereignisse herbeigeführt werden.

Wenden wir nun diese Zweiteilung auf den Weg vom Problem zur Lösung an. Dann müssen wir noch überlegen, welche geeigneten *Akteure* die Tätigkeiten oder Operationen ausführen bzw. die Ereignisse auslösen.

7.2.3.1 Tätigkeiten

Die folgenden Tätigkeiten sind auszuführen.

1. Das Problem beschreiben.
2. Die Dinge des Problembereichs identifizieren, durch Daten beschreiben und in Module gruppieren.
3. Die Tätigkeiten des Problembereichs identifizieren, den Modulen zuordnen und Algorithmen dazu entwerfen.

¹ Es gibt Programme, die sich selbst verändern - diese bleiben hier außer Betracht.

4. Programmieren: die Module, Daten und Algorithmen in der höheren Programmiersprache formulieren.
5. Übersetzen: das Programm in die Maschinsprache transformieren.
6. Ausführen: das Programm auf dem Rechner zum Ablauf bringen.
7. Die von dem Programmablauf erzeugten Ausgabedaten interpretieren und überprüfen, ob sie als Lösung des Problems akzeptabel sind.

7.2.3.2 Vorgegebenes und Ergebnisse

Die folgenden Dinge sind vorgegeben bzw. ergeben sich der Reihe nach.

1. Das Problem.
2. Der Lösungsentwurf mit Modulen, Daten und Algorithmen.
3. Das Quellprogramm.
4. Das Objektprogramm.
5. Die Eingabedaten.
6. Die Ausgabedaten.
7. Die Lösung.

7.2.3.3 Akteure

Die Tätigkeiten müssen von einem Subjekt oder Organ ausgeführt werden.

1. Der Mensch.
2. Die Maschine.²

7.2.3.4 Darstellungen

1. Als Flußdiagramm.
2. Als Zustandsübergangsdiagramm.
3. Als Petri-Netz.

Siehe Folien.

7.3 Entwurf von Algorithmen

Unsere Devise bei der Software-Entwicklung sei stets:

■ **Erst denken, dann handeln!**

■ **Erst entwerfen, dann programmieren!**

² Aus der formalen Gleichbehandlung von Mensch und Maschine in diesem Modell soll nicht auf ihre Gleichsetzung oder Austauschbarkeit geschlossen werden.

7.3.1 Prinzipien

Wir nennen einige wichtige Grundsätze, die uns als Leitfäden beim Algorithmenentwurf dienen sollen.

7.3.1.1 Abstraktion

- Betrachte die wesentlichen Merkmale, Beziehungen, Größen eines Problems!
- Vernachlässige unwesentliche Details!

7.3.1.2 Zerlegung

- Teile und (be-)herrsche!
- Wenn Du ein Problem nicht direkt lösen kannst, dann zerlege es in kleinere Teilprobleme!

7.3.1.3 Strukturierung

- Ordne die Merkmale, Eigenschaften, Größen eines Problems und die Elemente eines Algorithmus überschaubar an!

7.3.2 Konzepte und Methoden

Die genannten Prinzipien sind zwar richtig, aber so allgemein, daß sie uns bei einem konkreten Problem noch keinen Lösungsweg aufzeigen. Die folgenden Ideen und Ansätze helfen uns schon eher, eine Vorgehensweise zu finden, nach der wir Algorithmen entwickeln können.

7.3.2.1 Top-down-Entwurf

- Gehe vom Problem aus,
- formuliere einen Algorithmus in abstrakter Form,
- konkretisiere diesen dann schrittweise,
- bis er eine für die Maschine geeignete Form hat!

7.3.2.1.1 Beispiel: Sieb des Eratosthenes

Version 1:

Aufgabe: Primzahlenbestimmung

Gegeben: $n \in \mathbb{N}$

Gesucht: Alle Primzahlen $p_1, \dots, p_k \in \{1, \dots, n\}$.

Lösungsweg:

Schritt 1: Stelle Anfangssituation her.

Schritt 2: Suche die Primzahlen.

Schritt 3: Zeige die gefundenen Primzahlen an.

Dies ist eine abstrakte, grobe Formulierung des Problems und seiner Lösung, die aber schon ordentlich strukturiert ist. In nachfolgenden Entwurfsschritten konkretisieren und verfeinern wir die Lösung - siehe weitere Versionen in der Vorlesung.

7.3.2.2 Schrittweise Verfeinerung

Schrittweise Verfeinerung (*stepwise refinement*) bezeichnet eine Vorgehensweise für den Algorithmenentwurf, mit der man von einer groben Problemlösung zu immer feiner ausgearbeiteten Lösungen kommt. Dieser Ansatz verbindet Konkretisierung und Zerlegung mit der Top-Down-Strategie.

- Führe den Entwurf als Folge von Verfeinerungsschritten aus!
- Zerlege bei jedem Schritt die bereits erreichten (abstrakten) algorithmischen Lösungselemente in detailliertere (konkretere) Elemente!

7.3.2.3 Strukturierte Programmierung

Strukturierte Programmierung (*structured programming*) ist ein Konzept zur Entwicklung und Strukturierung von Programmen. Es wurde Ende 60er bis Mitte 70er Jahre entwickelt, um Programme verständlicher zu machen. Strukturiert programmieren kann man mit den meisten höheren imperativen Programmiersprachen. Manche Sprachen erlauben auch unstrukturiertes Programmieren, während Sprachen wie Component Pascal strukturiertes Programmieren besonders unterstützen. Wir stellen hier kurz den Teil des Konzepts dar, der sich mit der Strukturierung von Algorithmen befaßt.

Entwerfe und programmiere nach folgenden Regeln:

- Ein Algorithmus besteht aus **Strukturblöcken**.
- **Atomare** Strukturblöcke sind
 - **Zuweisung**,
 - **Bezugnahme** auf einen Strukturblock.
- Weiteres Strukturelement: **Bedingung**.
- Strukturblöcke können durch
 - **Aneinanderreihung** (Sequenz),
 - **Auswahl** (Selektion),
 - **Wiederholung** (Iteration),
 - **Schachtelung**zu neuen Strukturblöcken zusammengesetzt werden.
- Andere Strukturierung gibt es nicht.

7.3.2.4 Zerlegung in Kommandos und Prozeduren

Top-down-Entwurf und schrittweise Verfeinerung lassen sich anwenden, wenn sich das Problem mit einem einzigen Algorithmus lösen läßt. Oft muß man aber mehrere Algorithmen entwickeln, um eine Aufgabe zu lösen.

Mit Component Pascal lösen wir eine komplexe Aufgabe typischerweise dadurch, daß wir sie in Teilaufgaben zerlegen, die durch einzelne Kommandos und andere Prozeduren bearbeitet werden. Jede Prozedur enthält dann nur einen kleinen, verständlichen Algorithmus. Die Problemlösung erhalten wir, indem wir einige Kommandos aufrufen.

7.3.3 Graphische Darstellungsmittel

7.3.3.1 Flußdiagramme

Flußdiagramme (Programmablaufpläne, Blockdiagramme, *flow charts*) sind ein altes, nach DIN 66001 (1977) genormtes Darstellungsmittel für Ablaufstrukturen von Algorithmen. Ihre wesentlichen Strukturierungselemente sind Aktionen und Bedingungen.

Flußdiagramme sind immer noch weit verbreitet, so daß man sie passiv verstehen können muß. Wir raten jedoch von ihrer aktiven Nutzung ab, weil sie technisch überholt sind: Flußdiagramme ermöglichen zwar auch strukturierte Programmierung, unterstützen es aber nicht. Man kann mit ihnen leicht schlechte Ablaufstrukturen entwerfen - sogenannte Spaghetti-Algorithmen, die schwer durchschaubar sind. Außerdem vernachlässigen sie den Datenaspekt - sie stellen dar, *was* gemacht wird, aber es bleibt undeutlich, *woran* es gemacht werden soll.

7.3.3.2 Struktogramme

Struktogramme (*Nassi-Shneiderman-Diagramme*) sind ein graphisches Darstellungsmittel für die Strukturblöcke der strukturierten Programmierung.

Siehe Extrablätter und Vorlesung.

7.3.3.2.1 Hinweise zu Struktogrammen

Struktogramme sind vor allem nützlich, wenn man lernen will, Algorithmen strukturiert zu entwerfen. Sie sind programmiersprachenunabhängig und daher frei von unnötigem, zufälligem syntaktischem Zucker. Sie visualisieren das Wesentliche - die algorithmische Struktur - anschaulich zweidimensional, im Unterschied zu einem sequentiellen Programmtext.

Die wesentliche Denkarbeit beim Algorithmenentwurf ist, die algorithmische Struktur zu finden. Dabei können Struktogramme helfen. Die Struktogramme in eine Programmiersprache umzusetzen ist dann lediglich schematische Routinearbeit.

Struktogramme zeichnen kann mühsam sein. Zur Erleichterung der Arbeit gibt es Software-Werkzeuge, die einen beim Zeichnen unterstützen. Die Frage ist: Lohnt sich der Einsatz eines Werkzeugs? Kleine Struktogramme zum Üben zeichnet man schneller auf Papier.

Struktogramme nicht bis ins letzte Detail verfeinern! Meist reicht ein Struktogramm für den Grobentwurf. Feinheiten lassen sich am Programm ausfeilen.

Bei größeren Algorithmen muß man zuerst die Grobstruktur erkennen können. Das ist für die Verständlichkeit wichtig und deshalb bei der Dokumentation zu beachten. Eine Hierarchie schrittweise verfeinerter Struktogramme kann hier helfen.

7.3.3.2 Regeln zum Programmieren

Die algorithmischen Strukturen sollen auch am fertigen Quellprogramm erkennbar sein. Beim Übertragen von Struktogrammen, beim Zusammensetzen verfeinerter Lösungselemente besteht die Gefahr, daß die Übersicht über die Grobstruktur im Quellprogramm verlorengeht. Deshalb halten wir uns beim Programmieren an folgende Regeln:

- Die Grobstruktur eines Algorithmus muß im Programmtext sichtbar sein.
 - Eine Möglichkeit: Die Grobstruktur läßt sich durch Kommentare zwischen wesentlichen Strukturblocken hervorheben.
 - Besser als Kommentare ist die Möglichkeit, wesentliche Strukturblocke durch prägnant benannte Prozeduren zu realisieren.
- Die Feinstruktur eines Algorithmus läßt sich durch systematische Einrückung geschachtelter Strukturblocke optisch herausheben.

7.3.3.3 Zustandsübergangsdiagramme

Flußdiagramme und Struktogramme betonen Aktionen, Anweisungen, Ereignisse als wesentliches Element von Algorithmen. Es gibt aber auch eine dazu komplementäre Sicht, die Zustände, Situationen, Gegebenheiten hervorhebt - also das, was zwischen verändernden Aktionen eine Weile unverändert andauert.

Zustandsübergangsdiagramme (endliche Automaten, *state transition diagram*, *finite state machine*) sind ein Darstellungsmittel für die zustandsorientierte Sicht von Algorithmen. Ihr wesentliches Strukturierungselement sind Zustände.

Siehe Extrablätter und Vorlesung.

Sie lernen Zustandsübergangsdiagramme als endliche Automaten in einer streng formalisierten Variante in der Vorlesung *Digitaltechnik* genauer kennen. Zustandsübergangsdiagramme sind aber auch ein wichtiges Mittel beim Entwerfen von Programmen. Dabei verwendet man oft halbformale Varianten.

Zwischen Zustandsübergangsdiagrammen und formalen Sprachen gibt es grundlegende Beziehungen: Jedes Zustandsübergangsdiagramm beschreibt eine formale Sprache. Umgekehrt gibt es zu jeder Sprache, die sich z.B. mittels EBNF-Regeln ohne Rekursion oder mittels einer einzigen EBNF-Regel beschreiben läßt, ein Zustandsübergangsdiagramm, das dieselbe Sprache beschreibt. Das Zustandsübergangsdiagramm ist dann praktisch das Syntaxdiagramm der Sprache. Eine so darstellbare Sprache heißt **regulär**. Zu **regulären Sprachen** gibt es also unterschiedliche, aber äquivalente Beschreibungen.

Das Arbeiten mit Beschreibungen und Transformieren von Beschreibungen in andere Beschreibungen, die dieselben Sachverhalte beschreiben - das ist eine wesentliche Aufgabe der Informatik.

Jeder Rechner ist im Grunde ein endlicher Automat. Trotzdem sind reguläre Sprachen eine spezielle Sprachklasse mit eingeschränkter Ausdrucksfähigkeit, die noch nicht einmal zur Beschreibung arithmetischer Ausdrücke ausreicht. Programmiersprachen sind daher nicht regulär, sondern gehören zu größeren, mächtigeren Sprachklassen. Die Component-Pascal-Syntax definiert z.B. eine **kontextfreie Sprache**, bei diesen steht auf der linken Seite jeder EBNF-Regel nur ein Nichtterminal (nicht mehrere). Programmierspra-

chen enthalten aber reguläre Teilsprachen. Die **lexikalische Ebene** behandelt nur reguläre Teilsprachen, während die **syntaktische Ebene** die nichtregulären Teile umfaßt.

7.3.3.4 Petri-Netze

Struktogramme und Zustandsübergangsdiagramme heben jeweils einen Aspekt hervor: Aktionen oder Zustände - den anderen Aspekt vernachlässigen sie jeweils. Dies kann sich als nachteilig erweisen.

Petri-Netze (*Petri nets*) sind ein Darstellungsmittel für Ablaufstrukturen, das beide Aspekte berücksichtigt. Ihre wesentlichen Strukturierungselemente sind Bedingungen und Ereignisse bzw. Stellen und Transitionen, die die Bedeutung von Zuständen und Aktionen haben.

Petri-Netze sind ein sehr mächtiges Modellierungsmittel, das Sie in der Vorlesung *Software Engineering* genauer kennenlernen werden.

7.4 Korrektheit von Algorithmen

Ob ein Algorithmus korrekt ist oder nicht, kann nur in Bezug auf eine vollständige, genaue und eindeutige Aufgabenstellung entschieden werden. Wo die Aufgabe nicht klar definiert ist, erübrigt sich die Diskussion um Korrektheit. Zwei Fragen sind von Bedeutung:

- Löst der Algorithmus die gestellte Aufgabe? Das ist besonders wichtig, denn ein Algorithmus, der nicht tut, was er soll, ist meist unbrauchbar.
- Bricht der Algorithmus nach endlich vielen Schritten ab? Das ist auch wichtig, denn ein Algorithmus, der zwar auf dem richtigen Weg ist, aber sein Ziel nie erreicht, weil er läuft und läuft und läuft...

7.5 Komplexität von Algorithmen

Hier geht es um den Aufwand, den ein Algorithmus benötigt. Zwei Fragen sind von Bedeutung:

- Wieviele Operationen, d.h. Ausführungszeit braucht der Algorithmus? Wir können ja nicht Jahre auf das Ergebnis warten.
- Wieviel Hilfsmaterial - bei EDV Hilfsdaten, d.h. (im Vorgriff auf Späteres) Speicherplatz - braucht der Algorithmus? Wir haben ja nur beschränkte Betriebsmittel zur Verfügung.

Die Komplexitätstheorie befaßt sich mit der Berechnung des Zeit- und Speicherplatzbedarfs von Algorithmen, bzw. von unteren und oberen Schranken dafür. Interessiert ist man selbstverständlich an möglichst **effizienten** (*efficient*) Algorithmen, d.h. solchen, die möglichst schnell und platzsparend arbeiten. Die Verständlichkeit und Lesbarkeit eines Programms darf aber andererseits nicht bedenkenlos der Effizienz geopfert werden. Rechnerleistung und Speicherkapazität werden immer kostengünstiger, so daß man auf andere Programmqualitäten mehr Wert legen kann.

Kapitel 8

Daten

Dieses Kapitel gibt einen Überblick über verschiedene Aspekte des Begriffs Daten. Wir halten die Darstellung allgemein, unabhängig von einer speziellen Programmiersprache; um die Gedanken zu veranschaulichen dienen uns aber Beispiele in Component Pascal.

8.1 Exemplare und Typen

Bei Daten müssen wir - wie bei anderen Dingen - zwischen Objekten und Klassen unterscheiden. Die Realität besteht aus vielen einzelnen, unterscheidbaren, *konkreten Objekten*, Dingen, Gegenständen. Gedanklich können wir Objekte mit gleichen oder ähnlichen Merkmalen und Eigenschaften zu einer Einheit zusammenfassen, zu einer *abstrakten Klasse*. Umgekehrt definiert eine Klasse einen *Typ*, ein Muster für gleichartige Objekte. Ein Objekt ist dann ein *Exemplar* eines bestimmten Typs.¹

Beispiele:

Exemplar, Objekt

Mia Maier, Hans Schmidt
mein Hund, deine Katze, diese Maus
Reutlingen, Tübingen
0815, 2001, 4711

Typ, Klasse

Mensch
Säugetier
Stadt
natürliche Zahl

Eine ähnliche Abstraktion brauchen wir bei Daten.

Bearbeitet werden konkrete **Datenelemente** (*data element*), auch Datenexemplare oder Datenobjekte genannt. Jedes Datenelement ist ein **Exemplar** (*instance*) oder **Objekt** (*object*) eines bestimmten Typs, der auch als Typ des Datenelements, Exemplars oder Objekts bezeichnet wird.²

Ein **Typ** (*type*) oder **Datentyp** (*data type*) ist eine Abstraktionsklasse von Datenelementen mit gewissen Merkmalen, genauer: mit gleicher Struktur. Ein Typ ist durch

- eine Menge von Werten und
- eine Menge von Operationen

¹ Die Begriffe „Objekt“ und „Klasse“ sind in der objektorientierten Programmierung mit spezielleren Bedeutungen belegt. Hier gebrauchen wir sie mit allgemeinerer Bedeutung. Um Verwechslungen vorzubeugen, verwenden wir im folgenden meist „Exemplar“ statt „Objekt“ und „Typ“ statt „Klasse“.

² In der Literatur wird „instance“ manchmal mit „Instanz“ übersetzt, aber „Instanz“ bedeutet im Deutschen „Behörde“, d.h. nichts hier Passendes.

beschrieben. Der **Wertebereich** legt die Werte fest, die Exemplare dieses Typs annehmen können, d.h. ihre möglichen Zustände. Die **Operationen** legen fest, wie die Exemplare dieses Typs bearbeitet werden können, d.h. die zulässigen Aktionen.

Beispiel:

Typ:	Ganze Zahlen;
Wertebereich:	$\{-2^{31}, \dots, 2^{31}-1\}$;
Operationen:	Addition, Subtraktion, Multiplikation, Division,...

Beispiel in Component Pascal:

Typ:	INTEGER
Wertebereich:	$-2^{31} \dots 2^{31}-1$
Operationen:	+ - * DIV MOD = # < > <= >=

Zu einer konkreten Implementierung eines Typs gehört außerdem eine **Speicherstruktur**. Diese legt fest, wieviele Bytes Speicherplatz ein Exemplar benötigt, und welche Bitkombinationen in diesen Bytes welchen Wert repräsentieren, d.h. die Interpretation eines Bitmusters als Wert des Typs.

Zu einem Typ kann es mehrere - viele - Exemplare dieses Typs geben. Exemplare werden entweder statisch vereinbart oder dynamisch erzeugt.³

8.2 Werte und Namen

Exemplare besitzen einen **Wert** (*value*). Der Wert eines Exemplars liegt in dem durch seinen Typ festgelegten Wertebereich. Er kann sich i.a. ändern.

Es gibt mehrere Gründe, **Namen** oder **Bezeichner** (*identifier*) einzuführen.

- Ein Name kann als Symbol für einen Wert benutzt werden.

Beispiel: π für 3,14...

- Ein Name kann als Bezeichner für ein Exemplar benutzt werden. Auf das Exemplar wird mittels seines Namens zugegriffen.

Beispiel: n für eine ganze Zahl.

- Mehrere Exemplare (desselben oder unterschiedlichen Typs) werden erst durch Namen unterscheidbar. Es gibt eine Zuordnung zwischen Namen und Exemplaren.

Beispiel: n, m für zwei ganze Zahlen.

- Ein Name kann als Bezeichner für einen Typ verwendet werden.

Beispiel: **INTEGER** für den Typ „ganze Zahl“.

- Ein Name kann als Bezeichner für eine Prozedur verwendet werden.

Beispiel: **DO** für eine einfache Kommandoprozedur.

³ Die Erzeugung eines Exemplars eines bestimmten Typs heißt manchmal wegen ungeschickter Übersetzung **Instanzierung** (*instantiation*).

- Ein Name kann als Bezeichner für ein Modul verwendet werden.

Beispiel: `I1Primes` für das Modul, das sich mit Primzahlen befaßt.

- Ein Name kann als Bezeichner für ein Verbundfeld (Attribut) verwendet werden.

Beispiel: `name` für ein Attribut des Verbundtyps `AddressDesc`.

Zusammengefaßt: Namen bezeichnen **Größen**. Bezeichnete Größen sind z.B. Werte, Exemplare, Typen, Prozeduren, Module usw.

Mit Namen und Größen - insbesondere Exemplaren - erscheinen folgende Probleme.

- (1) Exemplare ohne Namen: **Anonyme Exemplare** sind nicht zugreifbar. Die Situation muß (sollte) vermieden werden.
- (2) Ein Name für mehrere Exemplare: **Namenskonflikte** müssen gelöst werden. Jedem Namen muß - zu einem Zeitpunkt - genau ein Exemplar zugeordnet sein.
- (3) Mehrere Namen für ein Exemplar: **Aliasing** sollte vermieden werden.
- (4) Namen ohne Exemplare: Daß ein Name vereinbart ist, ohne daß gleichzeitig ein Exemplar dafür vorhanden ist, sollte tunlichst ausgeschlossen sein.

In Component Pascal sind die Probleme (1), (2) und (4) durch Regeln der Sprache gelöst. Der Umgang mit Problem (3) ist der ProgrammiererIn überlassen; Exemplare mit mehreren Namen sind nicht unbedingt problematisch, sondern bei manchen Programmieraufgaben sogar notwendig.

Sowohl Werte als auch Namen werden durch Worte über bestimmten Alphabeten dargestellt. Es muß bei einem Wort stets feststellbar sein, ob es sich um einen Wert oder einen Namen handelt. Die Sprache muß dazu eindeutige Festlegungen treffen.

Beispiele in Component Pascal:

Literale Werte: `1 02H 3.4E5 0AAX 'b' "value" FALSE`

Namen: `one H02 E345 AAX b value false BOOLEAN`

8.3 Vereinbarung und Benutzung

Allgemein ist bei Größen zwischen **Vereinbarung** (*declaration*) und **Benutzung** (*usage*) oder **Zugriff** (*access*) zu unterscheiden. In manchen Programmiersprachen werden Exemplare **implizit** bei der ersten Benutzung vereinbart, was hinsichtlich Verständlichkeit und Sicherheit problematisch ist. Deshalb ist es in vielen Programmiersprachen - darunter Component Pascal - üblich, daß Exemplare vor ihrer ersten Benutzung **explizit** vereinbart sein müssen, d.h. im Programmtext steht eine Vereinbarung.

Literale Werte müssen nicht vereinbart werden, sie fallen sozusagen mit der Programmiersprache vom Himmel und können so direkt im Programm benutzt werden. (Es gehört aber zu gutem Programmierstil, literale Werte nicht in Anweisungen zu verwenden, sondern sie in Konstantenvereinbarungen mit Synonymen zu benennen.)

Namen hingegen sind entweder durch die Sprache vordefiniert oder werden vereinbart, wobei in der Vereinbarung gleichzeitig angegeben wird, was für eine Art von Größe der

Name bezeichnen soll: ob ein Modul, eine Konstante, einen Typ, eine Variable, eine Prozedur, einen Parameter usw.

Beispiele in Component Pascal:

Vereinbarung

```
MODULE I1MyModule;

CONST capacity = 1024 * 1024;
TYPE Name = ARRAY 20 OF CHAR;
VAR
    current,
    resistance,
    voltage    : REAL;
PROCEDURE F (x : REAL) : REAL;
```

Benutzung

```
IMPORT I1MyModule;
I1MyModule.Init;
IF x + y <= capacity ...
VAR myName : Name;

voltage :=
    resistance * current;

y := F (x);
```

8.4 Konstanten und Variablen

In der Realität gibt es sowohl feste als auch veränderliche Größen. Dies widerspiegelt sich bei den Daten.

Eine **Konstante** (*constant*) steht für eine feste Größe. Eine **Variable** steht für eine veränderliche Größe.

Bei Konstanten gibt es zwei unterschiedliche Auffassungen.

- Konstante als ausgezeichnete Wert, z.B. π . Sie werden meist durch Synonyme dargestellt, z.B. `Pi`. Programmtechnisch handelt es sich um **Übersetzungszeitkonstanten**, da sie bereits während der Übersetzung des Programms verarbeitet werden können.
- Konstante als unveränderliches Exemplar. Sie werden wie andere Exemplare behandelt, nur mit der Einschränkung, daß ihr Wert nicht verändert werden kann bzw. darf. Man kann auch von Nur-Lese-Variablen sprechen. Programmtechnisch handelt es sich um **Laufzeitkonstanten**, da sie erst während eines Programmlaufs verarbeitet werden.

In Component Pascal gibt es nur Übersetzungszeitkonstanten. Sie werden in Konstantendeklarationen definiert, wobei ihr Wert angegeben wird. Die Darstellung des Werts bestimmt den Typ der Konstanten.

Beispiel in Component Pascal:

Vereinbarung

```
CONST
    maxWidth = 100;
```

Benutzung

```
meanWidth :=
    maxWidth DIV 2;
```

Der Begriff der Variable ist aus der Mathematik bekannt. Er ist aber - trotz gewisser Ähnlichkeiten - von dem Begriff der Variable in der Informatik zu unterscheiden.

- In der Mathematik erscheinen Variablen im Zusammenhang mit Funktionen:

$$y = f(x).$$

Hier ist x die unabhängige und y die abhängige Variable; sie stehen als Platzhalter für viele mögliche Werte aus gewissen Zahlenbereichen. Die Gleichung drückt einen allgemeinen, zeitunabhängigen funktionalen Zusammenhang aus.

- In der Informatik sind Variablen benannte Exemplare: Seien etwa x und y Variablen (irgendeines Typs). Die Gleichung

$$y = f(x)$$

ist ein Ausdruck, dessen Wert „wahr“ ist, wenn der momentane Wert des Ausdrucks auf der rechten Seite gleich dem momentanen Wert der Variable auf der linken Seite ist. Davon zu unterscheiden ist die Zuweisung

$$y := f(x),$$

die der Variable y den momentanen Wert des Ausdrucks auf der rechten Seite zuweist (und damit ihren alten Wert überschreibt).

Variablen werden in Component Pascal mit Variablendeklarationen vereinbart, wobei ihr Typ angegeben wird.

Beispiel in Component Pascal:

Vereinbarung

```
VAR
  meanWidth : INTEGER;
```

Benutzung

```
meanWidth :=
  maxWidth DIV 2;
```

8.5 Vordefinierte und definierbare Datentypen

In den meisten Programmiersprachen gibt es **vordefinierte** Datentypen, also Datentypen, die integraler Bestandteil der Sprache sind. Schließlich unterstützen ja auch die Rechner die wichtigsten Datentypen, die Zahlen.

Beispiele in Component Pascal:

```
BOOLEAN CHAR INTEGER REAL SET
```

Nur mit vordefinierten Datentypen kann man aber nicht - oder nur sehr eingeschränkt - problemorientiert programmieren. Jedes Problem erfordert zu seiner Lösung eigene Datenelemente, deren Struktur aus dem Anwendungsbereich heraus entwickelt wird. Deshalb bieten die meisten höheren Programmiersprachen die Möglichkeit **definierbarer** Datentypen, also von Datentypen, die bei der Erstellung eines Programms festgelegt werden.

Unter dem **Typsystem** einer Programmiersprache verstehen wir die Menge ihrer Typen (die unendlich groß sein kann), zusammen mit den Regeln, nach denen die Typen konstruiert und benutzt werden können. Das Typsystem einer Sprache gehört zu ihren wesentlichen Merkmalen; die verschiedenen Aspekte werden wir in folgenden Kapiteln beleuchten.

Neue Typen werden in Component Pascal bei Typangaben oder mit Typendeklarationen eingeführt, wobei gewisse Konstrukte angewandt werden können.

Beispiel in Component Pascal:

Vereinbarung

TYPE

```
Voltage = REAL;
Vector = ARRAY 9 OF REAL;
AddressDesc =
  RECORD
    name,
    street,
    city : ARRAY 30 OF CHAR
  END;
```

Benutzung

VAR

```
voltage : Voltage;
x, y, z : Vector;
a : AddressDesc;

a.name := "Rick Rap";
a.street := "Route 66";
a.city := "Rockcity";
```

8.6 Einfache Datentypen

Einfache Datentypen sind im Unterschied zu strukturierten Datentypen nicht aus Komponenten zusammengesetzt, nicht weiter in Bestandteile zerlegbar. In den meisten Programmiersprachen sind einige einfache Datentypen vordefiniert.

Beispiel in Component Pascal:

Alle Grundtypen (vordefinierte Typen) sind einfach (siehe 6.3 Grundtypen, S. 9).

Im folgenden befassen wir uns mit Eigenschaften, Wertebereichen, Operationen einfacher Typen.

Beispiel in Component Pascal:

Vordeclarierte Funktionen:

MIN (Grundtyp) : Grundtyp

MAX (Grundtyp) : Grundtyp

Bedeutung:

Liefert zu einem Grundtyp den kleinsten Wert des Wertebereichs.

Liefert zu einem Grundtyp den größten Wert des Wertebereichs.

8.6.1 Zeichen

Zeichen sind wir bereits in 2.1 Zeichen, Alphabete und Symbole, S. 1, 6.2.3 Zeichen, S. 6, und 6.3 Grundtypen, S. 9, begegnet.

Beispiele in Component Pascal:

Vereinbarung

VAR a, b, c : CHAR;

Benutzung

```
a := 'a';
b := "B";
c := 0CCX;
```

Der Wertebereich von Zeichen ist durch einen bestimmten Code festgelegt.

Die Operationen, die mit Zeichen möglich sind, umfassen zumindest die Tests auf Gleichheit und Ungleichheit. Da Zeichen als Binärzahlen codiert und diese geordnet

sind, kann man auch die Zeichen selbst als geordnet betrachten. Damit sind Vergleichsoperationen sowie Vorgänger- und Nachfolgerfunktionen möglich. Außerdem kann man Zeichen und ganze Zahlen im Wertebereich des Codes ineinander konvertieren. Bei Buchstaben kann man zwischen kleinen und großen wechseln.

Beispiel Zeichentypen in Component Pascal:

Typ: SHORTCHAR
 Format: 8 Bit
 Wertebereich: Latin1-Erweiterung des ASCII-Zeichensatzes
 MIN (SHORTCHAR) .. MAX (SHORTCHAR)
 0X .. 0FFX

Typ: CHAR
 Format: 16 Bit
 Wertebereich: Unicode-Zeichensatz
 MIN (CHAR) .. MAX (CHAR)
 0X .. 0FFFFX

Operationen: = # < > <= >=

Vordeclarierte Funktionen:

Bedeutung:

CAP (c : Zeichentyp) : Typ von c

Großbuchstabe zu c
 (falls c Kleinbuchstabe).

ORD (c : CHAR) : INTEGER

ORD (c : SHORTCHAR) : SHORTINT

Ordnungszahl zu c
 (codierter Wert).

CHR (x : Integertyp) : CHAR

Zeichen mit der Ordnungszahl x.

LONG (c : SHORTCHAR) : CHAR

Identischer Wert, Einbettung.

SHORT (c : CHAR) : SHORTCHAR

Identischer Wert, Projektion.

8.6.2 Zahlen und Arithmetik

Aus der Mathematik kennen wir die Mengen der natürlichen, der ganzen, der rationalen, der reellen und der komplexen Zahlen und die arithmetischen und Vergleichs-Operationen, die das Rechnen mit Zahlen bestimmen. Ihnen entsprechen gewisse Datentypen, die in den meisten höheren Programmiersprachen vordefiniert sind. Dabei ist allgemein folgende Einschränkung zu beachten.

- Zahlen können maschinenintern immer nur mit endlich vielen Stellen dargestellt werden. Das bedeutet, daß der Wertebereich jedes Zahlen-Datentyps beschränkt ist. Beim maschinellen Rechnen mit Zahlen können Unter- und Überschreitungen des zugrundeliegenden Wertebereichs auftreten.

Die verschiedenen Zahlentypen haben gewisse gemeinsame Eigenschaften, die wir uns am Beispiel Component Pascal anschauen.

Beispiel Zahlentypen in Component Pascal:

Operationen: + - * = # < > <= >=

<i>Vordeclarierte Funktionen:</i>	<i>Bedeutung:</i>
ABS (x : Zahlentyp) : Typ von x	Absolutbetrag von x.
MAX (x, y : Zahlentyp) : Typ von x, y	Maximum von x, y.
MIN (x, y : Zahlentyp) : Typ von x, y	Minimum von x, y.

8.6.2.1 Ganze Zahlen

Die meisten Programmiersprachen bieten einen Datentyp für ganze Zahlen, meist heißt er - wie in Component Pascal - **INTEGER**, oder kurz **int**. Manchmal gibt es auch einen Typ für natürliche Zahlen (**natural**, **cardinal**, **unsigned int**).

Die Möglichkeiten der Darstellung von Zahlen in verschiedenen Zahlensystemen und die Konvertierungen (Umrechnungen) zwischen Zahlensystemen werden in der Vorlesung „Digitaltechnik“ besprochen. In der Informatik sind Dezimal-, Dual- und Hexadezimalzahlen wichtig.

Component Pascal bietet mehrere ganzzahlige Typen, **Integertypen** genannt, die sich im Speicherplatzbedarf und damit im Wertebereich unterscheiden.

Beispiel Integertypen in Component Pascal:

<i>Typ:</i>	BYTE
<i>Format:</i>	8 Bit
<i>Wertebereich:</i>	MIN (BYTE) .. MAX (BYTE) -128 .. 127
<i>Typ:</i>	SHORTINT
<i>Format:</i>	16 Bit
<i>Wertebereich:</i>	MIN (INTEGER) .. MAX (INTEGER) -32768 .. 32767
<i>Typ:</i>	INTEGER
<i>Format:</i>	32 Bit
<i>Wertebereich:</i>	MIN (INTEGER) .. MAX (INTEGER) -2147483648 .. 2147483647
<i>Typ:</i>	LONGINT
<i>Format:</i>	64 Bit
<i>Wertebereich:</i>	MIN (LONGINT) .. MAX (LONGINT) -9223372036854775808 .. 9223372036854775807

Operationen: + - * **DIV MOD** = # < > <= >=

<i>Vordeclarierte Funktionen:</i>	<i>Bedeutung:</i>
ASH (x, n : Integertyp) : INTEGER	Arithmetische Verschiebung, Ergebnis: $x \cdot 2^n$.
BITS (x : INTEGER) : SET	Interpretation als Menge, Ergebnis: $\{i \mid \text{ODD}(x \text{ DIV } 2^i)\}$.
ODD (x : Integertyp) : BOOLEAN	Test auf Ungerade, Ergebnis: $x \text{ MOD } 2 = 1$.

LONG (x : BYTE) : SHORTINT	Identischer Wert, Einbettung.
LONG (x : SHORTINT) : INTEGER	
LONG (x : INTEGER) : LONGINT	
SHORT (x : LONGINT) : INTEGER	Identischer Wert, Einschränkung.
SHORT (x : INTEGER) : SHORTINT	
SHORT (x : SHORTINT) : BYTE	
<i>Vordeclarierte Prozeduren:</i>	<i>Bedeutung:</i>
DEC (VAR x : Integertyp)	Dekrementierung, $x := x - 1$.
DEC (VAR x, n : Integertyp)	Dekrementierung, $x := x - n$.
INC (VAR x : Integertyp)	Inkrementierung, $x := x + 1$.
INC (VAR x, n : Integertyp)	Inkrementierung, $x := x + n$.

8.6.2.2 Reelle Zahlen und Gleitpunktzahlen

Reelle Zahlen sind in der Mathematik und vielen naturwissenschaftlich-technischen Gebieten außerordentlich wichtig. Rechensysteme und Programmiersprachen bieten dafür ersatzweise **Gleitpunktzahlen** (*floating point number*). Meist heißen sie - wie in Component Pascal - REAL, manchmal float.

Component Pascal bietet zwei reelle Typen, **Realtypen** genannt, die sich im Speicherplatzbedarf, im Wertebereich und in der Genauigkeit unterscheiden. Sie sind beide nach dem *IEEE P754 Standard-Format* für Gleitpunktzahlen implementiert.

Beispiel Realtypen in Component Pascal:

Typ:	SHORTREAL
Format:	IEEE P754, 32 Bit, 23 Bit Mantisse, 8 Bit Exponent
Wertebereich:	MIN (REAL) .. MAX (REAL)
kleinste positive Zahl:	1.2E-38
größte positive Zahl:	3.4E38
Genauigkeit:	7-8 Dezimalstellen
Typ:	REAL
Format:	IEEE P754, 64 Bit, 52 Bit Mantisse, 11 Bit Exponent
Wertebereich:	MIN (LONGREAL) .. MAX (LONGREAL)
kleinste positive Zahl:	2.2E-308
größte positive Zahl:	1.8E308
Genauigkeit:	15-16 Dezimalstellen
Operationen:	+ - * / = # < > <= >=

<i>Vordeclarierte Funktionen:</i>	<i>Bedeutung:</i>
ENTIER (x : Realtyp) : LONGINT	Größte Ganzzahl kleiner oder gleich x.
LONG (x : SHORTREAL) : REAL	Identischer Wert, Einbettung.
SHORT (x : REAL) : SHORTREAL	Identischer Wert, Einschränkung (mit Genauigkeitsverlust).

Bei numerischen Berechnungen mit Gleitpunktzahlen ist man mit einer Reihe von Problemen konfrontiert. Der Grund liegt darin, daß Gleitpunktzahlen als Ersatz für reelle Zahlen inhärent ungenau sind und daher die Gleitpunktzahlenarithmetik anderen Gesetzen folgt als die mathematisch exakte Arithmetik der reellen Zahlen. Denn maschinenintern können nur endlich viele Gleitpunktzahlen dargestellt werden. Die Menge dieser Gleitpunktzahlen ist eine Teilmenge der rationalen Zahlen. Gleitpunktzahlen liegen längst nicht so dicht beieinander wie rationale Zahlen. Dadurch entstehen Darstellungsfehler bei der Eingabe und Rundungsfehler beim Rechnen. Diese Fehler können sich im Verlauf einer längeren Berechnung so fortpflanzen, daß das Endergebnis einen völlig falschen Wert liefert.

Da ein gewisses Verständnis dieser Problematik Voraussetzung für einen guten Programmierstil ist, untersuchen wir die Gleitpunktzahlenarithmetik etwas genauer. Fassen wir zunächst die wichtigsten Unterschiede der beiden Zahlenarten zusammen

<i>Theorie</i>		<i>Praxis</i>
<i>Mathematik</i>		<i>Rechner, Programmierung</i>
reelle Zahlen		Gleitpunktzahlen
	Anzahl	
unendliche viele		endlich viele
	Abstand zwischen zwei benachbarten Zahlen	
beliebig klein		durch Rechnergenauigkeit vorgegeben
	Eingabegrößen, Ergebnisse	
exakte Werte		ungenau, Näherungswerte
	Berechnungen	
exakte Arithmetik		approximative Arithmetik

Im folgenden bezeichnen

N	die Menge der natürlichen Zahlen 1, 2, 3,...;
N₀	die Menge der nichtnegativen ganzen Zahlen 0, 1, 2,...;
Z	die Menge der ganzen Zahlen 0, 1, -1, 2, -2,...;
R	die Menge der reellen Zahlen.

8.6.2.2.1 Darstellungen von Gleitpunktzahlen

Bekanntermaßen sind reelle Zahlen als **Dezimalbrüche** darstellbar.

Beispiele:

123,456

7,296493451394732445...

$0,000834602 \cdot 10^{-5}$

$0,834602 \cdot 10^{-8}$

Beispiel in Component-Pascal-Notation:

8.34602E-9

Kommt in der Darstellung eine 10er-Potenz als Faktor vor, so handelt es sich um eine **halblogarithmische** oder **Exponentialdarstellung**. Zu einer Zahl gibt es unendlich viele solcher Darstellungen. Gesucht ist eine eindeutige, „normalisierte“ Darstellung, die als maschineninterne Darstellung geeignet ist. Eindeutigkeit erreichen wir, indem wir fordern, daß vor dem Komma eine 0 steht, die erste Ziffer nach dem Komma keine 0 ist und nach dem Komma ab jeder Stelle nicht nur 9en auftreten. (Beachte $0,999... = 1!$)

Jede reelle Zahl $x \in \mathbf{R}$, $x \neq 0$ besitzt eine eindeutige **normalisierte Dezimalbruchdarstellung** der Form

$$x = \pm 0,z_1z_2z_3... * 10^e$$

mit $z_1 \neq 0$, $z_k \in \{0,..., 9\}$, $e \in \mathbf{Z}$, wobei es zu jedem Ziffernindex i einen Index $k > i$ gibt mit $z_k \neq 9$. Die Ziffernfolge $z_1z_2z_3...$ nach dem Komma heißt **Mantisse**, 10 ist die **Basis** und e der **Exponent**. Einen Dezimalbruch **normalisieren** bedeutet, ihn in die normalisierte Darstellung bringen. Dazu muß ggf. die Mantisse bzw. das Komma verschoben und der Exponent entsprechend angepaßt werden.

Die Darstellung ist leicht für andere Basen verallgemeinerbar. Die Mantisse ist bei reellen Zahlen eine unendliche Ziffernfolge. Beschränken wir die Länge der Mantisse, so erhalten wir Gleitpunktzahlen.

Es seien $B \in \mathbf{N}$, $B \geq 2$, $m \in \mathbf{N}$, $e_1, e_2 \in \mathbf{N}_0$. Eine **B-adische und m-stellige Gleitpunktzahl** hat die Form $x \neq 0$ oder

$$x = \pm \sum_{k=1}^m z_k \times B^{e-k}$$

mit $z_1 \neq 0$, $z_k \in \{0, 1,..., B-1\}$, $e \in \mathbf{Z}$, $-e_1 \leq e \leq e_2$. Es heißen die

z_k	Ziffern,
B	Basis,
m	Mantissenlänge oder Stellenzahl,

$\sum_{k=1}^m z_k \times B^k$	Mantisse,
-------------------------------	------------------

$[-e_1, e_2] \cap \mathbf{Z}$	Exponentenbereich,
e	Exponent,
$-e_1$	kleinster Exponent,
e_2	größter Exponent.

Als eindeutige Darstellung der 0 legen wir

$$0 = + 0,0...0 * B^{-e_1}$$

fest.

Je nachdem, ob die Basis $B = 2, 8, 10$ oder 16 ist, sprechen wir von **dualen**, **oktalen**, **dezimalen** bzw. **hexadezimalen** Gleitpunktzahlen. Statt des Kommas schreibt man oft einen Punkt.

Beispiel einer dezimalen Gleitpunktzahl mit Mantissenlänge 5:

$$0.37249 * 10^4$$

Dieselbe Zahl in Component-Pascal-Notation:

$$0.37249E4$$

Der **Zahlbereich** eines Gleitpunktzahlentyps ist das Intervall vom kleinstmöglichen bis zum größtmöglichen Wert; er ist im wesentlichen durch den Exponentenbereich bestimmt. Die **Genauigkeit** einer Gleitpunktzahl ist durch die Mantissenlänge festgelegt.

Die Darstellung in Rechnern ist heutzutage meist dual. Die Stellenzahl für den Exponenten bewegt sich etwa im Bereich von 6 bis 15 Bit, die Mantissenlänge etwa von 9 bis 112 Bit. Manche Prozessoren bieten auch dezimale Gleitpunktzahlen.

Die maschineninterne Darstellung kann noch optimiert werden:

- Die Stelle vor dem Komma kann weggelassen werden, sie hat ja immer den Wert 0.
- Das Komma bzw. der Punkt kann weggelassen werden.
- Die Basis kann weggelassen werden.
- Bei dualen Gleitpunktzahlen kann die erste Stelle nach dem Komma weggelassen werden, sie muß ja stets - außer bei der Zahl 0 - gleich 1 sein.

Beispiele:

Wir haben 32 Bit für eine duale Gleitpunktzahl zur Verfügung. Wir wählen $e_1 = e_2$, nehmen 7 Bit für den Exponenten und dazu ein Vorzeichenbit. Dann bleiben 23 Bit für die Mantisse und ein Vorzeichenbit.

Damit ergibt sich ein Zahlbereich etwa von 10^{-38} bis 10^{38} und eine Genauigkeit von etwa 7 Dezimalstellen.

Mit dem vierfachen Speicherplatz und 112 Bit Mantissenlänge ist eine Genauigkeit von ca. 33 Dezimalstellen erreichbar.

In der Praxis (auch in Component Pascal) wird für die maschineninterne Darstellung oft das durch den *IEEE P754 Standard* festgelegte Format verwendet. Es unterscheidet sich in Details von unserer Darstellung und legt genau fest, wo welche Bits innerhalb eines Maschinenworts zu liegen haben. Außerdem liegt der Exponent in **Exzeßdarstellung** vor. Das bedeutet, daß nicht der vorzeichenbehaftete Exponent e , sondern die stets nicht-negative **Charakteristik** $c = e + e_1$ gespeichert wird. Zum IEEE-Format gehört auch der besondere Wert `inf` oder `INF` für „unendlich“ (*infinite*). Er kann beim Multiplizieren gigantischer Zahlen oder beim Dividieren durch 0 auftreten. Näheres siehe [BSSS92], S. 69.

8.6.2.2 Maschinendarstellung reeller Zahlen

Nicht jede reelle Zahl kann exakt im Rechner dargestellt werden. Eine Rundungsfunktion $rd : \mathbf{R} \rightarrow \mathbf{R}$ ordnet jedem $x \in \mathbf{R}$ eine Gleitpunktzahl $y = rd(x)$ zu, die **Gleitpunktdarstellung** von x . Dabei können verschiedene Fälle auftreten.

Die 0 wird exakt abgebildet: $rd(0) := 0$.

Ist $x \neq 0$ mit der Darstellung

$$x = \pm 0, z_1 z_2 \dots z_m z_{m+1} \dots * B^e, z_1 \neq 0,$$

so definiert man den **relativen Darstellungsfehler**

$$\delta_x := (rd(x) - x) / x.$$

Im Fall $e < -e_1$ liegt **Exponentenunterlauf** vor. Vernünftigerweise wird $rd(x) := 0$ gesetzt. Damit ist der Betrag des relativen Darstellungsfehlers $|\delta_x| = 1$, d.h. 100%.

Im Fall $e > e_2$ liegt **Exponentenüberlauf** vor. Dann ist eine Zuordnung $rd(x)$ sinnlos, der Rechner sollte die Berechnung mit einer Fehlermeldung abbrechen.

Im Normalfall $-e_1 \leq e \leq e_2$ kann gerundet werden. Zwei Lösungen liegen auf der Hand.

Bei **unsymmetrischer Rundung (Abbruch, Abschneiden)** werden nur die ersten m Stellen der Mantisse genommen, der Rest wird abgeschnitten:

$$rd(x) := \pm 0, z_1 z_2 \dots z_m * B^e.$$

Für den relativen Darstellungsfehler ergibt sich die Abschätzung

$$|\delta_x| < B^{e-m} / |x|.$$

Bei **symmetrischer Rundung** wird je nach Wert der $(m+1)$ -ten Stelle ab- oder aufgerundet:

$$rd(x) := \pm 0, z_1 z_2 \dots z_m * B^e, \quad \text{falls } z_{m+1} \leq B / 2;$$

$$rd(x) := \pm (0, z_1 z_2 \dots z_m + B^{-m}) * B^e, \quad \text{falls } z_{m+1} > B / 2.$$

Beachte, daß im letzten Fall durch das Aufrunden ein Übertrag entstehen kann und dann u.U. renormalisiert werden muß. Für den relativen Darstellungsfehler ergibt sich

$$|\delta_x| < B^{e-m} / 2 |x|.$$

Der relative Darstellungsfehler ist nach oben durch die vom Rechner R abhängige **relative Rechengenauigkeit** δ_R beschränkt. Diese Zahl ist definiert durch

$$\delta_R := B^{1-m} \quad \text{bei unsymmetrischer Rundung und}$$

$$\delta_R := B^{1-m} / 2 \quad \text{bei symmetrischer Rundung.}$$

Beispiel:

Bei $B = 2$, $m = 23$ und symmetrischer Rundung ergibt sich $\delta_R = 1/2^{23} \sim 0,125 * 10^{-6}$.

In einem Rechner kann (zwecks Vereinfachung der Schaltkreislogik) auch eine andere Rundungsfunktion realisiert sein. Man sollte jedoch erwarten dürfen, daß die relative Rechengenauigkeit nicht schlechter als bei unsymmetrischer Rundung ist.

8.6.2.2.3 Fehler bei Gleitpunktzahlen

Mit den Darstellungsfehlern haben wir eine spezielle Fehlerart kennengelernt. Klassifizieren wir nun die Fehler, die beim Rechnen mit Gleitpunktzahlen auftreten.

Genauigkeitsfehler entstehen

- als *Eingabefehler* schon bei den Eingabedaten und
- als *Rundungsfehler* bei arithmetischen Operationen.

Eingabefehler entstehen

- als *Darstellungsfehler* wegen der maschineninternen Beschränkung der Stellenzahl,
- als *Konvertierungsfehler* beim Übergang zwischen Zahlssystemen (dezimal $\leftrightarrow$ dual).

Sie bewirken selbst dann, wenn mit den maschineninternen Daten arithmetisch exakt weitergerechnet wird, *unvermeidbare Fehler* bei den Ausgabedaten, da sie den Gesetzen der *Fehlerfortpflanzung* unterliegen. Hinzu kommen dann die Rundungsfehler durch die Gleitpunktarithmetik, auf die wir im nächsten Abschnitt eingehen. Auch diese verursachen *Fortpflanzungsfehler*. Im schlechten Fall verstärken sich diese Fehler und verfälschen die Ergebnisse.

Das Problem der Fehlerfortpflanzung wird in der Numerischen Mathematik eingehend analysiert. Uns geht es hier darum, problembewußt programmieren zu lernen.

8.6.2.2.4 Gleitpunktzahlenarithmetik

Mathematische Notation:

Es seien

x, y	Gleitpunktzahlen,
$\bullet \in \{+, -, *, /\}$	eine arithmetische Operation,
$x \bullet y$	das exakte Ergebnis der Rechnung,
$z = gl(x \bullet y)$	das Gleitpunktergebnis , das ist die Gleitpunktdarstellung des exakten Ergebnisses, sie ist mit einem Rundungsfehler behaftet.

Component-Pascal-Notation:

```
VAR x, y, z : REAL;
z := x + y;
```

Es ist sinnvoll,

$$gl(x \bullet y) = rd(x \bullet y)$$

zu fordern. Man kann aber auch auf die Forderung $gl = rd$ verzichten und stattdessen die Gültigkeit des **allgemeinen Rundungsgesetzes**

$$|rd(x) - x| \leq |x| * \delta_R$$

$$|gl(x \bullet y) - (x \bullet y)| \leq |x \bullet y| * \delta_R$$

für $x, y \in \mathbf{R}$ mit einer durch den Rechner R festgelegten Rechnergenauigkeit δ_R fordern. Die Details der implementierten Rechnerarithmetik sind dann unwesentlich.

Prinzip einer Rechnung

1. x und y als Dezimalbrüche betrachten.
2. Exakte Rechnung durchführen. Das exakte Ergebnis $x \bullet y$ ist wieder ein Dezimalbruch.
3. Das Ergebnis $x \bullet y$ normalisieren.

4. Das normalisierte Ergebnis runden, so daß es auf eine Gleitpunktzahl $gl(x \bullet y)$ abgebildet wird.

Bei den folgenden Beispielen gilt $B = 10$, $m = 3$, und es wird unsymmetrisch gerundet (abgeschnitten).

Addition

$$x = 0.972E4$$

$$y = 0.861E3$$

$$z = gl(x + y)$$

1. Die betragsmäßig kleinere Zahl an die andere anpassen:

- ◆ Den Exponenten erhöhen, bis beide Exponenten gleich sind.

- ◆ Die Mantisse entsprechend verschieben.

$$x = 0.972E4 = 0.9720E4$$

$$y = 0.861E3 = 0.0861E4$$

2. Die Mantissen der beiden Zahlen addieren.

$$x + y = 1.0581E4$$

3. Normalisieren.

$$= 0.10581E5$$

4. Runden.

$$z = gl(0.10581E5) = 0.105E5$$

Subtraktion

Analog.

Multiplikation

$$x = 0.279E4$$

$$y = 0.168E3$$

$$z = gl(x * y)$$

1. Die Mantissen der beiden Zahlen multiplizieren.

$$0.279 * 0.168 = 0.046872$$

2. Die Exponenten der beiden Zahlen addieren.

$$4 + 3 = 7$$

3. Exaktes Ergebnis:

$$x * y = 0.279 * 10^4 * 0.168 * 10^3$$

$$= 0.279 * 0.168 * 10^4 * 10^3$$

$$= 0.046872 * 10^7 = 0.046872E7$$

4. Normalisieren.

$$= 0.46872\text{E}6$$

5. Runden.

$$z = \text{gl}(0.46872\text{E}6) = 0.468\text{E}6$$

Division

Analog.

8.6.3 Logische Größen und Ausdrücke

Logische Größen oder Aussagen können einen der beiden Wahrheitswerte „falsch“ oder „wahr“ annehmen. Man braucht sie vor allem zur Formulierung von Bedingungen. Viele Programmiersprachen bieten dafür einen Datentyp an, meist heißt er **BOOLEAN** nach Boole, dem Begründer der algebraischen Behandlung der Aussagenlogik. Die Wahrheitswerte heißen **FALSE** und **TRUE**. Man spricht auch von *Boole'schen Daten* oder *Logischen Daten*.

Die Boole'sche Algebra, Operationen wie Negation, Konjunktion, Disjunktion und Implikation sowie Wahrheitstabellen werden in der Vorlesung „Digitaltechnik“ behandelt.

Beispiel in Component Pascal:

Typ:	BOOLEAN
Format:	8 Bit
Wertebereich:	MIN (BOOLEAN) .. MAX (BOOLEAN) FALSE TRUE
Operationen:	~ OR & = #

8.7 Strukturierte Datentypen

Eine **Datenstruktur** (*data structure*) ist eine Zusammenfassung von Daten zu einer höheren Einheit. Ein **strukturierter Datentyp** (*structured data type*) ist eine Abstraktionsklasse von Datenstrukturen.

Strukturierte Datentypen behandeln wir in nachfolgenden Kapiteln. Hier soll eine kurze Vorausschau genügen.

8.7.1 Reihungen

In der Mathematik arbeitet man oft mit Vektoren und Matrizen. Das sind strukturierte Gebilde, die sich aus einer endlichen Folge gleichartiger Elemente zusammensetzen. Sie finden ihre Entsprechung in Programmiersprachen in **Reihungen** (*Feld; array, row*).

Beispiel in Component Pascal:

Typkonstrukt:	ARRAY ConstExpr OF Type
Beispiel:	ARRAY 10 OF REAL

8.7.2 Zeichenketten

Textverarbeitung bedeutet Verarbeitung von **Zeichenketten** (*string*). Zeichenketten können als Spezialfall von Reihungen betrachtet werden, der Elementtyp ist Zeichen. Da es zwei Zeichentypen gibt, erhält man auch zwei Zeichenkettentypen.

Beispiele in Component Pascal:

Beispiele für	ARRAY strLng OF CHAR
Zeichenkette:	ARRAY OF CHAR
Beispiele für	ARRAY strLng OF SHORTCHAR
Kurzzeichenkette:	ARRAY OF SHORTCHAR

8.7.3 Verbunde

Oft benötigt man strukturierte Daten, bei denen die Komponenten verschiedenartig sind. Für diesen Zweck bieten Programmiersprachen **Verbunde** (*record, struct*).

Beispiel:

Personaldaten =	(Name, Adresse, Geburtsdaten);
Name =	(Nachname, Vornamen);
Adresse =	(Straße, Hausnummer, Postleitzahl, Wohnort);
Geburtsdaten =	(Geburtsdatum, Geburtsort),
usw.	

Beispiel in Component Pascal:

Typkonstrukt:	RECORD FieldList END
Beispiel:	RECORD
	name : Name;
	address : Address;
	birthdate : Date;
	END

8.7.4 Mengen

Mengen sind in der Mathematik von grundlegender Bedeutung. Es ist angenehm, daß manche Programmiersprachen einen Datentyp **Menge** (*set*) anbieten (auch wenn die Anzahl der Elemente meist maschinenbedingt recht klein ist).

Endliche Mengen natürlicher Zahlen kann man durch Bitfolgen darstellen. Ist das Bit mit der Nummer *i* gleich 1, so gehört *i* zur Menge, sonst nicht. Mit dieser Implementierung ist ein Mengentyp eigentlich kein strukturierter, sondern ein einfacher Typ.

Beispiel in Component Pascal:

Typ:	SET
Format:	32 Bit
Wertebereich:	MIN (SET) .. MAX (SET)
	Teilmenge der Menge {0,...,31}
Operationen:	+ - * / = # IN

<i>Vordeclarierte Prozeduren:</i>	<i>Bedeutung:</i>
EXCL (VAR v : SET ; x : Integertyp)	Element entfernen, $v := v - \{x\}$.
INCL (VAR v : SET ; x : Integertyp)	Element hinzufügen, $v := v + \{x\}$.

8.7.5 Zeiger

Für komplexere Datenstrukturen braucht man Exemplare, deren Wert ein Bezug auf ein anderes Exemplar ist. Der entsprechende Datentyp heißt **Zeiger** (*Referenz; pointer, reference*). Wir behandeln ihn später.

8.7.6 Dateien

Eine **Datei** (*file*) besteht wie eine Reihung oder eine Zeichenkette aus einer Aneinanderreihung von gleichartigen Datenelementen. Der Unterschied ist, daß eine Datei i.a. etwas Großes ist, ihre Länge ist sozusagen unbeschränkt. Dateien existieren i.a. auch nicht nur während eines Programmablaufs, sondern sie dienen vor allem der längerfristigen Aufbewahrung von Daten. Meist werden Dateien auf externen Datenträgern (Disketten, Platten, Bänder, CD-ROMs) gespeichert. Die Datenelemente, aus denen eine Datei aufgebaut ist, sind im einfachsten Fall Bytes, es können aber auch komplexe Verbunde sein.

8.7.7 Datenbanken

Eine **Datenbank** (*data base*) ist ein großer Datenbestand, bei dem die Datenelemente in vielfältigen Beziehungen zueinander stehen und miteinander verknüpft sein können. Datenbanken sind i.a. mittels mehrerer - vieler - Dateien realisiert.

Zu diesem Thema gibt es eine eigene Vorlesung im Hauptstudium.

Kapitel 9

Vereinbarungen

Synonym: **Deklaration** (*declaration*).

9.1 Arten von Vereinbarungen

- Konstantenvereinbarungen,
- Typvereinbarungen,
- Variablenvereinbarungen,
- Prozedurvereinbarungen.

9.1.1 Syntax

DeclSeq =

```
{ CONST { ConstDecl „;“ } |  
TYPE { TypeDecl „;“ } |  
VAR { VarDecl „;“ } }  
{ ProcDecl „;“ | ForwardDecl „;“ }.
```

9.2 Konstantenvereinbarungen

9.2.1 Beispiele

CONST

```
phoneNbr = 07121;  
epsilon  = 1E-10;  
title    = "Informatik";  
  
department* = "Elektronik";  
  
maxArea = maxWidth * maxDepth;  
halfSet =  
    {MIN(SET) .. MAX(SET) DIV 2};
```

9.2.2 Syntax

ConstDecl = *IdentDef* „=“ *ConstExpr*.

IdentDef = *ident* [„*“ | „-“].

ConstExpr = *Expr*.

Expr = ...

9.2.3 Semantik

- Eine **Konstantenvereinbarung** definiert eine *Übersetzungszeitkonstante*.
- Der **Konstantenname** wird mit einem einfachen *Wert* gleichgesetzt.
- Der Wert muß *zur Übersetzungszeit berechenbar* sein,
- deshalb muß auf der rechten Seite der Deklaration ein *konstanter Ausdruck* stehen.
- Der Wert ist bei der Definition *nicht unbedingt typbehaftet*,
- sondern wird im Benutzungskontext an einen Typ angepaßt.
- Der Wert ist zur Laufzeit *nicht veränderbar*.
- Einem Konstantennamen ist i.a. *kein Datenexemplar* bzw. keine Speicherzelle zugeordnet.
- Konstanten können in Ausdrücken auftreten.

9.2.4 Wozu und wie sind Konstanten zu benutzen?

Programmierrichtlinien

- Definiere für jeden mehrfach vorkommenden Wert einen Konstantennamen!

Dadurch wird das Programm leichter änderbar.

- Verwende Konstanten anstelle von Variablen, deren Wert nicht verändert wird!

Das Programm wird stabiler & effizienter.

- Drücke die Unabhängigkeit bzw. Abhängigkeit von Werten in Konstantendefinitionen aus!

Das Programm wird änderungsstabiler.

9.3 Typvereinbarungen

9.3.1 Beispiele

TYPE

```
Voltage = REAL;
```

```
Vector  = ARRAY 10 OF REAL;
```

```
Circle* =
```

```
    RECORD
```

```
        centre : Point;
```

```
        radius  : REAL;
```

```
    END;
```

```
MyModel = TextModels.Model;
```

9.3.2 Syntax

TypeDecl = *IdentDef* „=“ *Type*.

Type =

Qualident

| **ARRAY** [*ConstExpr* { „,“ *ConstExpr* }]
OF *Type*

| **RECORD** [„(“ *Qualident* „)“]
FieldList { „;“ *FieldList* } **END**

| **POINTER TO** *Type*

| **PROCEDURE** [*FormalPars*].

Qualident = [*ident* „.“] *ident*.

9.3.3 Semantik

- Eine **Typvereinbarung** führt einen neuen **Typnamen** ein (linke Seite).
- Die **Typangabe** ist ein bereits definierter Datentyp oder führt einen neuen Datentyp ein (rechte Seite).
- Ein *Datentyp* legt
 - den Wertebereich,
 - die Operationen,
 - die Speicherstrukturvon Exemplaren dieses Typs fest.
- Typen treten auf der rechten Seite von Variablenvereinbarungen, in Formalparameterlisten & in Feldvereinbarungen von Verbunden auf.
- Näheres zu definierbaren Typen später.

9.4 Variablenvereinbarungen

9.4.1 Beispiele

VAR

```
number    : INTEGER;
isPrime   :
    ARRAY maxNbr OF BOOLEAN;
title,
author    : ARRAY 64 OF CHAR;
v         : Voltage;
x, y, z   : Vector;
circle-   : Circle;
m*        : TextModels.Model;
```

9.4.2 Syntax

VarDecl = *IdentList* „:“ *Type*.

IdentList = *IdentDef* { „,“ *IdentDef* }.

9.4.3 Semantik

- Eine **Variablenvereinbarung** führt einen **Namen** für eine neue **Variable** ein, d.h.
- dem Namen wird ein *Datenexemplar* & damit Speicherplatz *zugeordnet*.
- Jedes Datenexemplar ist von einem *unveränderlichen statischen Typ*, der bei der Vereinbarung festgelegt wird, es ist ein *Exemplar* dieses Typs,
- deshalb steht nach dem : eine Typangabe.
- Der Typ einer Variablen bestimmt den Wertebereich, die zulässigen Operationen & die Speicherstruktur.
- Der Wert einer Variablen ist nach der Vereinbarung i.a. *unbestimmt*, d.h.
- das Datenexemplar wird vor Ausführung des Anweisungsteils i.a. *nicht initialisiert*.
- Der Wert ist zur Laufzeit *veränderbar*.
- Variablen können in Ausdrücken, auf der linken Seite von Zuweisungen & als Aktualparameter auftreten.

9.4.4 Initialisierung von Variablen

Initialisieren :=

mit Anfangswerten versehen.

Implizite Initialisierung :=

Daten werden bei Vereinbarung mit Anfangswerten versehen (automatisch durch Übersetzer).

Explizite Initialisierung :=

Daten werden durch Anweisungen mit Anfangswerten versehen.

Semantik in Component Pascal

- Modulglobale & dynamische Variablen sowie Variablen von Zeiger- & Prozedurtypen werden ***implizit*** initialisiert.
- Prozedurlokale Variablen werden ***nicht implizit*** initialisiert, ausgenommen die von Zeiger- & Prozedurtypen.
- Diese Variablen müssen also ***explizit*** initialisiert werden!

Problem

- Fehlende oder falsche Initialisierung ist einer der häufigsten Programmierfehler!

Programmierleitlinie

- Initialisiere alle Variablen explizit, am besten am Beginn des Anweisungsteils!

Das verringert die Fehlerrate & man sieht, was das Programm tut.

9.5 Prozedurvereinbarungen

9.5.1 Beispiele

```
PROCEDURE Do*;
```

```
END Do;
```

```
PROCEDURE WriteChars;
```

```
    VAR i : INTEGER;
```

```
BEGIN
```

```
    FOR i := 0 TO 255 DO
```

```
        Out.Char (CHR (i));
```

```
    END;
```

```
END WriteChars;
```

```
PROCEDURE
```

```
    Abs (x, y : REAL) : REAL;
```

```
BEGIN
```

```
    RETURN
```

```
        Math.Sqrt (x * x + y * y);
```

```
END Abs;
```

■ Näheres zu Prozeduren später.

9.6 Namengebung

- BlackBox ist gemäß bestimmter Programmier-richtlinien geschrieben; siehe Online-Hilfe *Programming Conventions*.
- Es ist vernünftig, sich als AnwenderIn auch an diese Richtlinien zu halten.

Programmierleitlinien

- Kleinbuchstaben: für Normalfall, Großbuchstaben: für Hervorhebungen.
- Modul-, Typ- und Prozedurnamen beginnen mit einem Großbuchstaben.
- Konstanten- und Variablennamen beginnen mit einem Kleinbuchstaben.
- Großbuchstaben markieren den Beginn eines neuen Teilworts in einem Namen.
- Ein Variablenname kann bis auf den Anfangsbuchstaben gleich dem Typnamen sein.
- *Konstanten, Typen, Variablen, Funktionen* bezeichnen meist Dinge → dafür sind *Substantive* (evtl. ergänzt) angebracht.
- (Reine) *Prozeduren* bezeichnen Aktionen → dafür sind *Verben* (evtl. ergänzt) angebracht.

9.7 Reihenfolge von Vereinbarungen

- Bei

CONST

TYPE

VAR

ist beliebige Reihenfolge syntaktisch möglich.

Programmierleitlinie

- Halte bei Vereinbarungen die Reihenfolge

CONST

TYPE

VAR

PROCEDURE

möglichst ein!

Einheitliche Ordnung erhöht die Lesbarkeit.

Kapitel 10

Ausdrücke

Ein **Ausdruck** (*expression*) ist eine Vorschrift zur Berechnung eines Werts, ein nach gewissen Regeln strukturiertes Gebilde, das nach gewissen Regeln abgearbeitet, d.h. *ausgewertet* wird. Ein Ausdruck setzt sich zusammen aus

- Operanden,
- Operatoren und
- ggf. Klammern.

Ein **Operand** ist ein *passives* Teil eines Ausdrucks, das einen Wert liefert. Ein Operand kann eine Konstante, eine Variable, ein Funktionsaufruf oder wiederum ein aus anderen Teilen zusammengesetzter Ausdruck sein.

Ein **Operator** ist ein *aktives* Teil, das seine Operanden miteinander verknüpft und aus ihren Werten einen neuen Wert bildet.

Klammern (*parantheses*) sind *ordnende* Teile, die die Reihenfolge der Anwendung der Operatoren bestimmen.

Eine **Operation** ist die Anwendung eines Operators auf seine Operanden. Die **Auswertung** (*evaluation*) eines Ausdrucks ist damit eine Folge von Operationen.

Die Operanden und der Wert eines Ausdrucks sind jeweils von einem bestimmten Typ.¹

Beispiel in Component Pascal:

Ausdruck:

```
((a + 3) * 4 DIV b < c.d[e]) & (f(x) >= Math.Exp(5.67))
```

Operanden:

a, 3, 4, b, c.d[e], f(x), Math.Exp(5.67), a + 3, (a + 3) * 4,
(a + 3) * 4 **DIV** b, (a + 3) * 4 **DIV** b < c.d[e],
f(x) >= Math.Exp(5.67).

Operatoren:

+, *, **DIV**, <, &, >=.

Klammern:

(,).

Ausdrücke kommen vor

- auf der rechten Seite von Zuweisungen,

¹ Eine Ausnahme bilden konstante Ausdrücke, die nicht typbehaftet sein müssen.

- als aktuelle Parameter von Prozeduraufrufen,
- als Bedingungen und Bereichsgrenzen in Auswahl- und Wiederholungsanweisungen,
- in Rückgabeanweisungen von Funktionen,
- bei der Bildung von Mengen und
- als Indizes von Reihungen.

Ein Spezialfall sind **konstante Ausdrücke**, bei diesen sind alle Operanden konstant. Konstante Ausdrücke treten auf

- in Konstantenvereinbarungen,
- als Indexanzahlen in Typangaben von Reihungen,
- als Fälle in mehrfachen Auswahlanweisungen.

Die Auflistung verdeutlicht, daß Ausdrücke wichtige Bausteine von Programmen sind. Ihr Verständnis ist von grundlegender Bedeutung für das Programmieren.

Im Zusammenhang mit Ausdrücken und Operatoren sind folgende Fragen zu klären bzw. durch Regeln einer Programmiersprache zu beantworten.

- Gegeben ist ein bestimmter Typ. Welche Operatoren sind auf Exemplare dieses Typs anwendbar?
- Gegeben ist ein bestimmter Operator. Welche Operanden benötigt er? Von welchem Typ dürfen seine Operanden sein? Welches Ergebnis liefert er? Von welchem Typ ist das Ergebnis?
- Wie ist die Reihenfolge der Berechnung, d.h. in welcher Reihenfolge werden die Operationen ausgeführt? Welche Vorrangregeln sind für die Operatoren festgelegt?

10.1 Syntax

Expr = *SimpleExpr* [*Relation SimpleExpr*].

SimpleExpr = [„+“ | „-“] *Term* { *AddOp Term* }.

Term = *Factor* { *MulOp Factor* }.

Factor =

Designator [„(“ [*ExprList*] „)”] | *number* | *character* | *string* | **NIL** | *Set* | „(“ *Expr* „)” | „~“ *Factor*.

Set = „{“ [*Element* { „,“ *Element* }] „}“.

Element = *Expr* [„.“ *Expr*].

Relation = „=“ | „#“ | „<“ | „<=“ | „>“ | „>=“ | **IN** | **IS**.

AddOp = „+“ | „-“ | **OR**.

MulOp = „*“ | „/“ | **DIV** | **MOD** | „&“.

Designator = *Qualident* { „.“ *ident* | „[“ *ExprList* „]“ | „^“ | „(“ *Qualident* „)” }.

ExprList = *Expr* { „,“ *Expr* }.

10.2 Operatoren

10.2.1 Arten

Die Operatoren in Component Pascal lassen sich inhaltlich wie folgt gliedern.

■ **Arithmetische Operatoren:** +, -, *, /, **DIV**, **MOD**.

■ **Relationen:** =, #, <, <=, >, >=, **IN**, **IS**.

■ **Boole'sche Operatoren:** ~, **OR**, &.

■ **Mengenoperatoren:** +, -, *, /.

■ **Zeichenkettenoperatoren:** +.

Eine andere Einteilung ist die nach Anzahl der Operanden.

■ **Einstellige Operatoren:** +, -, ~.

■ **Zweistellige Operatoren:** +, -, *, /, **DIV**, **MOD**, =, #, <, <=, >, >=, **IN**, **IS**, **OR**, &.

10.2.2 Vorrangregeln

Component Pascal unterscheidet vier **Prioritäten** für Operatoren. Von hoher zu niedriger Priorität sind dies:

1. **Negationsoperator:** ~.
2. **Multiplikationsoperatoren:** *, /, **DIV**, **MOD**, &.
3. **Additionsoperatoren:** +, -, **OR**.
4. **Relationen:** =, #, <, <=, >, >=, **IN**, **IS**.

Es gilt also auch die aus der Arithmetik bekannte Regel *Punkt vor Strich*.

Bei Operatoren gleicher Priorität wird *von links nach rechts* ausgewertet.

Beispiel: $x / y / z$ wird ausgewertet als $(x / y) / z$.

Die Auswertungsreihenfolge kann durch Klammerung beeinflusst werden. Es wird wie in der Mathematik üblich *von innen nach außen* ausgewertet.

Ist ein Ausdruck mathematisch mehrdeutig, so ist es guter Programmierstil zu klammern, um die Lesbarkeit zu erhöhen, auch wenn die Programmiersprache dies nicht fordert.

Beispiel: Statt $x - y - z$ schreibe $(x - y) - z$.

Die Prioritätsregeln verschiedener Programmiersprachen unterscheiden sich. Arbeitet man abwechselnd mit verschiedenen Sprachen, so können einem leicht Fehler unterlaufen, die man durch zusätzliche Klammern vermeiden kann. Eine Programmierempfehlung ist daher: Lieber zu viel als zu wenig klammern!

10.2.3 Reihenfolge der Operandenauswertung

Eine andere Frage ist, in welcher Reihenfolge die Operanden zweistelliger Operatoren ausgewertet werden.

Solange bei der Auswertung eines Operanden nur ein Wert berechnet wird, ist diese Reihenfolge egal, da sie das Ergebnis der Operation nicht beeinflusst. Es könnte jedoch Operanden geben, die nicht nur einen Wert liefern, sondern bei ihrer Auswertung auch den Zustand einer (oder mehrerer) Variablen verändern, d.h. einen **Seiteneffekt** (*side effect*) bewirken.² Wenn Seiteneffekte auftreten, können je nach Reihenfolge der Operandenauswertung verschiedene Ergebnisse herauskommen - ein nicht wünschenswerter Sachverhalt!

Zum Glück sind in Component Pascal alle Operatoren **seiteneffektfrei**, d.h. sie bewirken nur die Auswertung ihrer Operanden und die Berechnung des Ergebniswerts. Seiteneffekte können daher höchstens durch Funktionsaufrufe verursacht werden.

Die Sprachdefinition von Component Pascal legt die Reihenfolge der Operandenauswertung nicht fest, außer bei den Boole'schen Operatoren **OR**, **&** (siehe 10.6, S. 11). Es bleibt der Implementierung eines Übersetzers überlassen, ob er zuerst den linken oder den rechten Operanden auswertet.

Manche Operatoren, z.B. **+**, *****, **=**, sind in der Mathematik **kommutativ**, d.h. die Operanden sind vertauschbar.

Beispiel: $a + b = b + a$.

Nach der Sprachdefinition hat ein Übersetzer die Freiheit, implizit das Kommutativgesetz anzuwenden, z.B. um die Registerzuteilung zu optimieren.

10.2.4 Überladungen

Manche Operatoren sind **überladen** (*overloaded*), d.h. sie können verschieden oder auf Operanden verschiedener Typen angewandt werden. Der Typ der Operanden bestimmt dann die Wirkung der Operation.

Beispiele:

In $a + b$ können a und b beide Zahlen, Mengen oder Zeichenketten sein.

Plus- und Minuszeichen treten als ein- und zweistellige Operatoren auf: $-a$, $b - c$.

10.3 Typregeln

10.3.1 Typbindung

Eine Programmiersprache hat **Typbindung** und heißt **typisiert** oder **typgebunden** (*typed*), wenn jeder Wert, jede Variable, jeder Ausdruck, jeder Parameter, jedes Funktionsergebnis usw. (mindestens) einen Typ hat, der unabhängig von den Operationen ist, die damit ausgeführt werden. Es ist umgekehrt der Typ eines Werts, einer Variable usw., der die möglichen Operationen damit festlegt. Viele höhere Programmiersprachen, insbesondere imperative Sprachen, sind typgebunden.

² Die Problematik der Seiteneffekte wird später im Zusammenhang mit Funktionen behandelt.

Es gibt aber auch nichttypisierte Programmiersprachen; sie heißen **typfrei** (*type free*). Beispiele für typfreie Sprachen sind die meisten Maschinensprachen: Sie sehen Daten als Bitfolgen, die mit beliebigen Befehlen verarbeitet werden können.

10.3.1.1 Statische Typbindung

Eine Programmiersprache hat **statische Typbindung** und heißt **statisch typisiert** (*statically typed*), wenn jedem Wert, jeder Variable usw. (mindestens) ein statisch - d.h. zur Übersetzungszeit - festgelegter, unveränderlicher Typ zugeordnet ist. Die Typen der Größen sind also ohne Ausführung des Programms aus dem Quellprogrammtext ermittelbar. Die Typen bestimmen, welche Operationen mit welchen Werten möglich sind. In Ausdrücken entscheiden die Typen der Operanden über die Bedeutung der Operatoren.

Es gibt auch **dynamisch typisierte** (*dynamically typed*) Programmiersprachen; bei diesen kann sich der Typ einer Variablen durch Zuweisung eines neuen Werts ändern. Mit diesem Aspekt beschäftigen wir uns später im Zusammenhang mit objektorientierter Programmierung, denn objektorientierte Sprachen sind dynamisch typisiert.

Die Typbindung heißt **stark** (*strongly typed*), wenn jedem Wert, jeder Variable usw. genau ein statisch festgelegter, unveränderlicher Typ zugeordnet ist und mit Größen nur die für den zugehörigen Typ definierten Operationen möglich sind.

Beispiel: Der nicht-objektorientierte Teil von Component Pascal ist stark typisiert.

Eine Operation heißt **typsicher**, wenn jede Ausführung korrekte Werte beteiligter Typen erzeugt. Eine Programmiersprache heißt **typsicher**, wenn alle ihre Operationen typsicher sind. Dies ist eine sehr starke Bedingung. Ist beispielsweise eine Ganzzahl vereinbart:

```
VAR i : INTEGER;
```

so kann eine einfache Addition

```
i := i + 1;
```

zu einem **Überlauf** führen, d.h. einen falschen Wert erzeugen. Diese Art von Typfehler - **Wertebereichsüberschreitung** - ist nicht statisch (d.h. vom Übersetzer) erkennbar, und eine Prüfung zur Laufzeit ist aufwendig.

Unter **Typprüfung** (*type checking*) verstehen wir das Kontrollieren, ob bei allen Operationen eines Programms alle beteiligten Operanden den richtigen Typ haben. **Statische Typprüfung** findet zur Übersetzungszeit durch den Übersetzer statt, **dynamische Typprüfung** zur Laufzeit. Statische Prüfung ist günstiger, da Fehler früher erkannt werden und Laufzeitaufwand gespart wird. Wo jedoch statische Prüfung nicht möglich ist, ist dynamische Prüfung besser als gar keine.

Absolute Typsicherheit ist schwer zu erreichen. Programmiersprachen unterscheiden sich in ihren Typregeln, in der Art, wie sie diese Regeln prüfen und in der daraus resultierenden Sicherheit. Je schwächer die Typregeln und je lockerer Typprüfungen, desto weniger Sicherheit. Je besser die Regeln, je konsequenter die Kontrolle, je weniger Lücken, desto größer die Sicherheit. Dies ist besonders wichtig bei Software für sicherheitskritische Anwendungen wie Autos, Flugverkehr, Atomkraftwerke, denn ein Programm kann nicht sicherer sein als die Sprache, in der es geschrieben ist.

Wir nennen eine Programmiersprache **streng typisiert** (*strongly typed*), wenn sie folgende Merkmale aufweist (neue Begriffe erklären wir in folgenden Abschnitten):

- + Für die Anwendbarkeit von Operatoren auf Operanden und die Zuweisbarkeit bei Zuweisungen und Parameterübergaben gelten strenge Typkompatibilitätsregeln.
- + Implizite Typanpassung erfolgt zurückhaltend.
- + Explizite Typanpassung birgt keine Sicherheitsrisiken.
- + Die Einhaltung der Typregeln wird geprüft, auch über die Grenzen von Übersetzungs- und Ladeeinheiten hinweg.
- + Wo statische Prüfungen nicht ausreichen, wird dynamisch geprüft.

Demgegenüber nennen wir eine Programmiersprache **schwach typisiert** (*weakly typed*), wenn sie wenigstens eines der folgenden Merkmale aufweist:

- In Ausdrücken sind Operatoren freizügig auf Operanden verschiedenen Typs anwendbar.
- Bei Zuweisungen und Parameterübergaben gelten lasche Regeln.
- Werte werden großzügig implizit an andere Typen angepaßt.
- Die Typbindung kann leicht durch explizite Typanpassungen unterlaufen werden.
- Die statischen Typprüfungen sind lückenhaft, z.B. nur innerhalb einer Übersetzungseinheit.
- Auf Laufzeitprüfungen wird prinzipiell (z.B. aus Effizienzgründen) verzichtet.

Die Übergänge zwischen strenger und schwacher Typisierung sind fließend. Man kann aber getrost sagen, daß Programmiersprachen in der Tradition von Pascal eher zu strenger Typisierung tendieren, während C und C++ als schwach typisiert einzuordnen sind. Component Pascal gehört nach Einschätzung des Autors zu den Sprachen mit den strengsten Typprüfungen, was für seine Eignung als erste Lehrsprache spricht.

Starke statische Typbindung unterstützt strenge Typisierung und bietet folgende Vorteile:

- + Sie ermöglicht statische Typprüfung und hilft so, bestimmte Programmierfehler früh - zur Übersetzungszeit - zu erkennen.
- + Sie erleichtert, die Korrektheit eines Programms zu prüfen, weil man die Bedeutung aller Operationen kennt, ohne das Programm mit Testdaten ausführen zu müssen.
- + Sie ermöglicht dem Übersetzer, effizienten Code zu erzeugen.
- + Sie macht ein Programm besser verständlich, da sie seinen Dokumentationswert erhöht.
- + Sie erlaubt es, Schnittstellen von Programmkomponenten (Modulen, Prozeduren) sauber zu definieren.

10.3.2 Typeinschluß

Aus der Mathematik sind die Teilmengenbeziehungen

$$N \subseteq Z \subseteq Q \subseteq R \subseteq C$$

der Zahlenmengen bekannt, d.h. eine natürliche Zahl ist auch eine ganze Zahl, die ganzen Zahlen sind in den rationalen Zahlen enthalten, die reellen Zahlen umfassen die rationalen Zahlen, die komplexen Zahlen enthalten alle anderen Zahlen. In Programmiersprachen wünscht man sich ein ähnliches Verhalten der verschiedenen Zahlentypen, auch wenn sie maschinenintern unterschiedlich gespeichert sind und unterschiedlich verarbeitet werden.

Auf der anderen Seite gibt es Datentypen, die den gleichen Speicherplatz benötigen, z.B. ein Byte, und dabei gleiche Bitmuster benutzen, aber trotzdem Unterschiedliches bedeuten, z.B. `BOOLEAN`, `BYTE` und `SHORTCHAR`. Sie sollten in der Programmiersprache strikt unterscheidbar behandelt werden.

In Component Pascal gilt folgendes. Ein **Zahlentyp** oder **numerischer Typ** (*numeric type*) ist ein Integertyp oder ein Realtyp. **Ganzzahlige Typen** oder **Integertypen** (*integer types*) sind `BYTE`, `SHORTINT`, `INTEGER`, `LONGINT`. **Reelle Typen** oder **Realtypen** (*real types*) sind `SHORTREAL`, `REAL`. Die numerischen Typen bilden eine Hierarchie, **Typeinschluß** (*type inclusion*) genannt:

$$\text{BYTE} \subseteq \text{SHORTINT} \subseteq \text{INTEGER} \subseteq \text{LONGINT} \subseteq \text{SHORTREAL} \subseteq \text{REAL}.$$

Ein größerer Typ (rechts) **schließt** die kleineren Typen (links), d.h. ihre Wertebereiche **ein**. Ein Typ ist zu einem größeren Typ **kompatibel**, d.h. ein Exemplar eines kleineren Typs ist ohne Verlust führender Stellen in ein Exemplar eines größeren Typs konvertierbar.³ Beachte, daß die Umkehrung nicht gilt, d.h. ein Typ ist nicht zu einem kleineren kompatibel, da bei der Konvertierung führende Stellen verlorengehen können.

Die Typeinschlußregel gilt auch für die beiden **Zeichentypen**:

$$\text{SHORTCHAR} \subseteq \text{CHAR}.$$

10.3.3 Typanpassungen

In Programmiersprachen versteht man unter **Typanpassung** die Umwandlung eines Werts eines bestimmten Typs in einen Wert eines anderen Typs. Man kann zwischen impliziter und expliziter Typanpassung unterscheiden. **Implizite Typanpassung** (*coercion*) bedeutet, daß die Sprache Regeln für die Anpassung enthält und die Umwandlung automatisch (durch den Übersetzer) erfolgt. **Explizite Typanpassung** (*casting*) heißt, daß die ProgrammiererIn selbst die Umwandlung mit sprachlichen Mitteln, z.B. einer Konversionsfunktion, erzwingt. Die Typwandlung selbst kann durch bloßes **Uminterpretieren** eines Bitmusters geschehen (z.B. `CHAR` als `SHORTINT`, `SET` als `INTEGER`), oder durch **Konvertieren** einer Darstellung in eine andere (z.B. von `INTEGER` nach `REAL`).

Manche Programmiersprachen passen ausgiebig implizit an, z.B. größere Typen an kleinere (wobei führende Stellen ohne Fehlermeldung verlorengehen können), oder unterschiedliche Typen wie `CHAR` und `INTEGER` (manche Sprachen unterscheiden diese gar nicht als verschiedene Typen).

Streng typisierte Sprachen lassen solches nicht zu und schließen damit unnötige Fehlerquellen. In Component Pascal gelten folgende Regeln.

³ Genauigkeit kann bei der Konvertierung verlorengehen, z.B. von `LONGINT` nach `SHORTREAL`.

10.3.3.1 Implizite Typanpassung

Implizite Typanpassung erfolgt nur zwischen kompatiblen Typen, d.h. bei numerischen Typen von kleineren zu größeren.

Für die Auswertung von Ausdrücken bedeutet das: Kommen in einem Ausdruck verschiedene numerische Operanden vor, so wird vor der Anwendung des Operators der kleinere an den größeren angepaßt, das Ergebnis ist dann auch von diesem Typ. In einem komplexen arithmetischen Ausdruck ist damit der Ergebnistyp gleich dem größten Typ aller Operanden. Die Typanpassung erfolgt nur nach Bedarf, d.h. in der Reihenfolge der Auswertung der Teilausdrücke.

10.3.3.2 Explizite Typanpassung

Für explizite Typanpassungen stehen folgende vordeklarierte Konversionsfunktionen zur Verfügung.

<i>Vordeclarierte Funktion:</i>	<i>Ergebnistyp:</i>	<i>Bedeutung:</i>
ORD (c : CHAR)	: INTEGER	Ordnungszahl zu c.
ORD (c : SHORTCHAR)	: SHORTINT	
CHR (x : Integertyp)	: CHAR	Zeichen mit der Ordnungszahl x.
LONG (c : SHORTCHAR)	: CHAR	Identischer Wert, Einbettung.
LONG (s : Kurzzeichenkette)	: Zeichenkette	
LONG (x : BYTE)	: SHORTINT	
LONG (x : SHORTINT)	: INTEGER	
LONG (x : INTEGER)	: LONGINT	
LONG (x : SHORTREAL)	: REAL	
SHORT (c : CHAR)	: SHORTCHAR	Identischer Wert, Projektion. ⁴
SHORT (s : Zeichenkette)	: Kurzzeichenkette	
SHORT (x : LONGINT)	: INTEGER	Identischer Wert, Einschränkung. ⁵
SHORT (x : INTEGER)	: SHORTINT	
SHORT (x : SHORTINT)	: BYTE	
SHORT (x : REAL)	: SHORTREAL	Identischer Wert, Einschränkung (mit Genauigkeitsverlust). ⁶
ENTIER (x : Realtyp)	: LONGINT	Größte ganze Zahl $\leq x$. ⁷
BITS (x : INTEGER)	: SET	Interpretation als Menge, Ergebnis: $\{i \mid \text{ODD}(x \text{ DIV } 2^i)\}$.

Gelegentlich braucht man diese Funktionen. Häufige angewandte Typanpassungen deuten jedoch darauf hin, daß Datenstrukturen schlecht entworfen wurden. Was richtig paßt, muß man nicht anpassen! Wir halten uns daher an die folgende

⁴ Überlauf ist möglich.

⁵ Überlauf ist möglich.

⁶ Überlauf ist möglich.

⁷ Überlauf ist möglich.

Programmierrichtlinie

Explizite Typanpassungen sind verpönt und dürfen nur in begründeten Ausnahmefällen verwendet werden. Stattdessen sind Variablen von vornherein mit dem richtigen Typ zu vereinbaren.

10.3.4 Ausdruckskompatibilität

In Component Pascal sind z.B. die Ausdrücke

```
1 + 'a'
2 = TRUE
'369' < {3, 6, 9}
```

syntaktisch korrekt. Aber was sollten sie bedeuten? Da ihnen kein offensichtlicher Sinn unterliegt, sollte eine Programmiersprache sie nicht erlauben.⁸ In Component Pascal sind sie falsch, da sie die Regeln der Ausdruckskompatibilität nicht erfüllen.

Die Typen der Operanden eines Operators heißen **ausdruckskompatibel** (*expression compatible*), wenn sie folgender Tabelle entsprechen. Ein Ausdruck ist nur dann korrekt, wenn alle Operanden ausdruckskompatibel sind. Die Tabelle zeigt auch den Typ des Ergebnisses des Ausdrucks.

Operator:	zulässige Operandentypen:	Ergebnistyp:
+ - * DIV MOD	Integertypen	größter Integertyp der Operanden, mindestens INTEGER
+ - *	Realtypen	REAL
/	numerische Typen	REAL
+ - * /	SET	SET
+	kurze Zeichenkette Zeichenkette, nicht beide kurz	kurze Zeichenkette Zeichenkette
OR & ~	BOOLEAN	BOOLEAN
= # < <= > >=	numerische Typen Zeichentypen Zeichenkettentypen	BOOLEAN BOOLEAN BOOLEAN
= #	BOOLEAN SET Zeigertyp T0 oder T1 , NIL Prozedurtyp T , NIL	BOOLEAN BOOLEAN BOOLEAN BOOLEAN
IN	links: Integertyp, rechts: SET	BOOLEAN
IS	links: Typ T0 , rechts: Typ T1	BOOLEAN

Der Typ **T1** muß eine Erweiterung des Typs **T0** sein.⁹

⁸ Es gibt Programmiersprachen, die solche Ausdrücke zulassen.

10.4 Arithmetische Ausdrücke

Ein **arithmetischer Ausdruck** setzt sich aus numerischen Konstanten und Variablen, Funktionen mit numerischen Ergebnissen und folgenden Operatoren zusammen.

Operator:	Bedeutung:
+	Addition (zweistellig), Identität (einstellig)
-	Subtraktion (zweistellig), Vorzeichenwechsel (einstellig)
*	Multiplikation
/	reelle Division
DIV	ganzzahlige Division
MOD	ganzzahliger Rest

Bei +, -, * und / können numerische Typen beliebig gemischt vorkommen. Die Werte der Operanden werden implizit entlang der Typeinschlußhierarchie angepaßt, wobei „kleine“ Typen übergangen werden. Genauer: Ist der größte Operandentyp **INTEGER**, so werden beide Operandenwerte an **INTEGER** angepaßt und das Ergebnis ist vom Typ **INTEGER**. Ist der größte Operandentyp **LONGINT**, so wird nach **LONGINT** angepaßt und ein **LONGINT**-Ergebnis geliefert. Kommt ein reeller Operand vor oder handelt es sich um die Division /, so werden beide Operandenwerte an **REAL** angepaßt und der Ergebnistyp ist **REAL**. **DIV** und **MOD** sind nur auf Integertypen anwendbar; es wird bis mindestens **INTEGER** angepaßt. Sie sind durch folgende Aussagen definiert (d.h. diese Boole'schen Ausdrücke liefern immer **TRUE**).

$$x = (x \text{ **DIV** } y) * y + (x \text{ **MOD** } y).$$

$$(0 \leq (x \text{ **MOD** } y) < y) \text{ **OR** } (0 \geq (x \text{ **MOD** } y) > y).$$

Außerdem gilt - auch bei negativen Werten - die Gleichung

$$x \text{ **DIV** } y = \text{ENTIER} (x / y).$$

Beispiel:

x	y	x DIV y	x MOD y
11	4	2	3
-11	4	-3	1
11	-4	-3	-1
-11	-4	2	-3

⁹ Typenerweiterung ist ein objektorientiertes Konzept und wird später behandelt.

10.5 Relationale Ausdrücke

Ein **relationaler Ausdruck** setzt sich aus folgenden Operatoren

Operator:	Bedeutung:
=	gleich
#	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
IN	enthält
IS	ist vom Typ

und aus Konstanten, Variablen und Funktionen mit Ergebnissen, die mit diesen Operatoren ausdrucks kompatibel sind, zusammen. Das Ergebnis ist immer vom Typ **BOOLEAN**, so daß relationale Ausdrücke eigentlich einen Spezialfall der Boole'schen Ausdrücke bzw. mögliche Operanden Boole'scher Ausdrücke darstellen.

Die **Ordnungsrelationen** <, <=, >, >= entsprechen bei numerischen Operanden der üblichen mathematischen Ordnung, bei Zeichen und Zeichenketten der alphabetischen Ordnung, die durch die Ordnung des zugrundeliegenden Zeichencodes erweitert ist. Numerische Typen können wieder beliebig gemischt vorkommen, der kleinere Typ wird implizit vor dem Vergleich an den größeren angepaßt. Entsprechend wird ein Zeichen als Zeichenkette der Länge 1 interpretiert. Die genaue Definition der Vergleichsregeln für Zeichenketten wird später gegeben, wenn der Begriff der offenen Reihung bekannt ist.

Die **Enthaltenseinsbeziehung** oder **Elementrelation** **i IN s** gilt zwischen einem ganzzahligen Operanden **i** und einem Operanden **s** vom Typ **SET** und liefert **TRUE**, wenn der Wert von **i** in der Menge **s** enthalten ist.

Die **Typrelation** **IS** gehört zu den objektorientierten Konstrukten und wird später behandelt.

Beispiele:

Relationaler Ausdruck:	Wert:
123 < 456.789	TRUE
'a' >= 'bcd'	FALSE
12 IN {0 .. 9, 20, 30}	FALSE

10.6 Boole'sche Ausdrücke

Ein **Boole'scher Ausdruck** setzt sich aus Boole'schen Variablen, Funktionen mit Boole'schen Ergebnissen, relationalen Ausdrücken und folgenden Operatoren zusammen. Die Boole'schen Konstanten **FALSE** und **TRUE** können natürlich auch vorkommen, sind aber stets durch Vereinfachung des Ausdrucks eliminierbar.

Operator:	Bedeutung:	Zeichen der math. Logik:
OR	oder, Disjunktion, $\vee$	
&	und, Konjunktion, $\wedge$	
~	nicht, Negation, $\neg$	

In der Boole'schen Algebra der Logik ist der Effekt der Operatoren bekanntlich durch folgende Wertetabelle beschrieben. (a und b sind Boole'sche Ausdrücke.)

a	b	a OR b	a & b	~ a
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
TRUE	FALSE	TRUE	FALSE	FALSE
TRUE	TRUE	TRUE	TRUE	FALSE

In Programmiersprachen unterscheidet man dagegen zwischen langer und kurzer Auswertung Boole'scher Ausdrücke. **Lange** oder **vollständige Auswertung** bedeutet, daß jeder Operand bei einer Operation vollständig ausgewertet wird und dann das Ergebnis gemäß obiger Wertetabelle bestimmt wird. Dabei müssen alle Operanden einen Wert besitzen.

Bei **kurzer** oder **bedingter Auswertung** (*lazy evaluation*) werden die Operanden von links nach rechts ausgewertet, wobei die Auswertung abgebrochen wird, sobald das Ergebnis feststeht, d.h. der zweite Operand wird ggf. nicht ausgewertet.

a **OR** b ausgewertet wie **IF a THEN TRUE ELSE b END**

a **&** b ausgewertet wie **IF a THEN b ELSE FALSE END**

Der zweite Operand muß also gar nicht unbedingt einen Wert besitzen. Diese Eigenschaft kann genutzt werden, um Abfragen, die sonst mit geschachtelten **IF**-Anweisungen zu lösen wären, in einer kurz auszuwertenden Bedingung zu formulieren.

Beispiel:

```
IF x > 0 THEN
  IF y / x < 1 THEN
    z := x + y;
  END;
END;
```

ist reduzierbar auf

```
IF (x > 0) & (y / x < 1) THEN
  z := x + y;
END;
```

Im Fall $x = 0$ ist y/x undefiniert bzw. führt zu einem Laufzeitfehler. Die zweite Formulierung der Anweisung ist also bei langer Auswertung fehlerhaft und kann zum Programmabbruch führen, während sie bei kurzer Auswertung immer korrekt funktioniert. Die erste Formulierung ist in beiden Fällen korrekt, aber aufwendiger.

Als Folge kurzer Auswertung sind die Boole'schen Operatoren aber nicht wie in der Logik kommutativ, d.h. die Operanden können nicht beliebig vertauscht werden!

Component Pascal wertet Boole'sche Ausdrücke generell kurz aus. Diese Regelung führt zu kompakteren und trotzdem sicheren Programmen. Bei Programmiersprachen, die die Art der Auswertung nicht festlegen, ist man hingegen vom Übersetzer abhängig und hat die Wahl zwischen Sicherheit mit Aufwand und Effizienz mit Portierbarkeitsproblemen.

Boole'sche Ausdrücke sollte man nicht unnötig kompliziert, sondern möglichst kompakt formulieren. Zur Vereinfachung eines Ausdrucks kann man ihn mittels der Rechenregeln der Boole'schen Algebra in einen äquivalenten umformen. Wichtige Regeln sind (mit obiger Einschränkung) die Kommutativ-, Assoziativ-, Distributiv-, Absorptions- und Idempotenzgesetze, die Gesetze der neutralen Elemente, der Komplemente und die de Morgan'schen Gesetze.

Beispiele:

Boole'scher Ausdruck:

vereinfacht zu:

$a = \text{TRUE}$

a

$a = \text{FALSE}$

$\sim a$

$\sim a \text{ OR } \sim b$

$\sim(a \ \& \ b)$

$a \ \& \ b \text{ OR } c \ \& \ (\sim d = \text{FALSE}) \ \& \ a$

$a \ \& \ (b \text{ OR } (c \ \& \ d))$

$(x < 9) \ \& \ \sim(0 \geq y) \ \& \ \sim(x \geq 6)$

$(x < 6) \ \& \ (y > 0)$

10.7 Zeichenkettenausdrücke

Ein **Zeichenkettenausdruck** setzt sich aus Zeichenkettenkonstanten und -variablen und dem $+$ -Operator für die Verkettung (Konkatenation) zweier Zeichenketten zusammen.

Operator: *Bedeutung:*

$+$ Verkettung

Beispiel:

Zeichenkettenausdruck:

Wert:

"strin" + "g e" + "xpression"

"string expression"

10.8 Mengenausdrücke

Ein **Mengenausdruck** setzt sich aus Mengenkonstanten und -variablen, mittels ganzzahligen Ausdrücken gebildeten Mengen, und folgenden Operatoren zusammen.

Operator: *Bedeutung:*

$+$ Vereinigung

$-$ Differenz (zweistellig), Komplement (einstellig)

$*$ Durchschnitt

$/$ symmetrische Differenz

In einer Menge stehen zwei Punkte $\dots$ für einen Bereich.

Beispiele:

Mengenausdruck:

Wert:

$\{\}$

leere Menge

$\{1 \dots 8\}$

$\{1, 2, 3, 4, 5, 6, 7, 8\}$

$\{i \dots i + k, l, 2 * m \dots 3 * n\}$

$\{9\} + aSet * bSet - cSet / dSet.$

Zusammenhänge zwischen den Mengenoperatoren und mit der Elementrelation sind durch folgende Aussagen gegeben (d.h. diese Boole'schen Ausdrücke liefern immer TRUE).

$s - t = s * (-t).$

$s / t = (s - t) + (t - s).$

$s \setminus t = (s + t) - (s * t).$

$-(s + t) = (-s) * (-t).$

$-(s * t) = (-s) + (-t).$

$(s = s + t) = (t = s * t).$

$(i \text{ IN } (-s)) = (i \text{ IN } \{0 \dots \text{MAX}(\text{SET})\}) \& \sim(i \text{ IN } s).$

$(i \text{ IN } (s + t)) = (i \text{ IN } s) \text{ OR } (i \text{ IN } t).$

$(i \text{ IN } (s - t)) = (i \text{ IN } s) \& \sim(i \text{ IN } t).$

$(i \text{ IN } (s * t)) = (i \text{ IN } s) \& (i \text{ IN } t).$

$(i \text{ IN } (s / t)) = ((i \text{ IN } s) \# (i \text{ IN } t)).$

10.9 Vordeklarierte Funktionen

Wir stellen hier die vordeklarierten Funktionen von Component Pascal in alphabetischer Reihenfolge zusammen; ausgenommen die in 10.3.3.2, S. 8 zusammengestellten Konversionsfunktionen. Einige Funktionen sind bereits von vorhergehenden Abschnitten bekannt. Aufrufe dieser Funktionen können als Operanden in Ausdrücken auftreten.

<i>Vordeklarierte Funktion:</i>	<i>Ergebnistyp:</i>	<i>Bedeutung:</i>
ABS (x : Zahlentyp)	: Typ von x	Absolutbetrag von x .
ASH (x, n : Integertyp)	: INTEGER	Arithmetische Verschiebung, Ergebnis: $x * 2^n$.
CAP (c : Zeichentyp)	: Typ von c	Großbuchstabe zu c (falls c Kleinbuchstabe).
LEN (a : Arraytyp; n : Integerkonstante)	: INTEGER	Länge von a in Dimension n . (Erste Dimension = 0.)
LEN (a : Arraytyp)	: INTEGER	Äquivalent zu LEN ($a, 0$).

LEN (s : Zeichenkette)	: INTEGER	Länge von s (ohne 0X).
MAX (Grundtyp)	: Grundtyp	Größter Wert des Grundtyps.
MAX (SET)	: INTEGER	Größter Wert einer Menge (31).
MAX (x, y : Zahlentyp)	: Typ von x, y	Maximum von x, y.
MIN (Grundtyp)	: Grundtyp	Kleinster Wert des Grundtyps.
MIN (SET)	: INTEGER	Kleinster Wert einer Menge (0).
MIN (x, y : Zahlentyp)	: Typ von x, y	Minimum von x, y.
ODD (x : Integertyp)	: BOOLEAN	Test auf Ungerade, Ergebnis: $x \bmod 2 = 1$.
SIZE (Typ)	: INTEGER	Anzahl der Bytes, die ein Exemplar des Typs als Speicherplatz benötigt.

Kapitel 11

Anweisungen

Anweisung (*statement*) :=

Aktion, die bei ihrer Ausführung den Zustand von Variablen verändert.

11.1 Arten von Anweisungen

■ **Einfache** (elementare) Anweisungen:

- Wertzuweisungen,
- Prozeduranweisungen,
- Steueranweisungen,
- leere Anweisungen.

■ **Strukturierte** Anweisungen :=

aus anderen Anweisungen zusammengesetzt:

- Auswahlanweisungen,
- Wiederholungsanweisungen.

11.1.1 Syntax

Statement =

[*Assignment* |
ProcedureCall |
EXIT | **RETURN** [*Expr*] |
IfStatement | *CaseStatement* | *WithStatement* |
WhileStatement | *RepeatStatement* |
LoopStatement | *ForStatement*].

StatementSequence =

Statement { „;“ *Statement* }.

- Das Semikolon ist Trennzeichen zwischen Anweisungen (nicht Abschlußzeichen nach einer Anweisung).
- Da leere Anweisungen erlaubt sind, ist das Semikolon als Abschlußzeichen verwendbar.

11.1.2 Formatierung

- Pro Anweisung (min.) eine Zeile vorsehen.

11.2 Einfache Anweisungen

11.2.1 Wertzuweisungen

11.2.1.1 Beispiele

```
x := 0
```

```
sGELSemester1 [stuNr + 1].  
noteInFachbeiVersuch [I1, 0] :=  
1.0
```

```
area := radius * radius * Math.Pi
```

11.2.1.2 Syntax

Assignment =
 Designator „:=“ *Expr* .

11.2.1.3 Semantik

- Der Ausdruck auf der rechten Seite wird ausgewertet.
- Der Wert des Ausdrucks wird der Variablen auf der linken Seite zugewiesen.

11.2.1.4 Typregel und Zuweisungskompatibilität

- Der Typ dieses Ausdrucks muß mit dem Typ der Variable auf der linken Seite *zuweisungskompatibel* sein, d.h. zur Übersetzungszeit wird eine *Typprüfung* durchgeführt.
- Ein Ausdruck e vom Typ T_e ist **zuweisungskompatibel** (*assignment compatible*) mit einer Variablen v vom Typ T_v , wenn eine der folgenden Bedingungen erfüllt ist:
 1. T_e und T_v sind *derselbe* Typ.
 2. T_e und T_v sind numerische Typen & T_v schließt T_e ein.
 3. bis 8. Siehe *Component Pascal Language Report* bzw. später.

11.2.2 Prozeduranweisungen

11.2.2.1 Beispiele

```
Out.String ( 'Hello' )
```

```
MyModule.Do
```

```
a[b].c[d, e + f].g.h (i, k[l], m.n)
```

■ Näheres in Kapitel 12.

11.2.3 Steueranweisungen

■ Siehe 11.3.2.4, S. 22 & Kapitel 12.

11.3 Strukturierte Anweisungen

11.3.1 Auswahlanweisungen

11.3.1.1 Ein-/zwei-/mehrfache Auswahl

Synonyme: **Alternative**, **Selektion**.

Syntax

IfStatement =

```
IF Expr THEN StatementSequence  
  { ELSIF Expr THEN StatementSequence }  
  [ ELSE StatementSequence ]  
END.
```

Typregel

- Die Ausdrücke müssen Boole'sche Ausdrücke sein; sie heißen **Bedingungen** (*guards*).

Formatierung

```
IF Guard1 THEN  
    Statements1  
ELSIF Guard2 THEN  
    Statements2  
ELSE  
    Statements3  
END
```

Semantik

1. Die Bedingung zwischen **IF** und **THEN** wird ausgewertet.
2. Liefert sie **TRUE**, so werden die Anweisungen des zugehörigen **THEN**-Zweigs ausgeführt & dann die **IF**-Anweisung beendet.
3. Andernfalls werden - falls vorhanden - weitere Bedingungen ausgewertet.
4. Die **ELSIF**-Bedingungen werden in der Reihe der Aufschreibung solange ausgewertet, bis eine Bedingung **TRUE** liefert. In diesem Fall gilt 2.
5. Liefern alle Bedingungen **FALSE**, so wird - falls vorhanden - der **ELSE**-Zweig ausgeführt.

Beispiel

```
IF mark <= 1.0 THEN  
    BeHappy  
ELSIF mark > 4.0 THEN  
    KeepOnTrying  
ELSE  
    DontWorry  
END
```

Beispiel

```
IF x < 2 THEN  
    IF y > 4 THEN  
        . . .  
    END  
END
```

ist zu vereinfachen zu

```
IF (x < 2) & (y > 4) THEN  
    . . .  
END
```

Beispiel

```
IF i <= 0 THEN  
    . . .  
ELSIF i = 1 THEN  
    . . .  
ELSIF i = 2 THEN  
    . . .  
ELSIF i = 3 THEN  
    . . .  
ELSE  
    . . .  
END
```

- Abfrage eines Ausdrucks auf verschiedene konstante Werte.
- Wiederholter Test derselben Größe ist ineffizient.

11.3.1.2 Mehrfache Auswahl

Beispiel

```
CASE i OF  
| MIN (INTEGER) .. 0 : ...  
| 1 : ...  
| 2 : ...  
| 3 : ...  
ELSE  
    ...  
END
```

Syntax

CaseStatement =

```
CASE Expr OF Case { „|“ Case }  
[ ELSE StatementSequence ]  
END.
```

Case = [*CaseLabelList* „:“ *StatementSequence*].

CaseLabelList = *CaseLabels* { „,“ *CaseLabels* }.

CaseLabels = *ConstExpr* [„. “ *ConstExpr*].

Regeln

- Der Typ des Ausdrucks muß ein Integertyp oder CHAR sein.
- Die **Marken** (*case labels*) müssen von demselben Typ wie der Ausdruck sein bzw. der Ausdruckstyp muß die Typen aller Marken einschließen.
- Die Marken sind Konstanten.
- Keine Marke darf mehrmals in der Falliste auftreten, bzw.
jeder Wert des Ausdrucks darf in höchstens einem **Fall** (*case*) einmal als Marke auftreten.

Formatierung

```
CASE Expression OF  
| Label11, Label12 :  
  Statements1  
| Label21 .. Label22 :  
  Statements2  
ELSE  
  Statements3  
END
```

Semantik

- Der Ausdruck wird ausgewertet, sein Wert bestimmt den Zweig, der ausgeführt wird.
- Tritt der Wert als Marke in einem Fall auf, so werden die zugehörigen Anweisungen ausgeführt.
- Fehlt der Wert in der Falliste, so wird ggf. der **ELSE**-Zweig ausgeführt.
- Nach Ausführung des gewählten Zweigs wird die **CASE**-Anweisung beendet.
- Fehlen ein Fall für den Wert & der **ELSE**-Zweig, so wird der Programmablauf *abgebrochen*!
- Beachte den Unterschied zur **IF**-Anweisung!

Problem

- Häufiger Programmierfehler: Fehlender Wert in der Falliste & fehlender **ELSE**-Zweig.

Programmierleitlinie

- Die Falliste muß jeden Wert des Ausdrucks enthalten, besser noch sieht man einen **ELSE**-Zweig *mit einer Fehlerbehandlung* vor (s. 11.4.2.1, S. 29).

11.3.1.3 With-Anweisung

- Die **WITH**-Anweisung ist ein objektorientiertes Konstrukt & wird später behandelt.

11.3.2 Wiederholungsanweisungen

Synonym: **Iteration**.

11.3.2.1 Syntax

WhileStatement =

WHILE *Expr* **DO** *StatementSequence* **END.**

RepeatStatement =

REPEAT *StatementSequence* **UNTIL** *Expr.*

LoopStatement =

LOOP *StatementSequence* **END.**

ForStatement =

FOR *ident* „:=“ *Expr* **TO** *Expr*
[**BY** *ConstExpr*]
DO *StatementSequence* **END.**

11.3.2.2 Allgemeine Semantik

- Die Anweisungsfolge heißt **Schleifenrumpf**.
- Der Schleifenrumpf wird wiederholt ausgeführt.
- Spezielle Klauseln steuern die Anzahl der Wiederholungen.

11.3.2.3 Bedingungsschleifen

Abweisende Schleife

```
WHILE continueCondition DO  
    Statements  
END
```

Typregel

- Der Ausdruck muß ein Boole'scher Ausdruck sein; er heißt **Fortsetzungsbedingung** (*guard*).

Semantik

- Die Fortsetzungsbedingung wird ausgewertet.
- Liefert sie **TRUE**, so wird der Schleifenrumpf ausgeführt.
- Nach Ausführung des Rumpfs wird die Fortsetzungsbedingung erneut ausgewertet & ggf. der Rumpf wieder ausgeführt.

Nichtabweisende Schleife

REPEAT

Statements

UNTIL *breakCondition*

Typregel

- Der Ausdruck muß ein Boole'scher Ausdruck sein; er heißt **Abbruchbedingung**.

Semantik

- Der Rumpf wird mindestens einmal ausgeführt.
- Nach Ausführung des Rumpfs wird die Abbruchbedingung ausgewertet.
- Liefert sie **FALSE**, so wird der Rumpf wiederholt ausgeführt.
- Die **REPEAT**-Schleife ist äquivalent zu
Statements;
WHILE *~breakCondition* **DO**
Statements
END

Folgerungen

- Bei Bedingungsschleifen wird die Anzahl der Schleifendurchgänge u.U. erst zur Laufzeit bei der Ausführung der Schleife bestimmt.
- Normalerweise soll eine Schleife nach endlich vielen Wiederholungen **terminieren**.
- Eine Schleife, die nicht terminiert, heißt **Endlosschleife**.
- Damit eine Bedingungsschleife terminiert, muß irgendwann die Fortsetzungsbedingung **FALSE** bzw. die Abbruchbedingung **TRUE** liefern.
- Die Veränderung der Bedingung muß durch den Rumpf bewirkt werden.

Programmierregeln

- Bei Bedingungsschleifen muß die Schleifenbedingung mindestens eine Variable enthalten, deren Wert innerhalb des Schleifenrumpfs verändert wird.
- Fortsetzungsbedingungen sind möglichst stark, Abbruchbedingungen möglichst schwach zu wählen.
- Zum Abbruch von Endlosschleifen von außen siehe Beispielm modul `TestTestUtilities`.

Beispiel

REPEAT

```
    ' ' '
    index := index + 1
UNTIL index = stopIndex
```

REPEAT

```
    ' ' '
    index := index + 1
UNTIL index >= stopIndex
```

Problem

- Wie kann man nachweisen, daß eine Schleife terminiert?

Schleifenvariante

- Nichtnegativ-ganzzahliger Ausdruck, dessen Wert bei jedem Schleifendurchgang kleiner wird.
- Durch Angabe einer Schleifenvarianten ist garantiert, daß die Schleife terminiert.

Beispiel aus Euklidischem Algorithmus

```
p := a;  
q := b;  
(*  
    VARIANT:  
        MIN (p, q)  
*)  
WHILE q > 0 DO  
    r := p;  
    p := q;  
    q := r MOD p;  
END;
```

Programmierung mit Zusicherungsprozedur

IMPORT

Math, BEC := BasisErrorConstants;

p := a;

q := b;

min := MIN (p, q);

ASSERT

(min >= 0, H.invariantLoop);

WHILE q > 0 **DO**

 r := p;

 p := q;

 q := r **MOD** p;

 ASSERT

 (MIN (p, q) < min,
 BEC.invariantLoopVariant);

 min := MIN (p, q);

 ASSERT

 (min >= 0, BEC.invariantLoop);

END;

11.3.2.4 Bedingungslose Schleifen

LOOP

```
Statements;  
IF breakCondition THEN  
    EXIT  
END;  
Statements  
END;  
StatementAfterExit
```

Semantik

- Die unbedingte Schleife muß eine **EXIT**-Anweisung enthalten, sonst ist es eine Endlosschleife.
- Die **EXIT**-Anweisung ist nicht syntaktisch, aber kontextuell an eine Schleife gebunden, d.h. sie darf nur im Rumpf einer **LOOP**-Schleife auftreten.
- Bei Ausführung der **EXIT**-Anweisung wird die nächste umfassende **LOOP**-Anweisung beendet & mit der darauf folgenden Anweisung fortgefahren.

Problem

- Eine **LOOP**-Schleife kann mehrere **EXIT**-Anweisungen, d.h. mehrere Ausgänge an beliebigen Stellen des Rumpfs besitzen.
- **LOOP-EXIT**-Konstrukte widersprechen einem Grundsatz strukturierter Programmierung, nach dem jedes Kontrollkonstrukt genau einen Eingang (am Anfang) & genau einen Ausgang (am Ende) haben soll.
- **LOOP-EXIT**-Schleifen sind als **WHILE**-Schleifen formulierbar, allerdings u.U. mit zusätzlichen Hilfsvariablen.

Programmierrichtlinien

- **LOOP-EXIT**-Schleifen werden nur in begründeten Ausnahmefällen mit Bedacht verwendet.
- Das **EXIT** wird im Quelltext **fett** geschrieben, um die Absprungstelle hervorzuheben.

Entwurfsmuster

```
OpenDialogBox;  
LOOP  
    ReadInput;  
    IF input = exit THEN  
        EXIT  
    ELSE  
        ProcessInput  
    END;  
END;  
CloseDialogBox;
```

- Wie kann das Entwurfsmuster vereinfacht werden?

11.3.2.5 Zählschleifen

```
FOR i := init TO term BY step DO  
    Statements  
END
```

Regeln

- Der Bezeichner (*i*) ist eine Integervariable, sie heißt **Zähl-** oder **Laufvariable** (*control variable*).
- Die Ausdrücke (*init*, *term*, *step*) sind von Integertypen, die vom Typ der Zählvariablen eingeschlossen sind. Sie heißen **Zählbereichsgrenzen** bzw. **Schrittweite**.
- *step* muß ein konstanter Ausdruck $\neq 0$ sein.
- Fehlt die **BY**-Klausel, so gilt $\text{step} = 1$.

Semantik

```
aux := term;  
i := init;  
IF step > 0 THEN  
    WHILE i <= aux DO  
        Statements;  
        i := i + step  
    END  
ELSE  
    WHILE i >= aux DO  
        Statements;  
        i := i + step  
    END  
END
```

- Beachte, daß das Zählsbereichsende nur einmal am Anfang ausgewertet wird!
- `aux` ist eine implizite Hilfsvariable von einem Typ, der den Typ von `term` einschließt.

Programmierrichtlinien

- Zählschleifen verwendet man da, wo die Anzahl der Wiederholungen vor Schleifenbeginn bekannt ist.
- Die Zählvariable & die Zählbereichsgrenzen dürfen innerhalb des Schleifenrumpfs nicht verändert werden!

Schlechtes Beispiel

```
FOR i := minIndex TO maxIndex DO  
    . . .  
    IF breakCondition THEN  
        i := maxIndex  
    END;  
    . . .  
END
```

- Formulieren Sie diese Schleife als Bedingungs-
schleife!

11.4 Vordeklarierte Prozeduren

11.4.1 Alphabetisch sortierte Liste

- Vollständige Angaben siehe *Component Pascal Language Report* oder Verweise.

ASSERT (x), ASSERT (x, n)

S. 11.4.3.3, S. 31.

DEC (v), DEC (v, n)

Dekrementiert Integerzahl, s. 8.6.2.1, S. 8.

EXCL (v, x)

Entfernt Element aus Menge, s. 8.7.4, S. 17.

HALT (n)

S. 11.4.2, S. 29.

INC (v), INC (v, n)

Inkrementiert Integerzahl, s. 8.6.2.1, S. 8.

INCL (v, x)

Fügt Element zu Menge hinzu, s. 8.7.4, S. 17.

NEW (v), NEW (v, x₀, ..., x_n)

Erzeugt dynamische Variable, später behandelt.

11.4.2 Programmabbruch

HALT (n : Integerkonstante)

- Bricht die Programmausführung ab; öffnet dann ein Trap-Fenster, das n als Trap-Nummer & die Geschichte des Programmlaufs zeigt.
- Im Modul `BasisErrorConstants` sind **Abbruchkonstanten** (*termination numbers*) definiert.

11.4.2.1 Beispiel

```
IMPORT BEC := BasisErrorConstants;  
    . . .  
IF (1 <= mark) & (mark <= 4) THEN  
    Report (mark, succeeded)  
ELSIF mark <= 5 THEN  
    Report (mark, failed)  
ELSE  
    HALT  
    (BEC.invariantUnreachableBranch)  
END
```

11.4.3 Zusicherungen

11.4.3.1 Allgemeine Definition

Zusicherung (*assertion*) :=

Aussage über den Zustand der Daten eines Programms.

11.4.3.2 Bedeutung

- Eine Zusicherung kann überall stehen, wo eine Anweisung stehen kann.
- Erreicht eine Programmausführung eine Zusicherung, so muß diese erfüllt sein, andernfalls ist das Programm fehlerhaft.
- Eine Zusicherung gibt an, welche Teilergebnisse die vorangehenden Anweisungen bewirkt haben & mit welchen Voraussetzungen die nachfolgenden Anweisungen weiterarbeiten.

11.4.3.3 Realisierung

ASSERT (b : BOOLEAN)

ASSERT

(b : BOOLEAN; n : Integerkonstante)

- Falls b, so ist die Prozedur wirkungslos.
- Falls $\sim b$, so bricht der Programmablauf ab; es öffnet sich ein Trap-Fenster ggf. mit n als Trap-Nummer.
- Äquivalent mit

IF $\sim b$ THEN HALT (n) END

11.4.3.4 Beispiel

IMPORT BEC := BasisErrorConstants;

...

ASSERT

((0 ≤ x) & (x ≤ 1),
BEC.invariantExprInRange);

- Im Fehlerfall ist nicht ersichtlich, ob $x < 0$ oder $x > 1$ war. Oder doch?

11.4.3.5 Wozu sind Zusicherungen nütze?

- Sie drücken die mit der Programmierung verfolgten Absichten aus & machen ihre Verwirklichung überprüfbar.
- Sie erhöhen den Dokumentationswert von Programmen.
- Sie unterstützen das Schreiben korrekter, zuverlässiger & fehlertoleranter Programme.
- Sie helfen beim Testen & Fehlersuchen.

11.4.3.6 Programmierrichtlinie

- Programmieren Sie möglichst oft mit Zusicherungen!

Grund: Wer mit Zusicherungen arbeitet, zeigt guten Programmierstil.

Kapitel 12

Prozeduren

Strukturierte Programmierung

Struktogramm

- Was kennen wir inzwischen davon in Component Pascal-Notation?
- Was fehlt noch?
- Was bedeutet *Bezugnahme* auf einen *Strukturblock*?

⇒ **Prozedur (Routine, Unterprogramm)**

Component Pascal:

- **Gewöhnliche Prozedur** (*proper procedure*),
- **Funktion** (*function procedure*), Funktionsprozedur.

12.1 Allgemeines

12.1.1 Kennen wir schon Prozeduren?

Ja,

- *Kommandos* = exportierte parameterlose gewöhnliche Prozeduren.
- Nicht exportierte parameterlose gewöhnliche Prozeduren.
- Von der Sprache Component Pascal *vordekla-*
rierte gewöhnliche Prozeduren & Funktionen.
 - Beispiele: INC, ODD.
- In der Entwicklungsumgebung BlackBox deklarierte Prozeduren; diese gehören immer zu einem bestimmten Modul bzw. sind an einen bestimmten Typ gebunden.
 - Ein-/Ausgabe: In.Open, Out.Real.
 - Mathematik: Math.Eps, Math.Sin.
 - Graphische Benutzungsoberfläche: Text-Models.dir.New, Views.OpenAux.

12.1.2 Was kommt Neues hinzu?

- Durch ProgrammiererIn vereinbarte Prozeduren.
(Beachte Analogie bei Datentypen!)
- Prozeduren
 - ohne/mit Parametern,
 - ohne/mit Resultat.
- Prozeduren mit eigenem Vereinbarungsteil.
- Schachtelung von Prozeduren,
Sichtbarkeitsbereiche.
- Spezifikation von Prozeduren.
- Rekursive Aufrufe.
- Prozedurtypen & -variablen.
- Typgebundene Prozeduren.
- ...

12.1.3 Wozu brauchen wir Prozeduren?

- Zur Kapselung von Algorithmen.
Prozedur = abgeschlossenes Stück Arbeit;
kleinste aufruf- & ausführbare Einheit.
- Zur Gliederung eines Moduls.
Modul = Daten & dazugehörige Prozeduren;
kleinste übersetz- & ladbare Einheit.
- Zur Gliederung einer größeren Prozedur.
 - Bisher: Kommentare.
 - Jetzt: Unterprozeduren.
 - + Übersichtlichkeit,
 - + Sicherheit.
- Zur Vereinfachung eines Programms, wenn gleiche oder ähnliche Anweisungsfolgen auftreten.
 - Bisher: Mehrfach aufschreiben.
 - Jetzt: Einmal als Prozedur aufschreiben, durch Bezugnahme (= Aufruf) mehrfach benutzen.
 - + Wiederverwendbarkeit,
 - + Änderbarkeit.

12.1.4 Merkmale von Prozeduren

- Ist in ein Modul eingebettet;
wichtiger Baustein für Module.
Modul ohne Prozedur = Schwabe ohne Spätzle.
- Besitzt einen *Namen*.
- Besteht aus
 - *Vereinbarungsteil* &
 - *Anweisungsteil*.
- Kann Parameter besitzen.
- Unterscheidung zwischen
 - *Vereinbarung* in einem Vereinbarungsteil,
 - *Aufruf* in einem Anweisungsteil.
- Arten von Prozeduren:
 - *gewöhnliche Prozeduren* tun etwas,
 - *Funktionen* liefern einen Wert.

12.2 Gewöhnliche Prozeduren

Gewöhnliche Prozedur (*proper procedure*) :=
Zusammenfassung von Anweisungen;
Abstraktion des Begriffs Anweisung.

- Vereinbarung durch (spezielle) **Prozedurvereinbarung** (*procedure declaration*).

Bestandteile:

- **Prozedurkopf** (*heading*),
 - ◆ **Prozedurname**,
 - ◆ **formale Parameterliste**,
- **Prozedurrumpf** (*body*),
 - ◆ **Vereinbarungsteil**,
 - ◆ **Anweisungsteil**.

- Aufruf durch **Prozeduranweisung**.

Bestandteile:

- (vollqualifizierter) **Prozedurname**,
- **aktuelle Parameterliste**.

12.2.1 Beispiel

Vereinbarung

```
PROCEDURE Init  
  (OUT x : ARRAY OF REAL; r : REAL);  
  VAR i : INTEGER;  
BEGIN  
  FOR i := 0 TO LEN (x) - 1 DO  
    x [i] := r;  
  END;  
END Init;
```

. . .

```
VAR  
  a : ARRAY 10 OF REAL;  
  b : ARRAY 100 OF REAL;
```

Aufruf

```
Init (a, 1.23);  
Init (b, a [0] + 4.56);
```

12.3 Funktionen

Funktion (*function procedure*) :=
Zusammenfassung eines Ausdrucks;
Abstraktion des Begriffs Ausdruck.

- Vereinbarung durch (spezielle) **Prozedurvereinbarung**.

Bestandteile:

- Funktionskopf,
 - ◆ Funktionsname,
 - ◆ formale Parameterliste,
 - ◆ **Ergebnistyp**,
- Funktionsrumpf,
 - ◆ Vereinbarungsteil,
 - ◆ Anweisungsteil
mit **RETURN-Anweisung**.

- Aufruf durch **Funktionsaufruf**, d.i. ein Faktor in einem Ausdruck.

Bestandteile:

- (vollqualifizierter) **Funktionsname**,
- **aktuelle Parameterliste**.

12.3.1 Beispiel

Vereinbarung

```
PROCEDURE    Mean
    (IN x : ARRAY OF REAL) : REAL;
VAR
    i          : INTEGER;
    result     : REAL;
BEGIN
    result := 0;
    FOR i := 0 TO LEN (x) - 1 DO
        result := result + x [i];
    END;
    RETURN result / LEN (x);
END Mean;
```

...

```
VAR mean : REAL;
```

Aufruf

```
mean := (Mean (a) + Mean (b)) / 2;
```

12.4 Syntax

DeclSeq =

{ **CONST** { *ConstDecl* „;“ } |
TYPE { *TypeDecl* „;“ } | **VAR** { *VarDecl* „;“ } }
{ *ProcDecl* „;“ | *ForwardDecl* „;“ }.

ProcDecl =

PROCEDURE [*Receiver*]
IdentDef [*FormalPars*] „;“ [*MethAttributes*]
DeclSeq [**BEGIN** *StatementSeq*] **END** *ident*.

FormalPars =

„(“ [*FPSection* { „;“ *FPSection* }] „)“
[„:“ *Qualident*].

FPSection =

[**VAR** | **IN** | **OUT**] *ident* { „,“ *ident* } „:“ *Type*.

Statement =

Designator [„(“ [*ExprList*] „)“] |
RETURN [*Expr*] | ...

Factor =

Designator [„(“ [*ExprList*] „)“] | ...

12.5 Semantik

1. Ein Prozeduraufruf aktiviert die Prozedur gleichen Namens. Dabei geht die Ablaufkontrolle von der Aufrufstelle an den Anfang des Anweisungsteils der Prozedur über.
2. Bevor die Anweisungen ausgeführt werden, werden ggf. Parameter übergeben.
3. Der Anweisungsteil der Prozedur wird bis zum Ende bzw. bis zur im Ablauf ersten **RETURN**-Anweisung ausgeführt.
4. Mit dem Ende der Prozedurabarbeitung kehrt die Ablaufkontrolle an die Aufrufstelle zurück.
5. Dabei wird ggf. ein Resultat zurückgegeben.
6. Der Ablauf wird fortgesetzt
 - bei gewöhnlichen Prozeduren mit der auf den Prozeduraufruf folgenden Anweisung,
 - bei Funktionen mit der Auswertung des Ausdrucks, der den Funktionsaufruf enthält.

12.5.1 Die Return-Anweisung

- Gewöhnliche Prozeduren enden **implizit** mit der letzten Anweisung oder **explizit** mit einer **RETURN**-Anweisung ohne Ausdruck.
- Funktionen enden **explizit** mit **RETURN** und einem Ausdruck für das Resultat.
- Gewöhnliche Prozeduren können, Funktionen müssen eine **RETURN**-Anweisung enthalten.

12.5.1.1 Programmierregel

- Der Abbruch einer gewöhnlichen Prozedur mit **RETURN** ist verpönt.
- Wird er in Ausnahmefällen trotzdem eingesetzt, so ist darauf zu achten, daß die Endebehandlung von **VAR**- oder **OUT**-Parametern bzw. von Resultaten nicht übersprungen wird, da undefinierte Ergebnisse unbedingt zu vermeiden sind.

12.6 Empfehlung zur Namengebung

Wie soll ich Prozeduren benennen?

12.6.1 Gewöhnliche Prozedur

- = Zusammenfassung von Anweisungen,
 - + tut etwas, Ausführung bewirkt einen Effekt, eine Zustandsänderung von Daten,
 - liefert aber keinen Wert.
- ⇒ **Name** = Name einer Anweisung = **Verb** (Infinitiv oder Imperativ) oder VerbSubstantiv.

Beispiele

Sort

Sort (list)

SearchList (element, position)

Search (list, element, position)

12.6.2 Funktion

= Zusammenfassung eines Ausdrucks,

+ liefert einen Wert,

- tut aber sonst nichts, Ausführung bewirkt keinen Effekt.

⇒ **Name** = Name eines Ausdrucks = **Substantiv** (Singular), AdjektivSubstantiv oder (mathematische) Abkürzung.

Beispiele

Age (birthday)

GreatestPrimeNumber (100)

GCD (a, b)

Ausnahme: Boole'sche Funktion.

12.6.3 Boole'sche Funktion

= Zusammenfassung eines Boole'schen Ausdrucks, Abstraktion einer Bedingung,

+ liefert einen Boole'schen Wert.

⇒ **Name** = Name einer Boole'schen Größe = **Adjektiv**, Verb (Partizip Perfekt), VerbSubstantiv oder **IS**Substantiv (sonst **IS** weglassen).

Beispiele

Even (10)

HasApple (listOfFruits)

IsPrime (13)

Beispiel

```
IF IsNumber (string) THEN  
    Extend (list, Number (string));  
END;
```

12.7 Datenübergabekonzepte

Parameter, Argument :=

Größe, die zur Datenübergabe zwischen Prozeduren dient.

Begriffe

- Formalparameter, formale Parameter,
- Aktualparameter, aktuelle Parameter,
- Wertparameter,
- Referenzparameter, Ein-/Ausgabeparameter,
- Namensparameter,
- Stellungparameter.

12.7.1 Formale und aktuelle Parameter

Formale Parameter (*formal parameters*)

- treten auf in:
 - Prozedurvereinbarung;
- legen fest:
 - Namen,
 - Typen,
 - Anzahl,
 - Reihenfolge (Stellungsparameter),
 - Art der Übergabe;
- Eigenschaften:
 - Platzhalter für aktuelle Parameter;
 - nur innerhalb der Prozedur existent & sichtbar,
 - innerhalb der Prozedur wie Variable verwendet.

Aktuelle Parameter (*actual parameters*)

- treten auf in:
 - Prozeduraufruf;
- legen fest:
 - Speicherplatz (bei Referenzparametern),
 - Wert;
- Anforderungen:
 - an Aufrufstelle existent & sichtbar,
 - Anzahl, Typen & Reihenfolge passend zu formaler Parameterliste,
 - insbes. zuweisungskompatibel mit formalen Parametern.

Folgerungen

- Für die aufrufende Prozedur sind die Namen der formalen Parameter bedeutungslos ($\Rightarrow$ *keine Namensparameter*).
- Für die aufgerufene Prozedur sind die Namen der aktuellen Parameter bedeutungslos.
- Es kommt nur auf die Typen & die Stellung der Parameter an ($\Rightarrow$ **Stellungsparameter**).
- Aufrufende & aufgerufene Prozedur können weitgehend unabhängig voneinander entwickelt werden.
- Zum Verständnis einer Prozedur genügt es, ihren Effekt bzw. Rückgabewert zu verstehen.
- Von den Details des zugrundeliegenden Algorithmus kann abstrahiert werden.

$\Rightarrow$ **Funktionale bzw. prozedurale Abstraktion**

Was passiert bei der Parameterübergabe?

Parameter-(Übergabe-)Arten

Wertparameter

Referenzparameter

IN-Parameter

OUT-Parameter

IN-OUT-Parameter = VAR-Parameter

12.7.2 Wertübergabe

call by value, value parameter

- Prozedur arbeitet mit lokalen Variablen, die ihre Initialisierungswerte von den aktuellen Parametern erhalten.
- ⇒ Aktuelle Parameter können Ausdrücke sein, die mit formalen Parametern zuweisungskompatibel sind.
- Die aktuellen Parameter werden vor der Aktivierung der Prozedur ausgewertet & den formalen Parametern zugewiesen.
 - Wertparameter treten in der Formalparameterliste nur mit ihrem Namen auf.

Beispiele

Prozeduren, Funktionen.

`Out.Int (3 * i + k)`

12.7.3 Referenzübergabe

call by reference, variable parameter

- Prozedur arbeitet direkt mit den aktuellen Parametern.

⇒ Aktuelle Parameter müssen Variable sein, die mit formalen Parametern zuweisungskompatibel sind.

- Die Speicherplätze der aktuellen Parameter werden vor der Aktivierung der Prozedur bestimmt & die formalen Parameter arbeiten dann mit diesen Speicherplätzen.
- Referenzparameter treten in der Formalparameterliste mit **VAR**, **IN** oder **OUT** vor ihrem Namen auf.

Beispiele

VAR, **IN**, **OUT** bei Prozeduren,
IN bei Funktionen.

Aufrufe

```
In.Int (i)
IsSorted (myArray)
Sort (myArray)
```

Vereinbarungen

```
PROCEDURE IsSorted
  (IN x : ARRAY OF REAL) : BOOLEAN;
BEGIN
  . . .
  RETURN result;
END IsSorted;
```

```
PROCEDURE Sort
  (VAR x : ARRAY OF REAL);
BEGIN
  . . .
  ASSERT (IsSorted (x),
          BEC.postcondition);
END Sort;
```

12.7.4 Resultatübergabe

Rückgabewert, *return value*

- Nur bei Funktionen.
- Der Ausdruck bei der **RETURN**-Anweisung wird ausgewertet & sein Wert an der Stelle des Funktionsaufrufs eingesetzt.
- Der Ausdruck muß mit dem im Funktionskopf festgelegten Ergebnistyp zuweisungskompatibel sein.

Beispiele

```
c := XYplane.Readkey ()  
y := Math.Exp (x)  
lower := MIN (x, y)
```

```
IF F(MAX(x, y)) < M.Sin(a) * G(b)  
THEN ...
```

12.7.5 Spezielle Regeln bei Funktionen

- Eine parameterlose Funktion muß mit leerer Formalparameterliste vereinbart werden.

PROCEDURE

FunctionWithoutArgs () : REAL;

- Eine parameterlose Funktion muß mit leerer Aktualparameterliste aufgerufen werden.

x := FunctionWithoutArgs ();

Grund: Im Zusammenhang mit Prozedurtypen & -variablen (s. 16.1, S. 2) müssen Funktion & Funktionsaufruf unterscheidbar sein.

- Der Ergebnistyp einer Funktion darf kein strukturierter Typ sein, d.h. keine Reihung & kein Verbund.

12.7.6 Programmierrichtlinien

Wo soll ich welche Parameterübergabeart verwenden?

- Falls es nur auf die Anfangswerte ankommt: Wertparameter.
- Falls eine Größe eingegeben, bearbeitet und zurückgegeben werden muß: **VAR**-Parameter (ggf. kombiniert mit Wertparametern).
- Falls es nur auf die Anfangswerte ankommt, aber dies etwas Großes ist (z.B. eine große Reihe): statt Wertparameter **IN**-Parameter.
 - + Effizienz: Kopieren großer Datenmengen vermeiden.
 - + Schutz: Kunde ist sicher, daß Prozedur seine Aktualparameter nicht verändert.
 - + Einsatzbereich: auch in verteilten Systemen.
- Falls etwas Strukturiertes, Großes zurückgegeben werden muß: **OUT**-Parameter.

- Falls nur ein einfacher Wert zurückgegeben werden muß & sonst nichts Globales verändert wird: Funktion mit Resultat & ggf. Wert- und **IN**-Parametern.
- Problematisch sind Funktionen mit Referenzparametern, die zur Rückgabe dienen (**VAR**, **OUT**).

12.8 Existenz und Sichtbarkeit vereinbarter Größen

Größe := Etwas, das vereinbart werden kann.

12.8.1 Lokale und globale Größen

- Eine Größe heißt **lokal** (bzgl. einer Prozedur), wenn sie in der Prozedur vereinbart ist.
- Eine Größe heißt **global** (bzgl. einer Prozedur), wenn sie in dem umfassenden Modul oder einer umfassenden Prozedur vereinbart ist.

12.8.2 Existenzdauern von Variablen

Auch: Lebensdauer.

- Eine in einem Modul vereinbarte Variable ist **existent**, solange das Modul geladen ist.
- Eine lokale Variable ist **existent** für die Dauer der Ausführung der Prozedur, in dem sie vereinbart ist.
- **Folgerung:** Moduldaten überdauern Aufrufe, Prozedurdaten überdauern Aufrufe nicht, sondern werden bei jedem Aufruf neu angelegt.

12.8.3 Sichtbarkeitsbereiche

scope

- Eine Größe ist in einer Prozedur **sichtbar**, wenn sie bzgl. der Prozedur
 - entweder lokal ist
 - oder global ist & es keine lokale Größe gleichen Namens gibt.
- Ein Prozedur ist halbdurchlässig:
 - Alles, was außerhalb der Prozedur (in umfassenden Prozeduren oder im Modul) vereinbart ist, ist innerhalb der Prozedur sichtbar & damit zugreifbar.
 - Alles, was innerhalb der Prozedur (lokal & in lokalen Prozeduren) vereinbart ist, ist außerhalb der Prozedur nicht existent & damit unsichtbar & nicht zugreifbar.
- **Überdeckung:** Bei Vorkommen gleicher Namen verdeckt das Innere das Äußere. Für jeden mehrfach vorkommenden Namen ist die am lokalsten vereinbarte Größe sichtbar.

12.9 Aliasing

- **Aliasing** (Mehrfachbenennung) bedeutet, daß eine Größe unter mehreren Namen zugreifbar ist.
- Siehe Beispielm modul `I1Aliasing`.

12.9.1 Programmierregeln

- Vermeide Aliasing.
- Entscheide für jede globale Größe,
 - ob Prozeduren auf sie zugreifen dürfen sollen,
 - oder ob sie Prozeduren über Parameter zugänglich gemacht wird.

Vermeide die Mischung beider Möglichkeiten in einem Modul.

- Eine Variable soll in einem Prozeduraufruf nicht mehrfach als aktueller **VAR**- oder **OUT**-Parameter auftreten.

12.10 Seiteneffekte

- Funktionsbegriff
 - in der Mathematik,
 - in Programmiersprachen.
- Eine Funktion bewirkt einen **Seiteneffekt** (*side effect*), wenn bei ihrer Ausführung außer der Berechnung des Werts, den sie zurückgeben soll, zusätzlich globale Größen verändert werden.
- Auch Ausdrücke können Seiteneffekte hervorrufen, wenn sie Aufrufe seiteneffektbehafteter Funktionen enthalten.
- Seiteneffekte bei Funktionen können durch
 - Referenzparameter (**VAR**, **OUT**),
 - Zugriffe auf globale Größen,
 - Prozeduranweisungenentstehen.
- **Seiteneffekte sind äußerst problematisch!**
- Siehe Beispielm modul `I1SideEffects`.

- Durch Seiteneffekte können arithmetische Regeln - z.B. das Kommutativgesetz - verletzt werden, & zwar in viel stärkerem Maße als durch die Gleitpunktarithmetik.

12.10.1 Programmierregeln

- Eine Funktion soll stets - bei jedem Aufruf - (wie in der Mathematik) denselben Ergebniswert liefern.

Ausnahmen z.B.:

`XYplane.ReadKey,`
`MathRandom.UniformI.`

- Vermeide Seiteneffekte!
- Seiteneffekte sind strengstens verboten!

Bemerkung

Es gibt Programmiersprachen, bei denen Seiteneffekte integrales Konzept sind (z.B. C/C++).

Für Component Pascal ist ein seiteneffektfreier Programmierstil deutlich vorzuziehen.

12.11 Spezifikation von Prozeduren

Spezifikation einer Prozedur :=

exakte Beschreibung dessen, *was* die Prozedur leistet.

Implementation einer Prozedur :=

Prozedurrumpf, d.h. lokale Daten & Algorithmus; legt fest, *wie* die Prozedur die von ihr geforderte Leistung erbringt.

Syntaktische Spezifikation =

Prozedurkopf.

Semantische Spezifikation = ?

12.11.1 Merkmale von Spezifikationen

- Eine Spezifikation ist unabhängig von einem speziellen Algorithmus.
- Es gibt i.a. viele Implementationen/Algorithmen, die eine Spezifikation erfüllen.
- Eine Spezifikation umfaßt die Information,
 - die ein Benutzer der Prozedur zu ihrer Benutzung wissen muß, ohne auf den Prozedur-rumpf schauen zu müssen;
 - ein Programmierer für das Implementieren der Prozedur wissen muß.

12.11.2 Das Kunden-Lieferanten-Modell

- Eine Prozedur ist ein **Lieferant** (*server, supplier*) einer Dienstleistung.
- Ein Aufrufer einer Prozedur ist ein **Kunde** (*client*), der diesen Dienst in Anspruch nehmen möchte.
- Ein Lieferant stellt **Vorbedingungen** (*preconditions*) an den Kunden.
 - Was muß der Kunde erfüllen, damit der Dienst erbracht werden kann?
 - Welchen Bedingungen müssen die Eingabegrößen genügen?
- Ein Kunde stellt **Nachbedingungen** (*postconditions*) an den Lieferanten.
 - Was muß der Lieferant leisten, damit der Dienst als erbracht gilt?
 - Welchen Bedingungen müssen die Ausgabegrößen (in Abhängigkeit von den Eingabegrößen) genügen?
- Vor- & Nachbedingungen sind als Boole'sche Ausdrücke bzw. Zusicherungen formulierbar.

12.11.3 Spezifikation mit Vor- und Nachbedingungen

Beispiel

IMPORT

```
M      := Math,  
BEC    := BasisErrorConstants;
```

```
PROCEDURE Arcsin (x : REAL) : REAL;
```

```
  VAR result : REAL;
```

BEGIN

```
  ASSERT
```

```
    (ABS (x) <= 1,  
     BEC.precondPar1InRange);
```

```
  result :=
```

```
    M.Arctan (x / M.Sqrt(1 - x*x));
```

```
  ASSERT
```

```
    (ABS (M.Sin (result) - x) <=  
     M.Eps () * ABS (x),  
     BEC.postcondResultInRange);
```

```
  RETURN result;
```

```
END Arcsin;
```

Kapitel 13

Definierbare Datentypen

Datentypen

Grundtypen

numerische: INTEGER, REAL,...

andere: CHAR, BOOLEAN, SET

Reihungen **ARRAY**

Verbunde **RECORD**

einfache
Datentypen

strukturierte
Datentypen

vordefinierte
Datentypen

definierbare
Datentypen

13.1 Typkonstruktoren und Typangaben

- Wie erhalten wir ausgehend von Grundtypen & Werten neue Datentypen?

13.1.1 Beispiele für Konstruktionen

ARRAY *n* **OF** **BOOLEAN** Reihung

Typangabe

ganzzahlige Konstante

Typkonstruktor

Typangabe

RECORD Verbund

articleNbr : INTEGER;

price : REAL;

END

Typangaben

Typkonstruktor

Typangabe

13.1.2 Arten von Typangaben

- Einfacher Typ (gegeben durch Typnamen).
- Strukturierter Typ.
- Prozedurtyp.
- Zeigertyp.
- Typname.

Syntax siehe 9.3.2

13.1.3 Anonyme Typen

- Mit einer Typangabe ist ein neuer Datentyp eingeführt.
- Solange er keinen Namen hat, ist es ein **anonymer Typ**.

13.1.3.1 Programmierregel

- Vermeide anonyme Typen!
Gründe:
 - Sicherheit des Programms.
 - ◆ Deutlich machen, daß Typgleichheit von Größen gewollt & nicht zufällig ist.
 - ◆ Übersetzer überprüft Typkompatibilität.
 - Änderbarkeit des Programms.
Änderungen nur an einer Stelle, bei der Vereinbarung, nicht bei der Verwendung des Typs.
- **Ausnahmen:**
 - Gibt es nur eine Variable eines Typs, so kann der Typ anonym bleiben.
Beispiel: Eine einzelne Reihungsvariable.
 - Offene Reihungen.

13.2 Typnamen und Typvereinbarungen

- Wie geben wir einem neuen Datentyp einen Namen?

13.2.1 Beispiel für Namengebung

Vector = **ARRAY** length **OF** REAL

Typname

Typangabe

Typvereinbarung

- Für Typnamen gelten dieselben
 - Vorschriften wie für alle anderen Namen,
 - Regeln für Sichtbarkeit wie für Variablen.
- Wo bringen wir die Typvereinbarungen unter?
Im Vereinbarungsteil eines Moduls oder einer Prozedur!

Syntax siehe 9.3.2, Semantik siehe 9.3.3

13.3 Strukturierte Datentypen

13.3.1 Allgemeine Eigenschaften

- Strukturierte Typen sind aus anderen Datentypen - **Komponenten** - zusammengesetzt.
- Bei einer Variablen von strukturiertem Typ ist zu unterscheiden, ob
 - das Exemplar als *Ganzes* oder
 - eine einzelne *Komponente* gemeint ist.
- Variable eines strukturierten Typs werden als Ganzes vereinbart. Damit sind zugleich alle Komponenten vereinbart.
- *Rekursive Konstruktion:*
Die Komponenten strukturierter Typen können selbst strukturierte Typen sein.

13.3.2 Reihungen

13.3.2.1 Beispiele für Typangaben

ARRAY maxNbr **OF** Item

ARRAY colorNbr, placeNbr **OF** State

ARRAY 1 **OF**

ARRAY 2 **OF**

ARRAY 3 **OF**

ARRAY 4 **OF**

ARRAY 5 **OF**

ARRAY 6 **OF**

ARRAY 7 **OF** Element

ARRAY **OF** CHAR

ARRAY **OF**

RECORD

x, y : REAL;

END

13.3.2.2 Definition

- Eine **Reihung** (Feld, *array*) besteht aus einer
 - *Anzahl*
 - *gleicher Komponenten* (**Elemente**),
 - die in einer *Folge angeordnet* sind.

- Bei einer Variablen (Exemplar) einer Reihung ist die Anzahl der Elemente *unveränderlich festgelegt*:
 - bei einer statischen Variablen in der Vereinbarung, also *zur Übersetzungszeit*;
 - bei einer dynamischen Variablen bei der Erzeugung, also *zur Laufzeit* (siehe später).

- In einer Typangabe einer Reihung kann die Angabe der Anzahl der Elemente fehlen; man spricht dann von einer **offenen Reihung** (*open array*).

13.3.2.3 Typangabe

- Zur *Typangabe* einer Reihung gehört
 - ein beliebiger **Elementtyp**,
 - ggf. die Anzahl der **Dimensionen**, d.h. eine natürliche Zahl $n > 0$,
 - ggf. die **Länge** - d.h. die Anzahl der Elemente - in der Dimension i , d.h. n natürliche Zahlen $l_i > 0$ ($i = 0, \dots, n - 1$).
- Der neu definierte Typ heißt **Reihungstyp**.

13.3.2.4 Syntax

ArrayType =

ARRAY [*ConstExpr* { „,“ *ConstExpr* }]
OF *Type*.

- Die Dimensionszahl einer Reihung ist implizit durch die Anzahl der in der Typangabe aufgeführten Ausdrücke festgelegt.
- **ARRAY** $l_0, l_1, \dots, l_n$ **OF** T
ist eine Abkürzung für
ARRAY l_0 **OF** **ARRAY** $l_1 \dots l_n$ **OF** T .

13.3.2.5 Eigenschaften von Reihungen als Ganzes

- Der *Wertebereich* einer Reihung ist das m -fache kartesische Produkt des Wertebereichs des Elementtyps, wobei m das Produkt der Längen in allen Dimensionen ist ($m = l_0 * \dots * l_{n-1}$).
- *Zulässige Operationen* auf Reihungen als Ganzes:
 - Wertzuweisungen,
 - Parameterübergabe.
- Die *vordeklarierte Funktion*
 `LEN (arr, dim)`
liefert die Länge der Reihung `arr` in der Dimension `dim`.
 `LEN (arr)`
ist äquivalent mit
 `LEN (arr, 0)`.

Beispiel

TYPE

```
Vector =  
    ARRAY length OF Numeric;
```

VAR

```
x, y, z : Vector;
```

PROCEDURE

```
AddVector  
    (IN x, y : Vector;  
     OUT z   : Vector);
```

```
...
```

```
x := y;
```

```
...
```

```
AddVector (x, y, z);
```

```
...
```

```
ASSERT (LEN (x) = length);
```

13.3.2.6 Eigenschaften von Elementen von Reihungen

- Der *Wertebereich* eines Elements ist natürlich durch den Elementtyp festgelegt.
- Jedes Element ist *explizit benennbar & direkt zugreifbar*.
- Der *Zugriff* auf ein Element erfolgt über den Namen der Reihung, ergänzt um Indizes. Ein **Index** ist ein ganzzahliger Ausdruck.
- Die Elemente einer Reihungsvariablen `arr` werden in jeder Dimension `dim` von 0 bis `LEN (arr, dim) - 1` numeriert.

Beispiele - welche sind korrekt, welche nicht?

`x [1]`

`y [3 * i + n]`

`TrafficLights.light [red, north]`

`x [-1]`

`y [LEN (y)]`

- Diese Art des Zugriffs heißt **Indizierung**. Die Elemente einer Reihungsvariablen heißen auch **indizierte Variable**.

Syntax

Designator = *Qualident* { „[“ *ExprList* „]“ | ... }.

- Ein Indexwert muß im Bereich
 $0 \dots \text{LEN}(\text{arr}, \text{dim}) - 1$
liegen; andernfalls liegt eine **Bereichsüberschreitung** (*index out of range*) vor.
- BlackBox prüft Bereichsüberschreitungen zur Laufzeit. Die Semantik ist, als ob vor jedem Zugriff der Art
`a [i]`
eine Zusicherung der Art
`ASSERT ((0 <= i) & (i < LEN(a)))`
steht.
 - + Erkennt Fehler, erhöht Sicherheit.
 - Kostet etwas Laufzeit.
 - Nicht von jeder Sprachumgebung geboten.
 - Beispiel siehe Modul `I1ArrayOverflow`.

- Ein Indexwert (bzw. Tupel von Indexwerten) bestimmt eineindeutig ein Reihungselement, für das tatsächlich Speicherplatz reserviert ist.
- Da Indizes ganzzahlig sind, sind **Indexberechnungen** möglich.
- *Zulässige Operationen* auf Elementen von Reihungen:
 - Alle Operationen, die für den Elementtyp zulässig sind, also insbesondere
 - ◆ Wertzuweisungen,
 - ◆ Parameterübergabe.
- $a [i_0, i_1, \dots, i_n]$
ist eine Abkürzung für
 $a [i_0] [i_1] \dots [i_n]$.

Beispiele

```

a := x [i + 1];
y [k - 2] := b;
c := x [3 * i] + y [k MOD 4];
z [i] := MAX (x [i], y [i]);
light [red, north] := on;

```

13.3.2.7 Eindimensionale Reihungen

- Der programmiersprachliche Begriff der eindimensionalen Reihung entspricht dem mathematischen Begriff des **Vektors**.
- *Vektorrechnung*: Arithmetische Operationen, Vergleichsoperationen usw. mit Reihungen numerischen Elementtyps sind nicht direkt möglich. Dafür sind Prozeduren bereitzustellen (siehe Übungsaufgabe).
- Reihungen können elementweise verarbeitet, eingelesen & ausgeschrieben werden, wie es der Elementtyp erlaubt. Die (meist) passende Anweisung ist die **FOR**-Schleife, bei der die Laufvariable als Index dient.

Beispiel

```
ASSERT (LEN (x) = LEN (y));  
FOR i := 0 TO LEN (x) - 1 DO  
    sum := sum + x [i] * y [i];  
    equal := equal & (x [i] = y [i]);  
    In.Real (x [i]);  
    Out.Real (y [i], 0);  
END;
```

13.3.2.8 Zweidimensionale Reihungen

- Der programmiersprachliche Begriff der zweidimensionalen Reihung entspricht dem mathematischen Begriff der **Matrix**.
- *Matrizenrechnung*: Wie bei eindimensionalen Reihungen ist nur elementweise Verarbeitung möglich. Am besten steckt man diese in Prozeduren. Siehe Beispielprogramme.
- „Gleichwertigkeit“ von Typvereinbarungen & Bezeichnern indizierter Variablen siehe Beispiel nächste Seite & Modul `I1ArrayEquivalence`.
- Beachte: Gleichwertigkeit von Typvereinbarungen bedeutet *nicht* Typidentität! Dazu später mehr.

Beispiel für Gleichwertigkeit

TYPE

```
Light1 =  
    ARRAY colorNbr, placeNbr OF  
        State;  
Light2 =  
    ARRAY colorNbr OF  
        ARRAY placeNbr OF State;  
PlaceState =  
    ARRAY placeNbr OF State;  
Light3 =  
    ARRAY colorNbr OF PlaceState;
```

VAR

```
light1 : Light1;  
light2 : Light2;  
light3 : Light3;  
...  
  
light1 [red, east]      := off;  
light2 [green, west]   := flashing;  
light3 [yellow, south] := on;  
light3 [red] [north]   := off;
```

13.3.2.9 Mehrdimensionale Reihungen

- Mehrdimensionale Reihungen sind eine Verallgemeinerung von zweidimensionalen Reihungen.

Beispiel

TYPE

```
Line =  
    ARRAY nbrOfChars OF CHAR;  
Page =  
    ARRAY nbrOfLines OF Line;  
Book =  
    ARRAY nbrOfPages OF Page;  
Location =  
    ARRAY nbrOfBooks OF Book;  
Library =  
    ARRAY nbrOfLocations OF  
        Location;
```

VAR

```
    library : Library;  
  
...  
library [0, 1, 2, 3, 4] := "?";
```

13.3.2.10 Offene Reihungen

- Offene Reihungen dürfen nur auftreten
 - als Typen formaler Parameter,
 - als Basistypen von Zeigern (siehe später),
 - als Elementtypen offener Reihungen.

Beispiel

PROCEDURE

```
IsSorted (IN x : ARRAY OF REAL) :  
    BOOLEAN;
```

...

PROCEDURE

```
Sort (VAR x : ARRAY OF REAL);
```

...

BEGIN

...

```
    ASSERT (IsSorted (x));
```

```
END Sort;
```

- Die Prozeduren können mit **REAL**-elementigen Reihungen beliebiger Länge als aktuelle Parameter aufgerufen werden.
- Sie erfahren die Länge des aktuellen Parameters mittels **LEN (x)**.

13.3.3 Zeichenketten

13.3.3.1 Definition

- Eine **Zeichenkette** (*string*) besteht aus einer
 - *variablen* (d.h. zur Laufzeit veränderlichen) *Anzahl*
 - *gleicher Komponenten*,
 - die vom Typ **CHAR** &
 - in einer Folge angeordnet sind.
 - Die Anzahl der Komponenten, d.h. die **aktuelle Länge** der Zeichenkette, ist durch eine sog. **Maximallänge** beschränkt.

13.3.3.2 Realisierung

- In Component Pascal gibt es keinen eigenen Datentyp für Zeichenketten, nur Konstanten.
- Zeichenketten werden durch Reihungen
ARRAY maxLength **OF** **CHAR**
ARRAY OF **CHAR**
realisiert.

13.3.3.3 Unterschiede zwischen ...

... normalen Reihungen

- Reihungen verarbeitet man hauptsächlich elementweise.
- Alle Elemente werden genutzt.
- Verkettung von Reihungen interessiert kaum.

... und Zeichenketten

- Zeichenketten verkettet oder zerlegt man hauptsächlich.
- Die aktuelle Länge variiert, es werden nur die Anfangselemente genutzt, die restlichen Elemente bis zur Maximallänge bleiben ungenutzt.
- Verknüpfung einzelner Elemente interessiert kaum.

13.3.3.4 Zeichenkettenvariablen

VAR string : **ARRAY** maxLen **OF** CHAR;

- Eine **Zeichenkettenvariable** (string) wird als Ganzes vereinbart. Damit wird Speicherplatz für alle Elemente bis zur Maximallänge (maxLen) reserviert.
- Alle Zeichen sind von 0 bis maxLen - 1 durchnummeriert.
- Die aktuelle Länge der Variablen kann dynamisch im Bereich 0 .. maxLen - 1 variieren.
- Das Zeichen 0X dient als **Endezeichen** (*terminator*) für die aktuelle Zeichenkette.
- Steht also beim Index i das erste 0X-Zeichen,
 string [i] = 0X,
dann hat string die aktuelle Länge i &
besteht aus den Zeichen
 string [0],..., string [i - 1].

13.3.3.5 Bearbeitung von Zeichenketten

- *Zulässige Operationen* auf Zeichenketten als Ganzes:
 - Wertzuweisungen,
 - Parameterübergabe,
 - Vergleichsoperationen,
 - Verkettung (mit Operator +).

- Mit obiger Vereinbarung gilt:
 `string$`
bezeichnet die null-terminierte Zeichenkette in der Reihung `string`. Der `$`-Selektor wird implizit angewandt, wenn `string` als Operand des Verkettungsoperators, von Vergleichen oder Typanpassungen erscheint.

- Das *BlackBox-Bibliotheksmodul* `Strings` enthält Prozeduren zur Zeichenkettenverarbeitung. Erkunden Sie ihn mit dem Browser!

Vereinbarungen

TYPE

String = **ARRAY 6 OF** CHAR;

VAR

s : **ARRAY 6 OF** CHAR;

t : **ARRAY 12 OF** CHAR;

z : String;

PROCEDURE Ps (s : **ARRAY 6 OF** CHAR);

...

PROCEDURE Pz (z : String);

...

PROCEDURE Po (o : **ARRAY OF** CHAR);

...

PROCEDURE

Pv (**VAR** v : **ARRAY OF** CHAR);

Wertzuweisungen - welche sind korrekt?

s := "Informatik";

t := "Praktikum";

z := "";

s := t;

z := s;

t := z;

z := s;

Parameterübergaben - welche sind korrekt?

Ps (s);

Pz (z);

Po (t);

Pv (s);

Ps (t);

Pz (s);

Po (z);

Pv (t);

Vergleichsoperationen - welche sind korrekt?

s = "Informatik"

t # "Praktikum"

z <= "Pruefung"

"Anf" < "Anfang"

"Anfang " >= "Anfang"

"Ende" > "Anfang"

- Verglichen wird zeichenweise von links nach rechts.
- Die Ordnung der Zeichen im Unicode/ASCII-Zeichensatz liegt zugrunde.
- Ist die kürzere Kette gleich dem Anfang der längeren Kette, so ist die kürzere kleiner.
- Zwei Ketten sind gleich, wenn sie gleich lang sind & in allen Zeichen übereinstimmen.

13.3.4 Verbunde

13.3.4.1 Beispiele für Typangaben

RECORD

```
re ,  
im : REAL
```

END

RECORD

```
location : ARRAY 2 OF REAL;  
state    : BYTE;
```

END

RECORD

```
name      : ARRAY 30 OF CHAR;  
address  :
```

RECORD

```
street,  
location : ARRAY 30 OF CHAR;
```

END;

END

13.3.4.2 Definition

- Ein **Verbund** (*record*) besteht aus einer
 - *festen* (d.h. zur Übersetzungszeit festgelegten) *Anzahl*
 - von *Komponenten* (**Felder**, Attribute, *fields*),
 - die *beliebig* (d.h. nicht notwendig vom gleichen Typ) sind.
 - Die Speicheranordnung kann sich von der Reihenfolge in der Typangabe unterscheiden.
- Beachte den Unterschied zwischen
 - *array* = Feld = Reihung;
 - *field* = (Verbund-)Feld = Attribut!

13.3.4.3 Typangabe

- Zur *Typangabe* eines Verbundes gehört eine Liste seiner Felder, deren Elemente aus
 - **Feldname** &
 - **Feldtyp**bestehen.
- Der neu definierte Typ heißt **Verbundtyp**. Die Komponentenliste heißt **Feldlistenfolge** (*field list sequence*).

13.3.4.4 Syntax

Type =

RecAttributes **RECORD** [„(“ *BaseType* „)“]
FieldList { „;“ *FieldList* } **END** | ...

FieldList = [*IdentList* „:“ *Type*].

13.3.4.5 Eigenschaften von Verbunden als Ganzes

- Der *Wertebereich* eines Verbundes ist das kartesische Produkt der Wertebereiche der Feldtypen.
- Der programmiersprachliche Begriff des Verbundes entspricht dem mathematischen Begriff des **kartesischen Produkts** beliebiger Mengen.
- *Zulässige Operationen* auf Verbunden als Ganzes:
 - Wertzuweisungen,
 - Parameterübergabe.

Beispiel

TYPE

```
Complex =  
  RECORD  
    re ,  
    im : REAL  
  END;
```

VAR

```
x, y, z : Complex;
```

PROCEDURE AddComplex

```
(IN x, y : Complex;  
  OUT z   : Complex);
```

```
...
```

```
x := y;
```

```
...
```

```
AddComplex (x, y, z);  
ASSERT (z.re = x.re + y.re);  
ASSERT (z.im = x.im + y.im);
```

13.3.4.6 Eigenschaften von Verbundfeldern

- Jede Verbundkomponente ist *explizit benennbar* & *direkt zugreifbar*.
- Der *Zugriff* auf ein Feld erfolgt über den Namen des Verbundes, ergänzt um den Namen des Felds.

Beispiele

x.re

y.im

point.location

person.address.street

- Diese Art des Zugriffs heißt **Selektion**. Der Name

recordname.fieldname

heißt **Selektor**.

- *Zulässige Operationen* auf Verbundfeldern:
 - Alle Operationen, die für den Feldtyp zulässig sind, also insbesondere
 - ◆ Wertzuweisungen,
 - ◆ Parameterübergabe.

Beispiele

```
a := x.re;
```

```
y.im := b;
```

```
c :=  
  Math.Sqrt  
    (x.re * x.re + x.im * x.im);
```

13.3.4.7 Sichtbarkeitsregeln für Feldnamen

- Feldnamen sind *lokal* bezüglich des definierenden Verbundes.
- Sie können daher
 - nur untereinander konfligieren,
 - aber nicht mit anderen Namen außerhalb des Verbundes oder mit Feldnamen anderer Verbunde, weil sie wegen des Verbundnamens eindeutig sind.
- Feldnamen können *Exportmarken* (*, -) haben.
- Markierte Felder heißen **öffentlich** (*public*), nur sie sind in Kundenmodulen sichtbar.
- Nichtmarkierte Felder heißen **privat** (*private*), sie sind in Kundenmodulen unsichtbar.
- Wird ein Feldname exportiert, so sollte der Verbundname auch exportiert werden.
- Selektiver Export ist ein wichtiges Mittel zur Datenabstraktion!

Beispiel

MODULE Supplier;

TYPE

Public* = ...;

Readable* = ...;

Private = ...;

Record* =

RECORD

public* : Public;

readable- : Readable;

private : Private;

END;

END Supplier.

MODULE Client; **IMPORT** Supplier;

VAR

public : Supplier.Public;

readable : Supplier.Readable;

~~private : Supplier.Private;~~

record : Supplier.Record;

...

record.public := public;

readable := record.readable;

~~record.private := private;~~

13.4 Typgleichheit

- Beachte: *Derselbe Typ & der gleiche Typ* bedeuten nicht dasselbe!
- Exakte Definitionen von
 - Variablen **desselben** Typs (*of the same type*);
 - **gleiche** Typen (*equal types*)siehe Extrablätter bzw. *Component Pascal Language Report*.

13.4.1 Lasche Definition

- Zwei Typen sind **gleich** (*equal*), wenn sie
 - auf denselben Typnamen zurückführbar,
 - oder offene Reihungen mit gleichen Elementtypen,
 - oder Prozedurtypen mit gleichen Signaturensind.

13.4.2 Wozu ist Typgleichheit nütze?

- Es müssen bei
 - Zuweisungen: linke & rechte Seite,
 - Parameterübergabe: formaler & aktueller Parametervom gleichen Typ sein.
- Der Übersetzer überprüft die Einhaltung der Typregeln.
 - + Typfehler zur Übersetzungszeit erkannt.
 - + Mehr Sicherheit zur Laufzeit.
- Spezialfall: Typeinschlußregeln bei numerischen Typen.

13.4.3 Beispiele

INTEGER

REAL

- nicht gleich;
- u.U. erfolgt Typanpassung.

TYPE

```
Int  = INTEGER;
Itg  = Int;
T1   = ARRAY 2 OF INTEGER;
T2   = ARRAY 2 OF Int;
T3   = ARRAY 2 OF Itg;
T4   = T1;
T5   = T2;
T6   = T3;
T7   = ARRAY 2 OF INTEGER;
```

VAR

```
i    : INTEGER;
int  : Int;
itg  : Itg;
v1   : T1;
v2   : T2;
v3   : T3;
v4   : T4;
v5   : T5;
v6   : T6;
v7   : T7;
v8   : ARRAY 2 OF INTEGER;
```

...

```
i      := int;           i := itg;
int    := i;             int := itg;
vi    := vk; für welche i, k zulässig?
```

Kapitel 14

Sortiervverfahren

Was soll sortiert werden?

- Elemente eines Datentyps mit Ordnungsrelation ' $\leq$ ' (vergleichbarer Typ, Comparable).

Wo soll sortiert werden?

- Zu ordnende Elemente in einem Feld gespeichert, Änderungen direkt „am Ort“, „in situ“.

Was sind die elementaren Aktionen?

- Zwei Elemente vergleichen.
- Zwei Elemente vertauschen.
- Ein Element verschieben (auf freie Stelle).

14.1 Einfache Verfahren

- Austauschen, *exchangesort*, *bubblesort*
Austauschen mit Umkehren, *shakersort*
- Auswählen, *selectionsort*
- Einfügen, *insertionsort*
- Mischen, *mergesort* (kein in-situ-Verfahren)

14.2 Höhere Verfahren

- Quicksort
- Heapsort
- Shellsort

Literaturhinweis

Niklaus Wirth: *Algorithmen und Datenstrukturen*.

14.3 Aufwandsabschätzung

Maße für Laufzeiteffizienz von Sortierv Verfahren:

C	Anzahl erforderlicher Vergleiche
M	Anzahl erforderlicher Umstellungen
n	Anzahl der Elemente der Reihung

$$C = C(n)$$

$$M = M(n)$$

Laufzeit:

$$T = T(n) \sim C(n) + M(n)$$

Für einfache Verfahren:

$$T(n) = c * n^2$$

Für gute Verfahren:

$$T(n) = c * n * \log n$$

Unterscheidung:

minimaler, mittlerer, maximaler Aufwand

Verfahren	C_{mit}	M_{mit}
Austauschen	$(n^2 - n) / 2$	$3 * (n^2 - n) / 4$
Auswählen	$(n^2 - n) / 2$	$n * (\ln n + g)$
Einfügen	$(n^2 + n - 2) / 2$	$(n^2 + 9n - 10) / 4$
Quicksort	$c_q * n * \log n$	$c_q * (n * \log n) / 6$
Heapsort	$c_h * (n * \log n)$	
Shellsort	$c_s * n^{1.2}$	

Anmerkungen

$g = 0,577216\dots$, Euler'sche Konstante

Schrittweiten bei Shellsort: 1, 3, 7, 15, 31,...

$$c_h \cong 2.5 * c_q$$

Kapitel 15

Modularisierung und Datenabstraktion

Fragen

- Wie behandelt man „große“ Programme?
- Wie kann man Programmteile wiederverwenden (ohne zu kopieren)?
- Wie können mehrere Entwickler gleichzeitig an einem Programmsystem arbeiten?

Einheiten zur Strukturierung - traditionell

- Programme, Unterprogramme.
- Dateien.
- 1 Programm - 1 Datei.

Ideen

- (a) Programme und Dateien sind voneinander unabhängige Einheiten.
- (b) Ein Programm auf mehrere Dateien aufteilen, die Dateien strukturieren.
- (c) Neue Programmstrukturen einführen.

15.1 Konzept der Modularisierung

- Das Quellprogramm wird in Module zerlegt, die getrennt übersetzbar sind.
- Ein Modul erhält einen Namen und einen eigenen Namensraum (*name space*) für die im Modul gekapselten Merkmale (*encapsulation*).
- Ein Modul besteht aus einem Schnittstellenteil (*interface*) und einem Implementierungsteil (*implementation*).
- Die Schnittstelle enthält Größen, die an andere Module exportiert werden und von diesen benutzt werden können.
- Die Implementierung enthält Größen, die der Realisierung der exportierten Merkmale dienen, aber für andere Module verborgen, nicht sichtbar, nicht zugreifbar sind (*information hiding*).

Wie soll eine Modulschnittstelle vernünftigerweise aussehen?

15.2 Konzept der Datenabstraktion

- Die Schnittstelle eines Moduls enthält nur
 - Vereinbarungen von Konstanten und Typen,
 - Köpfe von Routinen.Es werden keine Daten exportiert!
- Exportiert werden nur Routinen, sie heißen auch **Operationen**; ein Modul mit einer solchen Schnittstelle heißt **operationales Modul**. Die exportierten Konstanten und Typen sind nur zur Beschreibung der Parameter der Operationen notwendig.
- Die Implementierung eines Moduls enthält
 - Vereinbarungen und
 - die Rümpfe der exportierten Routinen.
- Die Moduldaten sind im Implementierungsteil verborgen, das Modul ist eine **Datenkapsel**.
- Auf die Moduldaten kann von außen nicht direkt zugegriffen werden, sondern nur über Aufrufe der Modulooperationen.

- Die Datenstruktur, die das Modul realisiert, ist abstrakt durch die Wirkung seiner Operationen definiert, deshalb heißt sie **abstrakte Datenstruktur**.

15.3 Modulare Programmierung

- Entwerfe ein Programm als strukturierte Ansammlung von operationalen Modulen,
- programmiere diese Module.

15.4 Getrennte Übersetzbarkeit

- Programm besteht aus mehreren Übersetzungseinheiten.
- Die Schnittstellen zwischen den Übersetzungseinheiten sind zur Übersetzungszeit bekannt.
- Der Übersetzer überprüft die Einhaltung der Schnittstellen zur Übersetzungszeit, z.B. Anzahl und Typen der Parameter, wie bei einem einzelnen Programm.

Literaturverzeichnis

Ist ein Buch in der Bibliothek des Studiengangs Elektronik vorhanden,
so ist seine Signatur mit angegeben.

Verlage

Addison-Wesley	Addison-Wesley Publishing Company, Reading, Massachusetts Addison-Wesley (Deutschland) GmbH, Bonn
Hanser	Carl Hanser Verlag, München, Wien
Oldenbourg	R. Oldenbourg Verlag GmbH, München, Wien
Springer	Springer-Verlag, Berlin, Heidelberg, Wien, Barcelona, Budapest, Hong Kong, London, New York, Paris, Tokyo, u.a.
Teubner	B. G. Teubner, Stuttgart
Vieweg	Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, Braunschweig, Wiesbaden

Nachschlagewerke

- [Eng93] Herrmann Engesser (Hrsg.): *Duden Informatik. Ein Sachlexikon für Studium und Praxis*. Dudenverlag, Mannheim (1993) 2. vollst. überarbeit. u. erweit. Auflage, 800S (HB El: I 1-20-49)
- [GrI93] Franz Grieser, Thomas Irlbeck: *Computer-Lexikon. Das Nachschlagewerk zum Thema EDV*. dtv 50302 (1993) 779S (HB El: I 1-20-86)
- [Kle89] Martin Klein: *Einführung in die DIN-Normen*. Teubner (1989) 10. Auflage, 1028S (HB El: MA 1-15-5)
- [LMW92] Helmut Löffler, Jürgen Meinhardt, Dieter Werner: *Taschenbuch der Informatik. Hardware - Software - Anwendungen*. Fachbuchverlag Leipzig (1992) 515S
Der Stoff ist inhaltlich gegliedert, jedes Teilgebiet ist zusammenhängend in einem Abschnitt beschrieben.
- [ReP99] Peter Rechenberg, Gustav Pomberger (Hrsg.): *Informatik-Handbuch*. Hanser (1999) 2. aktual. u. erweit. Auflage, 1166S (HB El: I 1-20-95)
Empfehlenswertes Nachschlagewerk mit Beiträgen von 47 Experten zu Themen aus Theoretischer, Technischer, Praktischer, Angewandter und Wirtschaftsinformatik sowie zu Daten, Normen und Spezifikationen.

Informatik - Grundlagen und Einführungen

- [ApL92] Hans-Jürgen Appelrath, Jochen Ludewig: *Skriptum Informatik - eine konventionelle Einführung*. Teubner (1992) 2. Auflage, 448S

- [Bal99] Helmut Balzer: *Lehrbuch Grundlagen der Informatik. Konzepte und Notationen in UML, Java und C++, Algorithmik und Software-Technik, Anwendungen*. Spektrum Akademischer Verlag, Heidelberg, Berlin (1999) 927S (HB El: I 3-100-124)
Führt mittels Java in Themen der Programmierung ein.
- [Bro92] Manfred Broy: *Informatik. Eine grundlegende Einführung. Teil I. Problemnahe Programmierung*. Springer (1992) 250S (HB El: I 1-10-129)
Erläutert Konzepte applikativer Programmiersprachen unabhängig von einer speziellen Sprache.
- [Bro93] Manfred Broy: *Informatik. Eine grundlegende Einführung. Teil II. Rechnerstrukturen und maschinennahe Programmierung*. Springer (1993) 195S (HB El: I 1-10-129)
Befasst sich auch mit Schaltnetzen und -werken.
- [BSSS92] J. Blieberger, G.-H. Schildt, U. Schmid, S. Stöckler: *Informatik*. Springer (1992) 2. neubearbeit. Auflage, 380S
Bietet keine Programmier Einführung, sondern behandelt u.a. theoretische Grundlagen, logische Schaltungen, Maschinencodes und Betriebssysteme.
- [Ern00] Hartmut Ernst: *Grundlagen und Konzepte der Informatik. Eine Einführung in die Informatik ausgehend von den fundamentalen Grundlagen*. Vieweg (2000) 822S (HB El: I 1-10-145)
Führt in die Themen Information, Codierung, Schaltalgebra, Digitaltechnik, Rechnerarchitekturen, Betriebssysteme, Programmiersprachen, Softwareentwicklung, Automatentheorie, formale Sprachen, Algorithmen, Datenstrukturen, Kommunikations- und Informationstechnik ein.
- [GmR93] Rainer Gmehlich, Heinrich Rust: *Mehr als nur Programmieren... Eine Einführung in die Informatik*. Vieweg (1993) 233S (HB El: I 1-10-126)
Behandelt nach einer Programmier Einführung in Pascal auch den Problemkreis Korrektheit - Fehlerhaftigkeit - Validierung von Programmen.
- [GoL90] Les Goldschlager, Andrew Lister: *Informatik. Eine moderne Einführung*. Hanser (1990) 3. bearbeit. u. erweit. Auflage, 366S (HB El: I 1-10-28)
Stellt den Begriff des Algorithmus in den Mittelpunkt, behandelt Entwurf, Theorie, Hard- und Software-Grundlagen sowie Anwendungen.
- [GuS94] Heinz-Peter Gumm, Manfred Sommer: *Einführung in die Informatik*. Addison-Wesley (1994) 579S (HB El: I 1-10-127)
Enthält eine Einführung in Turbo Pascal und deckt Algorithmen, Datenstrukturen, Rechnerarchitekturen, Betriebssysteme und Softwareentwicklung ab.

- [Kla91] Herbert Klaeren: *Vom Problem zum Programm. Eine Einführung in die Informatik*. Teubner (1991) 2. neubearbeit. u. erweit. Auflage, 240S (HB El: I 1-60-11)
Eine schöne, kompakte, fundierte und pascalorientierte Einführung in das Programmieren, die schnell zum Begriff des abstrakten Datentyps kommt. (Der Untertitel übertreibt etwas.)
- [MaL98] Fritz Mayer-Lindenberg: *Eine Einführung in die Informatik*. Vieweg (1998) 320S
- [Pep92] Peter Pepper: *Grundlagen der Informatik*. Oldenbourg (1992) 355S
Eine moderne Einführung jenseits von Pascal und anderer imperativer Programmiersprachen, die von mathematischen Grundlagen der Programmierung und den Ideen der Funktionsapplikation und Rekursion ausgeht.
- [PMK94] Precht, Meier, Kleinlein: *EDV-Grundwissen*. Addison-Wesley (1994) 2. Auflage, 320S
- [PSG94] Jörg Puchan, Wolffried Stucky, Jürgen Frhr Wolff von Gudenberg: *Programmieren mit Modula-2. Grundkurs Angewandte Informatik I*. Teubner (1994) 2. überarbeit. Auflage, 318S (HB El: I 1-10-128)
- [Rec94] Peter Rechenberg: *Was ist Informatik? Eine allgemeinverständliche Einführung*. Hanser (1994) 2. bearbeit. u. erweit. Auflage, 349S (HB El: I 1-10-111)
Wie [ReL99] eine Umschau, aber nach den Teilgebieten Technische, Praktische, Theoretische und Angewandte Informatik gegliedert.
- [ReL99] Ulrich Rembold, Paul Levi: *Einführung in die Informatik für Naturwissenschaftler und Ingenieure*. Hanser (1999) 3. vollst. überarbeit. u. erweit. Auflage, 610S (HB El: I 1-10-89)
Schlägt den Bogen von den mathematischen Grundlagen über Datenstrukturen, Algorithmen und Programmiersprachen bis hin zu Softwaretechnik, Rechnerarchitekturen, Betriebssysteme und Prozessautomatisierung.
- [Rol94] Dieter Roller: *Informatik. Grundlagen. Mit einer Einführung in Pascal*. Springer (1994) 224S (HB El: I 1-10-132)
- [RSS93a] Reinhard Richter, Peter Sander, Wolffried Stucky: *Problem - Algorithmus - Programm. Grundkurs Angewandte Informatik II*. Teubner (1993) 290S (HB El: I 1-10-128)
Stellt die Konzepte imperativer Programmiersprachen unabhängig von einer speziellen Sprache vor.
- [RSS93b] Reinhard Richter, Peter Sander, Wolffried Stucky: *Der Rechner als System - Organisation, Daten, Programme. Grundkurs Angewandte Informatik III*. Teubner (1993) 250S
- [SLA94] André Spiegel, Jochen Ludewig, Hans-Jürgen Appelrath: *Aufgaben zum Skriptum Informatik*. Teubner (1994) 2. durchgeseh. Auflage, 192S (HB El: I 1-10-115)

Allgemeines zu Programmiersprachen

- [BaG94] Henri Bal, Dick Grune: *Programming Language Essentials*. Addison-Wesley (1994) 288S (HB El: I 3-300-35)
- [Lou93] Kenneth C. Loudon: *Programming Languages*. PWS (1993) 592S (HB El: I 3-300-34)
Deutsche Ausgabe unter dem Titel „Programmiersprachen“.

Programmieren mit Oberon und Component Pascal

- [Hug00] Karlheinz Hug: *Module, Klassen, Verträge. Ein Lehrbuch zur komponentenorientierten Softwarekonstruktion mit Component Pascal*. Vieweg (2000) 446S (HB El: I 4-300-213)
- [MLK95] Jörg R. Mühlbacher, Bernhard Leisch, Ulrich Kreuzeder: *Programmieren mit Oberon-2 unter Windows. Mit kompletter Entwicklungsoberfläche Pow! und vielen Beispielprogrammen auf Diskette*. Hanser (1995) 353S (HB El: I 3-390-121)
Eine Einführung in Oberon-2, basierend auf der Entwicklungsumgebung Pow! von der Universität Linz, die sich von BlackBox dadurch unterscheidet, dass sie keine automatische Speicherplatzbereinigung enthält und komponentenorientierte Konzepte nicht unterstützt.
- [Mös94] Hanspeter Mössenböck: *Objektorientierte Programmierung in Oberon-2*. Springer (1994) 2. Auflage, 286S (HB El: I 3-390-81)
Eine Einführung in objektorientiertes Programmieren; setzt Programmiererfahrung und Kenntnisse von Pascal, Modula-2 oder Oberon voraus.
- [Nik98] Eric Nikitin: *Into the Realm of Oberon. An Introduction to Programming and the Oberon-2 Programming Language*. Springer (1998) 199S (HB El: I 3-390-356)
Geeignet, um blutige Anfänger bei ersten Gehversuchen zu stützen.
- [ReW94] Martin Reiser, Niklaus Wirth: *Programmieren in Oberon. Das neue Pascal*. Addison-Wesley (1994) 338S (HB El: I 3-390-95)
Ein einführendes Programmierlehrbuch für Oberon mit einem Kapitel über Oberon-2; gut für Anfänger geeignet.
- [War00] Stanley Warford: *Programming with BlackBox*. <ftp://ftp.pepperdine.edu/pub/compsci/prog-bbox/> (2000)

Programmieren mit anderen Sprachen

- [Kam97] Peter Kammerer: *Von Pascal zu Assembler*. Eine Einführung in die maschinennahe Programmierung für Intel und Motorola. Vieweg (1997) 280S (HB El: I 3-370-23)

Datenstrukturen und Algorithmen

- [Her92] Dietmar Herrmann: *Algorithmen Arbeitsbuch*. Addison-Wesley (1992) 415S (HB El: I 3-700-56)
Enthält 300 Algorithmen, darunter leider auch nicht strukturiert programmierte. Zum Nachschlagen geeignet.
- [Sed89] Robert Sedgewick: *Algorithms*. Addison-Wesley (1989) 2nd edition, 660S (HB El: I 3-700-20)
Gibt es in einer Pascal- und einer C-Variante sowie in deutschen Ausgaben. Alle enthalten mindestens ein fehlerhaftes Programm sowie nicht strukturiert programmierte Algorithmen - also Vorsicht beim Abschreiben!
- [Sta93] Thomas A. Standish: *Data Structures, Algorithms and Software Principles*. Addison-Wesley (1993) 748S (HB El: I 3-600-27)
- [Wir86] Niklaus Wirth: *Algorithmen und Datenstrukturen mit Modula-2*. Teubner (1986) 4. neubearbeit. u. erweit. Auflage, 299S (HB El: M 3-300-184)
Ein Klassiker des Schöpfers von Pascal, bis zur 3. Auflage mit Pascal-Programmen, dann mit Programmen in der Nachfolgersprache Modula-2 geschrieben.
- [Wir93] Niklaus Wirth: *Systematisches Programmieren. Eine Einführung*. Teubner (1993) 6. Auflage, 160S (HB El: I 3-100-26)
Noch ein Klassiker, den man gelesen haben muss.