Prof. Dr. Hans-Werner Six

# Kurs 01613

# Einführung in die imperative Programmierung

**LESEPROBE**

Fakultät für
**Mathematik und Informatik**

FernUniversität in Hagen

# Allgemeine Studierhinweise

Liebe Fernstudentin, lieber Fernstudent!

Der vorliegende Kurs 1613 „Einführung in die imperative Programmierung“ steht am Anfang des Bachelorstudiengangs Informatik. Wenn Sie den Kurs durchgearbeitet haben, sollten Sie kleinere, gut strukturierte und wohl-dokumentierte Pascal Programme bzw. Prozeduren selbständig entwickeln und elementare Verfahren der analytischen Qualitätssicherung auf einfache Beispiele anwenden können.

Gliederung

Der Kurs gliedert sich organisatorisch in 5 Kurseinheiten:

KE I: Grundlagen; Programmierkonzepte orientiert an Pascal (Teil 1)
KE II: Programmierkonzepte orientiert an Pascal (Teil 2)
KE III: Programmierkonzepte orientiert an Pascal (Teil 3)
KE IV: Programmierkonzepte orientiert an Pascal (Teil 4);
Exkurs: Prozedurale Programmentwicklung;
KE V : Analytische Qualitätssicherung

Exkurse

Als *Exkurs* bezeichnete Kapitel gehören nicht zum Pflichtpensum des Kurses. Die Ausführungen dienen lediglich dem Abrunden des Wissens interessierter Kursteilnehmer. Zu Exkursen gibt es weder Einsendeaufgaben noch Aufgaben in den abschließenden Klausuren.

Der Kurs besteht aus verschiedenfarbigen Teilen: Gelbe Seiten enthalten allgemeine Studierhinweise, der eigentliche Lehrtext steht auf weißen Seiten und die blauen Seiten sind für die Lösungen der Übungsaufgaben im Kurstext reserviert.

Viele Studierende haben gerade im ersten Semester Schwierigkeiten, mit dem Lehrstoff zurechtzukommen. Dies ist ganz normal, da zu Beginn des Studiums eine geeignete Arbeitstechnik erst noch entwickelt werden muss. Obwohl eine erfolgreiche Arbeitstechnik bei verschiedenen Studierenden unterschiedlich ausfallen kann, haben sich doch im Laufe der Zeit einige Erfolgsfaktoren herausgebildet:

1. Probieren Sie alle Programme auf Ihrem eigenen Rechner aus.
2. Arbeiten Sie alle Beispiele des Kurses intensiv durch (versuchen Sie, einen Fehler zu finden, oder denken Sie sich weitere neue Beispiele aus).
3. Versuchen Sie, alle Übungsaufgaben des Kurses zu lösen.
4. Versuchen Sie, alle Einsendeaufgaben zu lösen.
5. Beteiligen Sie Sich an der Newsgroup des Kurses (vgl. Informationsschreiben zum Kurs).
6. Arbeiten Sie möglichst in einem Team von 3 - 5 Personen, das sich in regelmäßigen Abständen trifft.
7. Besuchen Sie die Studientage.
8. Besuchen Sie regelmäßig ein Studienzentrum (sofern sich ein Studienzentrum in Ihrer Nähe befindet und der Kurs dort mentoriell betreut wird).

Literatur

Es kann auch sinnvoll sein, zusätzliche Literatur zu Rate zu ziehen. Eine vom Kurs abweichende Darstellung des Stoffs kann zum besseren Verständnis beitragen. Als Ergänzung undzur Vertiefung des Kursmaterials empfehlen wir:

[OtW04] Th. Ottmann, P. Widmayer: Programmierung mit Pascal, Vieweg+Teubner 2004

Dieses Buch führt in das Programmieren mit Pascal ein. In ihm können Sie insbesondere alle Pascal-Konstrukte vollständig und im Detail nachlesen. Es ist gut als Nachschlagewerk geeignet.

Zur Ergänzung und Vertiefung der analytischen Qualitätssicherung eignet sich:

[Lig02] P. Liggesmeyer: Software-Qualität, Spektrum Akademischer Verlag, 2002

Grundsätzlich kann der Kurs aber auch ohne zusätzliche Bücher erfolgreich bearbeitet werden.

Überarbeitete Kursversion

Noch ein abschließender Hinweis zur vorliegenden Kursversion: Dieser Kurs wurde zuletzt zum Wintersemester 2008/2009 überarbeitet. Die Überarbeitungen betreffen die Kapitel 1 (Einleitung), 2 (Grundlagen), 6.2 (Rekursion) und vor allem die gesamte fünfte Kurseinheit (Qualitätssicherung). Wegen dieser partiellen Überarbeitung werden Sie im Kurs teilweise noch auf alte, teilweise auf neue deutsche Rechtschreibung stoßen. Die Änderungen bestehen im Wesentlichen aus Korrekturen, Aktualisierungen sowie Verbesserungen und Ergänzungen der Stoffdarstellung. Klausurzulassungen aus Vorsemestern behalten ihre Gültigkeit, jedoch wird insbesondere in den Klausuren die Kenntnis der neuen Kursversion vorausgesetzt.

Bei der Bearbeitung des Kurses wünschen wir Ihnen viel Erfolg.

Ihre Kursbetreuer

# Inhaltsverzeichnis

# Kurseinheit I

# 1. Einleitung

## 1.1 Informatik, Computer, Algorithmus, Programm

In diesem Kurs wählen wir einen Zugang zur Informatik über die Programmierung von Computern. In einer ersten Annäherung an die Informatik greifen wir auf ihre Begriffsbestimmung zurück und stellen dazu eine pragmatische Variante vor:

Informatik

*Informatik* ist die Wissenschaft von der systematischen Verarbeitung von Informationen mit Hilfe von Computern.

Computer

Der zentrale Begriff ist hier der Begriff des *Computers*. Was verbirgt sich hinter einem Computer, der in so viele Bereiche unseres Lebens eindringt und mit der Informatik eine Wissenschaft hervorgebracht hat, die sich mit seinem Aufbau, seinen Eigenschaften und Möglichkeiten auseinandersetzt? Grundsätzlich ist ein Computer eine Maschine, die Daten speichern und auf ihnen einfache Operationen mit hoher Geschwindigkeit ausführen kann. Die Einfachheit der Operationen (typische Beispiele sind die Addition und der Vergleich zweier Zahlen) wird ausgeglichen durch die enorme Geschwindigkeit, mit der sie vollzogen werden (Milliarden von Operationen pro Sekunde). Diese Eigenschaften erlauben dem Computer, ein weites Spektrum von Aufgaben zu bearbeiten.

Natürlich kann ein Computer nur solche Aufgaben bearbeiten, die durch derart einfache Operationen erledigt werden können. Um einen Computer dahin zu bringen, dass er eine bestimmte Aufgabe bearbeitet, müssen wir ihm mitteilen, welche Operationen er dazu in welcher Reihenfolge ausführen soll. Wir müssen also beschreiben, *wie* die Aufgabe auszuführen ist. Ganz allgemein nennen wir ein Verfahren, das durch eine Folge von Anweisungen beschrieben ist, deren schrittweise Abarbeitung eine vorgegebene Aufgabe erledigt, einen *Algorithmus*. Beachten Sie, dass zwischen der *Beschreibung* und der *Ausführung* eines Algorithmus genau unterschieden wird.

Algorithmus

Algorithmen treten übrigens nicht nur als Lösungsverfahren für Computer auf, sondern begegnen uns überall im täglichen Leben. Ihre Anweisungen müssen dabei nicht notwendig den primitiven Operationen eines Computers entsprechen, sie können z.B. auch in natürlicher Sprache formuliert sein. Einige Beispiele für Algorithmen sind Kuchenbacken, Kleidernähen oder eine Hifi-Anlage anschließen. Die *Beschreibung des Algorithmus* ist beim Kuchenbacken in Form des Rezeptes festgehalten, beim Kleidernähen liegt sie als Schnittmuster vor und bei der Hifi-Anlage besteht sie aus der entsprechenden Passage in der Bedienungsanleitung. Eine *Anweisung des Algorithmus* ist dann beim Kuchenbacken z.B. „3 Eier schaumig schlagen“, beim Kleidernähen z.B. das Anfertigen einer Naht und bei der Installation der Hifi-Anlage z.B. das Anschließen des CD-Spielers an denVerstärker über ein Kabel, das in die entsprechenden Gerätebuchsen gesteckt wird. Die Durchführung des Backvorganges gemäß vorliegendem Rezept, das Anfertigen eines Kleides gemäß Schnittmuster und das Installieren einer Hifi-Anlage der Bedienungsanleitung folgend sind dann jeweils eine *Ausführung des Algorithmus*.

Beschreibung des Algorithmus

Anweisung des Algorithmus

Ausführung des Algorithmus

Der Computer ist eine Maschine, die Algorithmen ausführt. Dazu müssen die Algorithmen in einer Sprache abgefasst werden, die der Computer „versteht“. Eine solche Sprache nennen wir *Programmiersprache* und einen in dieser Sprache verfassten Algorithmus *Programm*. Die *Anweisungen einer modernen Programmiersprache* sind viel mächtiger und für den Menschen komfortabler zu benutzen als die primitiven Operationen eines Computers. Allerdings müssen diese Anweisungen erst in primitive Computeroperationen übersetzt werden, bevor der Computer sie ausführen kann. Die Übersetzung geschieht durch ein Programm. Ist dieses Übersetzungsprogramm erst einmal (von Menschen) entwickelt, sorgt der Computer von da an durch dessen vorgeschaltete Ausführung selbst dafür, dass er die Anweisungen der Programmiersprache „versteht“ und damit ausführen kann.

Programmiersprache

Programm

Ein Computer besitzt spezielle Vorzüge, sonst hätte er nicht derart rasch einen bedeutenden Einfluss auf so viele Lebensbereiche nehmen können. Zu diesen *positiven Eigenschaften* gehören Geschwindigkeit, Speicherfähigkeit, Zuverlässigkeit und Universalität.

1. *Geschwindigkeit*

   Geschwindigkeit

   Ein moderner Computer kann in einer Sekunde Milliarden von Operationen ausführen. Obwohl diese Operationen sehr einfach sind, führt die hohe Durchführungsgeschwindigkeit dazu, dass selbst komplexe Algorithmen mit einer großen Anzahl von auszuführenden Operationen im Allgemeinen sehr schnell abgearbeitet werden können.

   Allerdings bleiben trotz der hohen Computergeschwindigkeit Probleme, deren Lösungsalgorithmen zu viele Operationsausführungen erfordern, um praktisch durchführbar zu sein. Als Beispiel sei eine Gewinnstrategie beim Schachspielen genannt, die auf der Durchmusterung aller möglichen Spielkombinationen bei jedem Zug beruht.

2. *Speicherfähigkeit*

   Speicherfähigkeit

   Ein weiteres Hauptmerkmal des Computers ist seine Fähigkeit, große Datenmengen zu speichern, auf die er sehr schnell zugreifen kann. Speicherkapazität und Zugriffsgeschwindigkeit auf einzelne Speicherpositionen variieren dabei stark in Abhängigkeit vom Speichermedium. Einige Speichermedien können mehrere Milliarden Dateneinheiten speichern, auf die zum Teil in weniger als 10 Nanosekunden zugegriffen werden kann (eine Nanosekunde ist ein Milliardstel einer Sekunde).

   Allerdings sind Speicher so organisiert, dass auf eine Dateneinheit nur dann zugegriffen werden kann, wenn ihre Position (*Adresse*) im Speicher bekannt ist. Die Speicheradresse festzustellen, unter der die Dateneinheit abgelegt ist, bedeutet u.U. einen erheblichen Aufwand, der sich in der Ausführungszeit des Algorithmus niederschlagen kann.

   Speicheradresse

3. *Zuverlässigkeit*

   Zuverlässigkeit

   Entgegen vieler populärer Behauptungen machen Computer selbst sehr selten Fehler. Zwar kann ein elektronischer Fehler einen Computer veranlassen, ein Programm unkorrekt auszuführen, aber die Wahrscheinlichkeit hierfür ist gering und normalerweise werden solche Fehlfunktionen sofort aufgedeckt.

Fehlerhafte Ergebnisse oder gar ein „Abstürzen“, d.h. ein unkontrollierter Abbruch während des Programmlaufs, sind daher fast immer auf fehlerhafte Algorithmen oder Programme zurückzuführen.

Universalität

4. *Universalität*

   Der sicherlich größte Vorzug des Computers besteht in seiner Universalität. Die entscheidende Idee, nicht nur Daten sondern auch das Programm im selben Speicher abzulegen, wird John von Neumann zugeschrieben, der sie im Jahr 1945 gehabt hat. Dadurch kann ein Programm ebenso leicht wie gewöhnliche Daten geändert oder ausgetauscht werden. Diese Universalität, also die Fähigkeit, die verschiedensten Probleme ohne technische Umrüstung, sondern lediglich durch Einspeichern geeigneter Programme zu lösen, ist der wichtigste Grund für die enorme Vielseitigkeit und hohe Verbreitung des Computers.

   Jeder Computer kann damit potentiell alle Probleme lösen, für die ein Algorithmus existiert. Praktisch sind allerdings der Universalität Grenzen gesetzt. So kann der Programmtext so viele Anweisungen umfassen, dass dieser nicht mehr in den Speicher passt, und/oder seine Ausführung zu zeitintensiv sein. Letzteres ist beispielsweise etwa dann der Fall, wenn das Ergebnis erst dann feststeht, wenn es bereits bedeutungslos ist. Außerdem gibt es Probleme, für deren Lösung nachweislich überhaupt kein Algorithmus existiert. Diese Probleme werden Ihnen in den Kursen der Theoretischen Informatik wiederbegegnen.

## 1.2 Rolle des Kurses 1613 in der Programmierausbildung

Zum Abschluss der Einleitung noch ein paar Worte zu unserer Philosophie der universitären Programmierausbildung.

Wenn jemand ein Studium der Informatik beginnt, hat er gewisse Vorstellungen, die von seinem persönlichen Erfahrungshorizont geprägt sind. Dieser beruht zumeist auf dem Umgang mit bestimmten Computern und Programmiersprachen. Die daraus resultierenden Vorstellungen sind erfahrungsgemäß häufig etwas einseitig und manchmal sogar irreführend.

Bei der Programmierausbildung (und nicht nur da) legen wir Wert auf ein konzeptuelles Verständnis und weniger auf rasch veraltendes Spezialwissen, denn man muss sich auch nach dem Studium permanent mit neuen Herausforderungen auseinandersetzen. Im Informatik-Berufsfeld langfristig zu bestehen, erfordert daher eine universitäre Ausbildung, die zum einen lehrt, wie man lernt, und zum anderen kurzfristige Zeitströmungen überdauernde Konzepte bereitstellt, deren Allgemeingültigkeit den Einstieg in neue Techniken erleichtert.

Nun stellt sich kein gefestigtes konzeptuelles Verständnis ohne praktische Übung ein. Diese unumgängliche praktische Übung erzeugt als Nebeneffekt ein technisches Basiswissen, das auch noch aus einem anderen Grund wichtig ist. Schließlich kann jeder spätere Arbeitgeber von den Absolventen eines Informatikstudienganges in gewissen Umfang aktuelle technische Basiskenntnisse erwarten. Es sei jedoch an die Adresse der Industrie ein klares Wort gerichtet: Wer

das immer wieder in Anzeigen geforderte Spezialwissen erfüllt, mag vielleicht unmittelbar einsetzbar sein, hat aber später häufig Mühe, in anderen Einsatzgebieten oder bei gravierenden Neuerungen in seinem Spezialgebiet gleichwertige Leistungen zu erbringen. Auch führt eine zu sehr produktorientierte Ausbildung ohne fundierte Vermittlung der dahinterstehenden Grundlagen zu einer mangelhaften Beurteilungsfähigkeit und kritikloser Übernahme der Werbeversprechen der Anbieterindustrie. Ein Absolvent eines stärker an Grundlagen und Konzepten orientierten Studiums benötigt zwar eine gewisse Anlaufzeit, bis er mit einem Spezialisten konkurrieren kann, wird aber mittelfristig einen erheblich größeren Gewinn für das Unternehmen bedeuten.

Die skizzierten Vorstellungen über eine universitäre Programmierausbildung schlagen sich natürlich im Inhalt und Aufbau dieses Kurses nieder. Zunächst einmal sind Qualität und Quantität des Kursmaterials so ausgelegt, dass auch absolute Programmierneulinge – nach universitärem Maßstab – mit ihm zurechtkommen sollten. Studierende mit Programmiererfahrung können sich mit dem Stoff zum Teil deutlich leichter tun. An vielen Universitäten wird derzeit mit der objektorientierten Programmierung und z.B. der Programmiersprache Java begonnen. Unserer Meinung nach ist ein angemessenes Verständnis *objektorientierter Programmierkonzepte* in einem (5 Leistungspunkte, 2+1 SWS) Anfängerkurs kaum zu vermitteln. Dieser Kurs beginnt deshalb mit elementaren *imperativen Konzepten* (mit denen sich Anfänger bereits durchaus schwer tun können). Dabei achten wir darauf, dass die vorgestellten imperativen Konzepte und Techniken den Blick für die objektorientierte Programmierung nicht verstellen. Anschließend wird im zweiten Semester der Kurs 01618 *Einführung in die Objektorientierte Programmierung* (Programmiersprache Java) angeboten.

imperative und objektorientierte Programmierung

Als Programmiersprache haben wir Pascal gewählt, da sie die einfachste uns bekannte und allgemein verfügbare Programmiersprache ist, mit der sich die vorgestellten Konzepte sauber konkretisieren und praktisch anwenden lassen. Die Sprache ist gut strukturiert, ihre Syntax über Syntaxdiagramme wohldefiniert, und Pascal-Programme sind vergleichsweise gut lesbar. Insbesondere für das Fernstudium hat sich Pascal als erster Einstieg in die Programmierung bewährt, denn Pascal ist leicht zu erlernen und ein (Turbo) Pascal- System kann für den privaten PC billig beschafft und ohne fremde Hilfe leicht bedient werden. Es ist jedoch auf keinen Fall unser Anliegen, Ihnen die Programmiersprache Pascal mit allen ihren Möglichkeiten vollständig nahezubringen. Vielmehr werden wir uns auf diejenigen Pascal-Konstrukte konzentrieren, die zur praktischen Anwendung der vorgestellten elementaren Programmierkonzepte notwendig sind. Die behandelten Konzepte finden sich in allen aktuellen imperativen und den meisten objektorientierten Sprachen (wie z.B. Java) wieder.

Pascal
im Kurs 1613

Einen weiteren Schwerpunkt legen wir auf Aspekte, die für die Entwicklung qualitativ hochwertiger Programme von Bedeutung sind. Wir halten es für wichtig, dass die Anwendung entsprechender Methoden von klein auf, d. h. parallel zu den ersten Programmierversuchen, eingeübt wird. Hierzu gehören ein guter *Programmierstil* sowie *Testverfahren* als Teil der *analytischen Qualitätssicherung*, die das Aufspüren von Fehlern in Programmen zum Ziel haben.

Fortgeschrittene Programmierausbildung

Natürlich muss ein Informatiker erheblich weitergehende Programmierkenntnisse und -fertigkeiten besitzen als in diesem Kurs vermittelt werden. Dafür werden Kurse wie z.B. 01618 *Einführung in die Objekorientierte Programmierung* (Java), 01816 *Logisches und funktionales Programmieren* (Prolog, Scheme) sowie der weiterführende Kurs 01853 *Moderne Programmiertechniken und -methoden* angeboten. Zusätzlich gibt es das *Programmierpraktikum*, in dem Programmierfertigkeiten eingeübt werden. Wie qualitativ hochwertige Software für komplexe Problemstellungen entwickelt wird, ist Gegenstand der Kurse über *Software Engineering*. Im *Praktikum Software Engineering* werden diese Kenntnisse auf ein größeres Beispiel angewendet und das Arbeiten im Team geübt.

# Lernziele zum Kapitel 2

Nach diesem Kapitel sollten Sie

1. die Begriffe „Problembeschreibung“ und „Problemspezifikation“ gegeneinander abgrenzen und die einzelnen Teile einer Problemspezifikation erläutern können,

2. die Begriffe „Algorithmus“ und „Programm“ erläutern und voneinander abgrenzen können,

3. Maschinensprachen, Assemblersprachen und höhere Programmiersprachen erläutern und voneinander abgrenzen können,

4. das prozedurale imperative und das objektorientierte Programmierparadigma kennen und vom deklarativen Programmierparadigma abgrenzen können,

5. mit wenigen Worten die Funktion von Steuerwerk und Rechenwerk erklären und verschiedene Speichertypen voneinander abgrenzen können,

6. Anwendungssoftware vom Betriebssystem abgrenzen und die Hauptaufgaben des Betriebssystems erläutern können.

# 2. Grundlagen

## 2.1 Problem und Algorithmus

Ausgangspunkt jeder Problemlösung, ob sie nun manuell oder mit Hilfe des Computers erfolgt, ist zunächst das Problem selbst, das wir als *Realweltproblem* bezeichnen wollen. Durch eine sprachliche Fassung, verbunden mit einem Abstraktionsschritt, der überflüssige Details durch Konzentration auf die für das Problem relevanten Tatbestände eliminiert, erhalten wir eine *(informelle) Problembeschreibung* in textueller Form.

Realweltproblem

(informelle) Problembeschreibung

Betrachten wir beispielsweise das Realweltproblem der computergestützten Verwaltung der Girokonten von Privatkunden einer Bank. Für die Problembeschreibung sind Eigenschaften der Kunden wie Haarfarbe, Haustierbesitzer, Biertrinker, sympathisch oder unsympathisch unwesentlich. Für das Problem relevant sind dagegen beispielsweise Kontonummer, Monatseinkommen, Überziehungskredit und Adresse des Kunden. Daneben sind die auf den Daten auszuführenden Operationen wichtig, wie z.B. Buchung vornehmen, Kontoführungskosten berechnen oder Kontoauszug erstellen.

Wesentlich ist, dass die Problembeschreibung festlegt, **was** getan werden soll.

Die Problembeschreibung stellt einen ersten, wichtigen Schritt bei der Bearbeitung einer Aufgabe dar. Allerdings ist sie häufig noch zu unpräzise abgefasst, d. h. sie ist oft noch mehrdeutig, widersprüchlich und unvollständig. Durch Präzisierung und Formalisierung erhalten wir aus der Problembeschreibung eine *Problemspezifikation*. Auch die Problemspezifikation legt wiederum nur fest, was getan werden soll. Ausgehend von der Problemspezifikation wird dann ein *Lösungsalgorithmus* entwickelt, der beschreibt, **wie** das Problem gelöst wird.

Problemspezifikation
Lösungsalgorithmus

Das folgende Beispiel illustriert die Bedeutung der Problemspezifikation. Insbesondere zeigt sich, dass die Arbeit, die wir in die Erstellung der Problemspezifikation investieren, sich bei dem Finden des Lösungsweges auszahlen kann.

Beispiel 2.1.1

***Informelle Problembeschreibung:***

Auf einem Parkplatz stehen PKW und Motorräder ohne Beiwagen. Zusammen seien es $n$ Fahrzeuge mit insgesamt $r$ Rädern. Es soll die Anzahl $P$ der PKW bestimmt werden.

Bevor wir mit einer genaueren Betrachtung dieses Problems beginnen, zunächst eine Vorbemerkung: In Wirklichkeit wird hier eine ganze *Klasse von Problemen* beschrieben, nämlich je ein gesondertes Problem für jede mögliche Wahl von $n$ und $r$. Wir sagen auch: Die Problembeschreibung enthält $n$ und $r$ als *Parameter*. Das Vorhandensein solcher Parameter ist typisch für Probleme, zu denen Algorithmen entwickelt werden sollen. Es ist wenig interessant, einen Algorithmus zur Multipli-

Problemklasse

Parameter

parametrisiertes Problem

kation von 12 mit 253 zu entwickeln. Das *parametrisierte Problem* der Multiplikation von $a$ mit $b$ für beliebige Dezimalzahlen $a$ und $b$ dagegen lohnt es, aus allgemeiner Sicht betrachtet zu werden.

Nach dieser Vorbemerkung kehren wir zu unserem Problem zurück.

Wir überlegen, dass offensichtlich die Anzahl $P$ der PKW plus der Anzahl $M$ der Motorräder die Gesamtzahl $n$ der Fahrzeuge ergibt. Außerdem wissen wir, dass jeder PKW 4 Räder und jedes Motorrad 2 Räder hat und dass die Radzahlen der PKW und Motorräder zusammen $r$ ergeben müssen. Wir haben es also mit folgendem Gleichungssystem zu tun:

$$(1) \quad P + M = n$$

$$(2) \quad (4P + 2M) = r$$

Wir lösen Gleichung (1) nach $M$ auf:

$$M = n - P$$

und setzen in Gleichung (2) ein:

$$\begin{aligned} & 4P + 2(n - P) = r \\ \Leftrightarrow \quad & 2P = r - 2n \\ \Leftrightarrow \quad & P = \frac{r - 2n}{2} \end{aligned}$$

An dieser Stelle sind wir versucht, das Problem als gelöst zu betrachten, und könnten jetzt nach dieser Formel ein Computerprogramm schreiben. Allerdings erhalten wir dann für die Eingabe $n = 3$ und $r = 9$ die Rechnung:

$$P = \frac{9 - 2 \cdot 3}{2} = \frac{3}{2} = 1,5$$

Auf dem Parkplatz müssten also anderthalb PKW stehen, was keinen Sinn ergibt. Aber das ist noch nicht alles, wie die Rechnung für $n = 5$ und $r = 2$ zeigt:

$$P = \frac{2 - 2 \cdot 5}{2} = \frac{2(1 - 5)}{2} = 1 - 5 = -4$$

Die Antwort wäre also so etwas wie *„Es fehlen vier PKW“,* was auch unsinnig ist. Tatsächlich sind nur positive, ganzzahlige $P$ sinnvoll, d.h. $P$ muss eine *natürliche Zahl* sein.

Bei der Suche nach einer Lösung sind wir offensichtlich etwas zu schnell vorgegangen und haben einige wichtige Voraussetzungen bzw. Nebenbedingungen gar nicht berücksichtigt. Zum Beispiel ergibt die Aufgabe nur einen Sinn, wenn die Anzahl $r$ aller vorkommenden Räder gerade ist, da es keine Fahrzeuge mit ungerader Anzahl von Rädern gibt (4 Räder je PKW, 2 je Motorrad). Ebenfalls muss die Anzahl $r$ der Räder mindestens zweimal so groß wie die Anzahl $n$ der Fahrzeuge (nur Motorräder auf dem Parkplatz) und darf nur höchstens viermal so groß wie $n$ sein (nur PKW auf dem Parkplatz). Zusammengefasst muss $r$ gerade sein und $2n \le r \le 4n$ gelten.

Nach diesen Überlegungen sind wir nun in der Lage, eine präzise Problemspezifikation anzugeben. Ein möglicher formaler Rahmen hierfür besteht darin, an die Eingabe bestimmte Vorbedingungen und an die Ausgabe bestimmte Nachbedingungen zu knüpfen.

***Problemspezifikation:***

| | |
|---|---|
| Eingabe: | Zwei natürliche Zahlen $r$ und $n$ |
| Vorbedingung: | $r$ gerade, $2n \leq r \leq 4n$ |
| Ausgabe: | Eine natürliche Zahl $P$, falls Nachbedingung erfüllbar, sonst „keine Lösung“ |
| Nachbedingung: | Es gibt eine natürliche Zahl $M$ mit $P + M = n$ und $4P + 2M = r$ |

Dass $P$ eine natürliche Zahl sein muss, reicht zur Spezifikation der Lösung nicht aus. Es muss noch zusätzlich die Nachbedingung gelten, die – vereinfacht ausgedrückt – besagt, dass die Anzahl $P$ der PKW plus der Anzahl $M$ der Motorräder gleich $n$ und die Anzahl der Räder der PKW plus der Anzahl der Räder der Motorräder gleich $r$ ist, also die Gleichungen (1) und (2) gelten. Da nach der Anzahl $M$ der Motorräder nicht gefragt ist, taucht $M$ in der Ausgabe nicht auf.

Aus der Problemspezifikation erhalten wir die Beschreibung des Algorithmus, indem wir das eigentliche (Lösungs-)Verfahren ergänzen.

***Algorithmus:***

| | |
|---|---|
| Eingabe: | Zwei natürliche Zahlen $r$ und $n$ |
| Vorbedingung: | $r$ gerade, $2n \leq r \leq 4n$ |
| Ausgabe: | Eine natürliche Zahl $P$ |
| Nachbedingung: | Es gibt eine natürliche Zahl $M$ mit $P + M = n$ und $4P + 2M = r$ |
| Verfahren: | Berechne $P = \frac{r-2n}{2}$ |

Der Kern des Algorithmus ist unter dem Punkt „Verfahren“ angegeben. Das Verfahren besteht aus der Berechnung von $P$ gemäß der Auflösung der Gleichungen (1) und (2) zu Beginn des Beispiels. Dabei wird unterstellt, dass bei Verwendung des Verfahrens stets eine Lösung existiert. Das muss natürlich noch bewiesen werden, d.h. es muss gezeigt werden, dass für die Eingabe zweier natürlicher Zahlen $r$ und $n$ und bei erfüllter Vorbedingung die Berechnung von $P$ stets eine natürliche Zahl ergibt und zusätzlich die Nachbedingung erfüllt ist.

Nun hatten wir die Vorbedingung und die Berechnung von $P$ gerade so gewählt, dass $P$ sich als natürliche Zahl ergibt. Die beiden Gleichungen in der Nachbedingung entsprechen den Gleichungen (1) und (2). Da die Anzahl $P$ der PKW kleiner gleich der Anzahl $n$ aller Fahrzeuge und $P$ als natürliche Zahl ganzahlig ist, folgt unmittelbar, dass eine positive, ganze Zahl $M$, also eine natürliche Zahl $M$, existiert,

die Gleichung (1) erfüllt. Wegen der Berechnung von $P$ ist auch Gleichung (2) für $P$ und $M$ erfüllt.

❑

Das folgende Beispiel zeigt einen verfeinerten Algorithmus für das Parkplatzproblem, der schon eine große Nähe zu einem Computerprogramm aufweist. Zu Beginn ist eine kurze Problembeschreibung aufgeführt, es folgen die Vor- und Nachbedingung, den Schluss bildet das Verfahren. Beachten Sie, dass das Verfahren zunächst die Eingabe bezüglich der Vorbedingung überprüft.

Beispiel 2.1.2

*Parkplatz-Algorithmus:*

„Lies die Anzahl der Fahrzeuge und die Anzahl der Räder und gib die Anzahl der PKW aus, falls eine Lösung möglich ist."

Vorbedingung: *AnzahlRäder* gerade und
$2 * AnzahlFahrzeuge \leq AnzahlRäder$ und
$AnzahlRäder \leq 4 * AnzahlFahrzeuge$

Nachbedingung: Es gibt eine natürliche Zahl $M$, so dass gilt
$AnzahlPKW + M = AnzahlFahrzeuge$ und
$4 * AnzahlPKW + 2 * M = AnzahlRäder$

Verfahren:

Lies *AnzahlFahrzeuge*;
Lies *AnzahlRäder*;
Falls *AnzahlRäder* < 0 oder
*AnzahlRäder* ungerade oder
*AnzahlRäder* < 2 * *AnzahlFahrzeug*e oder
*AnzahlRäder* > 4 * *AnzahlFahrzeuge,*
dann schreibe „Falsche Eingabe, keine Lösung möglich",
andernfalls berechne
*AnzahlPKW* = (*AnzahlRäder* - 2 * *AnzahlFahrzeuge*) / 2;
schreibe *AnzahlPKW* „ist die Anzahl der PKW".

❑

Wir wollen nun den Begriff des Algorithmus etwas präziser fassen als dies bisher geschehen ist. Dabei werden wir von jetzt an die Begriffe Anweisung, Operation und Schritt synonym verwenden. Wesentlich bleibt weiterhin, zwischen (statischer) textueller Beschreibung und (dynamischer) Ausführung von Algorithmen und Schritten zu unterscheiden.

Definition 2.1.3

Algorithmus

Ein *Algorithmus* ist eine Menge von Regeln für ein Verfahren, um aus gewissen Eingabegrößen bestimmte Ausgabegrößen herzuleiten, wobei die folgenden Bedingungen erfüllt sein müssen:

*Finitheit der Beschreibung*: Das Verfahren muss in einem endlichen Text vollständig beschrieben sein. Die elementaren Bestandteile der Beschreibung nennen wir *Schritte*.

Finitheit der Beschreibung

*Effektivität*: Jeder einzelne Schritt des Verfahrens muss tatsächlich ausführbar sein.

Effektivität

*Terminierung*: Das Verfahren kommt in endlich vielen Schritten zu einem Ende.

Teminierung

*Determiniertheit*: Der Ablauf des Verfahrens ist zu jedem Punkt fest vorgeschrieben.

Determiniertheit

❑

Zu dieser Definition möchten wir noch einige Anmerkungen machen:

1. Ein Algorithmus löst i.A. eine Klasse von Problemen. Die Präzisierung des aktuell zu lösenden Problems aus der Klasse erfolgt durch geeignete Wahl der Parameter.
2. Es erscheint selbstverständlich, zu verlangen, dass das Verfahren durch einen endlichen Text beschrieben sein muss, da niemand unendliche Texte aufschreiben kann. Manchmal kommen unendliche Texte aber „in Verkleidung" vor, wie zum Beispiel in

   $$\text{Berechne } 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \ldots$$

   Derartige Vorschriften gehören nicht in einen Algorithmus.
3. Effektivität (prinzipielle Machbarkeit) darf nicht mit *Effizienz* (wirtschaftlich vernünftige Machbarkeit) verwechselt werden. Ein Beispiel für einen *nicht effektiven Rechenschritt* wäre etwa: „Falls die Dezimalbruchentwicklung von $x$ nur endlich oft die Ziffer 3 enthält, ist das Ergebnis 5." Die hier angegebene Bedingung lässt sich z.B. für $x = \pi$ nicht überprüfen.
   Ein *nicht effizienter Rechenschritt* demgegenüber wäre: „Berechne bei einem Schachspiel alle möglichen Fortsetzungen für eine gegebene Spielsituation jeweils bis zum Spielende und suche danach den besten Folgezug aus." Dieser Rechenschritt ist effektiv durchführbar, wenn man voraussetzt, dass in absolut festgefahrenen Situationen das Spiel abgebrochen wird. In diesem Fall gibt es zu jeder Schachstellung nur endlich viele Folgestellungen. Allerdings ist die Zahl der möglichen Fortsetzungen normalerweise so groß, dass wir sie weder alle (zum notwendigen Vergleich) speichern noch innerhalb eines Menschenlebens berechnen können.

   Effizienz

   nicht effektiver Rechenschritt

   nicht effizienter Rechenschritt
4. Von der Forderung nach Determiniertheit wird manchmal abgesehen, was bedeutet, dass an manchen Stellen des Verfahrens eine Wahlmöglichkeit besteht, die nicht durch feste Regeln abgefangen wird. (Hier dürfen wir „würfeln", d.h. der Zufall bestimmt, wie weiter fortgefahren wird.)

Abschließend noch eine Bemerkung. Wir können einen Algorithmus betrachten als eine Funktion f: $E \to A$ von der Menge der zulässigen Eingabedaten E in die Men-

ge der Ausgabedaten A, bei der die Funktion definiert ist durch ein Verfahren, mit dem Eingabedaten schrittweise in Ausgabedaten umgewandelt werden. Solche Funktionen heißen *berechenbar*. Eine wesentliche Tatsache ist, dass sich nicht jede Funktion durch einen Algorithmus definieren lässt. Es gibt also auch *nicht-berechenbare Funktionen*. Diese Problematik wird Ihnen in den Kursen der Theoretischen Informatik wiederbegegnen.

berechenbare Funktionen

nicht-berechenbare Funktionen

## 2.2 Programmiersprachen

Sollen Algorithmen von einem Computer ausgeführt werden, dann müssen sie in einer Form beschrieben werden, die der Computer „versteht“. Nun ist der Computer von seiner inneren Logik her eine schlichte Maschine, so dass der für ihn verständlichen Formulierung eines Algorithmus sehr enge Grenzen gesetzt und darüberhinaus noch eine Menge von Details festzuschreiben sind. Wir wissen schon, dass ein vom Computer ausführbarer Algorithmus Programm heißt. Die Regeln, denen ein Programm gehorchen muss, bestimmt die verwendete *Programmiersprache*.

Programmiersprache

Diese Regeln legen zum einen bestimmte Äußerlichkeiten, die *Syntax*, fest. Dazu gehören z.B. das Vokabular, d.h. die erlaubten Wörter, und Vorschriften über den Programmaufbau. So müssen z.B. in Pascal Angaben über die Art der verwendeten Daten am Anfang des Programms stehen, bevor der eigentliche Anweisungsteil beginnt.

Syntax

Zum anderen präzisiert die Programmiersprache die Bedeutung, die *Semantik*, der erlaubten Wörter und Anweisungen. Damit wird für Programmierer und Computer verbindlich festgelegt, was beispielsweise die Ausführung einer bestimmten Anweisung bewirkt.

Semantik

Als Maschine, die keine Flexibilität oder Toleranz kennt, verlangt der Computer vom Programmierer höchste Präzision bei der *Programmerstellung*. Diese Sperrigkeit überrascht gewöhnlich Programmierneulinge und macht den Umgang mit dem Computer oft frustrierend. Andererseits gewöhnt man sich so an Präzision und Disziplin, was sich im Studium und Alltag durchaus als nützlich erweist.

Programmerstellung

Ein Computer[1] versteht unmittelbar nur eine bestimmte prozessorspezifische[2] Programmiersprache, die als *Maschinensprache* bezeichnet wird. Eine Maschinensprache besteht im Wesentlichen aus einem Satz von Befehlen, die elementare Operationen der Maschine auslösen, wie z.B. einfache arithmetische Berechnungen, Speichern von Informationen oder Vergleichen von Werten, wobei das Ergebnis eines Vergleichs wiederum darüber entscheiden kann, welcher Befehl als nächstes ausgeführt wird. Da ein Computer intern nur mit Zahlen arbeitet, ist auch jeder Befehl der Maschinensprache eine Zahl. Zahlen werden für einen Computer wiederum als so genannte *Binärzahlen* (d.h. Zahlen, die nur aus den Ziffern 0 und 1 gebildet werden) dargestellt, was damit zusammenhängt, dass sich eine binäre Information technisch einfach übertragen und verarbeiten lässt (bei einer elektrischen

Maschinensprache

Binärzahlen

1. genauer: sein Prozessor, vgl. Abschnitt 2.5
2. Die Sprache, die ein Prozessor versteht, hängt vom Prozessormodell ab, d.h. die Maschinensprache ist nicht für alle Computer gleich.

Leitung kann man z.B. die 0 durch „Strom aus“ und die 1 durch „Strom an“ darstellen, bei optischen Leitungen entsprechend durch „Licht aus“ bzw. „Licht an“). Nehmen wir z.B. an, die Befehle einer Maschinensprache seien achtstellige Binärzahlen, so lassen sich insgesamt 256 verschiedene Befehle darstellen (von 00000000 bis 11111111).

Maschinenprogramm

Ein Programm, das in einer Maschinensprache formuliert ist, besteht aus einer Folge von als Binärzahlen codierten Befehlen und heißt *Maschinenprogramm*. Maschinenprogramme sind die einzigen Programme, die von einem Computer direkt ausgeführt werden können. Programme in anderen Programmiersprachen müssen erst in semantisch äquivalente, d.h. bedeutungsgleiche, Maschinenprogramme übersetzt werden, um ausgeführt werden zu können. Ein *Übersetzer* ist selbst wieder ein Programm, das Programme einer Sprache in semantisch äquivalente Programme einer anderen Sprache, z.B. Maschinensprache, transformiert.

Übersetzer

Weil das Programmieren in einer Maschinensprache sehr mühsam und fehleranfällig ist – als Binärzahlen codierte Befehle kommen der menschlichen Anschauung so gar nicht entgegen – wurden komfortablere und für den Menschen natürlichere Programmiersprachen (nebst Übersetzern) entwickelt.

Assemblersprachen

Erste Verbesserungen brachten *Assemblersprachen*, deren Befehle aus einer kurzen Folge von (möglichst suggestiven) Buchstaben bestehen, der sich z.B. Adressen von Speicherplätzen anschließen können. Zum Beispiel bedeutet der Assemblerbefehl `ADD R1,R2`: „Addiere den Inhalt des Registers[1] R2 zum Inhalt des Registers R1 (und speichere die Summe in R1)“. Der Übersetzer, der Programme einer Assemblersprache in ausführbare Maschinenprogramme übersetzt, heißt *Assembler*. Ein Befehl einer Assemblersprache wird dabei meist eins zu eins auf einen Maschinenbefehl abgebildet, so dass Assemblerprogramme – auch wenn sie für Menschen bereits deutlich besser lesbar sind als Folgen von Binärzahlen – praktisch aus den gleichen primitiven Computeroperationen bestehen wie Maschinenprogramme. Assembler- und Maschinenprogramme werden heute höchstens noch dazu benutzt, um interne Abläufe im Computer zu steuern.

Assembler

höhere Programmiersprache

Eine drastische Verbesserung des Komforts und der Fehlerrate beim Programmieren bedeuten *höhere Programmiersprachen.* Eine höhere Programmiersprache enthält Wörter und Sätze, die an natürlichen Sprachen angelehnt sind. So gibt es z.B. Anweisungen der Art „Wiederhole Anweisungsfolge A so lange bis Bedingung B erfüllt ist“. Der Übersetzer, der Programme einer höheren Programmiersprache in ausführbare Maschinenprogramme übersetzt, wird *Compiler* genannt.

Compiler

problemorientierte Programmiersprache

Der Erfolg höherer Programmiersprachen hat zu der Entstehung einer Vielzahl verschiedener höherer Sprachen mit unterschiedlichen Stärken und Schwächen geführt. Je nach Anwendungsgebiet werden daher auch unterschiedliche Programmiersprachen bevorzugt. Sprachen, die gezielt mit Blick auf bestimmte Arten von Problemstellungen entwickelt sind, bezeichnet man auch als *problemorientierte Programmiersprachen*.

1. Ein Register ist eine Art Zwischenspeicher des Rechenwerks, vgl. Abschnitt 2.5.

Eine der ersten problemorientierten Sprachen, FORTRAN (FORmula TRANslator), zielt hauptsächlich auf mathematisch-technische Anwendungen ab. COBOL (COmmon Business Oriented Language) wurde entwickelt als Standardsprache für kaufmännische Anwendungen, bei denen der Schwerpunkt eher auf der Verwaltung großer Datenmengen als der Durchführung komplexer Berechnungen liegt. Heute dominieren in der Softwareentwicklung Programmiersprachen wie z.B. C++, C#, Java, C, ADA, die für ein breites Spektrum von Anwendungsbereichen geeignet sind. Dennoch besitzen die Sprachen unterschiedliche Stärken. So wird man sich bei der Entwicklung einer Webapplikation in erster Linie für Java, bei einer sicherheitskritischen Anwendung eher für ADA entscheiden.

## 2.3 Vom Problem zur Computerlösung

In diesem Abschnitt wollen wir das bisher Gelernte zusammenfassen und abrunden.

Realweltproblem
Problembeschr.

Problemspez.
Algorithmus
Progr. in höherer S.
Maschinenprogr.
Programmlauf

Ausgangspunkt jeder Vorgehensweise zur Lösung eines Problems ist das zu lösende *Realweltproblem*. In einem ersten Schritt wird durch Abstraktion und Konzentration auf die für das Problem relevanten Sachverhalte eine *Problembeschreibung* erstellt. Aus der Präzisierung und Formalisierung der Problembeschreibung entsteht die *Problemspezifikation*. Ausgehend von der Problemspezifikation wird der *Algorithmus* entwickelt und danach in ein *Programm in einer höheren Programmiersprache* umgesetzt. Dieses Programm wird vom Compiler in ein semantisch äquivalentes, ausführbares *Maschinenprogramm* übersetzt. Werden die erforderlichen Eingabedaten bereitgestellt, dann kann in einem *Programmlauf* das zu der Eingabe zugehörige Ergebnis bestimmt werden. Als grobes graphisches Ablaufschema ergibt sich Abbildung 2.1.

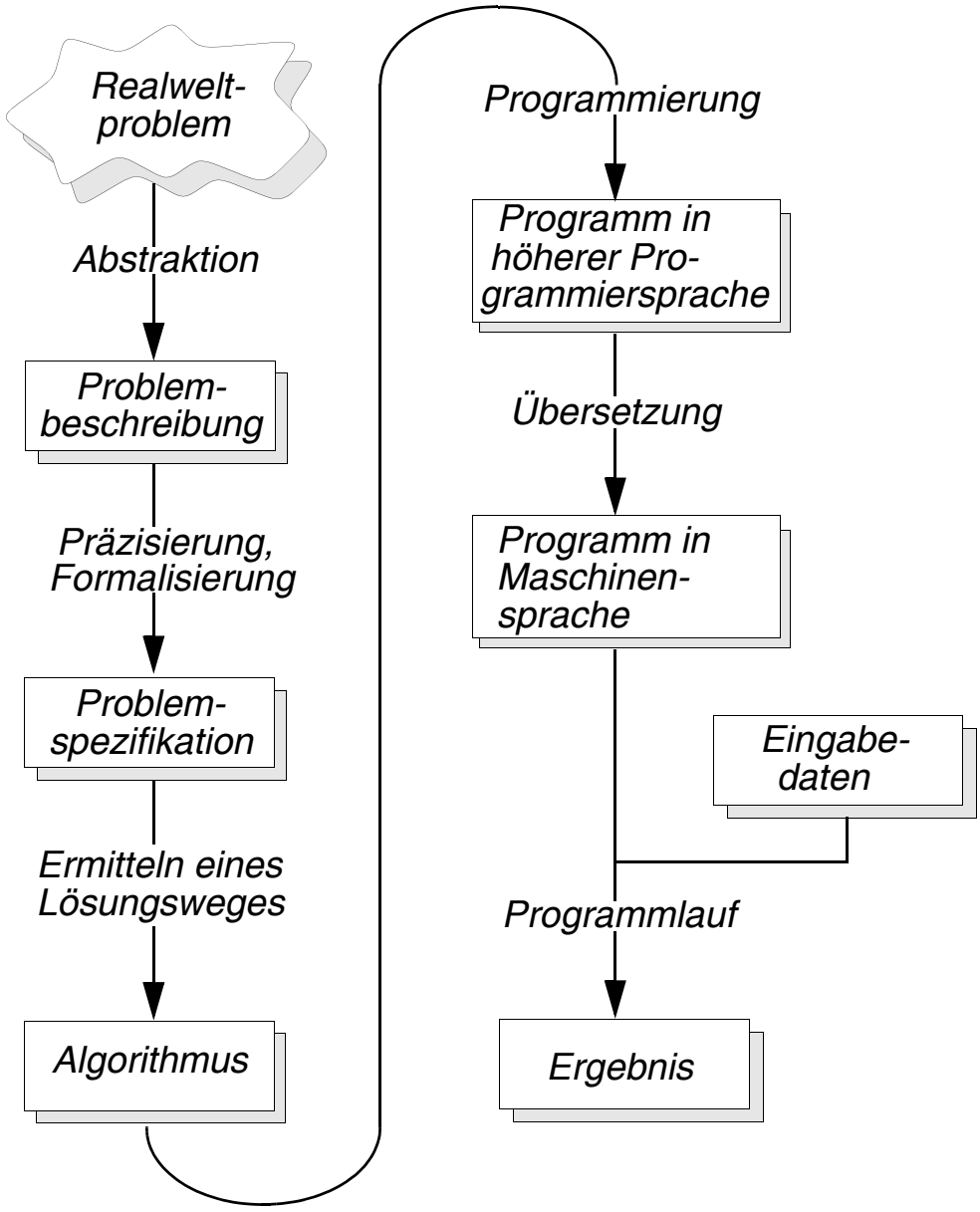



*Abbildung 2.1 Graphisches Ablaufschema: Vom Problem zur Computerlösung*

Bei der Tätigkeit „Ermitteln eines Lösungsweges“ sind wir implizit davon ausgegangen, dass der Nachweis der *Korrektheit* des Algorithmus darin enthalten ist. Intuitiv ist ein Algorithmus genau dann korrekt, wenn er tut, was er soll. Der *Korrektheitsbeweis* besteht im Kern darin, nachzuweisen, dass der Algorithmus für jede Eingabe, welche die Vorbedingung erfüllt, terminiert und die Ausgabe die Nachbedingung erfüllt. Die jeweiligen Vor- und Nachbedingungen sind in der Problemspezifikation angegeben. Ohne eine formale Problemspezifikation können also keine Beweise für die Korrektheit des Lösungsalgorithmus geführt werden. Für komplexe Anwendungen (z.B. integrierte Verwaltung der Kunden einer großen Versicherung mit Kontenverwaltung, termingerechter Zustellung der Beitragsrechnungen, Berücksichtigung von Schadensfällen und alles getrennt nach Versicherungsarten) ist es nicht möglich oder kostenmäßig nicht vertretbar, eine vollständige

Korrektheit

Korrektheitsbeweis

(formale) Problemspezifikation zu erstellen, so dass allein aus diesem Grund Korrektheitsbeweise höchstens partiell möglich sind. Weiterhin ist ein Korrektheitsbeweis nur möglich, wenn der Algorithmus selbst hinreichend formal beschrieben ist. Über Algorithmen, die beispielsweise in natürlicher Sprache angegeben sind, lassen sich keine formalen Beweise führen. In einem solchen Fall kann man dann die Korrektheit des zugehörigen Programms beweisen, falls die Syntax und Semantik der Programmiersprache formal definiert sind.

Testen

Für große und komplexe Programme können sich Korrektheitsbeweise als überaus aufwendig, ja undurchführbar erweisen, ganz zu schweigen von der in diesen Fällen gewöhnlich fehlenden oder unvollständigen Problemspezifikation. Hier sind wir auf das *Testen* angewiesen. Dabei wird dem Programm eine gewisse Auswahl möglichst charakteristischer oder repräsentativer Eingaben vorgelegt und die Ausgaben auf die Erfüllung der Nachbedingung hin überprüft. Ein solches Vorgehen ist natürlich kein Korrektheitsbeweis, sondern belegt lediglich die Funktionstüchtigkeit des Programms für die getesteten (Klassen von) Eingabedaten. Hand in Hand mit dem Testen wird die *Fehlerbeseitigung* vorgenommen.

Fehlerbeseitigung

Dokumentation

Schließlich ist auch eine geeignete *Dokumentation* zu erstellen, so dass nicht nur der Programmentwickler allein in der Lage ist, das Programm zu verstehen. Dies ist vor allem deswegen wichtig, weil oftmals notwendige Änderungen oder Erweiterungen des Programms erst zu einem späteren Zeitpunkt anfallen, an dem der Entwickler nicht mehr zur Verfügung steht.

## 2.4 Programmierparadigmen

Paradigma

In unseren bisheigen Ausführungen zu Programmen und Programmiersprachen sind wir stets stillschweigend von dem derzeit vorherrschenden *imperativen Programmierparadigma* (*Paradigma* bedeutet grundlegendes Prinzip oder Denkschema) ausgegangen. Dieses Programmierparadigma wollen wir nun etwas genauer betrachten und zwei seiner dominierenden Ausprägungen behandeln. Mit dem *deklarativen Programmierparadigma* und seinen wichtigsten Ausprägungen der logischen und funktionalen Programmierung stellen wir anschließend ein weiteres Programmierparadigma vor, das in deutlichem Kontrast zum imperativen Paradigma steht.

imperatives Programmierparadigma

von-Neumann-Architektur

Befehlsfolge

Bei dem *imperativen Programmierparadigma* (lat. imperare = befehlen) steht der Gedanke im Vordergrund, dem Computer den Problemlösungsweg, d.h. den Algorithmus in Form eines Programms, vorzugeben. Es wird also angegeben, **wie** ein Problem zu lösen ist. Das Paradigma geht von dem Konzept der *von-Neumann-Architektur* aus, die bis heute den Aufbau aller gängigen Computer prägt (vgl. Abschnitt 2.5). Ein imperatives Programm besteht aus einer *Folge von Befehlen*, die der Computer in der vorgegebenen Reihenfolge abarbeitet. Um von dieser Reihenfolge je nach Situation (dynamisch) abweichen zu können, gibt es Sprungbefehle, mit denen man an beliebige Stellen der Befehlsfolge wechseln kann, um dort mit der Abarbeitung fortzufahren. Die Befehle verarbeiten *Daten* („Datenverarbeitung“), die im Speicher abgelegt sind. Daten werden häufig in Variablen gespeichert. Eine *Variable* besteht aus einem „logischen“ Speicherplatz mit zugehörigem Wert. Logisch bedeutet hier, dass der Programmierer nicht mit technischen Details

Variable

belastet wird und sich weder um die zugehörige Speicheradresse noch die Größe des von der Variablen belegten Speicherbereichs kümmern muss. Über ihren logischen Namen (z.B. „Summe“) kann er auf die Variable zugreifen und z.B. ihren Wert abfragen oder abändern.

Die wichtigsten Ausprägungen des imperativen Programmierparadigmas sind die prozedurale imperative und die objektorientierte imperative Programmierung.

prozedurale imperative Programmierung

Prozedur

In der *prozeduralen imperativen Programmierung* werden die Daten und die sie manipulierenden Befehle separat behandelt. Im Vordergrund stehen dabei die Befehle (eigentlich: Befehlsfolgen) und nicht die Daten. Das zentrale Strukturierungskonstrukt ist die *Prozedur*. Eine Prozedur fasst logisch zusammengehörende Befehle zu einem „Unterprogramm“ zusammen, das über einen einzigen abstrakteren Befehl („Aufruf“) aktiviert wird. Dadurch wird das Programm kompakter und zugleich übersichtlicher. Wesentlich ist, dass sich die Strukturierungsform – als Folge der separaten Behandlung von Befehlen und Daten – ausschließlich auf Befehle und nicht auf Daten bezieht. Der prozedurale Ansatz ist Bestandteil aller höheren imperativen Programmiersprachen. Beispiele für prozedurale imperative Programmiersprachen sind C, COBOL, FORTRAN, Pascal und PL/1.

objektorientierte Programmierung

Objekt

In der *objektorientierten imperativen Programmierung* (kurz: *objektorientierte Programmierung*) werden logisch zusammengehörende Daten sowie die sie manipulierenden Operationen (das sind Befehlsfolgen in Form von Prozeduren) zu einem *Objekt* zusammengefasst. Ein Objekt repräsentiert oft Dinge aus der realen Welt, wie z.B. ein Bankkonto, einen Kunden oder eine Rechnung. Eine Operation auf einem Bankkonto könnte z.B. aus der Prozedur für eine Einzahlung, eine Operation auf einem Kunden z.B. aus der Prozedur für das Ausstellen einer Rechnung bestehen. Die Befehlsfolgen der prozeduralen imperativen Programmierung werden bei der objektorientierten Programmierung auf Objekte aufgeteilt, d.h. finden sich als Operationen von (i.A. verschiedenen) Objekten wieder. Im Gegensatz zur prozeduralen imperativen Programmierung, die auf Befehlsfolgen fokussiert und nur diese mit Prozeduren strukturiert, stehen in der objektorientierten Programmierung die Daten im Vordergrund, die – angereichert um die auf ihnen erlaubten Operationen – in Form von Objekten die zentrale Strukturierung (eigentlich *Modularisierung*) objektorientierter Programme bilden. Das gegenüber dem Prozedurkonzept verallgemeinerte und mächtigere Objektkonzept ermöglicht eine Modularisierung, mit der sich die Komplexität des Gesamtprogramms besser beherrschen lässt als es das Prozedurkonzept erlaubt. Typische objektorientierte Programmiersprachen sind C++, C#, Java und Smalltalk.

Modularisierung

deklaratives Programmierparadigma

Während das imperative Programmierparadigma darauf basiert, dem Computer in Form eines Programms vorzugeben, **wie** ein gegebenes Problem zu lösen ist, wird bei dem *deklarativen Programmierparadigma* dagegen das gewünschte Ergebnis formuliert, also das, **was** man gern hätte, und der Computer (genauer ein Programm) findet den Lösungsweg selbst und liefert eine passende Lösung. Deklarative Programmiersprachen finden deutlich weniger Verwendung als imperative (objektorientierte) Programmiersprachen und sind meist nur in speziellen Anwen-

dungsbereichen anzutreffen. Die wichtigsten Ausprägungen des deklarativen Programmierparadigma bilden die logische und die funktionale Programmierung.

logische Programmierung

Die *logische Programmierung* benutzt die Mathematische Logik zur Darstellung und Lösung von Problemen. In Form logischer Aussagen werden die Problemstellung und das Wissen über das Problem angegeben und der Computer versucht, mit Hilfe dieses Wissens selbständig eine Lösung des Problems zu finden. Die Problemstellung wird als Behauptung formuliert, die der Computer unter Ausnutzung des vorhandenen Wissens zu beweisen versucht. Im Fall einer parametrisierten Behauptung gibt der Computer alle Lösungen aus, für die die Behauptung wahr wird. Die bekannteste logische Programmiersprache ist PROLOG.

funktionale Programmierung

Die *funktionale Programmierung* benutzt mathematische Funktionen zur Formulierung von Programmen. Ein Programm besteht aus einer Menge von Funktionen, die Eingabedaten in Ausgabedaten abbilden. Die Funktionen werden üblicherweise aus anderen (häufig einfacheren) Funktionen zusammengesetzt, indem insbesondere Argumente und Resultate von Funktionen selbst wieder Funktionen sein können. Das Ende einer solchen Einsetzungskette ist erreicht, wenn man auf eine „Grundfunktion“ stößt, in die keine weitere Funktion mehr eingesetzt ist. Die erste funktionale Programmiersprache war Lisp (1969), eine modernere Variante ist Scheme (1987).

## 2.5 Rechner

Dieser Abschnitt verschafft Ihnen einen kurzen Überblick über den inneren Aufbau eines Computers und darüber, wie Hardware und Software auf elementarer technischer Ebene grundsätzlich zusammenspielen. Um ein guter Programmierer zu werden, muss man nur wenig über den inneren Aufbau, die Rechnerarchitektur, wissen. Die „Spielregeln“ der Programmierung werden durch die Syntax und Semantik der Programmiersprache eindeutig festgelegt, so dass die Bedeutung von Programmen (prinzipiell) unabhängig von dem verwendeten Computer und insbesondere davon ist, wie ein Computer auf technischer Ebene das Programm ausführt. Erfahrungen zeigen allerdings, dass Programmierneulinge „beruhigt“ sind, wenn sie eine Vorstellung davon haben, wie ein Computer grundsätzlich arbeitet und sie nicht eine „Black Box“ programmieren.

Rechnerarchitektur

### 2.5.1 Rechnerarchitektur

Den eigentlichen Durchbruch zu Computern, wie wir sie heute prinzipiell verwenden, brachte die Idee, nicht nur Daten, sondern auch Programme im Speicher abzulegen. Durch den einfachen Austausch von Programmen wurden Computer zu universell verwendbaren Datenverarbeitungsgeräten, die zur Ausführung beliebiger Algorithmen, beispielsweise zur Textverarbeitung, Bilderkennung oder Überwachung von Messgeräten, einsetzbar sind. Computer mit speicherbaren Programmen bezeichnen wir als *von-Neumann-Rechner*, da John von Neumann im Jahre 1945 als erster diese Idee vorgestellt hat.

von-Neumann-Rechner

Die *von-Neumann-Architektur* prägt bis heute alle gängigen Computer. Sie umfasst die fünf Funktionseinheiten Steuerwerk, Rechenwerk, Speicher, Eingabewerk und Ausgabewerk; die ersten drei davon werden nachfolgend kurz beschrieben.

von-Neumann-Architektur

Die Bezeichnung „Computer“ folgt aus der Tatsache, dass diese zunächst ausschließlich für Rechenaufgaben eingesetzt wurden. Im deutschen Sprachraum werden sie auch häufig *Rechner* genannt. Wir wollen daher die Begriffe Computer und Rechner synonym verwenden.

Rechner

**Steuerwerk**

Das *Steuerwerk* ist das „Herz“ des Rechners. Es hat folgende Aufgaben:

Steuerwerk

- Laden der Anweisungen (des gerade bearbeiteten Programms) aus dem Speicher in der richtigen Reihenfolge,
- Decodierung der Befehle,
- Interpretation der Befehle,
- Versorgung der an der Ausführung der Befehle beteiligten Funktionseinheiten mit den nötigen Steuersignalen.

Da an der Ausführung eines Befehls das Rechenwerk, der Speicher und die Geräteverwaltung beteiligt sind, ist das Steuerwerk mit all diesen Komponenten verbunden (vgl. Abbildung 2.2).

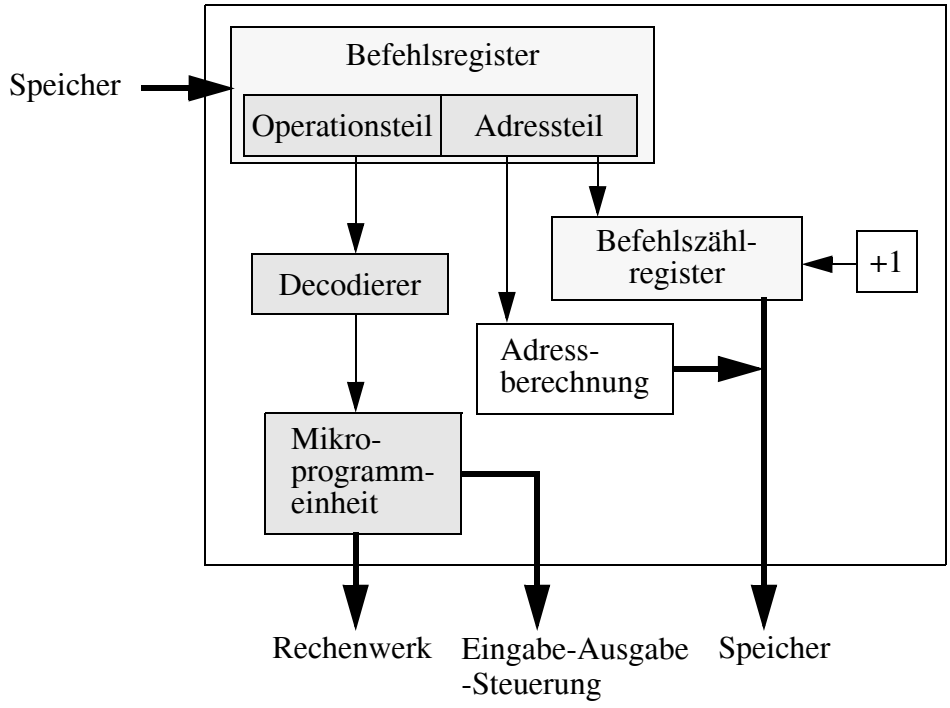



*Abbildung 2.2 Aufbau des Steuerwerks*

Das *Befehlsregister* enthält den Befehl, der gerade ausgeführt wird. Jeder Befehl besteht aus einem Operationsteil und einem Adressteil. Der Operationsteil wird in ei-

Befehlsregister

nem Decodierer entschlüsselt. Der Ausgang des Decodierers wählt für jeden möglichen Operationscode genau eine Eingangsleitung der Mikroprogrammeinheit aus.

Befehlszählregister

Das *Befehlszählregister* speichert die Adresse des nächsten auszuführenden Befehls. Es ist mit einer Schaltung verbunden, die den Inhalt des Registers nach der Ausführung eines Befehls um 1 erhöht. Bei Sprungbefehlen wird jedoch eine im Adressteil des Befehls stehende Adresse in das Befehlszählregister kopiert.

Mikroprogrammeinheit

Die *Mikroprogrammeinheit* erzeugt mit Hilfe der decodierten Informationen des Operationscodes eine Folge von Signalen zur Ausführung des Befehls. Die Mikroprogrammeinheit kann fest verdrahtet und unveränderbar oder programmierbar und variabel gestaltet sein (Mikroprogrammierung).

**Rechenwerk**

Rechenwerk

ALU

Das *Rechenwerk* ist die Komponente des Rechners, in der arithmetische (z.B. Addition, Subtraktion) und logische Verknüpfungen (z.B. und, oder, nicht) durchgeführt werden. Deshalb wird das Rechenwerk auch *ALU* (Arithmetic-Logic Unit) genannt. Die für die Verknüpfungen notwendigen Operanden (i.A. zwei) werden dem Rechenwerk vom Steuerwerk zugeführt.

Zentraleinheit
CPU, Prozessor

Steuerwerk und Rechenwerk fasst man unter der Bezeichnung *Zentraleinheit* (*CPU* = Central Processing Unit), oft auch *Prozessor* genannt, zusammen. Alle arithmetischen Operationen (incl. der vier Grundrechenarten) können auf die Basisoperationen „Verschieben der Stellen im Register“, „Stellenweises Komplementieren ( $0 \rightarrow 1$ , $1 \rightarrow 0$ )“ und „Addieren“ zurückgeführt werden. Subtraktion ist die Addition des Komplements, Multiplikation die wiederholte Addition, Division (mit Rest) die wiederholte Subtraktion. Daher sind die grundlegenden Verknüpfungselemente des Rechenwerks Addierwerk und Komplementierer (siehe Abbildung 2.3).

Das Steuerwerk versorgt das Rechenwerk mit den für die Durchführung der arithmetischen Operationen notwendigen Steuersignalen (z.B. geordnete Bereitstellung der notwendigen Operanden). Kompliziertere Algorithmen wie z.B. zur Multiplikation und Division können ebenfalls im Steuerwerk realisiert sein.

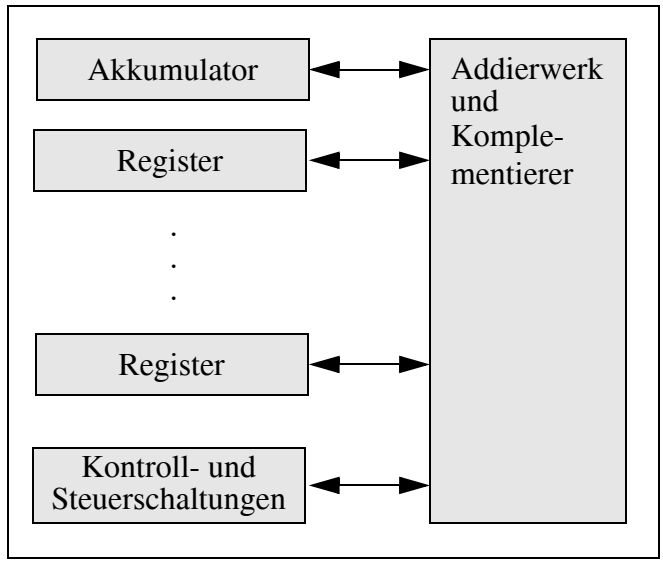



*Abbildung 2.3 Aufbau des Rechenwerks*

Weitere Einheiten des Rechenwerks sind verschiedene Hilfs- und Zwischen-Register (in denen z.B. die Operanden gespeichert werden) und der *Akkumulator* (der das Ergebnis enthält) sowie eine Operationssteuerung, die dafür sorgt, dass die durch das Steuerwerk veranlassten Rechenoperationen korrekt ablaufen. Meist enthält das Rechenwerk spezielle Schaltungen, um in speziellen Fällen einzugreifen, z.B. beim Versuch, durch Null zu teilen.

Akkumulator

### Speicher

Der *Speicher* ist die Rechnerkomponente zum „Aufbewahren“ von Daten und Programmen. Digitale Speicher bestehen aus Speicherelementen, die in der Lage sind, abhängig von einem äußeren Signal genau einen von mehreren erlaubten Zuständen anzunehmen und so lange in ihm zu verweilen, bis er durch ein anderes Signal geändert wird. Heute gibt es praktisch nur binäre Speicher mit zwei Zuständen, denen ein binäres Alphabet zugeordnet wird, etwa {0, 1}, {L, H}, (Low, High für niedriger, hoher Spannungspegel) oder {N, Y} (für No, Yes). Ein solches Speicherelement speichert 1 *Bit*.

Speicher

Bit

Da es mit kleinen und großen Buchstaben, Ziffern und einer großen Auswahl von Sonderzeichen fast 200 verschiedene Zeichen gibt, sind 8 Bit zur Speicherung eines Zeichens nötig. Daher spielt diese Einheit, das *Byte*, bei der Organisation des Speichers eine besondere Rolle. Ein Byte ist sowohl eine Gruppe aus acht Elementarspeichern als auch die Einheit der Speicherkapazität. Wie bei Einheiten üblich, verwendet man die Zusätze „k“, „M“, „G“ und „T“ für Kilo, Mega, Giga und Tera, die Einheit Byte wird, zumindest in solchen Zusammensetzungen, üblicherweise mit B abgekürzt (beispielsweise „Das Programm belegt 190 kB“).

Byte

Adresse

Speicherzelle

Speicherwort

Die Lokalisierung einer gespeicherten Dateneinheit erfolgt durch eine eindeutige Identifikation der Position im Speicher, an der sich die Dateneinheit befindet. Ein Identifikator ist die *Adresse*, die i.A. nicht jedem einzelnen Speicherelement des Speichers zugeordnet ist, sondern nur Gruppen von Speicherelementen, die alle eine gleiche für den Speicher charakteristische feste Größe haben. Eine solche Gruppe, also die kleinste adressierbare Einheit eines Speichers, heißt *Speicherzelle*. Eine Speicherzelle entspricht im Allgemeinen einem Byte. Die Zusammenfassung mehrerer Speicherzellen (meist 4 oder 8) nennt man *Speicherwort* oder kurz *Wort*.

Zugriff lesender, schreibender

Den Vorgang, eine Speicherzelle zu lokalisieren und die in ihr gespeicherte Dateneinheit abzufragen oder zu verändern, bezeichnen wir als *Zugriff*. Wir unterscheiden den *lesenden Zugriff* (Lesen), bei dem der Inhalt einer Speicherzelle abgerufen, aber nicht verändert wird, und den *schreibenden Zugriff* (Schreiben), bei dem die Speicherzelle mit einem neuen Inhalt versehen wird. Weiterhin spielen für die Beurteilung von Speichern folgende Kriterien eine Rolle:

Zugriffszeit

*Zugriffszeit*

Unter Zugriffszeit versteht man die Zeit, die zum Lokalisieren einer Speicherzelle benötigt wird, um anschließend deren Inhalt zu lesen oder zu schreiben. Die Zugriffszeit liegt je nach Typ des Speichers (Hauptspeicher oder externer Speicher, siehe unten) zwischen wenigen Nanosekunden (Nano = $10^{-9}$) und mehreren Sekunden.

Zugriffsart Speichertyp

*Zugriffsart* und *Speichertyp*

wahlfreier / direkter Zugriff, RAM

zyklischer Zugriff

sequentieller Zugriff

Speicher klassifiziert man nach der Methode, mit der auf Speicherzellen zugegriffen werden kann. Ist jede Speicherzelle eines Speichers unabhängig von ihrer Position auf die gleiche Weise mit dem gleichen zeitlichen Aufwand erreichbar, so spricht man von Speichern mit *wahlfreiem* oder *direktem Zugriff* (engl. *RAM* = Random Access Memory). Der Hauptspeicher eines Rechners ist immer ein Speicher mit wahlfreiem Zugriff. Speicher, bei denen die Speicherzellen rotieren und nur periodisch zugänglich sind (wenn sie den feststehenden Lesekopf passieren), erlauben nur einen *zyklischen Zugriff*. Externe Speicher (auch Hintergrund- oder Sekundärspeicher genannt) mit zyklischem Zugriff sind z.B. Festplatten, CDs oder DVDs. Externe Speicher mit *sequentiellem Zugriff* sind solche, bei denen auf eine Speicherzelle erst dann zugegriffen werden kann, wenn auf eine von der Position der Speicherzelle abhängige Anzahl anderer („Vorgänger“-)Speicherzellen zunächst zugegriffen wurde. Beispiel für einen Speicher mit sequentiellem Zugriff ist der Magnetbandspeicher.

Kapazität

*Kapazität*

Die Speicherkapazität wird bestimmt durch die Anzahl der Speicherzellen, die ein Speicher enthält. Sie wird i.A. in KB, MB oder GB gemessen. Externe Speicher haben eine erheblich größere Speicherkapazität als Hauptspeicher.

Im Hauptspeicher abgelegte Informationen gehen verloren, wenn der Computer abgeschaltet wird. Externe Speicher dienen dagegen der dauerhaften Ablage von Programmen und Daten. Sollen Daten geändert werden, so müssen sie in den Hauptspeicher geladen werden, denn nur dort können sie manipuliert werden. Ebenso müssen Programme sich im Hauptspeicher befinden, wenn sie ausgeführt oder geändert werden sollen. Die CPU kann nämlich nur auf den Hauptspeicher zugreifen, um sich z.B. Programmbefehle oder Daten zu holen. Sollen Programme oder Daten länger aufbewahrt werden, muss man sie wieder auf den externen Speicher zurückspeichern. Externe Speicher haben eine reine Archivierungsfunktion.

**von-Neumann-Prinzipien**

von-Neumann-Prinzipien

Nachfolgend sollen die wesentlichen Prinzipien der klassischen von-Neumann-Rechnerarchitektur noch einmal zusammengefasst werden:

1. Der Rechner besteht aus fünf Funktionseinheiten: dem Steuerwerk, dem Rechenwerk, dem Speicher, dem Eingabewerk und dem Ausgabewerk.
2. Die Struktur des von-Neumann-Rechners ist unabhängig von den zu bearbeitenden Problemen. Zur Lösung eines Problems muss von außen das Programm eingegeben und im Speicher abgelegt werden. Ohne dieses Programm ist die Maschine nicht arbeitsfähig.
3. Programme, Daten, Zwischen- und Endergebnisse werden in demselben Speicher abgelegt.
4. Der Speicher ist in gleichgroße Zellen unterteilt, die fortlaufend durchnumeriert sind. Über die Nummer (Adresse) einer Speicherzelle kann deren Inhalt abgerufen oder verändert werden.
5. Aufeinanderfolgende Befehle eines Programms werden in aufeinanderfolgenden Speicherzellen abgelegt. Das Ansprechen des nächsten Befehls geschieht vom Steuerwerk aus durch Erhöhen der Befehlsadresse um Eins.
6. Durch Sprungbefehle kann von der Bearbeitung der Befehle in gespeicherter Reihenfolge abgewichen werden.
7. Es gibt zumindest
   - arithmetische Befehle wie Addieren, Multiplizieren usw.,
   - logische Befehle wie Vergleiche, logisches nicht, und, oder usw.,
   - Transportbefehle, z.B. vom Speicher zum Rechenwerk und für die Ein-/Ausgabe,
   - bedingte Sprünge.

   Weitere Befehle wie Schieben, Unterbrechen, Warten usw. kommen u. U. hinzu.
8. Alle Daten (Befehle, Adressen usw.) werden binär codiert. Geeignete Schaltwerke im Steuerwerk und an anderen Stellen sorgen für die richtige Entschlüsselung (Decodierung).

Die Architektur des von-Neumann-Rechners fällt in die Klasse der sogenannten *SISD* (single instruction stream, single data stream)-*Architekturen*. Diese Architekturen sind gekennzeichnet durch

SISD-Architektur

- einen Prozessor (Ein-Prozessor-System), bestehend aus Steuer- und Rechenwerk, und

- die Erzeugung einer Befehls- und einer Operandenfolge mit streng sequentieller Abarbeitung.

### 2.5.2 Rechnersysteme

Rechnersystem

Die Komponenten eines *Rechnersystems* lassen sich in Hardware (physikalische Komponenten) und Software (veränderbare, immaterielle Komponenten in Form von Programmen oder Daten) unterscheiden. Neben dem eigentlichen Rechner, wie wir ihn bisher vorgestellt haben, gehören zur *Hardware* eines Rechnersystems periphere Geräte zur Datenerfassung, -speicherung, -ausgabe und -übertragung:

Hardware

- zur *Erfassung* z.B. Tastaturen, Scanner, Kameras, Grafiktabletts oder Touchscreens;
- zur *Speicherung* als Ergänzung zum Hauptspeicher z.B. Magnetbänder und -platten (wie insbesondere Disketten und Festplatten), optische Speichermedien (wie CD- oder DVD-ROM) oder nicht-flüchtige Speicherchips (wie Flash-Speicherkarten);
- zur *Ausgabe* z.B. Drucker, Plotter (Zeichengeräte), Bildschirme, oder „Soundkarten“ und Lautsprecher;
- zur *Übertragung* z.B. Modems, Netzwerkkarten, Funkschnittstellen (wie Bluetooth) oder Infrarotschnittstellen.

Software

Die *Software* eines Rechnersystems umfasst als wichtige Bestandteile die *Anwendungssoftware* und das *Betriebssystem*. Es gibt noch weitere wichtige Software wie z.B. Compiler oder Datenbanksysteme, die weder der einen noch der anderen Klasse eindeutig zugeordnet werden kann. Wir werden darauf aber nicht weiter eingehen.

Anwendungssoftware

*Anwendungssoftware* wird für die Lösung spezieller Aufgaben aus bestimmten Anwendungsbereichen entwickelt. Sie wird vom Endanwender für einen bestimmten Zweck benutzt und interagiert dabei oft mit ihrem Benutzer. Einige typische Vertreter von Anwendungssoftware sind

- Programme zur Textverarbeitung,
- Programme für Tabellenkalkulation,
- Programme für die Verwaltung von Kunden- oder Artikeldaten,
- Computerspiele.

Es wäre zwar grundsätzlich möglich, eine Anwendungssoftware zu entwickeln, die direkt und ohne zusätzliche Software auf einem Rechnersystem ausgeführt werden kann. Dann könnte aber z.B. nur diese Anwendung alleine ablaufen. Auch müsste die Anwendung sehr viele oft komplexe „Unterprogramme“ enthalten, die einerseits ausschließlich für den Zugriff auf benötigte Hardware (z.B. die Festplatte) zuständig sind und mitunter nur für bestimmte Hardware-Fabrikate funktionieren, andererseits aber in gleicher oder ähnlicher Form von fast jeder Anwendung benötigt werden.

Daher verfügen Computersysteme fast immer über ein *Betriebssystem*, welches das Laden und die Ausführung von Anwendungssoftware (oft auch mehrerer Anwendungen parallel) ermöglicht und dabei die Verwaltung und Steuerung der Hardware übernimmt. Die Anwendungssoftware muss nun nicht mehr aufwendig direkt mit der Hardware interagieren, sondern kann diese Aufgabe weitgehend ans Betriebssystem delegieren. Darüber hinaus unterstützt das Betriebssystem in der Regel die Unabhängigkeit der Anwendungssoftware von z.B. Modell und Hersteller der von ihr verwendeten peripheren Geräte. Dazu dienen so genannte *Gerätetreiber* (gerätespezifische Steuerungssoftware, oft vom Gerätehersteller angeboten), die vom Betriebssytem verwendet werden.

Betriebssystem

Gerätetreiber

Ein Betriebssystem bindet insbesondere Speichermedien an und stellt auf diesen ein so genanntes *Dateisystem* für die Ablage und Verwaltung von Daten und Anwendungssoftware zur Verfügung. Bei mehreren parallel laufenden Anwendungen ist das Betriebssystem verantwortlich für die *Zuteilung von Betriebsmitteln* (wie z.B. Rechenzeit, Arbeitsspeicher und Peripherie). Weiterhin kann ein modernes Betriebssystem auch mehrere Benutzer eines Computersystems unterscheiden und kümmert sich dann z.B. um die *Authentifizierung* und den *Schutz privater Daten* eines Benutzers vor dem Zugriff durch andere Benutzer.

Dateisystem

Zuteilung von Betriebsmitteln

Benutzerauthentifizierung, Schutz privater daten

# Lernziele zum Kapitel 3

Nach diesem Kapitel sollten Sie

1. den grundlegenden Aufbau einfacher Pascal-Programme kennen und die Struktur und Bedeutung vorgegebener einfacher Pascal-Programme erläutern können,

2. erste einfache Programmierrichtlinien zur Erhöhung der Lesbarkeit von Pascal-Programmen angeben können.

# 3. Programmierkonzepte orientiert an Pascal (Teil 1)

## 3.1 Einleitung

Zu Beginn der Kapitel über Programmierkonzepte möchten wir noch einmal daran erinnern, daß es uns nicht darum geht, Pascal vollständig mit all seinen Möglichkeiten zu erläutern. Vielmehr geht es uns darum, Ihnen die grundlegenden Konzepte imperativer Programmierung zu vermitteln. Diese Konzepte gelten aber nicht nur für die imperative Programmierung, sondern begegnen Ihnen auch in der objektorientierten Programmierung, wenn auch manchmal in abgewandelter Form, wieder. Anders ausgedrückt: Sie können kein objektorientiertes Programm erstellen, ohne die in diesem Kurs behandelten Konzepte verstanden zu haben. Natürlich kommen bei der Objektorientierung noch weitere (und schwierigere) Konzepte hinzu. Damit die im Kurs vorgestellten Konzepte nicht abstrakt bleiben, verwenden wir für ihre konkrete Realisierung und zu Übungszwecken die Programmiersprache Pascal.

Wir möchten noch auf einen weiteren wichtigen Punkt hinweisen.

Im Abschnitt 2 hatten wir betont, daß am Anfang jeder Problemlösung eine präzise Problemformulierung, möglichst eine Problemspezifikation, steht, um klare Vorgaben zu haben, an denen die Qualität und insbesondere die Korrektheit des Algorithmus bzw. des Programms gemessen werden können.

Entgegen diesen Richtlinien haben wir uns dazu entschlossen, bei den folgenden Programmbeispielen fast immer auf eine Problemspezifikation zu verzichten. Der Grund dafür besteht darin, daß die Beispiele grundsätzlich nur einen sehr kleinen Umfang und geringe Komplexität besitzen, ja häufig nur isolierte Teilprobleme aufgreifen, so daß wir von einer vollständigen und durchgehend formalen bzw. präzisen Vorgehensweise vom Realweltproblem zum Programm absehen wollen. Schwerpunkt der Kapitel über Programmierkonzepte ist nicht, Methoden der Problemlösung zu behandeln (sämtliche Problemstellungen und Programme sind elementar), sondern Programmiersprachenkonzepte und Basistechniken zu vermitteln.

Um Fußballtraining als analoges Beispiel zu bemühen: Bevor nicht technische Grundfertigkeiten der Ballbehandlung beherrscht werden, macht es wenig Sinn, über taktisches Verhalten zu reden.

Zum Abschluß der Einleitung wollen wir Ihnen eine kleine Hilfestellung geben, wie Programme auf einem Computer zum Laufen gebracht werden. Sie erkennen daran, daß wir davon ausgehen, daß Sie Zugriff auf einen Computer haben, sei es privat oder in einem Studienzentrum. Ohne praktischen Umgang mit dem Computer ist es kaum möglich, selbst einfache Programmierkonstrukte zu verstehen und anzuwenden.

Programmerstellung beginnt auf dem Papier. Nach der "Papiercodierung" wird versucht, in einer Art "Trockenübung" möglichst viele Fehler aufzuspüren und auszu-

Editor
File
Quellprogramm

bessern. Wir simulieren also den Computer, indem wir das Papierprogramm mit verschiedenen Eingabedaten durchspielen. Anschließend wird das Programm mit einem *Editor* (eine Art Textverarbeitungsprogramm) in den Computer eingegeben und in einem *File* (Datei) auf einem externen Speicher abgelegt. Dieses *Quellprogramm* muß nun vom Compiler in ein Maschinenprogramm übersetzt werden. Dieser Vorgang läuft in zwei Schritten ab. Zunächst untersucht der Compiler das Programm auf syntaktische Korrektheit und gibt Fehlermeldungen aus, falls gegen die Syntax verstoßen wurde. Nach einer entsprechenden Fehlerbeseitigung und anschließender Übersetzung liegt dann das compilierte Programm, auch *Objektprogramm* genannt, in Maschinensprache vor. Es ist gewöhnlich in einem weiteren File abgelegt.

Objektprogramm

Start-Kommando

Mit einem *Start-Kommando* wird die Ausführung des Objektprogramms angestoßen. Wendet man dieses Kommando auf ein Quellprogramm an, dann wird dies zunächst compiliert und, falls keine Syntaxfehler aufgetreten sind, ausgeführt.

Wie der Editor und der Compiler bedient bzw. aufgerufen werden, wie das Start-Kommando genau aussieht, hängt von dem verwendeten Rechnersystem ab. Diese Befehle sind nicht Bestandteil von Programmiersprachen.

## 3.2 Programmstruktur

Wir betrachten zunächst ein sehr einfaches, aber vollständiges Pascal-Programm:

```
program SehrEinfach (output);
begin
  writeln ('Hallo! Viel Spaß beim Programmieren!')
end.
```

Was dieses kurze Programm bewirkt bzw. ausgibt, läßt sich unschwer erraten:

```
Hallo! Viel Spaß beim Programmieren!
```

Syntaxdiagramm

Für die Beschreibung der Programmstruktur benutzen wir ein graphisches Hilfsmittel, das *Syntaxdiagramm*. Syntaxdiagramme wurden von N. Wirth zusammen mit Pascal eingeführt. Zur Einführung zitieren wir [ApL00]:

"Das Prinzip läßt sich durch eine Analogie veranschaulichen: Zwei Touristen fahren gemeinsam im Auto durch eine Region, in der es nur Einbahnstraßen gibt; sie heißt daher auch One-Way-County (...). Der Beifahrer fotografiert alle auf der Landkarte eingezeichneten 'Sehenswürdigkeiten' (durch abgerundete Rechtecke wie RR-Station und Motel repräsentiert). Es ist nun zu überlegen, welche Bildsequenzen auf dem Film möglich sind, welche nicht. Offenbar kann man die möglichen Bildsequenzen ermitteln, indem man die Fahrt nachvollzieht. Erreicht man dabei in der Landkarte One-Way-City (durch ein Rechteck repräsentiert), so zeigt

der Stadtplan von One-Way-City (...) den weiteren Weg, bis man diesen wieder verläßt und am Ausgang des Rechtecks One-Way-City ... fortfährt."

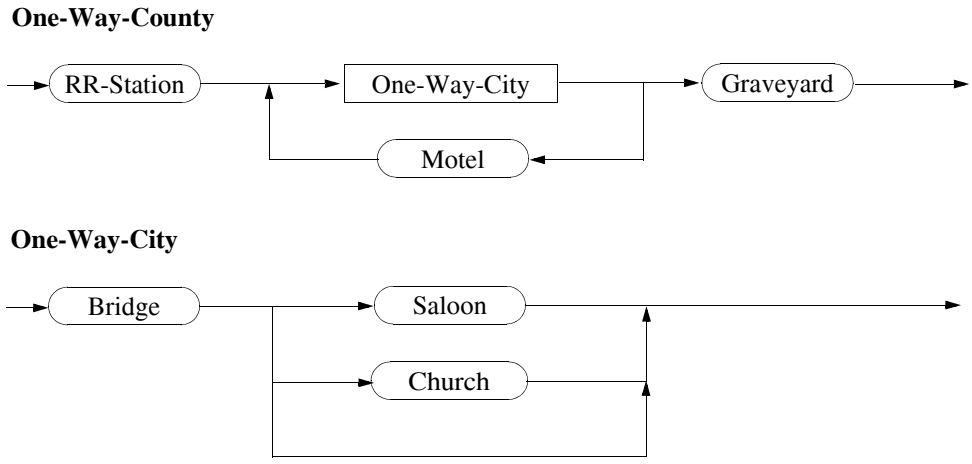


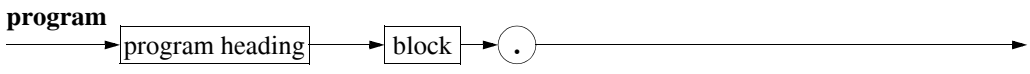

*Abbildung 2.4 : Landkarte von One-Way-County, Stadtplan von One-Way-City im Syntaxdiagramm*

Offenbar braucht man zur Zusammenstellung von möglichen Bildsequenzen nur den Pfeilen zu folgen. Rechtecke werden dabei in weitere Diagramme aufgegliedert, man bezeichnet sie als *Nichtterminale*; abgerundete Rechtecke (oder Kreise) können nicht weiter zerlegt werden, man spricht hier von *Terminalen*. Analog werden die Syntaxdiagramme für Pascal interpretiert.

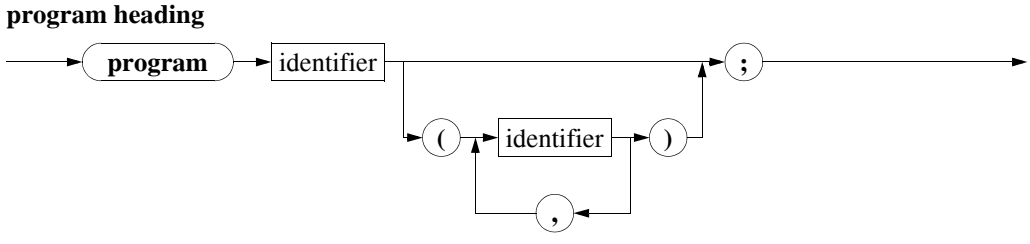
Terminale und Nichtterminale

Pascal-Programme haben stets die gleiche Struktur: Sie bestehen aus einem *Programmkopf* (*program heading*) und einem *Block* (*block*), gefolgt von einem abschließenden Punkt.

Programmkopf
Block



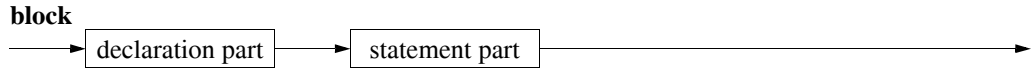

Der Programmkopf beginnt stets mit dem Schlüsselwort **`program`**, gefolgt vom Programmbezeichner. Danach können in Klammern Bezeichner von Ein- und Ausgabedateien angegeben werden, die außerhalb des Programms definiert sind. Der Programmkopf wird durch ein Semikolon vom Block getrennt. Wie in Pascal Bezeichner (identifier) zusammengesetzt sein dürfen, erfahren Sie im nächsten Abschnitt.



Unser Beispiel-Programm heißt `SehrEinfach`, die Angabe `output` weist darauf hin, daß Daten auf die Standardausgabe, also den Bildschirm, geschrieben werden sollen.

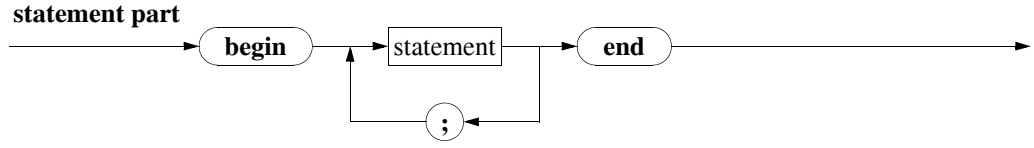
Deklarationsteil
Vereinbarungsteil

Der Block setzt sich zusammen aus dem *Vereinbarungsteil* (*declaration part*) und dem *Anweisungsteil* (*statement part*). Im Vereinbarungsteil werden die Daten beschrieben und benannt, die im Programm benutzt werden, im Anweisungsteil werden die auszuführenden Operationen angegeben.




Den Aufbau des Vereinbarungsteils untersuchen wir genauer im Abschnitt 3.5 "Beschreibung der Daten".

Der Anweisungsteil enthält die Aktionen, die ausgeführt werden sollen, als eine Folge von Anweisungen (statements), die zwischen den Schlüsselwörtern **`begin`** und **`end`** eingeschlossen werden:




In unserem Beispiel ist der Vereinbarungsteil leer, der Anweisungsteil besteht nur aus einer Anweisung. Die Ausgabe-Anweisung `writeln` gehört zu den Standard- (d.h. "eingebauten") Prozeduren von Pascal, daher braucht sie nicht deklariert zu werden. Mit einfachen Anweisungen beschäftigen wir uns im Abschnitt 3.6.1 "Einfache Anweisungen", mit Standardprozeduren zur Ein- und Ausgabe im Abschnitt 3.6.2 "Standard-Ein- und Ausgabe".

## 3.3 Bezeichner

Bezeichner, identifier

Zeichenvorrat von Pascal

Alle Objekte, die wir im Programm verwenden (Konstanten, Typen, Variablen, Prozeduren und Funktionen), müssen mit einem eindeutigen *Bezeichner* (*identifier*) versehen werden, um sie unterscheiden und ansprechen zu können. Bezeichner dürfen nur nach bestimmten Regeln gebildet werden. Die erste Vereinbarung betrifft den Zeichenvorrat, über dem Zeichenfolgen gebildet werden dürfen. Der *Zeichenvorrat* von *Pascal* unterscheidet die vier Zeichengruppen *Buchstaben* (*letters*), *Ziffern* (*digits*), *Spezialsymbole* (*special symbols*) und *Schlüsselwörter* (*reserved words*).

Buchstaben

*Buchstaben*

```
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m
n o p q r s t u v w x y z
```