

Hans-Werner Six  
unter Mitarbeit von Immo Schulz-Gerlach

# Einführung in die imperative Programmierung

Kurseinheiten 1-5

mathematik  
und  
informatik





# 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.

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

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).

Gliederung

Exkurse

## 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 und zur 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

<b>Kurseinheit I.</b>	<b>1</b>
<b>1. Einleitung</b>	<b>2</b>
1.1 Informatik, Computer, Algorithmus, Programm.	2
1.2 Rolle des Kurses 1613 in der Programmierausbildung.	4
<b>Lernziele zum Kapitel 2</b>	<b>8</b>
<b>2. Grundlagen</b>	<b>9</b>
2.1 Problem und Algorithmus	9
2.2 Programmiersprachen	14
2.3 Vom Problem zur Computerlösung.	16
2.4 Programmierparadigmen	18
2.5 Rechner	20
2.5.1 Rechnerarchitektur	20
2.5.2 Rechnersysteme.	26
<b>Lernziele zum Kapitel 3</b>	<b>28</b>
<b>3. Programmierkonzepte orientiert an Pascal (Teil 1)</b>	<b>29</b>
3.1 Einleitung	29
3.2 Programmstruktur	30
3.3 Bezeichner	32
3.4 Zahlen	35
3.5 Beschreibung der Daten	39
3.5.1 Datentypen	39
3.5.2 Definition von Konstanten	46
3.5.3 Deklaration von Variablen	49
3.5.4 Eindeutigkeit von Bezeichnern	49
3.6 Beschreibung der Aktionen	50
3.6.1 Einfache Anweisungen	51
3.6.2 Standard-Ein- und Ausgabe	56
3.6.3 Die bedingte Anweisung (if-then-else)	60
3.7 Einfache selbstdefinierbare Datentypen	68
3.7.1 Aufzählungstypen	69
3.7.2 Ausschnittstypen	70
3.8 Programmierstil (Teil 1)	72
<b>Kurseinheit II.</b>	<b>79</b>
<b>Lernziele zum Kapitel 4</b>	<b>80</b>
<b>4. Programmierkonzepte orientiert an Pascal (Teil 2)</b>	<b>81</b>
4.1 Wiederholungsanweisungen	81
4.1.1 Die for-Schleife.	81
4.1.2 Die while-Schleife.	85
4.1.3 Die repeat-Schleife	89

---

4.2 Strukturierte Datentypen . . . . .	93
4.2.1 Der array-Typ . . . . .	95
4.2.2 Der Verbundtyp. . . . .	108
4.3 Funktionen. . . . .	116
4.4 Programmierstil (Teil 2) . . . . .	126
<b>Kurseinheit III . . . . .</b>	<b>135</b>
<b>Lernziele zum Kapitel 5 . . . . .</b>	<b>137</b>
<b>5. Programmierkonzepte orientiert an Pascal (Teil 3) . . . . .</b>	<b>138</b>
5.1 Prozeduren. . . . .	138
5.1.1 Prozeduren und Parameterübergabe . . . . .	138
5.1.2 Prozeduren und Parameterübergabe in Pascal . . . . .	148
5.2 Blockstrukturen: Gültigkeitsbereich und Lebensdauer . . . . .	151
5.3 Hinweise zur Verwendung von Prozeduren und Funktionen . . . . .	162
5.4 Dynamische Datenstrukturen . . . . .	164
5.4.1 Zeiger . . . . .	164
5.4.2 Lineare Listen . . . . .	169
5.5 Programmierstil (Teil 3) . . . . .	182
<b>Kurseinheit IV . . . . .</b>	<b>193</b>
<b>Lernziele zum Kapitel 6 . . . . .</b>	<b>194</b>
<b>6. Programmierkonzepte orientiert an Pascal (Teil 4) . . . . .</b>	<b>195</b>
6.1 Fortsetzung Dynamische Datenstrukturen: Binäre Bäume . . . . .	195
6.2 Rekursion. . . . .	205
6.3 Programmierstil (Gesamtübersicht). . . . .	227
6.3.1 Bezeichnerwahl . . . . .	228
6.3.2 Programmtext-Layout . . . . .	229
6.3.3 Kommentare . . . . .	232
6.3.4 Prozeduren und Funktionen . . . . .	233
6.3.5 Seiteneffekte . . . . .	234
6.3.6 Sonstige Merkregeln . . . . .	236
6.3.7 Strukturierte Programmierung. . . . .	236
6.3.8 Zusammenfassung der Muss-Regeln . . . . .	238
<b>7. Exkurs: Prozedurale Programmentwicklung . . . . .</b>	<b>241</b>
7.1 Prozedurale Zerlegung und Verfeinerung . . . . .	241
7.2 Ein Beispiel . . . . .	243
7.2.1 Aufgabenstellung . . . . .	243
7.2.2 Lösungsstrategie . . . . .	243
7.2.3 Prozedurale Zerlegung und Verfeinerung. . . . .	245
<b>Kurseinheit V . . . . .</b>	<b>257</b>
<b>Lernziele zu Kurseinheit V . . . . .</b>	<b>258</b>
<b>8. Einführung in die Analytische Qualitätssicherung. . . . .</b>	<b>259</b>
8.1 Motivation und grundlegende Definitionen . . . . .	259

---

8.2 Klassifikation der Verfahren der analytischen Qualitätssicherung .....	264
<b>9. White-Box-Testverfahren .....</b>	<b>268</b>
9.1 Kontrollflussorientierte Testverfahren .....	268
9.1.1 Kontrollflussgraphen .....	268
9.1.2 Anweisungsüberdeckung .....	276
9.1.3 Zweigüberdeckung .....	278
9.1.4 Minimale Mehrfach-Bedingungsüberdeckung .....	281
9.1.5 Boundary-interior-Pfadtest .....	287
9.2 Exkurs: Datenflussorientierte Testverfahren .....	295
9.2.1 Kontrollflussgraphen in Datenflussdarstellung .....	295
9.2.2 Der Test nach dem all uses-Kriterium .....	300
<b>10. Black-Box-Testverfahren .....</b>	<b>303</b>
10.1 Funktionsorientierter Test: Funktionale Äquivalenzklassen .....	305
10.2 Fehlerorientierter Test: Test spezieller Werte .....	310
10.3 Zufallstest .....	311
<b>11. Regressionstest .....</b>	<b>313</b>
<b>12. Abschließende Bemerkungen zu dynamischen Tests .....</b>	<b>315</b>
<b>13. Statische Testverfahren .....</b>	<b>319</b>
13.1 Überblick .....	319
13.2 Reviews .....	319
13.3 Exkurs: Statische Analysatoren .....	321
<b>Lösungen zu den Aufgaben .....</b>	<b>323</b>
<b>Quellen- und Literaturangaben .....</b>	<b>347</b>
<b>Pascal-Referenzindex .....</b>	<b>349</b>
<b>Stichwortverzeichnis .....</b>	<b>351</b>





# 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.

Algorithmus

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.

Beschreibung des 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 den Verstärker 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*.

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.

## Universalität

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

#### 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 zu meist 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 Objektorientierte 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:

$$4P + 2(n - P) = r$$

$$\Leftrightarrow 2P = r - 2n$$

$$\Leftrightarrow P = \frac{r - 2n}{2}$$

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  $2n \leq r \leq 4n$  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 ganzzahlig 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  
 $AnzahlRäder < 2 * AnzahlFahrzeuge$  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

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*.

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

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

*Determiniertheit:* Der Ablauf des Verfahrens ist zu jedem Punkt fest vorge-schrieben.

Finitheit der  
Beschreibung

Effektivität

Terminierung

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

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

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.
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.)

Effizienz

nicht effektiver  
Rechenschritt

nicht effizienter  
Rechenschritt

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

berechenbare  
Funktionen  
nicht-berechenbare  
Funktionen

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.

## 2.2 Programmiersprachen

Programmiersprache

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*.

Syntax

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.

Semantik

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.

Programmerstellung

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.

Maschinensprache

Ein Computer<sup>1</sup> versteht unmittelbar nur eine bestimmte prozessorspezifische<sup>2</sup> 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

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).

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.

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.

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<sup>1</sup> 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.

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.

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*.

Maschinenprogramm

Übersetzer

Assemblersprachen

Assembler

höhere  
Programmiersprache

Compiler

problemorientierte  
Programmiersprache

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.

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.

Realweltproblem  
Problembeschr.

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



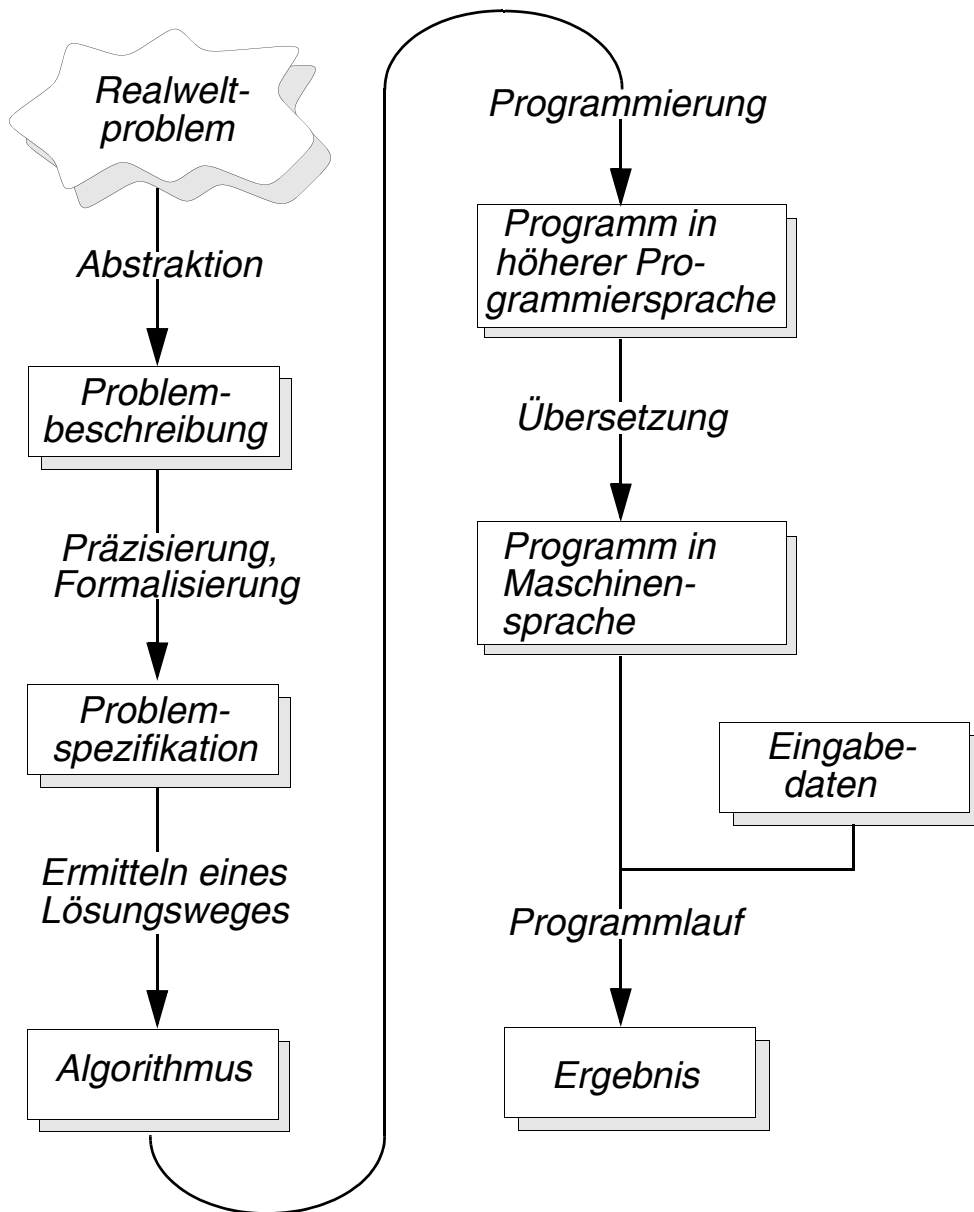


Abbildung 2.1 Graphisches Ablaufschema: Vom Problem zur Computerlösung

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.

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 bisherigen 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 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

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.

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.

prozedurale  
imperative  
Programmierung  
  
Prozedur

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.

objektorientierte  
Programmierung  
  
Objekt

Modularisierung

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-

deklaratives Pro-  
grammierparadigma

logische  
Programmierung

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

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.

### Steuerwerk

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

- 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).

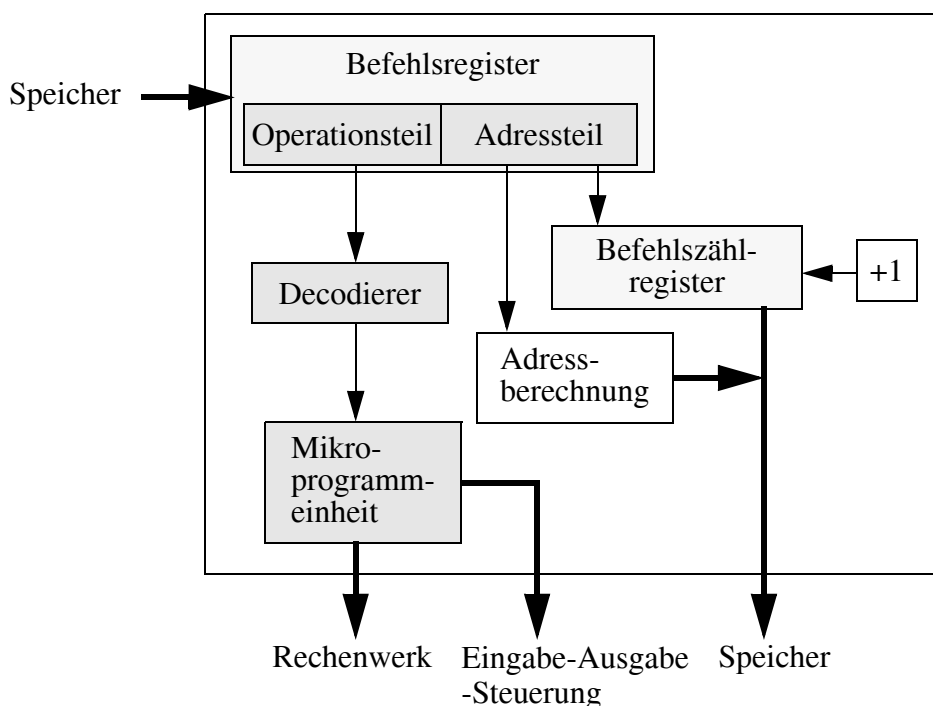


Abbildung 2.2 Aufbau des Steuerwerks

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

von-Neumann-  
Architektur

Rechner

Steuerwerk

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

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.

ALU

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.

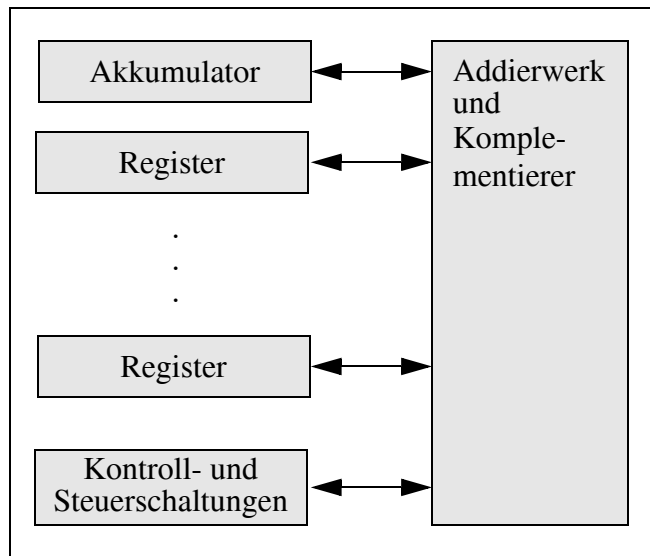


Abbildung 2.3 Aufbau des Rechenwerks

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.

## 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*.

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“).

Akkumulator

Speicher

Bit

Byte

Adresse	<p>Die Lokalisierung einer gespeicherten Dateneinheit erfolgt durch eine eindeutige Identifikation der Position im Speicher, an der sich die Dateneinheit befindet. Ein Identifikator ist die <i>Adresse</i>, 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 <i>Speicherzelle</i>. Eine Speicherzelle entspricht im Allgemeinen einem Byte. Die Zusammenfassung mehrerer Speicherzellen (meist 4 oder 8) nennt man <i>Speicherwort</i> oder kurz <i>Wort</i>.</p>
Speicherzelle	
Speicherwort	
Zugriff lesender, schreibender	<p>Den Vorgang, eine Speicherzelle zu lokalisieren und die in ihr gespeicherte Dateneinheit abzufragen oder zu verändern, bezeichnen wir als <i>Zugriff</i>. Wir unterscheiden den <i>lesenden Zugriff</i> (Lesen), bei dem der Inhalt einer Speicherzelle abgerufen, aber nicht verändert wird, und den <i>schreibenden Zugriff</i> (Schreiben), bei dem die Speicherzelle mit einem neuen Inhalt versehen wird. Weiterhin spielen für die Beurteilung von Speichern folgende Kriterien eine Rolle:</p>
Zugriffszeit	<p><i>Zugriffszeit</i></p> <p>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 = <math>10^{-9}</math>) und mehreren Sekunden.</p>
Zugriffsart Speichertyp	<p><i>Zugriffsart und Speichertyp</i></p> <p>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 <i>wahlfreiem</i> oder <i>direktem Zugriff</i> (engl. <i>RAM</i> = 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 <i>zyklischen Zugriff</i>. Externe Speicher (auch Hintergrund- oder Sekundärspeicher genannt) mit zyklischem Zugriff sind z.B. Festplatten, CDs oder DVDs. Externe Speicher mit <i>sequentiell</i>em 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 sequentiell</p>
wahlfreier / direkter Zugriff, RAM	
zyklischer Zugriff	
sequentieller Zugriff	
Kapazität	<p><i>Kapazität</i></p> <p>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.</p>



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**

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 durchnummeriert 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

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

von-Neumann-  
Prinzipien

SISD-Architektur

- 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, -abgabe und -übertragung:

Hardware

- zur *Erfassung* z.B. Tastaturen, Scanner, Kameras, Grafiktablets 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 *Abgabe* 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 Betriebssystem verwendet werden.

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.

Betriebssystem

Gerätetreiber

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!
```

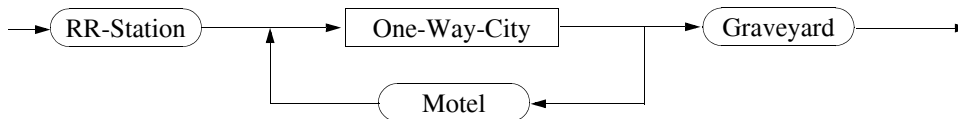
Syntaxdiagramm

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."

#### One-Way-County



#### One-Way-City

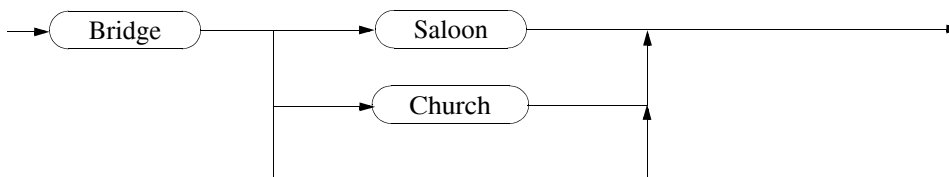


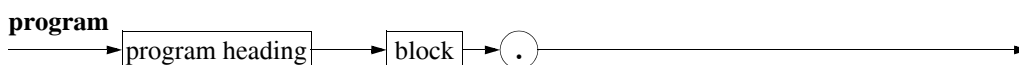
Abbildung 2.4 : Landkarte von One-Way-County, Stadtplan von One-Way-City im Syntaxdiagramm

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.

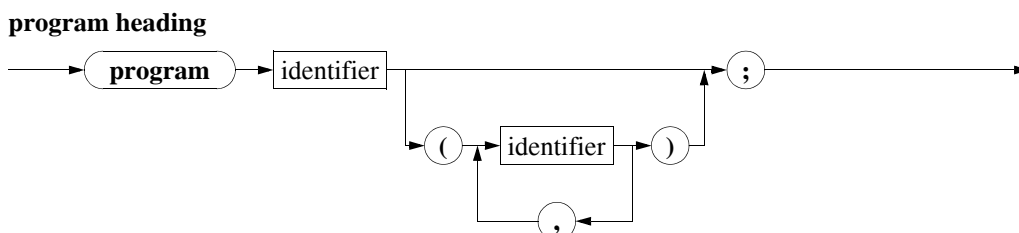
Terminale und Nichtterminale

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



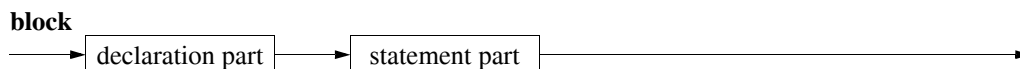
Der Programmkopf beginnt stets mit dem Schlüsselwort **program**, gefolgt vom Programmbezeichner. Danach können in Klammern Bezeichner von Ein- und Ausgabedateien angegeben werden, die außerhalb des Programms definiert sind. Der Programmkopf wird durch ein Semikolon vom Block getrennt. Wie in Pascal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



Deklarationsteil  
Vereinbarungsteil

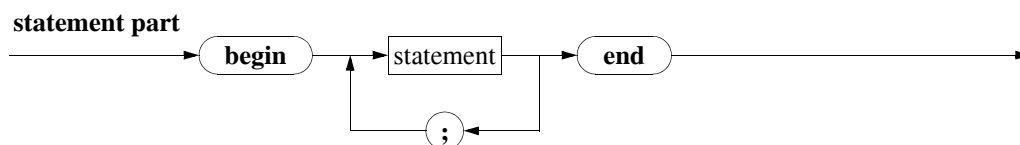
Unser Beispiel-Programm heißt `SehrEinfach`, die Angabe `output` weist darauf hin, daß Daten auf die Standardausgabe, also den Bildschirm, geschrieben werden sollen.

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,  
identifizier

Zeichenvorrat von  
Pascal

Buchstaben

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*).

<i>Buchstaben</i>	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



Umlaute gehören in Pascal nicht zu den Buchstaben. Groß- und Kleinbuchstaben werden nicht unterschieden.

<i>Ziffern</i>	0	1	2	3	4	5	6	7	8	9
<i>Spezialsymbole</i>	+	-	*	/						
	.	,	:	;						
	=	<>	<	<=	>	>=				
	:=	..	↑	'						
	(	)	[	]	{	}				

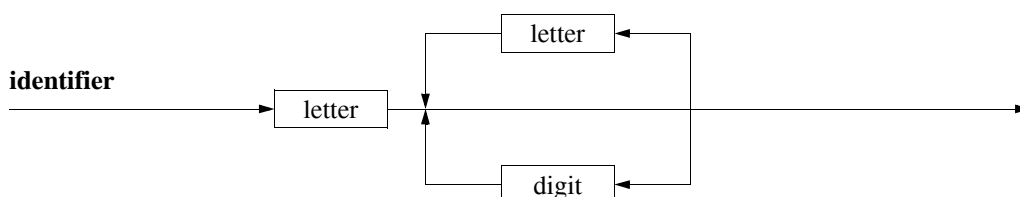
Spezialsymbole, die aus zwei Zeichen bestehen, werden zusammenhängend geschrieben, ohne einen Trenner dazwischen. Auf einigen Rechnern sind nicht alle Spezialsymbole verfügbar. Pascal sieht für diesen Fall folgende Ersatzdarstellungen vor:

<i>Ersatzdarstellungen</i>	( .	für	[
	. )	für	]
	( *	für	{
	* )	für	}
	@ oder ^	für	↑

*Schlüsselwörter*     **and array begin case const div do  
downto else end file for function  
goto if in label mod nil not of or  
packed procedure program record repeat  
set then to type until var while with**

Schlüsselwörter werden im handgeschriebenen Programm unterstrichen, um sie von frei wählbaren Bezeichnern zu unterscheiden (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)"). Diese Wörter dürfen nicht in einem anderen als dem durch die Pascal-Definition gegebenen Zusammenhang verwendet werden. Insbesondere dürfen diese Wörter nicht als Bezeichner verwendet werden, sie dürfen aber Bestandteil von Bezeichnern sein.

*Bezeichner* haben den folgenden Aufbau: Ein Bezeichner ist eine Folge von Buchstaben und Ziffern, die mit einem Buchstaben beginnt.



Demnach sind `SehrEinfach` und `output` aus unserem einführenden Beispiel gültige Bezeichner. Das Syntaxdiagramm "identifizier" kann in das Syntaxdiagramm

"program heading" überall eingesetzt werden, wo ein Kästchen mit der Beschriftung "identifizier" vorkommt. Ob die Zeichenfolgen `writeln`, `'Hallo'` usw. gültige Bezeichner sind, kann an dieser Stelle noch nicht entschieden werden. Hier sei schon soviel verraten: `writeln` ist der Bezeichner einer speziellen Anweisung, die übrigen Zeichenfolgen stellen keine Bezeichner dar, sondern bilden eine Zeichenkette (`string` vgl. Abschnitt "Der vordefinierte Datentyp `string`"), da sie in Hochkommata eingeschlossen sind.

Wie bereits erwähnt, unterscheidet Pascal nicht zwischen Groß- und Kleinbuchstaben. Die beiden Bezeichner

`BuchstabeUndZiffer` und `buchstabeundziffer`

sind also identisch. Um eine bessere Lesbarkeit zu erreichen, schreiben wir "zusammengesetzte Bezeichner" stets in der ersten Form (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").

Theoretisch können Bezeichner beliebige Längen haben und wir empfehlen, sinnvolle, d.h. "sprechende" und damit möglicherweise auch lange Bezeichner zu verwenden. In der Praxis beschränken viele Implementierungen die Zahl signifikanter Zeichen (leider) auf die ersten acht Stellen.

Die Bezeichner

`DiesIstEinLangerName` und `DiesIstEinSehrLangerName`

werden von solchen Implementierungen als identisch betrachtet, obwohl sie in Standard-Pascal verschieden sind.

Fünf Beispiele für gültige Bezeichner sind:

`pascal` `i` `Kurs1613` `E605` `H2CO3`

Beispiele für ungültige Bezeichner sind:

`58084Hagen` Ein Bezeichner muß mit einem Buchstaben beginnen.

`Hans-Werner` Ein Bezeichner darf keine Spezialsymbole enthalten.

`Mein Programm` Ein Bezeichner darf keine Leerzeichen enthalten.

Standardbezeichner

Einige Bezeichner, die *Standardbezeichner* (*standard identifier*, *predefined identifier*), werden von Pascal bereitgestellt, sie sind vordefiniert. In unserem Beispiel sind `output` und `writeln` Standardbezeichner.

Trennsymbole

Bezeichner, Zahlen und Schlüsselwörter dürfen nicht unmittelbar hintereinander stehen, sondern sie müssen durch Spezialsymbole, mindestens ein Leerzeichen (blank) oder mindestens ein Zeilenende-Zeichen (end-of-line = `eoln`), sog. *Trennsymbole* (*separator*), voneinander getrennt werden. Im übrigen dürfen Leerzeichen und Zeilenwechsel an jeder beliebigen Stelle in beliebiger Anzahl stehen, ausgenommen innerhalb von Bezeichnern, Zahlen oder Schlüsselwörtern. Aus Vereinfachungsgründen stellen wir diese Bedingungen nicht in Syntaxdiagrammen dar.

Wir können unser Beispielprogramm also auch folgendermaßen notieren:

```

program
  SehrEinfach
    (output);

begin
  writeln ('Hallo! Viel Spaß beim Programmieren!')
end.

```

oder so:

```

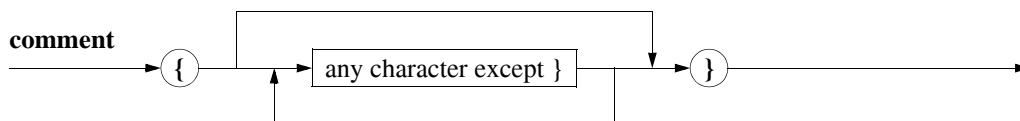
program SehrEinfach (output);begin writeln
('Hallo! Viel Spaß beim Programmieren!') end.

```

Die zweite Schreibweise ist zwar sehr platzsparend und syntaktisch durchaus korrekt, aber Sie werden zugeben, daß die Programmstruktur (ebenso wie bei der ersten Version) nur noch schwer erkennbar ist. Wir werden uns bei der Formatierung von Programmen fortan an gewissen Regeln orientieren, damit sie leicht lesbar sind und ihre Struktur klar erkennbar ist. Eine Zusammenstellung dieser Regeln finden Sie in Abschnitt 3.8 "Programmierstil (Teil 1)".

Als Trennsymbol wirkt außer dem Leerzeichen und dem Zeilenwechsel der *Kommentar* (*comment*). Er dient zum Einfügen von erklärenden Bemerkungen in den Programmtext und hat die Gestalt

Kommentar



Jede Zeichenfolge, die zwischen { und } eingeschlossen ist und nicht das Zeichen } enthält, wird als Kommentar angesehen. Kommentare haben keinerlei Einfluß auf die Ausführung des Programms, sondern dienen ausschließlich dem besseren Verständnis des Lesers. Es besteht keine Beschränkung hinsichtlich der Länge eines Kommentars, er kann sich auch über mehrere Zeilen erstrecken.

```

program SehrEinfach (output);
{ Dies ist das erste Programm-Beispiel
  im Kurs 1613 }
begin
  writeln ('Hallo! Viel Spaß beim Programmieren!')
end.

```

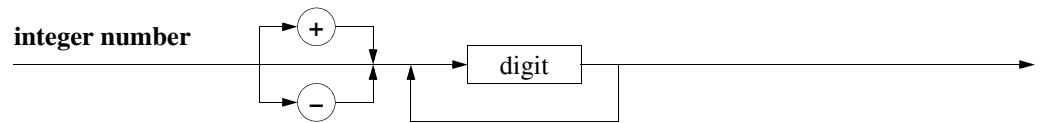
Auf einigen Rechnern stehen die Zeichen { und } nicht zur Verfügung, und werden durch (\* bzw. \*) ersetzt.

### 3.4 Zahlen

Pascal kennt zweierlei Arten von Zahlen: integer -Zahlen und real-Zahlen.

integer

Eine `integer`-Zahl stellt eine ganze Zahl als eine beliebig lange Folge von Ziffern dar, die bei Bedarf ein Vorzeichen tragen kann:



Beispiele für syntaktisch korrekte `integer`-Zahlen sind:

1613, -1613 und +1613

Folgende Konstruktionen sind keine syntaktisch korrekten `integer`-Zahlen:

6-437      Eine `integer`-Zahl darf nur Ziffern enthalten.

-6.0        Eine `integer`-Zahl darf keinen Dezimalpunkt enthalten.

Das Syntaxdiagramm erlaubt zwar die Verwendung von `integer`-Zahlen beliebiger Größe, doch jede Pascal-Implementierung hat bestimmte Intervallgrenzen, innerhalb derer ein `integer`-Wert liegen muß. Das ist deshalb nötig, weil der Platz zur Speicherung einer `integer`-Zahl im Rechner limitiert ist. Ältere DOS-Systeme nutzen z.B. 16 Bit (2 Byte) pro Integer, womit  $2^{16}$  `integer`-Zahlen darstellbar sind, neuere Systeme nutzen 32 Bit oder mehr. Von der Anzahl darstellbarer Zahlen entfällt die Hälfte auf den positiven Zahlenbereich inklusive der Null, die andere Hälfte auf den negativen Zahlenbereich. Der größte darstellbare `integer`-Wert ist durch den Standardbezeichner `maxint` definiert, der bei 16 Bit z.B. den Wert  $2^{15}-1=32767$  besitzt. Als `integer`-Zahlen darstellbar sind also in Pascal alle ganzen Zahlen, die in dem Intervall  $I=[-\text{maxint}-1, \text{maxint}]$  liegen.

Die folgende Rechnung zeigt, daß Rechenergebnisse von der Reihenfolge abhängen, in der Ausdrücke ausgewertet werden. So kommt es für `maxint` = 100 bei der Rechnung

$$(60 + 50) - 30$$

zu einem Überlauf (overflow), da die Addition  $60 + 50$  aus dem zur Verfügung stehenden Zahlenbereich herausführt. Die Rechnung

$$60 + (50 - 30) = 80$$

dagegen kann problemlos durchgeführt werden. Das Assoziativgesetz der Addition gilt also im Bereich `integer` nur, solange alle berechneten Zwischenergebnisse im Intervall  $I$  liegen.

Zahlen, die über die `integer`-Grenzen hinausgehen, oder Zahlen, die einen Fraktionsteil enthalten, können als `real`-Zahlen dargestellt werden. Da ein digitaler Computer keine unendlichen Dezimalbrüche, wie z.B.  $\pi=3,14159\dots$ , darstellen kann, sind `real`-Zahlen auf diejenigen Zahlen beschränkt, die in sogenannter *Gleitpunktdarstellung* eine vorgegebene Anzahl von Stellen nicht überschreiten.

`maxint`

`real`

Gleitpunktdarstellung

Die Gleitpunktdarstellung ist eine Methode zur Darstellung von Zahlen in halblogarithmischer Form. Seien mit  $\mathbb{N}$  die natürlichen Zahlen, mit  $\mathbb{Z}$  die ganzen Zahlen und mit  $\mathbb{R}$  die reellen Zahlen bezeichnet. Sei weiterhin  $z \in \mathbb{R}_+$ , d.h. eine beliebige positive reelle Zahl. Dann kann  $z$  bei gegebenem  $b$  stets eindeutig in der Form

$$z = m \cdot b^e \quad \text{mit } b \in \mathbb{N}, b \geq 2, e \in \mathbb{Z} \text{ und } m \in \mathbb{R}_+: 1 \leq m < b.$$

geschrieben werden.  $m$  heißt *Mantisse*,  $e$  *Exponent* und  $b$  *Basis*. In der Praxis wählt man als Basis  $b$  meist eine der Zahlen 2, 10 oder 16. Die Mantisse  $m$  gibt die Ziffernfolge von  $z$  zur Basis  $b$  an, wobei die sogenannte *Normalisierung*  $1 \leq m < b$  sicherstellt, daß genau eine Stelle vor dem Punkt und beliebig (sogar unendlich) viele Stellen nach dem Punkt auftreten. Dadurch wird die Darstellung eindeutig. Der Exponent  $e$  gibt an, mit welcher Potenz von  $b$  die Mantisse zu multiplizieren ist, um  $z$  zu erhalten, d.h. er charakterisiert die Größenordnung der Zahl.

Normalisierte Gleitpunktdarstellungen zur Basis 10 sind dann beispielsweise:

$$\begin{array}{ll} 1.89217 \cdot 10^2 & (= 189,217) \text{ und} \\ 1 \cdot 10^3 & (= 1000). \end{array}$$

Ist  $z \in \mathbb{R}_-$ , d.h. eine beliebige negative reelle Zahl, so wird der Absolutbetrag von  $z$  nach der oben beschriebenen Methode dargestellt und der Mantisse ein Minuszeichen vorangestellt. Damit bleibt die Darstellung eindeutig. Beispiele sind

$$\begin{array}{ll} -2.4 \cdot 10^0 & (= -2,4) \text{ und} \\ -1.3 \cdot 10^{-3} & (= -0,0013). \end{array}$$

Schließlich wird die Zahl 0 in der Form

$$0 \cdot b^0$$

dargestellt. Wegen der endlichen Speicherplatzgrenzen steht sowohl für die Mantisse als auch für den Exponenten nur eine begrenzte Anzahl von Stellen zur Verfügung. Daraus folgt, daß

- a) nur endlich viele reelle Zahlen als *real*-Zahl darstellbar sind und
- b) jede darstellbare reelle Zahl "nur" eine rationale Zahl mit endlicher Dezimalentwicklung ist (falls  $b=10$ ), also nur endlich viele Nachkommastellen besitzt.

Die Endlichkeit der Darstellung bedingt, daß viele Zahlen, die sich z.B. bei der Berechnung von Zwischenergebnissen ergeben, nur noch näherungsweise dargestellt werden können. Die daraus resultierenden sogenannten *Rundungsfehler* können sich im Laufe von Rechnungen zu total falschen Ergebnissen aufsummieren. Dies soll folgendes Beispiel zeigen.

Angenommen, es stehen für die Mantisse vier und für den Exponenten zwei Stellen zur Verfügung; die Basis  $b$  sei 10.

Wir berechnen  $2000 + 0.7 - 2000$  auf zwei verschiedene Arten:

Mantisse  
Exponent  
Basis  
Normalisierung

Rundungsfehler

$$\begin{aligned}
 \text{a) } (2000+0.7)-2000 &= (2.000 \cdot 10^3 + 7.000 \cdot 10^{-1}) - 2.000 \cdot 10^3 \\
 &= (2.000 \cdot 10^3 + 0.000 \cdot 10^3) - 2.000 \cdot 10^3 \\
 &= 0 \\
 \text{b) } (2000-2000)+0.7 &= (2.000 \cdot 10^3 - 2.000 \cdot 10^3) + 7.000 \cdot 10^{-1} \\
 &= 0.0000 \cdot 10^0 + 7.000 \cdot 10^{-1} \\
 &= 7.000 \cdot 10^{-1} \\
 &= 0.7 .
 \end{aligned}$$

Die Addition und Subtraktion zweier *real*-Zahlen wird auf die Addition bzw. Subtraktion der zugehörigen Mantissen zurückgeführt. Dazu müssen beide den gleichen Exponenten haben. Daher muß zuvor eine Angleichung an den größeren Exponenten erfolgen. Die Zahl 0.7 wird hierbei in Fall a) ausgelöscht. Wenn anschließend das Ergebnis mit 1000 multipliziert wird, so weicht das Ergebnis bereits um 700 vom exakten Wert ab. Auch dieser Effekt – entstanden durch Auslöschung kleinerer Zahlen – wird als Rundungsfehler bezeichnet.

Die Verwendung von *real*-Zahlen ist also nicht unproblematisch. Speziell bei der Programmierung numerischer Algorithmen muß beachtet werden, daß die Ergebnisse durch Rechenungenauigkeiten verfälscht werden können.

Festpunktdarstellung

Eine *real*-Zahl kann in Pascal auf zwei verschiedene Arten geschrieben werden. Die erste Form bezeichnet man als sogenannte *Festpunktdarstellung*. Jede solche Zahl enthält einen Dezimalpunkt; sowohl vor als auch hinter ihm muß mindestens je eine Ziffer stehen. Sie kann mit oder ohne Vorzeichen geschrieben werden. Dies entspricht einer (nicht normalisierten) Gleitpunktdarstellung zur Basis 10 mit dem Exponenten 0.

Beispiele für *real*-Zahlen in Festpunktdarstellung sind:

0.0, 0.873, -74.1 und 73.36789

Die folgenden Zeichenfolgen sind aus den angegebenen Gründen keine *real*-Zahlen:

0.            Der Dezimalpunkt steht nicht zwischen Ziffern.  
 .873        Dito.  
 -74,1       Das Dezimalkomma ist nicht erlaubt.

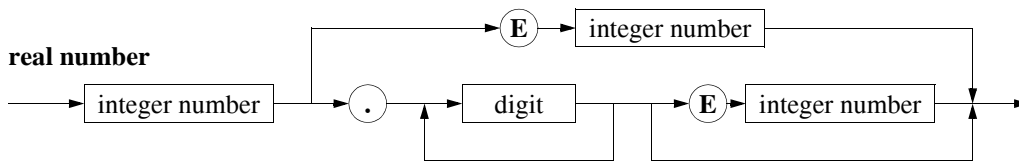
nicht normalisierte  
Gleitpunktdarstellung

Als zweite Form der Darstellung wird eine *nicht normalisierte Gleitpunktdarstellung* zur Basis 10 zugrundegelegt. Die Mantisse kann eine *integer*-Zahl oder eine *real*-Zahl in Festpunktdarstellung sein, die Basis 10 wird nicht mit aufgeführt und der Exponent – eine *integer*-Zahl – durch den Buchstaben E von der Mantisse getrennt.

Beispiele für *real*-Zahlen in Gleitpunktdarstellung sind:

0E0, 8.73E+02, 741E-1 und 0.7336789E2.

Als Syntaxdiagramm einer `real`-Zahl erhalten wir also:



Zum Schluß geben wir einige syntaktisch korrekte Schreibweisen für die `real`-Zahl `1613.0` an:

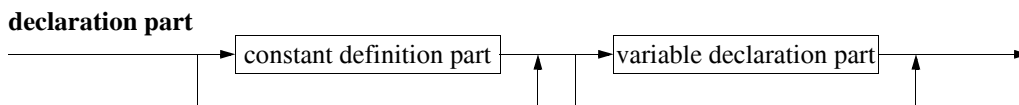
`1613.0`, `1.613E3`, `161.3E01`, `1613E0` und `16130E-1`.

Für die `integer`-Zahl `1613` gibt es dagegen nur zwei syntaktisch korrekte Schreibweisen:

`1613` und `+1613`.

### 3.5 Beschreibung der Daten

Wir haben bereits festgestellt, daß sich der Block eines Pascal-Programms aus einem Deklarations- und einem Anweisungsteil zusammensetzt. Der Deklarationsteil eines Programms muß alle nicht vordefinierten Bezeichner erklären, die im Programm (insbesondere im Anweisungsteil) verwendet werden. Wir erhalten das vorläufige Syntaxdiagramm:



Jeder der Teile kann weggelassen werden, wichtig ist allerdings die Reihenfolge: *Konstantendefinitionen* müssen in Pascal immer vor *Variablendeklarationen* erfolgen. Dies wird in anderen Programmiersprachen nicht zwingend gefordert.

Später werden wir noch Typdefinitionen und Prozedurdeklarationen kennenlernen. Beachten Sie bitte die Unterscheidung zwischen Definition und Deklaration: Konstanten und Typen werden definiert, d.h. es wird lediglich ein Name für die Konstante bzw. den Typ vereinbart. Variablen und Prozeduren werden dagegen deklariert, denn hier werden weitergehende Maßnahmen getroffen. Für eine Variable wird z.B. neben der Festlegung eines Namens der erforderliche Speicherplatz angelegt. Die Bezeichnungen Definition bzw. Deklaration werden in der Literatur nicht immer konsequent unterschieden.

Konstantendefinition  
Variablendeklaration

#### 3.5.1 Datentypen

Ein Programm manipuliert Daten, um schließlich ein benötigtes Ergebnis zu liefern. Die Daten, mit denen ein Programm arbeitet, werden im Computer in binärer Darstellung gespeichert. Höhere Programmiersprachen erlauben es dem Programmierer, die maschineninterne Repräsentation der Daten zu ignorieren und diese

Datentyp

typisierte  
Programmiersprache

stattdessen durch Angabe ihres Typs zu charakterisieren. Ein *Datentyp* definiert eine Menge von Werten mit auf dieser Menge ausführbaren Operationen. Pascal verlangt wie die meisten höheren Programmiersprachen, daß jedes Datenobjekt (Konstante, Variable) eindeutig einem Typ zugeordnet ist. Wir sprechen auch von *typisierten Programmiersprachen*. Dabei gilt im einzelnen:

- Der Typ eines Datenobjekts bestimmt die möglichen Werte, die das Datenobjekt annehmen kann, und die Operationen, die auf das Datenobjekt angewendet werden können.
- Jedes Datenobjekt ist von genau einem Typ.
- Jede Operation erfordert Operanden bestimmten Typs und produziert ein Ergebnis eines bestimmten Typs.

Durch diese Regeln wird der Programmierer davor bewahrt, ungeeignete Kombinationen von Daten und Operationen zu verwenden, die gewöhnlich zu schwer lokalisierbaren Fehlern während der Programmausführung führen. Dieser Schutz ist bei der Verwendung untypisierter Sprachen nicht gegeben.

Pascal kennt die vier Standarddatentypen `integer`, `real`, `char` und `boolean`. Diese Standarddatentypen gehören zu den unstrukturierten Datentypen. Der Programmierer kann selbst weitere unstrukturierte sowie strukturierte Datentypen definieren. Wir beginnen mit den vier Standarddatentypen.

Beim ersten Lesen sind die vielen Details oft verwirrend. Es kommt aber nicht darauf an, sofort alle Einzelheiten zu verstehen. Diese Abschnitte sind auch zum Nachschlagen beim Erstellen der ersten Programme gedacht.

### Der Standarddatentyp `integer`

Dieser Typ definiert ein Intervall der ganzen Zahlen; jeder Wert dieses Typs ist folglich eine ganze Zahl. Wie in Abschnitt 3.4 "Zahlen" bereits gesagt, steht in jeder Implementierung von Pascal nur ein beschränktes Intervall der ganzen Zahlen als Wertebereich für den Typ `integer` zur Verfügung. Die Darstellung von `integer`-Zahlen wurde im Abschnitt 3.4 "Zahlen" beschrieben.

Der Datentyp `integer` verfügt über die folgenden arithmetischen Standardoperationen:

- + Addition,
- Subtraktion,
- \* Multiplikation,
- div** ganzzahlige Division ohne Rest und
- mod** Restbildung bei ganzzahliger Division.



Es sind *dyadische Infixoperatoren*, d.h. sie werden zwischen zwei Operanden notiert, also  $a+b$ . Die Zeichen  $+$  und  $-$  können auch als *monadische Präfixoperatoren* verwendet werden, also  $+a$  oder  $-a$  zur Vorzeichenumkehr (Negation).

dyadischer Operator  
monadischer  
Operator

Addition, Subtraktion und Multiplikation sind in der üblichen Weise definiert, solange man den Wertebereich von  $-\text{maxint}-1$  bis  $+\text{maxint}$  nicht verläßt.

Für **mod** und **div** gilt:

7 **div** 2 = 3,      7 **mod** 2 = 1,  
14 **div** 3 = 4,      14 **mod** 3 = 2,  
-14 **div** 3 = -4 und -14 **mod** 3 = -2.



Wir erinnern uns, daß die arithmetischen Gesetze beim Datentyp `integer` nicht uneingeschränkt gelten, da `integer` ja nur eine Teilmenge der ganzen Zahlen definiert. Insbesondere dürfen die arithmetischen Operationen für `integer` nicht uneingeschränkt angewendet werden, da die (Zwischen-)Ergebnisse einer Berechnung aus dem Intervall herausfallen können.

Neben diesen Infix- und Präfixoperatoren werden in Pascal auch sogenannte *Standardfunktionen* bereitgestellt. Die Bedeutung solcher Funktionen ist wie in der Mathematik: Eine Funktion weist jedem Argument des Wertebereichs genau einen Wert im Bildbereich zu. Zwei Standardfunktionen sind:

Standardfunktionen

`abs (x)` Absolutbetrag und  
`sqr (x)` Quadrat.

Sowohl der Operand (das Argument) als auch das Ergebnis sind in beiden Fällen vom Typ `integer`. Beispiele zur Verwendung sind:

`abs (7) = 7,`  
`abs (-6) = 6,`  
`sqr (3) = 9 und`  
`sqr (-4) = 16.`

Das Quadrat eines `integer`-Wertes kann natürlich ebenso gut als `x*x` geschrieben werden. Die Verwendung von `sqr (x)` kann aber die Lesbarkeit des Programms erhöhen.

Ebenso wie andere unstrukturierte Datentypen (außer `real`) definiert `integer` eine lineare Ordnung auf den Werten - entsprechend der üblichen Ordnung auf den ganzen Zahlen. Jeder Wert außer dem kleinsten hat einen Vorgänger, jeder außer dem größten hat einen Nachfolger. Für einen `integer`-Wert kann der Nachfolger durch Addition von 1 und der Vorgänger durch Subtraktion von 1 ermittelt werden. Für alle Datentypen, auf denen eine Ordnung definiert ist, stellt Pascal zwei Standardfunktionen `succ` und `pred` zur Verfügung, die wie folgt definiert sind:

`succ (x)` ergibt den unmittelbaren Nachfolger von `x`, falls er existiert,  
`pred (x)` ergibt den unmittelbaren Vorgänger von `x`, falls er existiert.

Für `integer` gilt also:

$$\text{pred } (x) = \begin{cases} x - 1, & \text{falls } x > -\text{maxint} - 1 \\ \text{undefiniert}, & \text{falls } x = -\text{maxint} - 1 \end{cases}$$

$$\text{succ } (x) = \begin{cases} x + 1, & \text{falls } x < \text{maxint} \\ \text{undefiniert}, & \text{falls } x = \text{maxint} \end{cases}$$

Beispiele sind:

`pred (7) = 6,`  
`pred (-7) = -8,`  
`succ (7) = 8` und  
`succ (-7) = -6.`

### Der Standarddatentyp `real`

Der Wertebereich des Datentyps `real` umfaßt die Teilmenge der reellen Zahlen (genaugenommen der rationalen Zahlen), die man als `real`-Zahl darstellen kann (siehe Abschnitt 3.4 "Zahlen"). Pascal stellt eine Anzahl von Operationen zur Verfügung, die mit `real`-Operanden `real`-Ergebnisse erzeugen:

`+` Addition,  
`-` Subtraktion,  
`*` Multiplikation,  
`/` Division.

Wie auch bei `integer` sind dies dyadische Infixoperatoren, wobei `+` und `-` auch als monadische Präfixoperatoren benutzt werden können.

Daneben sind für `real`-Zahlen noch folgende Standardfunktionen verfügbar:

<code>abs (x)</code>	Betragsfunktion, analog wie bei <code>integer</code> ,
<code>sqr (x)</code>	Quadratfunktion, analog wie bei <code>integer</code> ,
<code>sin (x)</code>	Sinusfunktion,
<code>cos (x)</code>	Cosinusfunktion,
<code>arctan (x)</code>	Arcustangensfunktion,
<code>exp (x)</code>	Exponentialfunktion $e^x$ ,
<code>ln (x)</code>	natürlicher Logarithmus für $x > 0.0$ und
<code>sqrt (x)</code>	Quadratwurzelfunktion für $x \geq 0.0$ .

Beachten Sie, daß `abs` und `sqr` `integer`-Werte liefern, wenn die Argumente vom Typ `integer` sind, und `real`-Werte, wenn die Argumente vom Typ `real` sind.

### Der Standarddatentyp `char`

Der Wertebereich des Datentyps `char` (Abk. von engl. character = dt. Zeichen) ist die Menge aller darstellbaren Zeichen des Pascal-Zeichensatzes sowie Steuerzeichen, die nicht ausgegeben werden können, aber das Ausgabegerät beeinflussen können. In einem Pascal-Programm wird ein `char`-Zeichen durch Einschließen des Zeichens in Hochkommata (Apostrophe) definiert, also z.B:

```
'a'  '9'  'B'  '?'  ' ' (Leerzeichen)
```

Das Hochkomma selbst wird durch Verdoppelung des Zeichens und Einschluß in Hochkommata (also vier Hochkommata) definiert:

```
''''
```

Auf dem Wertebereich des Datentyps `char` existiert ebenfalls eine Ordnung. Es gibt also ein "kleinstes", ein "zweitkleinstes" usw. und ein "größtes" Zeichen. Wenn wir uns die Zeichen der Reihe nach aufsteigend sortiert hingeschrieben denken, dann gibt die Standardfunktion `ord (c)` die Position des Zeichens `c` in dieser aufsteigend sortierten Folge an, während `chr (i)` das Zeichen an der Position `i` angibt:

`ord (c)` die Ordinalzahl des Zeichens `c` (in der Menge aller Zeichen),  
`chr (i)` das Zeichen mit der Ordinalzahl `i`, wenn es existiert.

Offensichtlich sind also `ord` und `chr` inverse Funktionen. Daher gilt stets

$$\text{chr} (\text{ord} (c)) = c \quad \text{und} \quad \text{ord} (\text{chr} (i)) = i.$$

Welches Resultat die `ord`-Funktion für die einzelnen Zeichen genau liefert, hängt davon ab, wie der Computer die Zeichen intern verschlüsselt. Oft wird der ASCII-Code (sprich: aski, Abk. für American Standard Code for Information Interchange) verwendet, bei dem jedes Zeichen durch eine Zahl zwischen 0 und 127 (bzw. 255 inkl. Sonderzeichen) dargestellt wird.

In allen Rechnern sind die Zeichen für die Ziffern eine lückenlos geordnete Menge, so daß gilt:

```
ord ('0') = ord ('1') - 1,
ord ('1') = ord ('2') - 1,
...
ord ('8') = ord ('9') - 1.
```

In den meisten Rechnern bilden die Großbuchstaben eine ebensolche lückenlos geordnete Menge, also:

`char`

ASCII-Code

```
ord ('A') = ord ('B') - 1,
ord ('B') = ord ('C') - 1,
    ...
ord ('Y') = ord ('Z') - 1.
```

Analoges gilt für die Kleinbuchstaben.

Die Standardfunktionen `pred` und `succ` sind auch für Argumente vom Typ `char` definiert, und zwar derart, daß gilt:

```
pred (c) = chr (ord (c) - 1) und
succ (c) = chr (ord (c) + 1).
```

### Der Standarddatentyp `boolean`

Der Typ `boolean` besteht aus den beiden Wahrheitswerten `wahr` und `falsch`, die in Pascal durch die Standardbezeichner `true` (=wahr) und `false` (=falsch) vordefiniert sind. `true` und `false` können mittels der folgenden, sogenannten *booleschen Operatoren* verknüpft werden, die als Ergebnis einen Wert vom Typ `boolean` liefern:

**and** Konjunktion, logisches "und",  
**or** Disjunktion, logisches einschließendes "oder",  
**not** Negation, logische Verneinung.

Die Bedeutung ist die in der Logik übliche und durch folgende Wertetabelle festgelegt:

x	y	x <b>and</b> y	x <b>or</b> y	<b>not</b> x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

**x and y** liefert den Wert `true` dann und nur dann,  
wenn `x` den Wert `true` und `y` den Wert `true` besitzt.

**x or y** liefert den Wert `false` dann und nur dann,  
wenn `x` den Wert `false` und `y` den Wert `false` besitzt.

**not x** liefert den Wert `true` dann und nur dann,  
wenn `x` den Wert `false` besitzt.

Weitere Operatoren, die boolesche Werte liefern, sind die sogenannten *Vergleichsoperatoren*. Sie sind definiert für alle unstrukturierten Datentypen (also insbesondere für die vorgestellten Standarddatentypen `integer`, `real`, `char` und `boolean`):

boolescher Operator

Vergleichsoperator

```

=      gleich
<>    nicht gleich, ungleich
<      kleiner als
<=     kleiner als oder gleich, nicht größer
>      größer als
>=     größer als oder gleich, nicht kleiner

```

Diese Symbole entsprechen den üblichen Vergleichen, wobei statt " $\leq$ " das Spezialsymbol " $<=$ ", statt " $\geq$ " das Spezialsymbol " $>=$ " und statt " $\neq$ " das Spezialsymbol " $<>$ " verwendet wird, da die üblichen Zeichen nicht immer auf einer Rechnertastatur vorgesehen sind. Eine Vergleichsoperation liefert einen Wahrheitswert in Abhängigkeit von den Werten der Operanden. Die aufgeführten Operatoren können auf beliebige Paare von Operanden desselben unstrukturierten Typs angewandt werden. Für `integer`- und `real`-Operanden haben die Operatoren ihre normale Bedeutung, also beispielsweise:

```

7 = 11      ergibt false,
7 < 11      ergibt true und
3.4 <= 5.9  ergibt true.

```

Für Operanden vom Typ `char` entspricht das Ergebnis der Anwendung des Vergleichs auf die zugehörigen Ordinalzahlen. Es gilt also:

```
'd' < 'm'    ist äquivalent zu ord ('d') < ord ('m').
```

Außerdem gilt `false < true`.

Schließlich stellen wir noch die Standardfunktion `odd` vor, die auf einen `integer`-Operanden angewendet ein boolesches Ergebnis ergibt. Für ein Argument `x` vom Typ `integer` gilt:

```
odd (x) liefert das Ergebnis true, wenn x ungerade ist; andernfalls
false.
```

Eine Verknüpfung von Operanden mittels der vorgestellten Operatoren und der Standardfunktion `odd` führt zu booleschen Ausdrücken. Diese behandeln wir in einem späteren Unterabschnitt (vgl. Abschnitt "Zuweisung und Ausdruck").

Zum Abschluß der Standarddatentypen noch eine wichtige Bemerkung: Die Wertebereiche der Standarddatentypen sind paarweise disjunkt. Der Wert `6.0` vom Typ `real` ist also nicht identisch mit dem Wert `6` vom Typ `integer` und das Zeichen `'6'` vom Typ `char` ist mit keinem der beiden Werte identisch.

### Der vordefinierte Datentyp `string`

`string`

Der Datentyp `string` umfaßt die Menge aller Zeichenketten bis zu einer fest vorgegebenen, maximalen Länge (i.a. 255 Zeichen). Dabei verstehen wir unter einer Zeichenkette eine Aneinanderreihung von Zeichen vom Typ `char`.

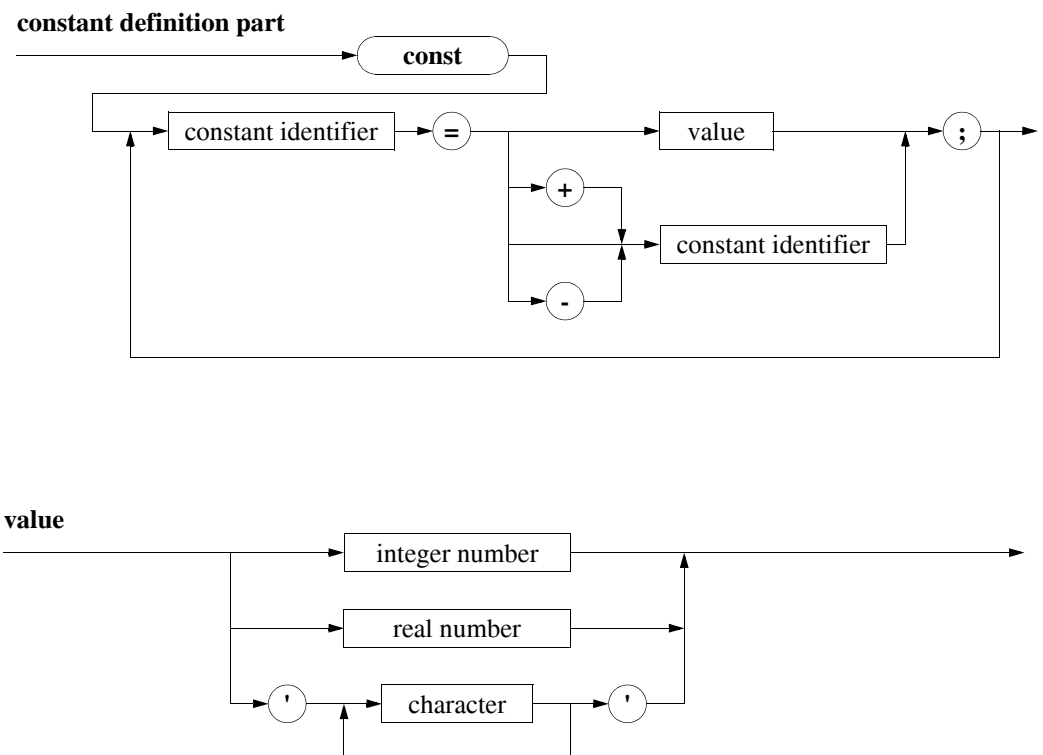
In Standard-Pascal ist der Typ `string` nicht vordefiniert; er wird aber von allen gängigen Pascal-Compilern bereitgestellt (insbesondere von Turbo-Pascal). Wir nehmen im folgenden an, daß der Typ `string` wie in Turbo-Pascal vordefiniert ist.

Werte vom Typ `string` werden analog zu einfachen Zeichen durch Einschließen in Hochkommata dargestellt, also z.B. `'Stringkonstante'` oder `'Hallo! Viel Spaß beim Programmieren!'`. Sie können auch in Ausgabeanweisungen auftreten, wie wir in unserem ersten Beispiel schon gesehen haben.

Auf dem Datentyp `string` sind analog zu den Standarddatentypen Vergleichsoperationen definiert. Die zugrundeliegende Ordnung ist allerdings von der rechnerinternen Darstellung abhängig. Sie wird im wesentlichen durch die Ordnung des Typs `char` festgelegt, da Vergleiche zeichenweise durchgeführt werden bis zum ersten verschiedenen Zeichen bzw. zum Ende der kürzeren Zeichenkette.

### 3.5.2 Definition von Konstanten

Wir geben oft benutzten oder wichtigen Werten einen leicht lesbaren Bezeichner, indem wir sie als Konstanten definieren. Wie der Begriff schon sagt, kann der Wert einer Konstanten während des Programmlaufs nicht verändert werden. Die Werte für Konstanten sind Zahlen oder Zeichenketten. Auch bereits definierte Konstanten dürfen, wahlweise mit Vorzeichen, den Wert neuer Konstanten definieren. Die Konstantendefinition (constant definition part) hat die Form:



Die Syntax-Diagramme zu "integer number" und "real number" kennen Sie aus Abschnitt 3.4 "Zahlen". Die Syntax eines "constant identifier" stimmt mit der eines "identifier" überein.

Durch die gewählte Darstellung eines Zahl-Wertes wird der Typ der Konstanten festgelegt. Beispiele hierzu sind

```
const
PI = 3.141592654;    {Konstante vom Typ real}
MAX = 1000;          {Konstante vom Typ integer}
MIN = -MAX;           {ebenfalls eine Konstante
                      vom Typ integer}
```

Besonders bequem sind Konstanten, wenn man lange Zahlen abkürzen will, wie das folgende Beispiel zeigt:

#### Beispiel 3.5.2.1

Ein mathematisches Programm macht ständig Gebrauch von dem real-Wert 3.141592654, einer Approximation der Zahl  $\pi$ . Wenn dieser Wert überall, wo er benötigt wird, ausgeschrieben würde, hätte das folgende Nachteile:

- Die Ziffernfolge hat weit weniger Aussagekraft als das Symbol  $\pi$ .
- Das wiederholte Niederschreiben der Ziffernfolge ist mühsam und kann leicht zu Fehlern führen.
- Eine Entscheidung, z.B. die Genauigkeit des verwendeten Wertes von  $\pi$  zu ändern, erfordert eine Änderung an jeder Stelle des Programms, an der die Ziffernfolge auftritt.

Eine ideale Lösung für den Programmierer wäre es, das Symbol  $\pi$  direkt zu verwenden und den Wert, den es repräsentieren soll, nur einmal im Programm zu definieren. Da aber griechische Buchstaben in Pascal nicht zur Verfügung stehen, verwenden wir sinnvollerweise einen mnemotechnischen Bezeichner, z.B. PI.

```
program Kreisumfang (input, output);
{ berechnet den Kreisumfang bei gegebenem Radius }

const
PI = 3.141592654;

{ Jetzt kommt eine Variablendeklaration. Sie finden
  sie im folgenden Abschnitt erläutert }

var
Radius,
Umfang : real;

begin
  writeln ('Berechnung des Kreisumfanges.');
```

input

```

write ('Geben Sie den Radius ein: ');
readln (Radius);
Umfang := 2.0 * PI * Radius;
writeln ('Der Umfang betraegt: ', Umfang)
end. { Kreisumfang }

```

Die Angabe input im Programmkopf weist darauf hin, daß Daten von der Standardeingabe, in den meisten Fällen also von der Tastatur, zu lesen sind. Wird dieses Programm ausgeführt und für den Radius die Zahl 22.5 eingegeben, erscheinen auf dem Bildschirm die Zeilen:

```

Berechnung des Kreisumfanges.
Geben Sie den Radius ein: 22.5
Der Umfang betraegt: 1.4137167E02

```



In diesem Kurs werden alle selbstdefinierten Konstanten-Bezeichner durchgängig mit Großbuchstaben geschrieben. Das ist keine syntaktische Forderung, sondern eine Frage des Programmierstils: So lassen sich Konstanten sofort von Variablen unterscheiden (vgl. auch Abschnitt 3.8 "Programmierstil (Teil 1)").

Konstanten können auch Zeichenketten repräsentieren. Beispiele für Textkonstanten sind:

```

const
LEER5 = '      ';
LINIE5 = '-----';

```

Das folgende Programm benutzt eine Textkonstante:

```

program Einfach (output);
{ Programmbeispiel fuer eine Textkonstante }

const
GRUSS = 'Guten Tag! ';

begin
  writeln (GRUSS, 'Viel Spaß beim Programmieren!')
end. { Einfach }

```

```

Guten Tag! Viel Spaß beim Programmieren!

```

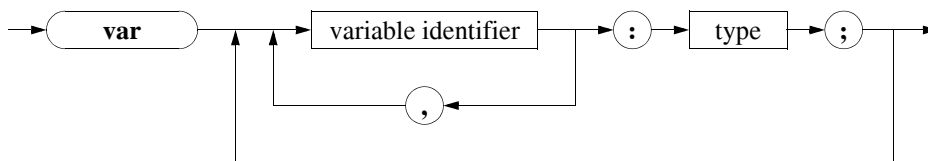


### 3.5.3 Deklaration von Variablen

Variablen können (im Gegensatz zu Konstanten) ihren Wert im Laufe der Ausführung eines Programms ändern. Die möglichen Werte einer Variablen (Wertebereich) und die zulässigen Operationen auf dieser Variablen werden durch den Typ der Variablen bestimmt.

Jede Variable wird durch eine Variablendeklaration eingeführt. Die Deklarationen werden im Variablendeklarations-Teil (variable declaration part) zusammengefaßt. Wir können auch mehrere Variablen desselben Typs zusammen deklarieren, indem wir die Bezeichner durch Kommata getrennt auflisten und dann den gemeinsamen Typ nach einem Doppelpunkt anfügen:

variable declaration part



Die Syntax von "variable identifier" stimmt mit der eines "identifier" überein. "type" kann ein Bezeichner für einen Standarddatentyp oder einen vor- bzw. selbstdefinierten Datentyp sein. Zur Zeit kennen wir außer dem vordefinierten Datentyp `string` nur die vier Standarddatentypen `integer`, `real`, `char`, `boolean`. Später führen wir selbstdefinierte Datentypen ein, die Bezeichner nach den bekannten Regeln tragen können.

Korrekte Variablendeklarationen sind also zum Beispiel:

```

var
min,
max : integer;
  
```

und

```

var
min : integer;
max : integer;
  
```

Im weiteren werden wir Konstanten- und Typ-Definitionen sowie Variablen-Deklarationen als *Vereinbarungen* bezeichnen.

Vereinbarungen

### 3.5.4 Eindeutigkeit von Bezeichnern

Vereinbarungen haben u.a. den Sinn, Variablen und Konstanten (später auch Datentypen und Prozeduren) Bezeichner zuzuordnen. Pascal verlangt, daß Bezeichner innerhalb eines Gültigkeitsbereichs einmalig sind. Das Konzept des Gültigkeitsbereichs werden wir später ausführlich behandeln. Im Augenblick gehen wir von der (zu restriktiven) Annahme aus, daß ein Bezeichner im gesamten Pro-

gramm nur einmal vereinbart werden darf. Somit sind die Definitionen und Deklarationen

```
const
MIN = 0;

var
x,
y : integer;
min : real;
```

falsch, da der Bezeichner `min` bzw. `MIN` zweimal verwendet wird. (Beachten Sie, daß Standard-Pascal nicht zwischen Groß- und Kleinbuchstaben unterscheidet.)

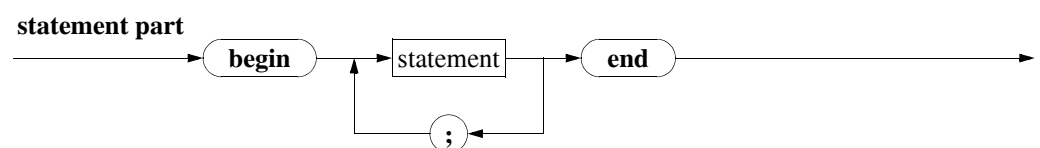
### 3.6 Beschreibung der Aktionen

Im letzten Abschnitt haben wir die Definition von Konstanten und Datentypen sowie die Deklaration von Variablen vorgestellt. Für diesen Teil eines Programms wurde der Begriff Deklarationsteil eingeführt. Die Manipulationen, die das Programm auf diesen Daten durchführt, werden im *Anweisungsteil* (statement part) zusammengefaßt.

Anweisungsteil

Der Anweisungsteil besteht aus einer Folge von Anweisungen (statements), wobei jede Anweisung einer zugehörigen Aktion entspricht. Pascal wird auch als sequentielle Programmiersprache bezeichnet, da die Anweisungen eine nach der anderen und niemals gleichzeitig ausgeführt werden.

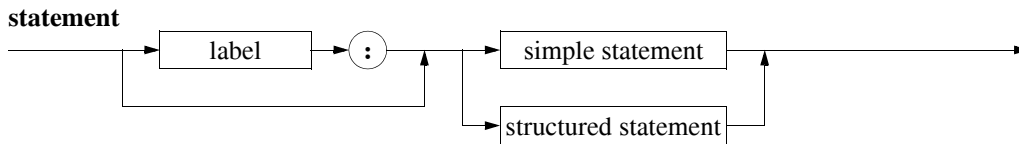
Diese sequentielle Ausführung der Anweisungen spiegelt auch die Struktur des Anweisungsteils wieder. Seine Syntax ist:



Wir haben bereits Programme vorgestellt, die einen einfachen Anweisungsteil enthielten. Wir wollen den Anweisungsteil des Programms *Kreisumfang* hier noch einmal als Beispiel anführen:

```
begin
  writeln ('Der Umfang eines Kreises wird ',
    'berechnet. ');
  write ('Geben Sie den Radius ein: ');
  readln (Radius);
  Umfang := 2.0 * PI * Radius;
  writeln ('Der Umfang betraegt: ', Umfang)
end
```

Pascal stellt sowohl einfache ("simple statements") als auch strukturierte Anweisungen ("structured statements") bereit, mit deren Hilfe Aktionen festgelegt werden können:

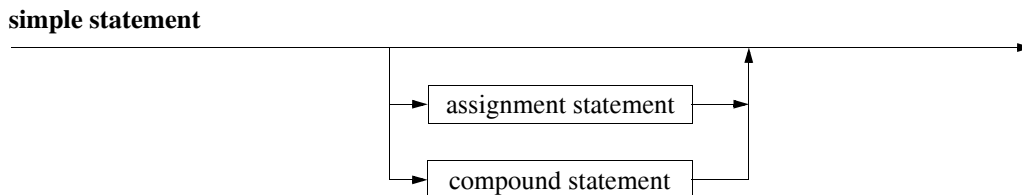


Anweisungsmarken ("label") werden wir nicht benötigen. Ihr Auftreten läßt gewöhnlich auf eine schlechte Programmstruktur schließen.

In diesem Kapitel wollen wir uns auf einfache Anweisungen konzentrieren. Bis auf eine Ausnahme (**if-then-else**) behandeln wir die strukturierten Anweisungen im nächsten Kapitel.

### 3.6.1 Einfache Anweisungen

Zu den *einfachen Anweisungen* gehören die leere Anweisung (empty statement), die Wertzuweisung (assignment statement) und die zusammengesetzte Anweisung (compound statement):



Die einfachste Anweisung von Pascal ist die *leere Anweisung*. Im Syntaxdiagramm ist sie dargestellt als durchgezogene Linie. Sie weist den Computer an, nichts zu tun. Diese scheinbar überflüssige Anweisung dient überwiegend dazu, ein Programm besser zu strukturieren und übersichtlicher zu machen.

### Zuweisung und Ausdruck

Die *Zuweisung*, auch *Wertzuweisung* genannt, ("assignment statement") ist die wichtigste einfache Anweisung und kommt so oder ähnlich in allen imperativen Programmiersprachen vor. Sie ist ein typisches Kennzeichen der imperativen Programmierung. Sie hat die allgemeine Form

$$x := \text{expr}$$

$x$  ist der Bezeichner einer Variablen,  $\text{expr}$  ist ein *Ausdruck* (*expression*). Wir sagen: " $x$  ergibt sich aus  $\text{expr}$ " oder "Der Ausdruck  $\text{expr}$  wird der Variablen  $x$  zugewiesen".

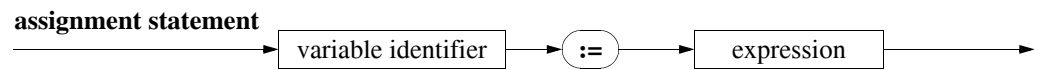
einfache  
Anweisungen

leere Anweisung

Zuweisung,  
Wertzuweisung

Ausdruck  
(expression)

Als Syntaxdiagramm:



Durch die Zuweisung  $x := \text{expr}$  wird der Wert der Variablen mit dem Bezeichner  $x$  durch den Wert des Ausdrucks  $\text{expr}$  ersetzt.  $x$  muß im Deklarationsteil als Bezeichner einer Variablen angegeben sein. Wir bezeichnen  $:=$  als Zuweisungsoperator. Sowohl die Variable als auch der Ausdruck müssen vom selben Datentyp sein. Abweichungen von dieser Vorschrift sind nur erlaubt, wenn die Variable vom Typ *real* und der Ausdruck vom Typ *integer* ist.

Der Ausdruck  $\text{expr}$  ist eine Zeichenfolge, die aus Konstanten, Variablen, Funktionsaufrufen, Operationssymbolen und Klammern aufgebaut ist und bestimmten syntaktischen Regeln genügt. Als Funktionsaufrufe sind uns bisher nur die Aufrufe von Standardfunktionen bekannt, wie z.B.  $\text{sqrt}(x)$  für eine *real*-Variable  $x$ . Der Wert eines Ausdrucks wird durch den Prozeß der *Auswertung* bestimmt. Die Auswertung eines Ausdrucks definiert eine Verarbeitungsvorschrift, die festlegt, welche Operationen in welcher Reihenfolge auf welche Operanden anzuwenden sind. Sie erfolgt abhängig von der Priorität der Operatoren und bei gleicher Priorität stets von links nach rechts.

So liefert die Auswertung der Zeichenfolge  $3+5$  den *integer*-Wert 8, die von  $\text{sqrt}(x)$  die positive Quadratwurzel des Wertes von  $x$ . Wird  $a-b \geq 2$  ausgewertet, so ist das Ergebnis der Wahrheitswert *true* oder *false*. Es wird dabei zunächst die Differenz der Werte von  $a$  und  $b$  bestimmt. Ist das Ergebnis mindestens 2, so ist der Wert des Ausdrucks *true*, anderenfalls *false*.

Der Datentyp des ausgewerteten Ausdrucks kann ebenfalls am Aufbau des Ausdrucks abgelesen werden. Wir betrachten hier zwei Klassen von Ausdrücken: *arithmetische Ausdrücke*, deren Werte vom Typ *integer* oder *real* sind, und *boolesche* bzw. *logische Ausdrücke*, deren Werte vom Typ *boolean* sind. Arithmetische Ausdrücke können neben *integer*- oder *real*-wertigen Funktionen arithmetische Operatoren ( $*$ ,  $/$ , **div**, **mod**,  $+$ ,  $-$ ) enthalten. Boolesche Ausdrücke enthalten arithmetische Operatoren nur in arithmetischen Ausdrücken, die mittels Vergleichsoperatoren zu Vergleichsausdrücken verknüpft sind. Sie können aber auch boolesche Operatoren (**and**, **or**, **not**), allgemeinere Vergleichsausdrücke (z.B. Vergleich von zwei *char*-Variablen) und *boolean*-wertige Funktionen enthalten.

Die folgenden Zeichenfolgen sind Ausdrücke:

$((3+5)*4)$	vom Typ <i>integer</i> ,
$((-3.1)-2.4)*(1.3E2 + 3.4)$	vom Typ <i>real</i> ,
$((a \text{ and } b) \text{ or } (b \text{ and } (\text{not } c)))$	vom Typ <i>boolean</i> ,
$((a=(-3)) \text{ or } (b>2))$	vom Typ <i>boolean</i> .

Auswertung

arithmetischer  
Ausdruck,  
boolescher bzw.  
logischer Ausdruck

Pascal kennt eine automatische Typanpassung von `integer` in `real`. Steht ein `integer`-Wert dort, wo ein `real`-Wert erwartet wird, beispielsweise in `sqrt(6)` oder `4.6 * 6`, so wird an dieser Stelle zunächst die entsprechende `real`-Zahl eingesetzt und anschließend der Wert des Ausdrucks berechnet. Automatische (wir sagen auch: *implizite*) *Typanpassung* ist eine typische Fehlerquelle. Meist hat der Programmierer versehentlich Werte verschiedenen Typs benutzt. Da solche Fehler vom Compiler bei der Syntaxprüfung nicht erkannt werden, entstehen Laufzeitfehler, die mühsam gesucht werden müssen. Implizite Typanpassung zeugt von schlechtem Programmierstil und wir werden ihren Gebrauch auf ein Minimum einschränken (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)"). In der umgekehrten Richtung findet aber keine Typanpassung statt, der Ausdruck

```
20.0 div 5
```

führt in Standard-Pascal unweigerlich zu einem Typfehler.

Die Operatoren sind in Pascal in vier Prioritätsklassen eingeteilt:

	<u>Klasse</u>	<u>Operator</u>
höchste Priorität	1	<b>not</b>
	2	<b>*</b> <b>/</b> <b>div</b> <b>mod</b> <b>and</b> (sog. <i>Punktoperatoren</i> )
	3	<b>+</b> <b>-</b> <b>or</b> (sog. <i>Strichoperatoren</i> )
niedrigste Priorität	4	<b>=</b> <b>&lt;&gt;</b> <b>&lt;</b> <b>&lt;=</b> <b>&gt;</b> <b>&gt;=</b> (Vergleichsoperatoren)

Bei Ausdrücken ohne Klammern wird gemäß der Prioritäten ausgewertet, d.h., zunächst werden alle Teilausdrücke mit **not**-Operatoren ausgewertet, dann die Teilausdrücke mit Punktoperatoren, dann alle mit Strichoperatoren und zum Schluß alle mit Vergleichsoperatoren. Folgen von Teilausdrücken der gleichen Prioritätsklasse werden - wie bereits bemerkt - von links nach rechts abgearbeitet.

#### Beispiel 3.6.1.1

```
6*2+4*3
```

Da der Operator **\*** die höchste Priorität hat, ist der Ausdruck gleichbedeutend mit  $(6*2) + (4*3)$  und ergibt 24.

```
10 mod 3 * 4
```

Die beiden Operatoren **mod** und **\*** haben dieselbe Priorität. Deshalb wird der Ausdruck von links nach rechts ausgewertet und ist damit gleich  $(10 \text{ mod } 3) * 4$ , also 4.

```
true and false or (true and false)
```

Der Klammerinhalt wird zuerst berechnet, er ergibt `false`. Danach wird der Ausdruck `true and false` ausgewertet (**and** hat höhere Priorität als **or**) und ergibt ebenfalls `false`. Zuletzt wird `false or false` ausgewertet und liefert das Endergebnis `false`. In diesem Beispiel sind die Klammern überflüssig. In einigen Fällen führt das Weglassen von Klammern zu falschen Ergebnissen oder gar zu unzulässigen Formen von Ausdrücken, z.B.

implizite  
Typanpassung

Prioritäten von  
Operationen

`(3>0) and (3<10)`

ergibt das Resultat `true`. Werden jedoch die Klammern weggelassen, wird also

`3>0 and 3<10`

geschrieben, so erhalten wir einen unzulässigen Ausdruck. Die Prioritäten erfordern nämlich, daß zuerst der Teilausdruck für den Operator **and** ausgewertet wird; der sich ergebende Ausdruck `0 and 3` ist aber unzulässig, da der Operator **and** boolesche Operanden erfordert. □

Für den Aufbau boolescher Ausdrücke gibt es stets mehrere gleichbedeutende Möglichkeiten. Setzen wir voraus, daß `a` und `b` boolesche Variablen sind, so läßt sich z.B. mit **and** und **not** der Operator **or** ausdrücken:

`a or b` liefert genau dann den Wert `true`, wenn der boolesche Ausdruck `(not a) and (not b)` den Wert `false` liefert.

Analog läßt sich mit **or** und **not** der Operator **and** ausdrücken:

`a and b` liefert genau dann den Wert `true`, wenn der boolesche Ausdruck `(not a) or (not b)` den Wert `false` liefert.

#### Beispiel 3.6.1.2

Sind `a` und `b` boolesche Variablen und `c` eine `integer`-Variable, so gilt:

Der Ausdruck `not a or b and not (c >= 10)` ist gleichbedeutend mit `not a or b and (c < 10)`. □

Einige wichtige Punkte im Zusammenhang mit Ausdrücken sollten wir uns stets vor Augen halten:

- Jeder Variablen in einem Ausdruck muß vor dessen Auswertung auf irgendeine Weise ein Wert zugewiesen worden sein, sonst ist die Auswertung nicht definiert.
- Die Verwendung möglicherweise überflüssiger Klammern schadet nie, deshalb sollten im Zweifelsfall Klammern gesetzt werden. Gelegentlich dienen Klammern auch als optische Hilfe zur besseren Lesbarkeit komplexer Ausdrücke.
- Boolesche Ausdrücke werden in Standard-Pascal stets vollständig ausgewertet, auch wenn der Wert des gesamten Ausdrucks bereits nach einer Teilauswertung feststeht. So wird beispielsweise im Ausdruck

`true or (N > 10)`

auch die Bedingung `N > 10` ausgewertet, obwohl der Gesamtausdruck unabhängig von dem Ausdruck rechts vom **or** stets `true` ist. Es ist deshalb notwendig, darauf zu achten, daß alle Bedingungen in jedem Fall einen definierten Wert annehmen. (Turbo-Pascal beendet dagegen die Auswertung, sobald der Wert eines Ausdrucks sich nicht mehr ändern kann.)

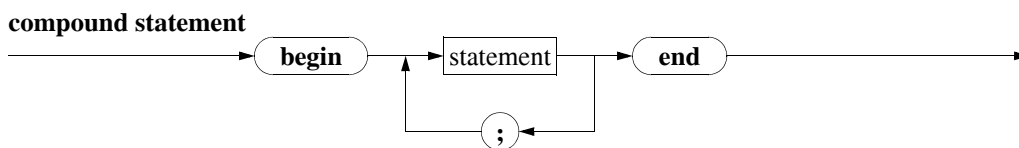
- Wir empfehlen größte Sorgfalt bei der Umformung boolescher Ausdrücke. Derartige Umformungen sind stets eine beliebte Fehlerquelle.

### Zusammengesetzte Anweisungen

Als *Sequenz* bezeichnen wir eine Folge von Anweisungen, die jeweils durch ein Semikolon getrennt sind. Man kann eine Folge von Anweisungen auch zu einer einzigen Anweisung zusammenfassen, *zusammengesetzte Anweisung* (*compound statement*) genannt. Sind  $A_1, A_2, \dots, A_n$  Pascal-Anweisungen, so ist

**begin**  $A_1; A_2; \dots; A_n$  **end**

eine solche zusammengesetzte Anweisung. Die Schlüsselwörter **begin** und **end** sind eine Art öffnende bzw. schließende Klammer für Anweisungen. Das Semikolon dient als Trennzeichen zwischen zwei Anweisungen. Wir können in Pascal mehrere Anweisungen in eine Zeile schreiben. Allerdings werden wir davon keinen Gebrauch machen, da die Übersichtlichkeit und Lesbarkeit des Programms darunter leidet (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").



Der Anweisungsteil eines Programms hat also die syntaktische Struktur einer zusammengesetzten Anweisung. Die Anweisungen einer zusammengesetzten Anweisung können (vorerst) leere Anweisungen, Zuweisungen oder wiederum zusammengesetzte Anweisungen sein. Also ist auch das folgende Beispiel eine zusammengesetzte Anweisung (die nur Demonstrationszwecken dient):

```

begin
  x := 1;
  x := x + 1;
  begin
    y := false;
    y := x > 0;
  end;
  begin
    end;
    x := x * x - 1;
    ;
    ;
    y := x > 4
  end

```

Vor dem ersten **end** in der siebten Zeile steht eine leere Anweisung, da ein **end** lt. Syntaxdiagramm nicht unmittelbar auf ein Semikolon folgt. Die Schlüsselwörter **begin** und **end** in der achten und neunten Zeile bilden eine zusammengesetzte Anweisung, die nur die leere Anweisung enthält. Oberhalb der vorletzten Zeile befinden sich zwei leere Anweisungen.

Sequenz

zusammengesetzte  
Anweisung

### 3.6.2 Standard-Ein- und Ausgabe

Wir haben in unseren Beispielen schon Anweisungen zur Ein- und Ausgabe benutzt und dabei ein intuitives Verständnis des Lesers vorausgesetzt. Da es sich bei diesem Kurs nicht um einen Programmierkurs handelt, gehen wir auf die technischen (und oft systemabhängigen) Einzelheiten der Ein- und Ausgabe nicht näher ein. Details entnehmen Sie bitte entsprechenden Programmierhandbüchern.

Das Einlesen und Ausgeben von Daten geschieht in Pascal durch die speziellen Anweisungen (die Prozeduren ähnlich sind)

`read`      und    `readln` für die Eingabe

sowie

`write`      und    `writeln` für die Ausgabe.

Zur Kontrolle des Eingabevorgangs dienen außerdem die Funktionen `eoln` zum Erkennen eines Zeilenendes und `eof` zum Erkennen des Eingabeendes.

Zu den Ein- und Ausgabeanweisungen im einzelnen:

#### 1. Die `read`-Anweisung

`read`

Mit `read` können Werte vom Datentyp `char`, `integer` oder `real` eingelesen werden. Eine `read`-Anweisung hat in unserem Kurs die allgemeine Form:

`read (x1, x2, ..., xn),`

Parameter

wobei  $x_1, x_2, \dots, x_n$  als *Parameter* bezeichnet werden.

$x_1, x_2, \dots, x_n$  sind Variablen vom Typ `char`, `integer` oder `real`. In dieser Form ist stets die Standardeingabe (i.a. über die Tastatur) gemeint. `read` liest  $n$  Eingabewerte ein und weist sie nacheinander den Variablen  $x_1, x_2, \dots, x_n$  zu. Falls ein Wert eingegeben wird, der nicht dem Datentyp der zugehörigen Variablen entspricht, erfolgt ein Fehlerabbruch.

#### Beispiel 3.6.2.1

Gegeben seien folgende Deklarationen:

```
var
i,
j : integer;
x,
y : real;
c : char;
```

Werden nacheinander die Werte



```
A    1    102.50    2    -7.620
```

eingegeben, dann haben nach Ausführung der Anweisung

```
read (c, i, x, j, y)
```

die Variablen folgende Werte:

```
i = 1, j = 2, x = 102.5,
y = -7.62, c = 'A'.
```



## 2. Die `readln`-Anweisung (Abk. von `read line`)

Diese Anweisung unterscheidet sich von der `read`-Anweisung nur insoweit, als nach dem Lesen zusätzlich der Bildschirmcursor auf die erste Position der nächsten Zeile gesetzt wird. Wird jetzt wieder gelesen oder geschrieben, dann geschieht dies ab dieser Position. `readln` ohne Parameter bewirkt, daß alle restlichen Eingabewerte der aktuellen Zeile ignoriert werden und lediglich der Cursor auf den Anfang der nächsten Zeile positioniert wird.

## 3. Die Standardfunktion `eof` (Abk. von `end of file`)

Mit Hilfe der Standardfunktion `eof` kann das Ende einer Eingabe überprüft werden. Wir kommen auf die Verwendung von `eof` im nächsten Kapitel zurück.

## 4. Die `write`-Anweisung

Mit `write` können die Werte von Variablen und Konstanten ausgegeben werden. Betrachten wir zunächst die `write`-Anweisung

```
write (A).
```

Dabei kann `A` eine der folgenden drei Formen haben:

```
x
x:b
x:b:k.
```

`x` ist in unserem Kurs entweder ein Ausdruck vom Typ `char`, `integer`, `real`, `boolean` oder eine `string`-Konstante. Beachten Sie, daß einzelne Konstanten und Variablen auch Ausdrücke sind.

`b` ist ein `integer`-Ausdruck mit positivem Wert und wird als *minimale Feldbreite* bezeichnet. `b` gibt die Anzahl der Zeichen an, die der Wert von `x` bei der Ausgabe mindestens belegen soll. Nimmt der Wert von `x` weniger als `b` Stellen ein, so wird vor `x` eine entsprechende Zahl von Leerzeichen ausgegeben. Wird `b` weggelassen, so wird ein Standardwert angenommen. Dieser ist für die Datentypen `integer`, `real` oder `boolean` abhängig von der Implementierung, für den Typ `char` ist er 1, und für `string`-Konstanten ist er gleich der Anzahl der Komponenten des Strings. Die Angabe von `b` eignet sich besonders zum Erstellen von Tabellen.

`k` darf nur verwendet werden, wenn `x` vom Typ `real` ist, und bestimmt die Zahl der Nachkommastellen. `k` ist ein positiver `integer`-Ausdruck. Ist `k`

`readln`

`eof`

`write`

Feldbreite

kleiner als die Zahl der Nachkommastellen von  $x$ , so wird  $x$  automatisch auf die gewünschte Stellenzahl gerundet. Wird  $k$  bei `real`-Größen  $x$  nicht spezifiziert, so wird  $x$  automatisch mit der maximalen Stellenzahl und in Gleitpunktdarstellung ausgegeben.

### Beispiel 3.6.2.2

Gegeben seien folgende Vereinbarungen

**const**

`N = 15;`

`L = '-----';`

**var**

`i : integer;`

`x,`

`y : real;`

`c : char;`

Die zusammengesetzte Anweisung

**begin**

`i := 7;`

`x := -2.7;`

`y := 8.1;`

`c := '$';`

`write (1);`

`write ('.');`

`write ('i*n=');`

`write (i:2);`

`write (c:3);`

`write (N:3);`

`write ('=':3);`

`write (i*N:3);`

`write ('2.':3);`

`write ('x*y=');`

`write (x:6:2);`

`write (c:3);`

`write (y:8:2);`

`write ('=':3);`

`writeln (x*y:7:1);`

`write (L, L)`

**end**

liefert folgende Ausgabe:

```
1.i*n= 7  $ 15  =105 2.x*y= -2.70  $      8.10  =  -21.9
-----
```



Mehrere aufeinanderfolgende `write`-Anweisungen können wir wie bei `read` zu einer einzigen zusammenfassen. Eine zusammengesetzte Anweisung der Form

```
begin
  write (a1);
  write (a2);
  ...;
  write (an)
end
```

kann abgekürzt werden durch

```
write (a1, a2, ..., an)
```

Ausgegeben wird über die Standardausgabe Drucker oder Bildschirm.

#### 5. Die `writeln`-Anweisung (Abk. von `write line`)

`writeln` hat bezüglich der Ausgabe eine ähnliche Wirkung wie `readln` bezüglich der Eingabe. `writeln (a1, a2, ..., an)` positioniert den Cursor nach der Ausgabe der Werte von `a1`, `a2`, ..., `an` an den Anfang der neuen Zeile. Die nächste `write`-Anweisung schreibt also in die nächste Ausgabezeile. `writeln` kann auch ohne Parameter verwendet werden und bewirkt dann nur den Übergang zu einer neuen Zeile.

`writeln`

#### Beispiel 3.6.2.3

Für ein letztes Beispiel nehmen wir an, daß eine `real`-Zahl, die einen Geldbetrag (unter einer Million) in EUR angibt, eingelesen und mit 2 Stellen hinter dem Dezimalpunkt, gefolgt von der Angabe EUR, ausgegeben werden soll. Das folgende Pascal-Programm löst diese Aufgabe:

```
program EURBetrag (input, output);
{ EUR-Betrag ausgeben }

const
  WAEHRUNG = 'EUR';

var
  Betrag : real;
```

```

begin
  write ('Bitte ', WAEHRUNG, '-Betrag (< 1 Mio.) ');
  write ('eingeben: ');
  readln (Betrag);
  write (Betrag:9:2, WAEHRUNG)
end. { EURBetrag }

```

```

Bitte EUR-Betrag (< 1 Mio.) eingeben: 1753862e-3
1753.86EUR

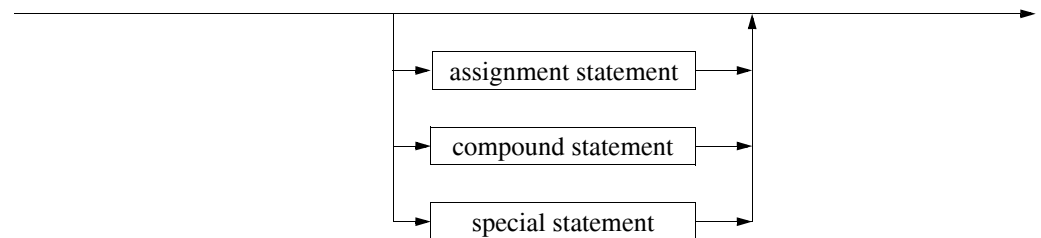
```



Bei den Ein- und Ausgabeanweisungen handelt es sich weder um Zuweisungen noch um zusammengesetzte Anweisungen. Später werden Sie feststellen, daß diese Anweisungen auch nicht ganz der Syntax von Prozeduraufrufen genügen. Ein- und Ausgabeanweisungen heißen deshalb *spezielle Anweisungen* (*special statements*). Damit können wir unser Syntax-Diagramm "simple statement" ergänzen zu:

spezielle  
Anweisungen

#### simple statement

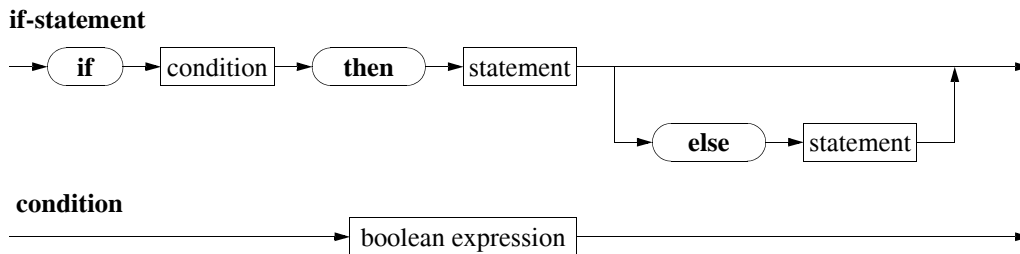


### 3.6.3 Die bedingte Anweisung (if-then-else)

Bei der Ausführung eines Programms mit einer Benutzer-Eingabe kann es vorkommen, daß eine Eingabe zwar vom richtigen Typ ist, aber nicht die vorgesehene Vorbedingung erfüllt (vgl. das Parkplatzproblem in Abschnitt 2.1). Bisher haben wir gegen solche Fälle keine Vorkehrungen getroffen. Im allgemeinen werden wir aber die Eingabe für ein Programm auf die Einhaltung der Vorbedingung prüfen müssen.

Die Möglichkeit, abhängig von der Erfüllung einer Bedingung Aktionen durchzuführen oder zu unterlassen, ist allerdings nicht nur für die Prüfung der Zulässigkeit der Eingabe, sondern für die Steuerung der Ausführung von Anweisungen ganz allgemein von Bedeutung.

Die **if**-Anweisung gestattet in Abhängigkeit von einer Bedingung die Ausführung einer Anweisung oder die Wahl zwischen zwei Anweisungen. Sie hat die Form:



Die bedingte Anweisung beginnt mit dem Schlüsselwort **if** und einer *Bedingung* (*condition*) in Gestalt eines logischen Ausdrucks, gefolgt vom Schlüsselwort **then** und einer Anweisung. In der einfachen Form ist die bedingte Anweisung hier abgeschlossen, in der erweiterten Form schließt sich das Schlüsselwort **else** und eine andere Anweisung an.

Bedingung

In der kürzeren Form wird der Ausdruck ausgewertet und, falls sein Wert `true` ist, wird die Anweisung nach **then** ausgeführt. Liefert die Auswertung `false`, dann wird die Anweisung nach **then** nicht ausgeführt und mit der darauf folgenden Anweisung fortgefahren.

In der längeren Form wird ebenfalls erst der logische Ausdruck ausgewertet. Ist sein Ergebnis `true`, so wird die erste Anweisung (nach **then**), sonst die zweite (nach **else**) ausgeführt.

Nun können wir das Programm `EURBetrag` so modifizieren, daß der eingelesene Betrag nur dann in der gewünschten Form ausgegeben wird, wenn er nicht negativ ist:

```

program EURBetrag2 (input, output);
{ EUR Betrag ausgeben mit Prüfung }

  const
    WAEHRUNG = 'EUR';

  var
    Betrag : real;

begin
  write ('Bitte ', WAEHRUNG, '-Betrag (< 1 Mio.) ');
  write ('angeben: ');
  readln (Betrag);
  if Betrag >= 0.0 then

```

```

        write (Betrag:9:2, WAEHRUNG)
    end. { EURBetrag2 }

```

*Bitte EUR-Betrag (< 1 Mio.) angeben: -1753862e-3*

Wegen der negativen Eingabe wird hier nichts ausgegeben. Im allgemeinen sollte allerdings auch eine Fehlermeldung ausgegeben werden, wenn die Eingabe nicht der Bedingung genügt:

```

program EURBetrag3 (input, output);
{ EUR Betrag ausgeben mit Prüfung und Fehlermeldung }

    const
    WAEHRUNG = 'EUR';

    var
    Betrag : real;

    begin
        write ('Bitte ', WAEHRUNG, '-Betrag (< 1 Mio.) ');
        write ('angeben: ');
        readln (Betrag);
        if Betrag >= 0.0 then
            write (Betrag:9:2, WAEHRUNG)
        else
            begin
                write ('Eingabefehler! Betrag ', Betrag);
                write (' ist negativ.')
            end
        end. { EURBetrag3 }

```

*Bitte EUR-Betrag (< 1 Mio.) angeben: -1753862e-3  
Eingabefehler! Betrag -1753862e-3 ist negativ.*



Da die **if**-Anweisung in fast allen Programmen vorkommt, ist hier auch die Fehlerhäufigkeit unverhältnismäßig groß. Drei der häufigsten Fehler sind die folgenden:

### falsche Leeraanweisung

Im Programmausschnitt

```
if Betrag >= 0.0 then; { Fehler: Leeraanweisung }
    write (Betrag:9:2, WAEHRUNG)
```

findet der Compiler im **then**-Zweig aufgrund des falsch gesetzten Semikolons nur die leere Anweisung. Dieser Fehler bewirkt also, daß, anders als durch die Einrückung suggeriert, die Ausgabeanweisung nicht mehr zur **if**-Anweisung gehört.

### fehlende Zusammenfassung

Ein weiterer häufig vorkommender Fehler ist das Vergessen von **begin** und **end** bei einer Folge von Anweisungen, innerhalb eines **then**-Zweiges:


```
if Betrag >= 0.0 then
    write (Betrag:9:2);      { Fehler: kein begin/end }
    write (WAEHRUNG)
```

Hier gehört nur die erste `write`-Anweisung zur **if**-Anweisung. Die zweite `write`-Anweisung wird unabhängig davon ausgeführt, obwohl dies durch die Einrückung wieder anders suggeriert wird.

### Semikolon vor else

Ein Fehler, der bereits vom Compiler erkannt wird, liegt vor, wenn vor einem **else**-Zweig ein Semikolon steht:

```
if Betrag >= 0.0 then
    write (Betrag:9:2, WAEHRUNG);
else { Fehler: Semikolon vor else }
    write ('Eingabefehler!')
```

Durch das Semikolon wird die **if**-Anweisung beendet. Die darauf folgende Anweisung würde mit **else** beginnen, was aber laut Syntaxdiagramm nicht sein darf. 

**if**-Anweisungen können auch geschachtelt auftreten (vgl. Syntaxdiagramm: ein inneres „statement“ kann wieder durch eine **if**-Anweisung ersetzt werden). Dabei ist auf die richtige Zuordnung von **else**-Zweigen zu den **if**-Anweisungen zu achten. Ein **else**-Zweig wird dem letzten **then**-Zweig zugeordnet, der keinen **else**-Zweig besitzt. (Geprüft wird rückwärts bis zum nächstliegenden **begin**, wobei vor dem **else**-Zweig abgeschlossene zusammengesetzte Anweisungen vollständig überlesen werden.) Das Problem der richtigen Zuordnung wird als „dangling else“-Problem („baumelndes else“-Problem) bezeichnet.

#### Beispiel 3.6.3.1

Ein Unternehmen verkaufe Waren zu einem festen Grundpreis von 200,00 EUR. Bestellt ein Kunde mehr als 10 Waren, so erhält er einen Rabatt. Dieser beträgt mindestens 10% des Gesamtpreises, bei einer Bestellung von mehr als 50 Teilen sogar 20%. Gesucht ist ein Programm, das zu einer Bestellzahl den Preis berechnet und ausgibt.

kein Semikolon  
vor **else**

"dangling else"

Ohne lange Überlegung können wir eine einfache Lösung angeben:

```

program Verkauf1 (input, output);
{ Rabattgewaehrung }

    const
    GRUNDPREIS = 200.00 {EUR};

    var
    Anzahl : integer;
    Preis : real;

begin
    writeln ('Bitte die Anzahl bestellter Waren ',
            'eingeben:');
    readln (Anzahl);
    if Anzahl > 0 then
    { es liegt eine Bestellung vor }
    begin
        Preis := Anzahl * GRUNDPREIS;
        if Anzahl > 10 then
        { ein Rabatt wird erteilt }
        if Anzahl > 50 then
        { grosser Rabatt kann gewaehrt werden }
        Preis := Preis * 0.8
        else
        { kleiner Rabatt kann gewaehrt werden }
        Preis := Preis * 0.9;
        writeln ('Der Preis betraegt ', Preis, ' EUR')
    end
    else
        writeln ('falsche Eingabe')
    end. { Verkauf1 }

```

Offensichtlich gehört der innere **else**-Zweig zur dritten, der äußere **else**-Zweig zur ersten **if**-Anweisung. Beachten Sie die verwendeten Einrückregeln und insbesondere, daß wir ein zusammengehöriges **if** und **else** sowie **begin** und **end** auf dieselbe Position eingerückt haben, um die Lesbarkeit und Übersichtlichkeit zu erhöhen. Ein solches *Layout* des Programmtextes verringert die Fehlerhäufigkeit (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").



Um die Übersichtlichkeit weiter zu erhöhen und die Zuordnung eines **else**-Zweiges zu seinem **then**-Zweig zu verdeutlichen, empfehlen wir, den Abstand zwischen beiden Zweigen so klein wie möglich zu halten. Dazu soll folgende *Faustregel* beachtet werden:

Bei geschachtelten **if**-Anweisungen wird nach Möglichkeit nur der **else**-Zweig geschachtelt.

Layout

Empfehlung:  
nur **else**-Zweig  
schachteln



Nur in Fällen, in denen diese Regel zu starken Abweichungen vom intuitiv verständlichen Algorithmus führt, werden wir gelegentlich darauf verzichten (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").

#### Beispiel 3.6.3.2

Wir greifen unser letztes Beispiel noch einmal auf und formen das Programm Verkauf1 gemäß unserer neuen Regel um.

```
program Verkauf2 (input, output);
{ Rabattgewaehrung: zweite Version }

const
  GRUNDPREIS = 200.00 {EUR};

var
  Anzahl : integer;
  Preis  : real;

begin
  writeln ('Bitte die Anzahl bestellter Teile ',
           'eingeben:');
  readln (Anzahl);
  if Anzahl <= 0 then
    writeln ('falsche Eingabe')
  else
    { es liegt eine Bestellung vor }
    begin
      Preis := Anzahl * GRUNDPREIS;
      if Anzahl > 50 then
        { grosser Rabatt kann gewaehrt werden }
        Preis := Preis * 0.8
      else
        if Anzahl > 10 then
          { kleiner Rabatt kann gewaehrt werden }
          Preis := Preis * 0.9;
      writeln ('Der Preis betraegt ', Preis, ' EUR')
    end
  end. { Verkauf2 }
```



Mit Hilfe der eingeführten Kontrollstruktur können wir schon etwas anspruchsvollere Programme schreiben. Wir wollen zum Abschluß dieses Abschnitts eine Methode des Programmentwurfs, also ein Vorgehen bei der Programmentwicklung, für kleine Programme aufzeigen.

Beispiel 3.6.3.3

Es soll ein Programm geschrieben werden, das drei positive integer-Werte (in aufsteigender Ordnung) einliest, welche die Längen der Seiten eines Dreiecks darstellen. Das Programm soll die Eingabewerte, eine der folgenden Informationen

```
kein Dreieck
ein gleichseitiges Dreieck
ein gleichschenkliges Dreieck
ein ungleichseitiges Dreieck
```

sowie - falls es sich um ein Dreieck handelt - die Fläche ausdrucken.

Die Grobstruktur kleiner Programme besteht oft aus einem Eingabe-, einem Verarbeitungs- und einem Ausgabeteil. Wir sprechen auch vom EVA-Prinzip (**E**ingabe, **V**erarbeitung, **A**usgabe). In unserem (sehr) einfachen Fall teilen wir das Programm in zwei Teile, die wir als Kommentare in einem Anweisungsteil festhalten:

```
begin
  {lese und drucke die Seitenlaengen}
  {bestimme und drucke die Charakteristika des
   Dreiecks}
end.
```

Der erste Teil kann unmittelbar in read- und writeln-Anweisungen mit den integer-Variablen a, b und c für die Seiten übertragen werden.

Der zweite Teil kann weiter unterteilt werden in

```
if { es kann kein Dreieck gebildet werden } then
  { schreibe 'kein Dreieck' }
else
  { bestimme seine Natur und Flaeche }.
```

Drei Strecken können nur dann ein Dreieck bilden, wenn die Summe der beiden kürzeren Strecken länger ist als die längere Strecke. Da a, b und c die Strecken in aufsteigender Ordnung darstellen, kann die Bedingung "es ist kein Dreieck" auch als  $a + b \leq c$  ausgedrückt werden. Die Bestimmung der Charakteristik des Dreiecks kann ebenfalls leicht auf Grund dieser geordneten Seitenlängen ausgedrückt werden, nämlich

```
if a = c then
  { Dreieck gleichseitig }
else
  if (a = b) or (b = c) then
    { Dreieck ist gleichschenkelig }
  else
    { Dreieck ist ungleichseitig }.
```

Die Fläche eines allgemeinen Dreiecks kann durch die Formel

$$Flaeche = \sqrt{s \cdot (s - a) \cdot (s - b) \cdot (s - c)}$$

berechnet werden, wobei  $s$  die Hälfte der Summe der Dreiecksseiten ist. In dieser Formel werden  $s$  und  $Flaeche$  als `real-Variable` deklariert, da ganzzahlige Werte eine zu grobe Näherung wären. Somit können wir jetzt das folgende Programm zusammenstellen:

```

program Dreiecke (input, output);
{ Dreiecksbestimmung }

var
  a,
  b,
  c : integer;
  s,
  Flaeche : real;

begin
  { lese und drucke Dreiecksseiten
    in aufsteigender Folge }
  writeln ('Drei Seitenlaengen in auf',
           'steigender Folge eingeben!');
  readln (a);
  readln (b);
  readln (c);
  writeln (a:3, b:3, c:3);
  if (b < a) or (c < b) then
    writeln (' ist nicht aufsteigend sortiert')
  else
    if a + b <= c then
      writeln (' ist kein Dreieck')
    else
      begin
        { bestimme die Art des Dreiecks: }
        if a = c then
          write ('ist ein gleichseitiges Dreieck')
        else
          if (a = b) or (b = c) then
            write ('ist ein gleichschenkliges Dreieck')
          else
            write ('ist ein ungleichseitiges Dreieck');
        { bestimme die Flaeche: }
        s := 0.5 * (a + b + c);
        Flaeche := sqrt (s * (s - a) * (s - b) * (s - c));
        writeln (' der Flaeche', Flaeche:8:2)
      end
  end

```

```

      end { if a + b <= c }
    end. { Dreiecke }

```

```

Drei Seitenlaengen in aufsteigender Folge eingeben!
7
7
9
    7  7  9
ist ein gleichschenkliges Dreieck der Flaeche    24.13

```



#### Bemerkung 3.6.3.4

Wir haben bereits in Abschnitt "Zuweisung und Ausdruck" darauf hingewiesen, daß in Standard-Pascal boolesche Ausdrücke stets vollständig abgearbeitet werden und nicht bereits abgebrochen wird, sobald der Wert feststeht. Es muß daher darauf geachtet werden, daß keine undefinierten Teilausdrücke in der Bedingung einer **if**-Anweisung auftreten. Beispielsweise führt die Anweisung

```

if (b <> 0) and (a div b > 100) then
    a := a - s;

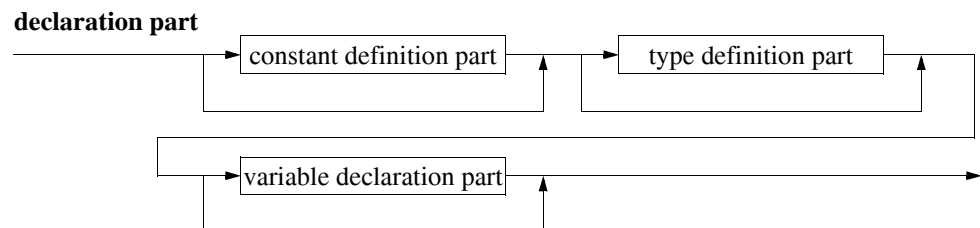
```

für  $b=0$  in Standard-Pascal zu einem Programmabbruch (Division durch Null). In Turbo-Pascal tritt dagegen kein Fehler auf.



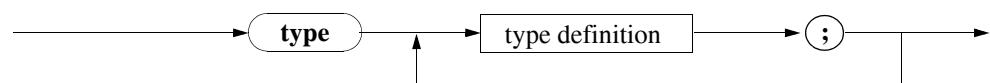
### 3.7 Einfache selbstdefinierbare Datentypen

Neben den Standarddatentypen `integer`, `real`, `char`, `boolean` können wir weitere Datentypen in einem Programm verwenden, die wir selbst definieren. Dazu müssen wir zunächst die Syntax des Deklarationsteils geeignet erweitern:



Der Typdefinitionsteil beginnt immer mit dem Schlüsselwort **type**. Anschließend werden (neue) Datentypen definiert.

**type definition part**

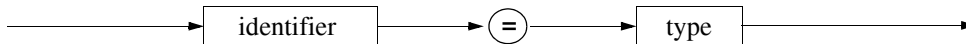


Typdefinition

In einer *Typdefinition* ("type definition") wird dem zu definierenden Typ ("type") ein Bezeichner zugeordnet. Zur Unterscheidung von Konstanten und Variablen las-

sen wir Typbezeichner stets mit dem Buchstaben t beginnen (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").

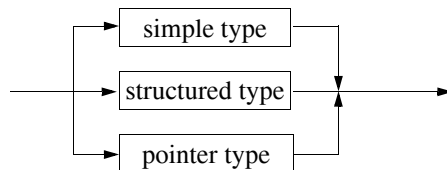
#### type definition



Wir unterscheiden *einfache* bzw. *unstrukturierte Typen* (*simple types*), *strukturierte Typen* (*structured types*) und *Zeigertypen* (*pointer types*). Auf die beiden letzteren kommen wir im vierten bzw. fünften Kapitel zurück.

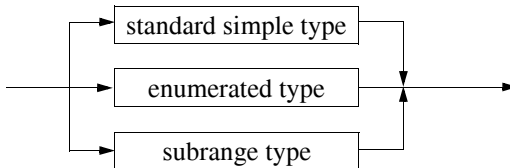
einfacher bzw.  
unstrukturierter Typ,  
strukturierter Typ

#### type



Unter die einfachen Datentypen fallen die vier Standarddatentypen `integer`, `real`, `boolean` und `char`, sowie die beiden in den folgenden Abschnitten besprochenen Datentypen, so daß wir hier noch folgendes Diagramm angeben können:

#### simple type



### 3.7.1 Aufzählungstypen

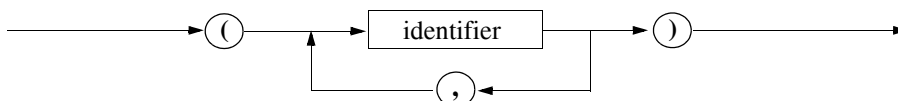
Ein *Aufzählungstyp* (*enumerated type*) wird in der Form

$$\text{Typname} = (\text{Name}_1, \text{Name}_2, \dots, \text{Name}_n)$$

definiert. Dabei genügen *Typname*, *Name*<sub>1</sub>, *Name*<sub>2</sub>, ..., *Name*<sub>n</sub> den Syntaxregeln für Bezeichner.

Aufzählungstyp

#### enumerated type



Definieren wir in einem Programm verschiedene Aufzählungstypen, dann darf kein Bezeichner in mehr als einem Aufzählungstyp auftreten. Dabei können Aufzählungstypen nicht aus Werten anderer Datentypen (z.B. `integer` oder `char`) gebildet werden, da diese nicht der Syntax von Bezeichnern genügen.

#### Beispiel 3.7.1.1

Ein Typdefinitionsteil, der nur aus Aufzählungstypen besteht, könnte etwa folgendermaßen aussehen:

**type**

```
tFahrzeug = (LKW, PKW, Bus, Zug);
tZeitraum = (Tag, Woche, Monat, Jahr);
tFarbe = (rot, gruen, blau, gelb, rosa, braun);
```

Nach unseren Regeln unzulässig ist dagegen die Typdefinition

```
tZiffer = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9);
```



Die Wertemenge eines Aufzählungstyps ist gemäß der in der Definition angegebenen Reihenfolge geordnet. Damit können Vergleichsoperatoren angewandt werden, die wie beim Standarddatentyp `char` über die Standardfunktion `ord` definiert sind. Es gilt z.B. für den Typ `tFarbe`:

```
ord (rot)      ergibt 0,
ord (gruen)    ergibt 1,
...
ord (braun)    ergibt 5.
```

Weiter ist für alle Elemente bis auf das erstgenannte ein Vorgänger definiert (Standardfunktion `pred`), für alle bis auf das letztgenannte ein Nachfolger (Standardfunktion `succ`).

Beispiel 3.7.1.2

Mit Bezug auf Beispiel 3.7.1.1 können wir schreiben:

- zur Anwendung der Standardfunktion `ord`:  
`ord (LKW) ergibt 0, ord (Monat) ergibt 2;`
- zur Anwendung von Vergleichsoperationen:  
`Tag < Jahr` (denn `ord (Tag) < ord (Jahr)`),  
`Monat >= Woche` (analog),  
`rot <> blau` (analog);
- zur Anwendung der Standardfunktionen `pred` und `succ`:  
`pred (gruen) ergibt rot,`  
`succ (Monat) ergibt Jahr,`  
`pred (LKW) ist undefiniert.`

**3.7.2 Ausschnittstypen**

Ein *Ausschnittstyp* (*subrange type*) definiert als Wertebereich stets ein Intervall des Wertebereichs eines bereits definierten *Grundtyps*, (engl. *host type*). Die Definition geschieht in der Form

Typname = kleinstesElement . . groesstesElement

Geeignete Grundtypen sind die Standarddatentypen `integer` und `char` sowie Aufzählungstypen.

Als *kleinstesElement* bzw. *groesstesElement* wird eine Konstante (constant), d.h.

Ausschnittstyp  
Grundtyp

ein bereits zuvor definierter Konstantenbezeichner oder ein Wert des entsprechenden Grundtyps, angegeben.

#### subrange type



Die auf dem Grundtyp zulässigen Operationen und Standardfunktionen werden automatisch für jeden Ausschnittstyp übernommen. Weiterhin sind Zuweisungen zwischen Grundtyp und Ausschnittstypen möglich, d.h. auch hier findet eine automatische Typanpassung statt. In beiden Fällen kann es zu Fehlern durch einen Bereichsüberlauf kommen, wenn dabei der Wertebereich des Ausschnittstyps verlassen wird.

Wir hatten bisher die Regel aufgestellt, daß verschiedene Datentypen disjunkte Wertebereiche aufweisen müssen. Ausschnittstypen bilden die einzige Ausnahme von dieser Regel. Ihre Wertebereiche und die der zugehörigen Grundtypen sind natürlich nicht disjunkt. Nicht-leere Durchschnitte von Wertebereichen verschiedener Datentypen sind also erlaubt, wenn es sich um eine Teilmengenbeziehung handelt.

#### Beispiel 3.7.2.1

Betrachten wir folgenden Typdefinitionsteil mit den zugehörigen Variablendeklarationen:

#### type

```
tMinute = 0..59;
tPlusMinus10 = -10..+10;
tFarbe = (rot, gruen, blau, gelb, rosa, braun);
tFarbausschnitt = rot..blau;
```

#### var

```
Uhr : tMinute;
Intervall : tPlusMinus10;
Testzahl : integer;
FarbtonGross : tFarbe;
FarbtonKlein : tFarbausschnitt;
```

#### Zuweisungen der Form (mit impliziter Typanpassung)

```
Testzahl := Intervall;
Testzahl := Uhr;
und
FarbtonGross := FarbtonKlein;
```

können nicht zu Fehlern führen. Dagegen können Zuweisungen der Form (mit impliziter Typanpassung)

```

Uhr := Intervall;
Uhr := Testzahl;
und
FarbtonKlein := FarbtonGross;

```

zu Laufzeitfehlern führen, wenn im Moment der Ausführung der Wertebereich verletzt ist. Zuweisungen der Form

```

Uhr := pred (0);
FarbtonKlein := succ (blau);
und
Intervall := Intervall + 100;

```

führen in jedem Fall zu einem Laufzeitfehler.



implizite  
Typanpassung

Wir werden soweit wie möglich keine implizite Typanpassung benutzen (vgl. Abschnitt 3.8 "Programmierstil (Teil 1)").

implizite  
Typdefinition

In Standard-Pascal ist es erlaubt, in einer Variablendeklaration nicht nur einen Typbezeichner anzugeben, sondern alternativ einen Typ "direkt" zu definieren. Man spricht im letzteren Fall von einer *impliziten Typdefinition*. Aus Programmierstilgründen erlauben wir keine impliziten Typdefinitionen, da sie zu unübersichtlichen Programmen führen. Der Leser erwartet Typdefinitionen im Typdefinitionsbereich und nicht im Bereich von Variablendeklarationen.

### 3.8 Programmierstil (Teil 1)

Programme können, nachdem sie implementiert und getestet wurden, in den seltensten Fällen ohne Änderung über einen längeren Zeitraum hinweg eingesetzt werden. Tatsächlich ist es meist so, daß die Anforderungen nach der Fertigstellung verändert oder erweitert werden und während des Betriebs bislang unerkannte Mängel oder Fehler auftreten, die zu beseitigen sind. Programme müssen während ihres Einsatzes gewartet werden. Zu Wartungszwecken muß der Programmtext immer wieder gelesen und verstanden werden. Die Lesbarkeit eines Programms spielt also eine große Rolle. Je größer ein Programm ist, desto wichtiger wird das Kriterium der Lesbarkeit. Hinzu kommt, daß in industriellen Anwendungen häufig der bzw. die Entwickler nicht mehr verfügbar ist bzw. sind und Änderungen von Dritten vorgenommen werden müssen.

Programmierstil

Die Lesbarkeit eines Programms hängt einerseits von der verwendeten Programmiersprache und andererseits vom *Programmierstil* ab. Der Programmierstil beeinflusst die Lesbarkeit eines Programms mehr als die verwendete Programmiersprache. Ein stilistisch gut geschriebenes C- oder COBOL-Programm kann besser lesbar sein als ein schlecht geschriebenes Pascal-Programm.

Wir bemühen uns bei unseren Beispielen um einen guten Programmierstil. Zum einen wollen wir Ihnen einen guten Programmierstil "von klein auf" implizit vermitteln, d.h. Sie sollen nur gute und gar nicht erst schlechte Beispiele kennenlernen. Zum anderen bezwecken wir durch die erhöhte Lesbarkeit einen leichteren Zugang



zu den Konzepten, die durch die Programme bzw. Prozeduren und Funktionen vermittelt werden.

Programmierstil ist, wie der Name vermuten läßt, in gewissem Umfang Geschmackssache. Auch können starre Richtlinien in Einzelfällen unnatürlich wirken. Dennoch sind die Erfahrungen mit der flexiblen Handhabung solcher Richtlinien durchweg schlecht, so daß gut geführte Programmierabteilungen und Softwarehäuser einen hauseigenen Programmierstil verbindlich vorschreiben und die Einhaltung - häufig über Softwarewerkzeuge - rigoros kontrollieren.

Genau das werden wir auch in diesem Kurs tun (dies erleichtert auch die Korrektur der über tausend Einsendungen je Kurseinheit) und Regeln vorschreiben, deren Nichteinhaltung zu Punktabzügen führt. Dabei unterscheiden wir zwischen "Muß-Regeln", die in jedem Fall befolgt werden müssen, und "Kann-Regeln", von denen im Einzelfall abgewichen werden kann. Muß-Regeln sind durch Fettdruck und entsprechende Marginalien gekennzeichnet. **Das Nichteinhalten von Muß-Regeln wirkt sich in Punktabzügen aus.**

Die folgenden Regeln sind auf unserem persönlichen Erfahrungshintergrund gewachsen. Für einen anderen Anwendungskontext können sie zum Teil weniger geeignet sein. Daß die Empfehlungen etwas trocken ausgefallen und mit nur wenigen Beispielen garniert sind, hängt damit zusammen, daß wir hier lediglich zusammentragen, was wir ohnehin in unseren Beispielen vorführen und meist auch schon dort angesprochen haben. Eine ausführlichere Diskussion der Programmierstilthematik finden Sie in den Kursen über Software Engineering.

Wir haben uns in den Programmbeispielen nicht immer streng an die Kann-Regeln gehalten. Das hängt damit zusammen, daß die Empfehlungen zum Teil bereits für den professionellen Einsatz gedacht sind und unsere kleinen Beispiele teilweise überfrachtet hätten. So können z.B. gewisse Layout-Regeln (etwa für den Vereinbarungsteil), die bei größeren Programmen sinnvoll sind, kleinere Programme unnötig aufblähen. Außerdem nehmen wir aus Gründen der Lesbarkeit auf den Seitenumbruch Rücksicht, was sich nicht immer mit den Regeln verträgt.

Die Ausführungen zum Programmierstil beginnen mit dem ersten Kapitel zu den Programmierkonzepten und werden von Kapitel zu Kapitel aktualisiert und ergänzt. Dabei werden die Regeln "kumuliert", d.h. die Programmierstilregeln von Kapitel 5 schließen die Regeln für Kapitel 3 und Kapitel 4 mit ein. Am Ende von Kapitel 6 (in Kapitel 6.3) finden Sie sämtliche kursrelevanten Regeln zusammengefasst aufgeführt.

### 3.8.1 Bezeichnerwahl

Stilistisch gute Programme werden oft als selbstdokumentierend bezeichnet. In erster Linie wird dabei auf die geschickte Wahl von Bezeichnern angespielt, die es dem Leser erlauben, die Programmlogik schnell und ohne die Hilfe zusätzlicher Kommentare zu verstehen. (Daraus folgt aber nicht, daß Kommentare überflüssig werden!) Dies wird erreicht, wenn die Bedeutung einer Anweisung aus den Namen

der beteiligten Variablen und der verwendeten Schlüsselwörter bzw. Operatoren hervorgeht.

Richtig gewählte Namen sind so kurz wie möglich, aber lang genug, um die Funktion oder das Objekt, das sie bezeichnen, so zu beschreiben, daß jemand, der mit der Aufgabenstellung vertraut ist, sich darunter etwas vorstellen kann. Selten gebrauchte Namen können etwas länger als häufig gebrauchte Namen sein.

Für die Bezeichner in Pascal-Programmen geben wir folgende Regeln vor:

Muß-Regel 1

- Bezeichner sind aussagekräftig (sprechend) und orientieren sich an dem Problem, welches das Programm löst.
- **Wird ein Bezeichner aus mehreren Worten zusammengesetzt, so beginnt jedes Teilwort mit einem Großbuchstaben, z.B. KnotenAnzahl.**  
(Standard-Pascal unterscheidet übrigens nicht zwischen Groß- und Kleinschreibung.)

Muß-Regel 2

- **Konstantenbezeichner bestehen nur aus Großbuchstaben. Typbezeichnern wird ein t vorangestellt.**  
Es ist hilfreich, wenn an einem Bezeichner direkt abgelesen werden kann, ob er eine Variable, eine Konstante oder einen Typ bezeichnet. Dies erspart lästiges Nachschlagen der Vereinbarungen und vermindert Fehlbenutzungen. Wir erreichen eine Unterscheidung durch kurze Präfixe sowie Groß- und Kleinschreibungsregeln.
- Für Variablen bieten sich häufig Substantive als Bezeichner an, für Bezeichner von Funktionen und Prozeduren werden am besten Verben (welche die Tätigkeit beschreiben) verwendet, für Zustandsvariablen oder boolesche Funktionen eignen sich Adjektive oder Adverben.

### Abkürzungen

Bei langen Namen ist eine gewisse Vorsicht angebracht, da diese umständlich zu handhaben sind und die Lesbarkeit von Programmen vermindern, weil sie die Programmstruktur verdecken können. Unnötig lange Namen sind Fehlerquellen, der Dokumentationswert eines Namens ist nicht proportional zu seiner Länge.

Allerdings ist es oft schwierig, kurze und gleichzeitig präzise Bezeichner zu finden. In diesem Fall werden für Bezeichner Abkürzungen gewählt. Wir wollen dafür einige Regeln zusammentragen:

- Ein spezieller Kontext erleichtert das Abkürzen von Bezeichnern, ohne daß diese dadurch an Bedeutung verlieren. Allgemein gilt: Wird eine Abkürzung benutzt, sollte sie entweder im Kontext verständlich oder allgemeinverständlich sein.
- Wenn Abkürzungen des allgemeinen Sprachgebrauchs existieren, sind diese zu verwenden (z. B. Pers für Person oder Std für Stunde).
- Häufig gebrauchte Namen können etwas kürzer als selten gebrauchte Namen sein.
- Eine Abkürzung muß enträtselbar und sollte aussprechbar sein.

- Beim Verkürzen sind Wortanfänge wichtiger als Wortenden, Konsonanten sind wichtiger als Vokale.
- Es ist zu vermeiden, daß durch Verkürzen mehrere sehr ähnliche Bezeichner entstehen, die leicht verwechselt werden können.

### 3.8.2 Programmtext-Layout

Die Lesbarkeit eines Programms hängt auch von der äußeren Form des Programms ab. Für die äußere Form von Programmen gibt es viele Vorschläge und Beispiele aus der Praxis, die sich aber oft am Einzelfall oder an einer konkreten Programmiersprache orientieren. Für unsere Zwecke geben wir die folgenden Regeln an:

- Die Definitionen und Deklarationen werden nach einem eindeutigen Schema gegliedert. Dies fordert bereits die Pascal-Syntax, aber nicht in allen Programmiersprachen ist die Reihenfolge von Definitionen und Deklarationen vorgeschrieben.
- **Jede Anweisung beginnt in einer neuen Zeile; `begin` und `end` stehen jeweils in einer eigenen Zeile.**
- Der Einsatz von redundanten Klammern und Leerzeichen in Ausdrücken kann die Lesbarkeit erhöhen. Insbesondere sind Klammern dann einzusetzen, wenn die Priorität der Operatoren unklar ist. Die Prioritäten in verschiedenen Programmiersprachen folgen eventuell unterschiedlichen Hierarchien. Aber auch hier gilt wie bei Bezeichnerlängen: Ein Zuviel hat den gegenteiligen Effekt!
- Die hervorgehobene Darstellung von Schlüsselwörtern einer Programmiersprache unterstützt die Verdeutlichung der Programmstruktur. Wir heben Schlüsselwörter durch Fettdruck hervor. In Programmtexten, die von Hand geschrieben werden, wird durch Unterstreichung hervorgehoben.
- Kommentare werden so im Programmtext eingefügt, daß sie einerseits leicht zu finden sind und ausreichende Erläuterungen bieten, andererseits aber die Programmstruktur deutlich sichtbar bleibt (vgl. auch Abschnitt "Kommentare")

Muß-Regel 3

### Einrückungen

**Die Struktur eines Programms wird durch Einrückungen hervorgehoben. Wir geben die folgenden Regeln an:**

- **Anweisungsfolgen werden zwischen `begin` und `end` um eine konstante Anzahl von 2 - 4 Stellen eingerückt. `begin` und `end` stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.**
- **Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.**
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **`end`** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist (vgl. auch Abschnitt "Kommentare").

Muß-Regel 4

Muß-Regel 5

Beispiele für die letzten drei Regeln:

```

if <Bedingung> then
begin
    <Anweisungsfolge>
end
else { if not <Bedingung> }
begin
    <Anweisungsfolge>
end { if <Bedingung> };

```

- In einem Block werden die Definitions- und Deklarationsteile ebenfalls eingerückt. Die entsprechenden Schlüsselwörter stehen eingerückt in einer eigenen Zeile. Die vereinbarten Konstanten, Datentypen und Variablen werden darunter linksbündig angeordnet, in jeder Zeile sollte nur ein Bezeichner stehen. **begin** und **end** des Anweisungsteils werden nicht eingerückt. Z.B:

```

program einruecken (input, output);
{ Beispielpogramm fuer das Programmtext-Layout }

const
PI = 3.1415927;
ANTWORT = 42;

type
tWochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);

var
Heute,
Gestern : tWochentag;

begin
    write ('Die Antwort ist: ');
    writeln (ANTWORT)
end. { einruecken }

```

In Ausnahmefällen, damit z.B. ein Programm (noch) auf eine Seite paßt, kann von diesen Regeln abgewichen werden. Falls das Programm in jedem Fall durch einen Seitenumbruch getrennt wird, gelten die Layoutregeln wieder.

### Leerzeilen

Leerzeilen dienen der weiteren Gliederung des Programmtextes. Es empfiehlt sich, zusammengehörige Definitionen, Deklarationen oder Anweisungen auch optisch im Programmtext zusammenzufassen. Die einfachste Art ist die Trennung solcher Gruppen durch Leerzeilen. Es bietet sich an, einer solchen Gruppe einen zusammenfassenden Kommentar voranzustellen.

Wir schlagen den Einsatz von Leerzeilen in folgenden Fällen vor:

- Die verschiedenen Definitions- und Deklarationsteile werden durch Leerzeilen gegliedert.

- Anweisungssequenzen von mehr als ca. 10 Zeilen sollten durch Leerzeilen in Gruppen unterteilt werden.

Bei den in diesem Abschnitt "Programmtext-Layout" angesprochenen Richtlinien ist wichtig, daß selbstaufgelegte Konventionen konsequent eingehalten werden, um ein einheitliches Aussehen aller Programmbausteine zu erreichen. Um dem gesamten Programmtext ein konsistentes Aussehen zu geben, werden automatische *Programmtext-Formatierer*, sogenannte *Prettyprinter*, eingesetzt. Solche Programme formatieren einen (syntaktisch korrekten) Programmtext nachträglich gemäß fester oder in einem gewissen Rahmen variabler Regeln. Durch *syntaxgesteuerte Editoren* erreicht man vergleichbare Ergebnisse direkt bei der Programmtext-Eingabe.

Programmtext-  
Formatierer  
Prettyprinter  
  
syntaxgesteuerter  
Editor

### 3.8.3 Kommentare

Kommentare sind nützliche und notwendige Hilfsmittel zur Verbesserung der Lesbarkeit von Programmen. Sie sind wichtige Bestandteile von Programmen, und ihre Abwesenheit ist ein Qualitätsmangel.

Die Verwendung von Kommentaren ist in jeder höheren Programmiersprache möglich. Wir unterscheiden Kommentare, welche die Historie, den aktuellen Stand (Version) oder die Funktion von Programmen beschreiben - sie werden an den Programmanfang gestellt - und solche, die Objekte, Programmteile und Anweisungen erläutern - sie werden im Deklarations- und Anweisungsteil benutzt.

Die folgenden Konventionen bleiben an einigen Stellen unscharf, wichtig ist auch hier, daß man sich konsequent an die selbst gegebenen Regeln hält.

- Kommentare werden während der Programmierung eingefügt und nicht erst nachträglich ergänzt. Selbstverständlich werden die Kommentare bei Änderungen angepaßt.
- Kommentare sind so zu plazieren, daß sie die Programmstruktur nicht verdecken, d.h. Kommentare werden mindestens so weit eingerückt wie die entsprechenden Anweisungen. Trennende Kommentare wie Sternchenreihen oder Linien werden so sparsam wie möglich eingesetzt, da sie den Programmtext leicht zerschneiden.
- **Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst (und nicht die Arbeitsweise des Programms).**  
**Eine Kommentierung der Ein- und Ausgabedaten gehört gewöhnlich dazu.**
- Die Bedeutung von Konstanten, Datentypen und Variablen wird erläutert, wenn sie nicht offensichtlich ist.
- Komplexe Anweisungen oder Ausdrücke werden kommentiert.
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **end** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist.
- Kommentare sind prägnant zu formulieren und auf das Wesentliche zu beschränken. Sie wiederholen nicht den Code mit anderen Worten, z.B.

Muß-Regel 6

`i := i + 1; {erhoehe i um 1},`

sondern werden nur da eingesetzt, wo sie zusätzliche Information liefern.

### 3.8.4 Sonstige Merkregeln

Einige weitere Regeln wollen wir kurz auflisten:

- Es werden nach Möglichkeit keine impliziten (automatischen) Typanpassungen benutzt.
- Selbstdefinierte Datentypen werden nicht implizit definiert.

# Kurseinheit II

## Lernziele zum Kapitel 4

Nach diesem Kapitel sollten Sie

1. die drei Wiederholungsanweisungen von Pascal kennen und wissen, in welcher Situation welche Schleifenart sinnvollerweise zu verwenden ist,
2. eine gegebene Wiederholungsanweisung in eine andere, bedeutungsgleiche Wiederholungsanweisung umformen können,
3. die Anwendungsbereiche der strukturierten Datentypen Feld (**array**) und Verbund (**record**) kennen,
4. mit Hilfe strukturierter Datentypen von Pascal problemadäquate, selbstdefinierte Datentypen festlegen und die darauf zulässigen Operationen anwenden können,
5. eine Funktion deklarieren und aufrufen können,
6. die Begriffe „formaler Parameter“, „aktueller Parameter“ und „lokale Variable“ kennen,
7. einfache Programme schreiben können, die ein übersichtliches Layout, sinnvolle Kommentierung sowie wohlüberlegte Datentypen und Kontrollstrukturen verwenden.



## 4. Programmierkonzepte orientiert an Pascal (Teil 2)

### 4.1 Wiederholungsanweisungen

Wie alle imperativen Programmiersprachen kennt auch Pascal (Steuer-)Anweisungen, die für die wiederholte Abarbeitung von Anweisungen sorgen. Diese sogenannten Wiederholungsanweisungen sind die **for**-Anweisung, die **while**-Anweisung und die **repeat**-Anweisung.

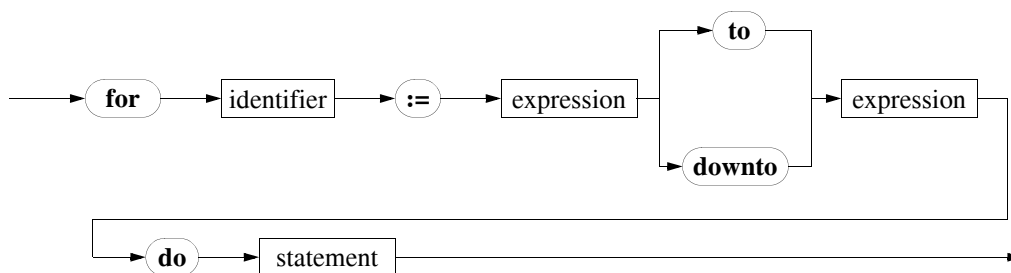
Eine *Wiederholungsanweisung*, auch *Schleife* genannt, führt eine Folge von Anweisungen, den *Schleifenrumpf*, solange aus, wie die sogenannte *Schleifenbedingung* erfüllt ist. Sobald die Schleifenbedingung nicht mehr erfüllt ist, wird die Wiederholungsanweisung beendet, und das Programm setzt mit der darauf folgenden Anweisung fort. Die Negation der Schleifenbedingung heißt *Abbruchbedingung* (oder *Abbruchkriterium*), denn sobald sie erfüllt ist, bricht die Schleife ab. Die einmalige Ausführung der zu wiederholenden Folge von Anweisungen bezeichnen wir als *Schleifendurchlauf*.

Ein fehlerhaft formuliertes Abbruchkriterium kann eventuell nie erfüllt werden, so daß die Schleife immer wieder durchlaufen wird und nicht abbricht. In diesem Fall sprechen wir von einer *Endlosschleife*.

#### 4.1.1 Die for-Schleife

Die **for**-Schleife, auch *Laufanweisung* genannt, wird verwendet, wenn die Anzahl der Schleifendurchläufe vor dem Eintritt in die Schleife bekannt ist. Sie hat die Form:

**for-statement**



Der Bezeichner wird *Kontrollvariable* oder *Laufvariable* genannt und muß einem unstrukturierten Typ ungleich `real` angehören. Der erste Ausdruck wird auch *Anfangsausdruck*, der zweite Ausdruck auch *Endausdruck* genannt. Beide Ausdrücke und die Kontrollvariable müssen vom selben Typ sein.

Bei der Durchführung der **for**-Schleife nimmt die Kontrollvariable zunächst den Wert des Anfangsausdrucks und dann alle zulässigen Zwischenwerte bis einschließlich des Wertes des Endausdrucks an. Bei Verwendung von **to** ist die Folge der Zwischenwerte aufsteigend, bei **downto** absteigend. Die zulässigen Zwischen-

Wiederholungs-  
anweisung, Schleife  
Schleifenrumpf  
Schleifenbedingung  
Abbruchbedingung  
Schleifendurchlauf

Endlosschleife

for-Schleife

Laufanweisung

Kontrollvariable,  
Laufvariable

Anfangsausdruck,  
Endausdruck

Schleifenbedingung  
der **for**-Schleife

werte sind durch den Typ der Kontrollvariablen festgelegt. Ist beispielsweise die Kontrollvariable vom Typ `integer`, der Anfangswert 10 und der Endwert 20, dann nimmt die Kontrollvariable bei Verwendung von **to** die Werte 10, 11, ... , 20 an. Für jeden von der Kontrollvariablen angenommenen Wert wird der Schleifenrumpf ausgeführt, der hinter dem Schlüsselwort **do** steht. Die *Schleifenbedingung der **for**-Schleife* lautet also: Der Wert der Kontrollvariablen ist kleiner oder gleich (im Fall von **to**) bzw. größer oder gleich (im Fall von **downto**) als der Wert des Endausdrucks und größer oder gleich (im Fall von **to**) bzw. kleiner oder gleich (im Fall von **downto**) als der Wert des Anfangsausdrucks. Ist der Wert des Anfangsausdrucks größer (im Fall von **to**) oder kleiner (im Fall von **downto**) als der Wert des Endausdrucks, so wird der Schleifenrumpf *keinmal* ausgeführt.

Namenswahl der  
Kontrollvariablen

Bei der *Namenswahl der Kontrollvariablen* werden üblicherweise kurze Bezeichnernamen wie `i`, `j`, `k` verwendet. Dies steht in gewissem Widerspruch zu unserem Hinweis, möglichst selbsterklärende Bezeichner zu verwenden. Die Verwendung von z.B. `Zaehler` oder `Index` führt aber schnell zu schlecht lesbaren Ausdrücken, so daß wir uns im folgenden an diese Konvention halten wollen.

#### Beispiel 4.1.1.1

Gesucht ist ein Programm, das die Quadratzahlen für die Zahlen 1 bis 10 in Form einer Tabelle ausgibt. Anstelle des Programms mit zehn Anweisungen

```
program Quadratzahl1 (input, output);
{ berechnet die Quadrate der Zahlen von 1 bis 10 }
```

**begin**

```
  writeln (1:2, sqr (1):5);
  writeln (2:2, sqr (2):5);
  writeln (3:2, sqr (3):5);
  writeln (4:2, sqr (4):5);
  writeln (5:2, sqr (5):5);
  writeln (6:2, sqr (6):5);
  writeln (7:2, sqr (7):5);
  writeln (8:2, sqr (8):5);
  writeln (9:2, sqr (9):5);
  writeln (10:2, sqr (10):5)
```

**end.** { Quadratzahl1 }

verwenden wir besser eine **for**-Schleife:

```
program Quadratzahl2 (input, output);
{ berechnet die Quadrate der Zahlen von 1 bis 10 }
```

**const**

```
MAXINDEX = 10;
```

**type**

```
tIndex = 1..MAXINDEX;
```

```

var
  i : tIndex; { Laufvariable }

begin
  for i := 1 to MAXINDEX do
    writeln (i:2, sqr (i):5)
end. { Quadratzahl2 }

```

1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100



#### Beispiel 4.1.1.2

Es ist ein Programm zu schreiben, das eine integer-Zahl  $\geq 0$  einliest, die Fakultät der Zahl berechnet und ausgibt. Die Fakultät  $f(n)$  einer natürlichen Zahl  $n$  ist definiert als:

$$f(n) = \begin{cases} 1, & \text{falls } n = 0 \\ \prod_{i=1}^n i, & \text{falls } n > 0. \end{cases}$$

Beispielsweise ist  $f(6) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$ .

Das zugehörige Programm könnte z.B. lauten:

```

program Fakultaet (input, output);
{ berechnet die Fakultaet einer einzulesenden
  natuerlichen Zahl }

type
  tNatZahl = 0..maxint;

var
  i : tNatZahl; { Laufvariable }
  Zahl, { einzulesende Zahl }
  Ergebnis : integer;

```

```

begin
  write ('Fuer welche Zahl soll die Fakultaet ',
        'berechnet werden? ');
  readln (Zahl);

  if Zahl < 0 then
    writeln ('Fuer negative Zahlen ist die ',
            'Fakultaet nicht definiert.')
  else
    begin
      Ergebnis := 1; { Initialisierung }
      for i := 1 to Zahl do
        Ergebnis := Ergebnis * i;
      writeln ('Die Fakultaet von ', Zahl, ' lautet ',
              Ergebnis, '.')
    end
  end. { Fakultaet }

```

*Fuer welche Zahl soll die Fakultaet berechnet werden? 6  
Die Fakultaet von 6 lautet 720.*



#### Beispiel 4.1.1.3

Gesucht ist ein Programm, das vom Benutzer die Eingabe einer vorgegebenen Anzahl von integer-Zahlen erwartet, das Maximum dieser Zahlen bestimmt und ausgibt.

```

program Maximum (input, output);
{ bestimmt das Maximum einer Folge von
  einzulesenden integer-Zahlen }

const
  ANZ = 4; { Anzahl der einzulesenden Zahlen }

type
  tIndex = 1..ANZ;

var
  i : tIndex; { Laufvariable }
  Zahl,
  Max : integer;

begin
  writeln ('Geben Sie ', ANZ, ' ganze Zahlen ein,');
  writeln ('deren Maximum bestimmt werden soll.');
```

```
write ('1. Wert: ');
readln (Zahl);
Max := Zahl;
for i := 2 to ANZ do
begin
    write (i, '. Wert: ');
    readln (Zahl);
    if Zahl > Max then
        Max := Zahl
    end; { for-Schleife }
writeln ('Das Maximum ist: ', Max, '.')
end. { Maximum }
```

Geben Sie 4 ganze Zahlen ein,  
deren Maximum bestimmt werden soll.

1. Wert: 9  
2. Wert: 19  
3. Wert: 23  
4. Wert: 11  
Das Maximum ist: 23.



Bei der Verwendung der **for**-Schleife sind folgende Punkte zu berücksichtigen:

- Die Anfangs- und Endwerte, die durch die entsprechenden Ausdrücke definiert sind, werden nur einmal beim Eintritt in die Schleife berechnet. Die in diesen Ausdrücken verwendeten Variablen können also innerhalb der Schleife unbeschadet verändert werden.
- Ist der Wert des Anfangsausdrucks größer (im Fall von **to**) oder kleiner (im Fall von **downto**) als der Wert des Endausdrucks, so wird der Schleifenrumpf keinmal ausgeführt.
- Der Wert der Kontrollvariablen wird definitionsgemäß automatisch mit jedem Durchlauf durch die Schleife verändert. Es sollte unbedingt vermieden werden, ihn zusätzlich durch Anweisungen innerhalb der Schleife zu verändern.
- Nach Beendigung der **for**-Schleife ist der Wert der Kontrollvariablen nicht mehr definiert. Es dürfen deshalb im weiteren Verlauf des Programms keine Annahmen über ihren Wert gemacht werden.

#### 4.1.2 Die while-Schleife

Die **while**-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe vor dem Eintritt in die Schleife nicht bekannt ist. Die Schleifenbedingung wird bei der

Charakteristika der  
**for**-Schleife

Einsatz der  
**while**-Schleife

**while**-Schleife stets vor der Ausführung des zu wiederholenden Schleifenrumpfes überprüft. Das Syntaxdiagramm für die **while**-Schleife lautet:

#### while-statement



Die Schleifenbedingung "expression" ist vom Typ `boolean`. Von ihrem Wert hängt es ab, ob die Anweisung ausgeführt wird (`expression = true`) oder nicht (`expression = false`). Vor jedem weiteren Schleifendurchlauf wird die Schleifenbedingung erneut ausgewertet. In Abhängigkeit vom ermittelten Wert wird entschieden, ob der Schleifenrumpf abermals auszuführen ist. Falls die Schleifenbedingung schon vor dem ersten Schleifendurchlauf nicht erfüllt ist, wird der Anweisungsteil komplett übergangen.

Eine typische Anwendung der **while**-Schleife ergibt sich, wenn eine vorher nicht bekannte Anzahl von Daten einzulesen ist. Hier gibt es verschiedene Möglichkeiten, das Eingabeende zu signalisieren. So kann z.B. die Eingabe von (von Null verschiedenen) `integer`-Zahlen durch eine Null abgeschlossen werden.

#### Beispiel 4.1.2.1

Das folgende Programm liest von Null verschiedene `integer`-Zahlen ein und gibt anschließend die kleinste eingelesene Zahl aus. Das Ende der Eingabefolge wird durch eine Null signalisiert. Der Sonderfall, daß nur eine Null eingegeben wird, ist berücksichtigt.

```

program kleinsteZahl1 (input, output);
{ gibt die kleinste Zahl der integer-Eingabezahlen aus }

var
  Zahl,
  Minimum : integer;

begin
  writeln ('Geben Sie die integer-Zahlen ein. ',
    '0 beendet die Eingabe. ');
  readln (Zahl);
  Minimum := Zahl;
  while Zahl <> 0 do
    begin
      if Zahl < Minimum then
        Minimum := Zahl;
      readln (Zahl)
    end;
  if Minimum <> 0 then
    { Minimum ist genau dann Null, wenn die erste (und

```

```

        einzige) Zahl eine Null war }
    writeln ('Die kleinste Zahl lautet ', Minimum, '.')
else
    writeln ('Es wurde nur eine 0 eingegeben.')
end. { kleinsteZahl1 }

```

```

Geben Sie die integer-Zahlen ein. 0 beendet die Eingabe.
20
36
-6
9
-2
0
Die kleinste Zahl lautet -6.

```

Beachten Sie, daß beim ersten Schleifendurchlauf stets `Zahl = Minimum` gilt, und damit lediglich die zweite Zahl gelesen wird.



#### Beispiel 4.1.2.2

Das folgende Programm `ggTBerechnung` berechnet für zwei natürliche Zahlen  $x$  und  $y$  den größten gemeinsamen Teiler nach dem Euklidischen Algorithmus:

*Euklidischer Algorithmus:*

"Lies zwei natürliche Zahlen  $x$  und  $y$  ungleich Null und gib ihren größten gemeinsamen Teiler `ggT` aus."

Vorbedingung<sup>1</sup>:  $x, y \in \mathbb{N}^*$

Nachbedingung: `ggT` ist die größte natürliche Zahl,  
die sowohl  $x$  als auch  $y$  teilt.

Verfahren:

```

Lese  $x$  und  $y$  ;
solange  $x \neq y$ 
    subtrahiere den kleineren der beiden Werte vom größeren;
schreibe "Der ggT ist "  $x$ .

```

Eine mögliche Realisierung des Euklidischen Algorithmus als Pascal-Programm lautet:

---

1. zur Erinnerung:  $\mathbb{N}^*$  bezeichnet die Menge der natürlichen Zahlen ungleich Null.

```

program ggTBerechnung (input, output);
{ berechnet den groessten gemeinsamen Teiler zweier
  einzulesender integer-Zahlen x und y groesser Null }

  var
    x,
    y : integer;

begin
  writeln ('Bitte geben Sie 2 natuerliche Zahlen > 0 ',
    'ein:');
  readln (x);
  readln (y);
  if (x <= 0) or (y <= 0) then
    writeln ('Eingabefehler!')
  else
    begin
      write ('Der ggT von ', x, ' und ', y, ' ist ');
      { Ausgabe von x und y schon hier, da nach der
        while-Schleife die Originalwerte nicht mehr
        existieren }
      while x <> y do
        if x > y then
          x := x - y
        else
          y := y - x;
        writeln (x, '.')
      end
    end. { ggTBerechnung }

```

```

Bitte geben Sie 2 natuerliche Zahlen >0 ein:
36
78
Der ggT von 36 und 78 ist 6.

```



Besteht die Schleifenbedingung einer **while**-Schleife aus mehreren logischen Teilausdrücken, die mit **and** verknüpft sind, dann ist nach Abbruch der Schleife nicht immer unmittelbar klar, welcher Teilausdruck zum Schleifenabbruch geführt hat. Die Schleife

```

while (a < b) and (c < d) do
  begin
    .
  
```



```

    •
    •
end

```

bricht ab, falls  $a \geq b$  oder  $c \geq d$  oder beides, nämlich  $(a \geq b) \textbf{ and } (c \geq d)$ , gilt. Hängt nun - was oft der Fall ist - der weitere Programmverlauf davon ab, welcher Fall zum Schleifenabbruch geführt hat, dann muß dies im Anschluß an die Schleife durch eine geeignete **if**-Anweisung festgestellt werden, also in unserem Beispiel durch

```

if (a >= b) and (c >= d) then
    •
    •
    •
else
    if (a >= b) then
        •
        •
        •
    else { hier gilt c >= d }
        •
        •
        •

```

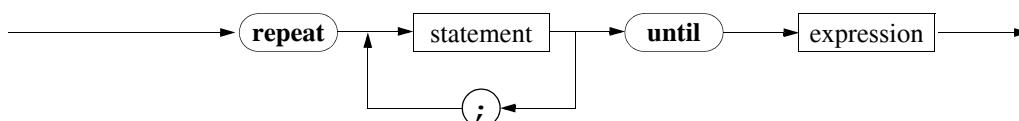
Noch eine abschließende Bemerkung:

Die Schleifenbedingung muß eine Variable enthalten, deren Wert sich bei der Bearbeitung des Schleifenrumpfes ändert. Andernfalls haben wir eine Endlosschleife programmiert, und die Programmbearbeitung bricht nicht ab. Falls das Betriebssystem des Rechners für diesen Fall keine Abbruchmöglichkeit vorsieht, kommen wir nicht umhin, den Rechner auszuschalten, was üblicherweise mit dem Verlust der aktuellen Programmdaten verbunden ist.

#### 4.1.3 Die repeat-Schleife

Die Überprüfung des Abbruchkriteriums erfolgt bei der **repeat**-Schleife im Gegensatz zur **while**-Schleife nicht vor, sondern nach dem zu wiederholenden Schleifenrumpf. Der Schleifenrumpf in einer **repeat**-Schleife wird somit stets mindestens einmal ausgeführt, während der einer **while**-Schleife komplett übergangen wird, falls bereits vor dem ersten Schleifendurchlauf die Abbruchbedingung erfüllt ist. Das Syntaxdiagramm der **repeat**-Schleife hat die Form:

**repeat-statement**



Wenn der Schleifenrumpf der **repeat**-Schleife aus mehreren Anweisungen besteht, brauchen wir diese nicht mittels **begin** und **end** zu einer zusammengesetzten Anweisung zu klammern, da hier die Schlüsselwörter **repeat** und **until** diese Funktion übernehmen.

Unsere Anmerkungen zu Endlosschleifen und zusammengesetzten Abbruchkriterien gelten hier sinngemäß wie bei der **while**-Schleife.

#### Beispiel 4.1.3.1

Das Programm aus Beispiel 4.1.2.1 wird nun mit Hilfe einer **repeat**-Schleife realisiert.

```
program kleinsteZahl2 (input, output);
{ gibt die kleinste Zahl der integer-Eingabezahlen aus }

var
  Zahl,
  Minimum : integer;

begin
  writeln ('Geben Sie die integer-Zahlen ein. ',
           '0 beendet die Eingabe. ');
  readln (Zahl);
  Minimum := Zahl;
  if Zahl <> 0 then
    repeat
      if Zahl < Minimum then
        Minimum := Zahl;
      readln (Zahl)
    until Zahl = 0;
  if Minimum <> 0 then
    { Minimum ist genau dann Null, wenn die erste und
      einzige Zahl eine Null war }
    writeln ('Die kleinste Zahl lautet ', Minimum, '.')
  else
    writeln ('Es wurde keine Zahl <> 0 eingegeben.')
end. { kleinsteZahl2 }
```

□

Ersetzung von  
**while**- durch  
**repeat**-Schleifen

Das obige Programm zeigt exemplarisch, wie grundsätzlich jede **while**-Schleife durch eine **repeat**-Schleife mit derselben Bedeutung ersetzt werden kann. Die **while**-Schleife

```
while expression do
  statement
```

läßt sich stets schreiben als

```

if expression then
  repeat
    statement
  until not expression

```

Umgekehrt läßt sich jede **repeat**-Schleife durch eine **while**-Schleife mit derselben Bedeutung ersetzen:

```

repeat
  statement
until expression

```

ist äquivalent zu

```

statement;
while not expression do
  begin
    statement
  end

```

In beiden Fällen führen die jeweiligen Ersetzungen zu komplizierteren Programmstücken. Dies zeigt, daß eine für das jeweilige Problem geeignete Schleifenwahl zu kompakteren und übersichtlicheren Programmen führt. Für die Wahl des geeigneten Schleifenkonstrukts läßt sich folgende Faustregel angeben:

Die **repeat**-Schleife ist zu wählen, wenn unabhängig von der Schleifenbedingung stets mindestens ein Schleifendurchlauf erfolgen soll. Hängt bereits der erste Schleifendurchlauf von der Erfüllung der Schleifenbedingung ab, so ist die **while**-Schleife die geeignete Wahl.

Zur Erinnerung:

Die **for**-Schleife ist gewöhnlich das geeignete Schleifenkonstrukt, wenn die Anzahl der Schleifendurchläufe vor Eintritt in die Schleife bekannt ist.

#### Aufgabe 4.1.3.2

Wir hatten für das Programm in Beispiel 4.1.3.1 die Version gewählt, die dem allgemeinen Ersetzungsschema einer **while**- durch eine **repeat**-Schleife entspricht. Dieses Programm läßt sich allerdings noch kompakter schreiben. Der Fall, daß die Eingabe nur aus einer Null besteht, wird nämlich zweimal abgefragt. Dies läßt sich verbessern, so daß wir nur mit einer einzigen **if**-Anweisung auskommen.



Die Abfrage nach einem speziellen Eingabewert (hier Null) zur Markierung des Eingabeendes ist keine sehr praktische Lösung. In unseren Beispielen können wir die Null nicht als "gewöhnliche" Eingabezahl verwenden, da sie zur Markierung des Eingabeendes reserviert ist. Angemessener ist daher meist die Verwendung der Standardfunktion `eof`, die wir schon im Kapitel 3 erwähnt haben. Um dem Rechner

Ersetzung von  
**repeat**- durch  
**while**-Schleifen

Auswahl der  
geeigneten Schleife

Standardfunktion  
**eof**

das Eingabeende mitzuteilen, muß unter UNIX die eof-Tastenkombination <ctrl> d bzw. <strg> d und bei Betriebssystemen, die auf MS-DOS basieren, die eof-Tastenkombination <ctrl> z bzw. <strg> z verwendet werden. Schließen Sie unter MS-DOS die Eingabe zusätzlich mit der Eingabetaste (Return) ab.

Alle Eingaben von der Tastatur werden dem Programm in einem Zwischenpuffer zur Verfügung gestellt. Ein Programm kann erst dann (z.B. mit read oder readln-Befehlen) daraus lesen, wenn eine Eingabezeile abgeschlossen wurde. Ist der Zwischenpuffer leer, weil noch nichts eingegeben wurde oder alle eingegebenen Zeichen schon gelesen wurden, so kann die Standardfunktion eof nicht erkennen, ob das Ende der Eingabe (input) erreicht ist. Deshalb hält der Aufruf von eof bei einem leeren Zwischenpuffer das Programm an, bis auf der Tastatur eine Eingabe erfolgt ist. Danach kann mit eof festgestellt werden, ob das Ende der input-Datei erreicht, die entsprechende Tastenkombination also eingegeben worden ist. Ist dies der Fall, liefert eof den Wert true zurück. Andernfalls liefert eof den Wert false zurück und die Eingabedaten stehen im Zwischenpuffer für read-Befehle bereit.

Ist das Ende der input-Datei erreicht (und damit eof = true), dürfen keinen weiteren read-Befehle mehr für die input-Datei erfolgen. Führt das Programm einen weiteren read-Befehl aus, so erfolgt meist ein Programmabbruch. Dies ist aber nicht bei allen Pascal-Compilern der Fall. Turbo-Pascal liefert in diesem Fall keine Fehlermeldung und läßt die Variablen unverändert. Sollten Sie mit unseren (allgemeinen) Ausführungen nicht zurechtkommen, bleibt Ihnen ein Blick in die entsprechenden Handbücher nicht erspart.

#### Beispiel 4.1.3.3

Das Programm aus Beispiel 4.1.2.1 läßt sich bei Verwendung der eof-Funktion schreiben als

```
program kleinsteZahl3 (input, output);
{ gibt die kleinste Zahl der integer-Eingabezahlen aus }

var
  Zahl,
  Minimum : integer;

begin
  writeln ('Geben Sie die integer-Zahlen ein. ',
          '<ctrl> z beendet die Eingabe. ');
  if eof then
    writeln ('Es wurde keine Zahl eingegeben.')
  else
    begin
      readln (Zahl);
      Minimum := Zahl;
      while not eof do
        begin
```

```

    readln (Zahl);
    if Zahl < Minimum then
        Minimum := Zahl
    end;
    writeln ('Die kleinste Zahl lautet ', Minimum, '.')
end
end. { kleinsteZahl3 }

```



## 4.2 Strukturierte Datentypen

Das Prinzip, komplexere Konstrukte durch die strukturierte Zusammensetzung einfacherer Konstrukte zu erhalten, gilt in imperativen Programmiersprachen nicht nur für Anweisungen, sondern auch für Datentypen. Betrachten wir etwa folgendes Beispiel: An einem Seminar der FernUniversität nehmen 12 Studenten teil, die einen Schein erhalten, wenn sie einen Vortrag gehalten haben. Gesucht ist ein Programm, das die Art der Teilnahme (aktiv oder passiv) jedes einzelnen Studenten einliest, die Matrikel-Nummern der erfolgreichen Teilnehmer ausdruckt und zum Abschluß sowohl eine Liste der aktiven als auch der passiven Teilnehmer erstellt.

Zwei Problemeigenschaften machen es schwierig, das Programm mit Hilfe unserer bekannten Programmstrukturen in Pascal zu entwickeln. Einerseits muß eine ähnliche Bearbeitung für jeden einzelnen Studenten durchgeführt werden - Einlesen der Matrikel-Nummer und der Teilnahmeart, eventuelle Ausgabe eines Scheins, Einsetzen der Matrikel-Nummer in eine der zwei Listen. Andererseits ist es nötig, die Nummern und Teilnahmearten aller 12 Studenten durch das gesamte Programm hindurch zu speichern, da nur ein Drucker zur Verfügung steht und Matrikelnummern und Listen nacheinander ausgegeben werden müssen.

Wir benötigen also 12 Variablen, um die Teilnahmeart zu speichern, und 12 Variablen für die Matrikelnummern. Wir nehmen der Einfachheit halber an, daß eine Matrikel-Nummer durch einen `integer`-Wert repräsentiert wird. Dann könnte eine (schon recht mühsame) Typ- und Variablenvereinbarung so aussehen:

```

...
type
tNatZahlPlus = 1..maxint;
tStatus = (aktiv, passiv);
...

var
StudMatr1,
StudMatr2,
...
StudMatr12 : tNatZahlPlus;

StudTeilnahme1,
StudTeilnahme2,

```

```

...
StudTeilnahme12 : tStatus;

Status : char; { Zeichen zum Einlesen des Studenten-
                status. Muss vom Typ char sein, um
                eingelesen werden zu koennen.
                'a' entspricht aktiv,
                'p' entspricht passiv }

...

```

Eine Programmierung wird dann aber ausgesprochen lästig, wie diese Programmfragmente zeigen:

```

begin
  readln (StudMatr1);
  readln (Status);
  if Status = 'a' then
    StudTeilnahme1 := aktiv
  else
    StudTeilnahme1 := passiv;
  readln (StudMatr2);
  readln (Status);
  if Status = 'a' then
    StudTeilnahme2 := aktiv
  else
    StudTeilnahme2 := passiv;
  ...
  readln (StudMatr12);
  readln (Status);
  if Status = 'a' then
    StudTeilnahme12 := aktiv
  else
    StudTeilnahme12 := passiv;

  writeln('Liste aller Seminar-Scheine');
  {Jeder aktive Teilnehmer bekommt einen Schein.}
  if StudTeilnahme1 = aktiv then
    writeln (StudMatr1);
  if StudTeilnahme2 = aktiv then
    writeln (StudMatr2);
  ...
  if StudTeilnahme12 = aktiv then
    writeln (StudMatr12);

  writeln('Liste aller aktiven Seminar-Teilnehmer');
  if StudTeilnahme1 = aktiv then
    writeln (StudMatr1);
  if StudTeilnahme2 = aktiv then

```

```

        writeln (StudMatr2);
    ...
    if StudTeilnahme12 = aktiv then
        writeln (StudMatr12);

    writeln('Liste aller Zuhörer im Seminar');
    if StudTeilnahme1 = passiv then
        writeln (StudMatr1);
    if StudTeilnahme2 = passiv then
        writeln (StudMatr2);
    ...
    if StudTeilnahme12 = passiv then
        writeln (StudMatr12)
end

```

Ein solches Programm zu schreiben, ist einigermaßen mühsam. Wieviel mühsamer wäre es erst, ein ähnliches Problem für einen großen Kurs zu lösen, der einige tausend Teilnehmer hat. Für die Lösung solcher Probleme bietet Pascal daher den Datentyp **array** an.

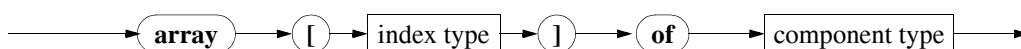
#### 4.2.1 Der array-Typ

Zur Lösung des oben beschriebenen Problems wäre es günstiger, wenn wir etwa die Variablen StudMatr1 bis StudMatr12 als Elemente einer Folge ansehen und das  $i$ -te Folgenglied über seinen Index, also StudMatr <sub>$i$</sub> , ansprechen könnten. Ein solches Vorgehen setzt zwei sprachliche Möglichkeiten voraus:

- Zum einen, daß eine Folge gleichartiger Elemente durch einen einzigen Bezeichner gekennzeichnet wird.
- Zum anderen, daß die Folgenglieder über ihren "Index", d.h. ihre Position in der Folge angesprochen werden können.

In Pascal ist dies durch die Verwendung eines *Feldes* (array) möglich, wobei wir einen *Feldtyp* vorläufig folgendermaßen definieren:

array type



Der *Indextyp* (index type) beschreibt eine endliche Folge von Werten und muß ein Aufzählungs- oder Ausschnittstyp oder vom Typ char oder boolean sein. Der *Komponententyp* (component type) darf ein beliebiger, insbesondere auch ein strukturierter Typ sein, also z.B. wieder ein Feldtyp.

In dem Beispiel

Feld, array  
Feldtyp

Indextyp

Komponententyp

**type**

```
tStatus = (aktiv, passiv);
tIndex = 1..12;
tTeilnahmeArt = array [tIndex] of tStatus;
```

**var**

```
StudTeilnahme : tTeilnahmeArt;
```

wird die Variable `StudTeilnahme` als Feld mit 12 Komponenten deklariert. Der Indextyp ist der (selbstdefinierte) Ausschnittstyp `tIndex` und der Komponententyp der (selbstdefinierte) Aufzählungstyp `tStatus`.

indizierte Variable

Eine einzelne Komponente einer Feldvariablen wird als *indizierte Variable* (indexed variable) dadurch angesprochen, daß der Feldname angegeben wird und danach in eckigen Klammern ein Ausdruck vom Indextyp:

```
StudTeilnahme[3] := aktiv;
StudTeilnahme[3 * 4 - 2] := passiv
```

Mit der zusätzlichen Vereinbarung von Variablen (hier z.B. `i` und `j` vom Typ `tIndex`) kann auf eine Feldkomponente auch über einen Ausdruck in der Form

```
StudTeilnahme[i + 3 * j] := passiv
```

Bereichs-  
überschreitung

zugegriffen werden, wenn sichergestellt ist, daß der Wert des Ausdrucks `i+3*j` im Bereich zwischen 1 und 12 liegt. Liegt der Wert außerhalb, wird eine Komponente angesprochen, die nicht existiert. Wir bezeichnen diesen Fehler als *Bereichsüberschreitung*. Die meisten Pascal-Implementierungen erkennen solche unzulässigen Zugriffe während der Laufzeit und brechen die Ausführung des Programms ab.

Ein Feldtyp ermöglicht es, daß eine Anzahl von Variablen identischen Typs als eine einzige Feldvariable vereinbart und behandelt werden können. Insbesondere in Kombination mit der **for**-Anweisung sind Felder eine mächtige und kompakte Möglichkeit zur Bearbeitung großer Datenmengen. Beispielsweise können wir mit einer Anweisung allen Komponenten unserer Beispiel-Variablen den Wert `passiv` zuweisen:

```
for i := 1 to 12 do
  StudTeilnahme[i] := passiv
```

Feld  
initialisieren

Auf diese Weise kann - analog zur Zuweisung eines Anfangswerts an eine einzelne Variable - ein *Feld initialisiert* werden.



Beispiel 4.2.1.1

Eine Folge von fünf integer-Zahlen soll eingelesen und in umgekehrter Reihenfolge wieder ausgegeben werden. Das Programm legt die Zahlen in einem Feld ab, um sie anschließend rückwärts auszugeben.

```

program FolgenInverter (input, output);
{ liest eine Folge von 5 integer-Zahlen ein und
  gibt sie in umgekehrter Reihenfolge wieder aus }

  type
    tIndex = 1..5;
    tZahlenFeld = array [tIndex] of integer;

  var
    Feld : tZahlenFeld;
    i : tIndex;

begin
  for i := 1 to 5 do
    begin
      write ('Bitte ', i, '. Zahl eingeben: ');
      readln (Feld[i])
    end;
  for i := 5 downto 1 do
    writeln (Feld[i])
end. { FolgenInverter }

```

```

Bitte 1. Zahl eingeben: 11
Bitte 2. Zahl eingeben: 4711
Bitte 3. Zahl eingeben: 1612
Bitte 4. Zahl eingeben: 42
Bitte 5. Zahl eingeben: 93
    93
    42
  1612
  4711
    11

```



Wenn Sie versuchen, dieses Programm so abzuändern, daß es auch für andere Feldgrößen benutzt werden kann, werden Sie feststellen, daß nicht nur die Typdefinition, sondern auch die Anfangs- bzw. Endwerte der **for**-Schleifen angepaßt werden müssen. Günstiger ist es, die *Grenzen eines Feldes als Konstanten* zu definieren und diese in den Operationen, die auf dem Feld arbeiten, konsequent einzusetzen. Wir werden fortan so verfahren.

Konstanten als  
Feldgrenzen

Beispiel 4.2.1.2

Das folgende Programm liest eine Folge von integer-Zahlen ein, speichert sie in einem Feld, bestimmt die größte Zahl und gibt sie aus:

```
program FolgenMaximum (input, output);
{ liest eine Folge von 5 integer-Zahlen ein
  und bestimmt das Maximum }

const
  FELDGROESSE = 5;

type
  tIndex = 1..FELDGROESSE;
  tZahlenFeld = array [tIndex] of integer;

var
  Feld : tZahlenFeld;
  Maximum : integer;
  i : tIndex;

begin
  { Einlesen des Feldes }
  writeln ('Geben Sie ', FELDGROESSE:2, ' Werte ein:');
  for i := 1 to FELDGROESSE do
    readln (Feld[i]);
  { Bestimmen des Maximums }
  Maximum := Feld[1];
  for i := 2 to FELDGROESSE do
    if Feld[i] > Maximum then
      Maximum := Feld[i];
  { Ausgabe des Maximums }
  writeln ('Die groesste Zahl ist ', Maximum, '.')
end. { FolgenMaximum }
```

```
Geben Sie  5 Werte ein:
20
36
-6
9
-2
Die groesste Zahl ist 36.
```



Beachten Sie, daß die Variable `Maximum` sinnvollerweise mit dem Wert der ersten Feldkomponente initialisiert wird, bevor die übrigen Komponenten geprüft werden, ob sie einen größeren Wert enthalten.

#### Aufgabe 4.2.1.3

Schreiben Sie ein Programm, welches das Minimum in einem Feld von 20 `integer`-Zahlen bestimmt und sowohl das Minimum als auch seine Position im Feld ausgibt.



Obwohl sich die Programme durch Ändern von Konstanten an andere Feldgrößen anpassen lassen, ist dieses Vorgehen doch recht inflexibel, da die Konstanten geändert werden müssen und das Programm neu übersetzt werden muß. Es wäre daher wünschenswert, *Feldgrenzen dynamisch* festlegen zu können, also z.B. zunächst die Feldgröße einzulesen, um dann ein Feld entsprechender Größe zu deklarieren. In Pascal ist ein solches Vorgehen leider nicht möglich. Eine Variable kann nicht als Grenze in einem Indextyp auftreten.

dynamische  
Feldgrenzen

Die einzige Möglichkeit, ein Feld für unterschiedlich große Datenmengen zu deklarieren, besteht darin, ein genügend großes Feld anzulegen und dieses jeweils nur zu einem Teil zu nutzen. Dieses Verfahren werden wir in einigen der folgenden Beispiele anwenden.

#### Beispiel 4.2.1.4

Das folgende Programm liest eine Folge von bis zu 100 verschiedenen `integer`-Zahlen ein. Anschließend erfragt es einen Suchwert und bestimmt, ob er unter den eingegeben Zahlen vorkommt. Falls der Wert im Feld vorkommt, wird seine Position, ansonsten eine Fehlanzeige ausgegeben.

```
program FeldSuche (input, output);
{ stellt fest, ob ein Suchwert in einem Feld
  von bis zu 100 integer-Zahlen vorkommt }

const
  MAX = 100;
  MAXPLUS1 = 101;

type
  tIndex = 1..MAX;
  tFeld = array [tIndex] of integer;
  tIndexPlus1 = 1..MAXPLUS1;

var
  Feld : tFeld;
  Groesse,
  Suchwert : integer;
  i : tIndexPlus1;
  gefunden : boolean;
```

```

begin
  { Eingabe der Feldgroesse solange, bis eine
    Feldgroesse zwischen 1 und MAX eingegeben wird. }
  repeat
    write ('Anzahl Werte (max. ', MAX, '): ');
    readln (Groesse)
  until (1 <= Groesse) and (Groesse <= MAX);

  { Eingabe der Zahlen und des Suchwerts }
  for i := 1 to Groesse do
    begin
      write (i, '. Wert: ');
      readln (Feld[i])
    end;
  write ('Suchwert: ');
  readln (Suchwert);

  { Pruefen des gesamten Feldes bis zum Sucherfolg oder
    bis zum letzten eingelesenen Wert }
  gefunden := false;
  i := 1;
  while (i <= Groesse) and (not gefunden) do
    if Feld[i] = Suchwert then
      gefunden := true
    else
      i := i + 1;
  { Ende der while-Schleife }
  { Falls Groesse = MAX ist und Suchwert nicht gefunden
    wird, gilt nach Verlassen der while-Schleife
    i = MAXPLUS1. Daher obige Deklaration für i. }
  if gefunden then
    writeln ('Der Wert steht an Position ', i, '.')
  else
    writeln ('Der Wert kommt nicht vor.')
end. { FeldSuche }

```

```

Anzahl Werte (max. 100): 5
1. Wert: 20
2. Wert: 36
3. Wert: -6
4. Wert: 9
5. Wert: -2
Suchwert: 9
Der Wert steht an Position 4.

```

Aufgabe 4.2.1.5

Schreiben Sie ein Programm, das die Seminarverwaltung realisiert, die in der Einleitung zum Abschnitt 4.2 beschrieben ist.



Eine kompaktere Version der Deklaration von `Feld` in Beispiel 4.2.1.4 kann auch so erfolgen:

```
var
  Feld : array [1..100] of integer;
```

Der Typ `array [1..100] of integer` ist innerhalb einer Variablendeklaration definiert. Es liegt hier also eine *implizite Typdefinition* vor. In unserem Beispiel ist zusätzlich noch der Indextyp `1..100` implizit definiert. Wir wollen jedoch implizite Typdefinitionen vermeiden, da sie zu unübersichtlichen Programmen führen. Der Leser erwartet Typdefinitionen im Typdefinitionsbereich und nicht im Bereich von Variablendeklarationen.

Der Index ist häufig ein Ausschnittstyp von `integer`, wenngleich dies nicht notwendig so sein muß. So kann beispielsweise der *Aufzählungstyp*

```
type
  tFarbe = (rot, blau, gelb);
```

als Indextyp verwendet werden, etwa:

```
type
  tBlumenzahl = array [tFarbe] of integer;

var
  Blumenzahl : tBlumenzahl;
```

Das Feld `Blumenzahl` besteht aus drei `integer`-Komponenten, wobei für jeden der Index-Werte `rot`, `blau` und `gelb` eine Komponente existiert.

Ebenso kann der Typ `char`, ein Ausschnitt davon oder der Typ `boolean` als Indextyp dienen.

Beispiel 4.2.1.6

Wir wollen einen Text, genauer: einen Satz, daraufhin untersuchen, wie viele kleingeschriebene Vokale und Konsonanten er enthält. Wir legen dazu ein Feld an, das für jeden Kleinbuchstaben einen Zähler bereitstellt. Der Satz, der durch einen Punkt abgeschlossen ist, wird zeichenweise eingelesen. Falls das Zeichen ein Kleinbuchstabe ist, wird der entsprechende Zähler im Feld inkrementiert. Die Anzahl der Vokale wird bestimmt, indem die Summe der Zählerwerte für die Vokale gebildet wird, die Anzahl der Konsonanten ist die Differenz der Vokal-Anzahl zur Gesamtzahl der eingelesenen Kleinbuchstaben:

implizite  
Typdefinition

Aufzählungstyp als  
Indextyp

char oder boolean  
als Indextyp

```

program VokaleUndKonsonanten (input, output);
{ bestimmt in einem einzulesenden Satz die Anzahl der
  vorkommenden Vokale und Konsonanten }

type
tBuchst = 'a'..'z';
tNatZahl = 0..maxint;
tHaeufigkeit = array [tBuchst] of tNatZahl;

var
Anzahl : tHaeufigkeit;
Zeichen : char;
Gesamtzahl,
Vokalzahl : tNatZahl;

begin
  { Initialisieren der Zaehler }
  for Zeichen := 'a' to 'z' do
    Anzahl[Zeichen] := 0;
  Gesamtzahl := 0;

  { Zeichenweises Einlesen des Textes
    und Aufaddieren der Zaehler }
  read (Zeichen);
  while Zeichen <> '.' do
    begin
      if (Zeichen >= 'a') and (Zeichen <= 'z') then
        begin
          Anzahl[Zeichen] := Anzahl[Zeichen] + 1;
          Gesamtzahl := Gesamtzahl + 1
        end; { if }
      read (Zeichen)
    end; { while-Schleife }
  writeln;
  { Ausgabe der Statistik }
  Vokalzahl := Anzahl['a'] + Anzahl['e'] + Anzahl['i'] +
    Anzahl['o'] + Anzahl['u'];
  writeln ('Anzahl der Vokale: ', Vokalzahl, '.');
  write ('Anzahl der Konsonanten: ');
  writeln (Gesamtzahl - Vokalzahl, '.')
end. { VokaleUndKonsonanten }

```

*dies ist ein kurzer satz.*  
*Anzahl der Vokale: 8.*  
*Anzahl der Konsonanten: 12.*

#### Beispiel 4.2.1.7

Der Median einer Folge von Buchstaben ist derjenige Buchstabe, der die mittlere Position (bzw. die kleinere der beiden mittleren Positionen) belegt, wenn die vorkommenden Buchstaben alphabetisch sortiert angeordnet werden, wobei mehrfach vorkommende Buchstaben in der sortierten Anordnung nur einmal aufgeführt werden.

Es ist ein Programm zu entwickeln, das den Median einer Folge von einzulesenden Buchstaben bestimmt. Wir gehen davon aus, daß nur Kleinbuchstaben eingegeben werden. In einem mit den Kleinbuchstaben indizierten Feld wird das Auftreten des jeweiligen Buchstabens markiert. Gleichzeitig werden die verschiedenen auftretenden Buchstaben gezählt. Der Median wird bestimmt, indem das Feld von 'a' an durchlaufen wird, bis die Hälfte der markierten Buchstaben besucht ist. Aus Gründen der Einfachheit gehen wir davon aus, daß die Eingabefolge mindestens zwei verschiedene Zeichen enthält. Die Eingabe wird durch ein Zeichen beendet, das kein Kleinbuchstabe ist.

```
program Median (input, output);
{ bestimmt den Median einer einzugebenden Buchstaben-
  folge aus mindestens zwei verschiedenen Klein-
  buchstaben }

type
  tBuchst = 'a'..'z';
  tNatZahl = 0..maxint;
  tFeld = array [tBuchst] of boolean;

var
  vorhanden : tFeld;
  gesamt,
  i,
  MedianPos : tNatZahl;
  Buchstabe : tBuchst;
  Zeichen : char;

begin
  { Eingabe anfordern }
  writeln
    ('Geben Sie eine Kleinbuchstaben-Folge ein. ');
  writeln
    ('Eingabe-Ende durch ein anderes Zeichen. ');

  { Initialisierung }
  for Buchstabe := 'a' to 'z' do
    vorhanden[Buchstabe] := false;
  gesamt := 0;
```

```

{ Einlesen und Markierungen setzen }
read (Zeichen);
while (Zeichen >= 'a') and (Zeichen <= 'z') do
begin
  if not vorhanden[Zeichen] then
    begin
      vorhanden[Zeichen] := true;
      gesamt := gesamt + 1
    end;
  read (Zeichen)
end;
writeln;

{ Alle Buchstaben und Marken ausgeben }
for Buchstabe := 'a' to 'z' do
  write (Buchstabe);
writeln;
for Buchstabe := 'a' to 'z' do
  if vorhanden[Buchstabe] then
    write ('*')
  else
    write (' ');
writeln;

{ Median suchen }
MedianPos := (gesamt + 1) div 2;
Buchstabe := 'a';
i := 0;
repeat
  if vorhanden[Buchstabe] then
    i := i + 1;
  Buchstabe := succ (Buchstabe)
until i = MedianPos;
writeln ('Der Median ist: ', pred (Buchstabe), '.')
end. { Median }

```

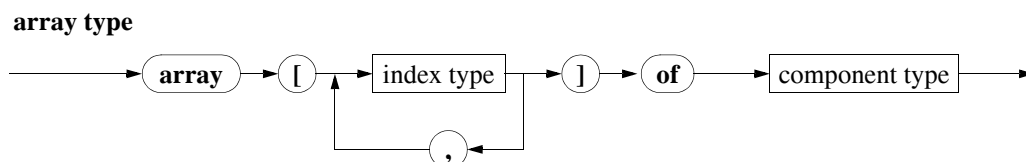
Geben Sie eine Kleinbuchstaben-Folge ein.  
Eingabe-Ende durch ein anderes Zeichen.  
diesisteinkurzersatz.

abcdefghijklmnopqrstuvwxyz  
\*   \*   \*   \*   \*   \*   \*   \*  
Der Median ist: n.



## Mehrdimensionale arrays

Bisher haben wir nur Felder mit einfacher Indizierung vereinbart. Sie werden als eindimensionale Felder bezeichnet. Pascal erlaubt aber auch die Verwendung mehrfacher Indizes bei der Vereinbarung von Feldern. Wir erweitern das Syntaxdiagramm des **array**-Typs entsprechend:



Eine gedruckte Seite, die z.B. aus 66 Zeilen mit je 120 Druckpositionen besteht, könnte durch die folgenden Definitionen und Deklarationen als Feldvariable vereinbart werden:

### type

```
tZeile = 1..66;
tSpalte = 1..120;
tSeite = array [tZeile, tSpalte] of char;
```

### var

```
Druckseite : tSeite;
```

Der *s*-te Buchstabe in der *z*-ten Zeile der Druckseite wird dann notiert als

```
Druckseite[z, s],
```

wobei *z* zwischen 1 und 66 und *s* zwischen 1 und 120 liegen muß. Die folgende Anweisung initialisiert jede der 7920 Druckpositionen mit einem Leerzeichen:

```
for z := 1 to 66 do
  for s := 1 to 120 do
    Druckseite[z, s] := ' ';
```

Das Feld Druckseite wird als *zweidimensionales Feld* bezeichnet. Zweidimensionale Felder werden häufig zur Darstellung von Daten verwendet, die nach dem mathematischen Konzept einer *Matrix* aufgebaut sind; eindimensionale Felder stellen in dieser Terminologie *Vektoren* dar.

### Beispiel 4.2.1.8

Das Programm liest eine  $3 \times 4$ -Matrix von integer-Zahlen in das zweidimensionale Feld A ein, legt die Zeilensumme im Vektor B und die Spaltensumme im Vektor C ab und gibt A, B und C aus.

```
program Matrix (input, output);
{ gibt die Zeilen- und Spaltensumme einer
  3x4 Matrix von integer-Zahlen aus }
```

zweidimensionales  
Feld  
Matrix  
Vektor

```
const
ZEILENMAX = 3;
SPALTENMAX = 4;

type
tZeile = 1..ZEILENMAX;
tSpalte = 1..SPALTENMAX;
tMatrix = array [tZeile, tSpalte] of integer;
tZeilensumme = array [tZeile] of integer;
tSpaltensumme = array [tSpalte] of integer;

var
A : tMatrix;
B : tZeilensumme;
C : tSpaltensumme;
i : tZeile;
j : tSpalte;

begin
  { Lesen der Matrixwerte in A }
  for i := 1 to ZEILENMAX do
    for j := 1 to SPALTENMAX do
      readln (A[i, j]);

  { Berechnen der Zeilensumme in B }
  for i := 1 to ZEILENMAX do
    begin
      B[i] := 0;
      for j := 1 to SPALTENMAX do
        B[i] := B[i] + A[i, j]
      end;

  { Berechnen der Spaltensumme in C }
  for j := 1 to SPALTENMAX do
    begin
      C[j] := 0;
      for i := 1 to ZEILENMAX do
        C[j] := C[j] + A[i, j]
      end;

  { Drucken von A, B und C in geeigneter Form }
  writeln;
  for i := 1 to ZEILENMAX do
    begin
      for j := 1 to SPALTENMAX do
        write (A[i, j]:5);
      writeln (B[i]:10)
    end;
```

```

writeln;
for j := 1 to SPALTENMAX do
    write (C[j]:5);
writeln
end. { Matrix }

```

```

1
2
3
4
5
6
7
8
9
10
11
12

      1      2      3      4      10
      5      6      7      8      26
      9     10     11     12     42

    15     18     21     24

```



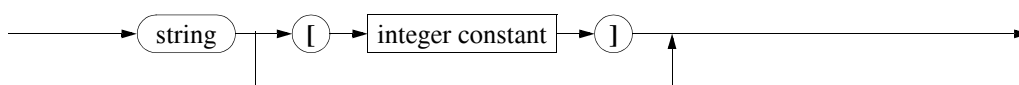
## Strings

In Kapitel 3 haben wir den Datentyp `string` als Aneinanderreihung von Zeichen vom Typ `char` vorgestellt. Mit unserem jetzigen Wissen wollen wir strings noch einmal genauer betrachten. Wir erinnern daran, daß der Typ `string` in Standard-Pascal nicht vordefiniert ist. Er wird aber von allen gängigen Pascal-Compilern bereitgestellt. Wir erklären hier die Turbo-Pascal-Variante.

Zeichenkette,  
`string`

Tatsächlich ist ein `string` ein **array** mit dem Komponententyp `char`. Wir können also mittels Indizierung auf einzelne Zeichen eines strings zugreifen. Der Indextyp eines strings ist immer ein Ausschnitt von `integer` beginnend mit der Zahl 1. Aus diesem Grund und da für Zeichenketten noch zusätzliche Operationen erlaubt sind, erfolgt die Definition einer Zeichenkette abweichend von der Definition für arrays in folgender Form:

**string type**



Die `integer`-Konstante gibt hier die maximale Länge des strings an. In den meisten Implementationen darf dieser Wert 255 nicht überschreiten. Die Deklaration

length

einer Variablen vom Typ `string` führt zur Anlage einer Zeichenkette mit maximaler Anzahl von Zeichen (z.B. 255).

Neben der Zuweisung von Konstanten an `string`-Variable sind auch Vergleichsoperationen definiert. Die *vordefinierte Funktion* `length` gibt die aktuelle Länge einer Zeichenkette, d.h. die Anzahl ihrer Zeichen zurück.

#### Beispiel 4.2.1.9

```
program StringTest (input, output);
  { Ein Beispiel fuer string-Operationen }

  const
    MAXLAENGE = 100;

  var
    Zeichenkette : string [MAXLAENGE];

begin
  Zeichenkette := 'Dies ist ein kurzer Satz!';
  writeln (Zeichenkette);
  writeln ('Das dritte Zeichen ist: ', Zeichenkette[3]);
  write ('Das letzte Zeichen ist: ');
  writeln (Zeichenkette[length (Zeichenkette)])
end. { StringTest }
```

```
Dies ist ein kurzer Satz!
Das dritte Zeichen ist: e
Das letzte Zeichen ist: !
```



### 4.2.2 Der Verbundtyp

Ein Feld besteht aus einer festgelegten Anzahl **typgleicher** Komponenten. Genau so oft benötigen wir Datenelemente, die aus Komponenten verschiedenen Typs bestehen.

Eine bestimmte Uhrzeit ist beispielsweise durch die Werte dreier Variablen gegeben, welche die unterschiedlichen "Komponenten" der Uhrzeit wiedergeben:

```
type
  tStunde = 0..23;
  tMinSek = 0..59;
```

**var**

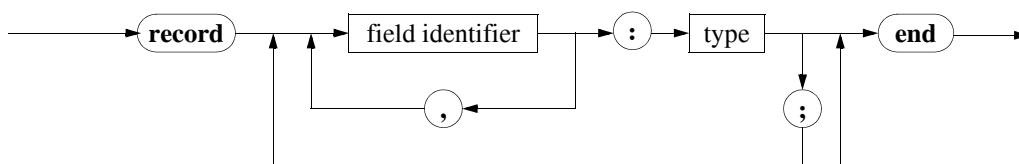
```
Stunde : tStunde;
Minute : tMinSek;
Sekunde : tMinSek;
```

Wir können nun die drei unterschiedlichen Variablen *Stunde*, *Minute* und *Sekunde* zu einer einzigen Variablen mit drei Komponenten zusammenfassen. Eine so zusammengesetzte Variable wird als Verbund (**record**) bezeichnet.

Die Definition eines *Verbundtyps* (record type) besteht aus den Schlüsselwörtern **record** und **end**, die eine Folge von Komponenten einschließen:

Verbundtyp  
Verbund, record

record type



Die Komponenten werden auch Felder (fields) eines Verbunds genannt. Um sie von arrays, die ebenfalls als Felder bezeichnet werden, zu unterscheiden, werden wir den Begriff Feld im Zusammenhang mit Verbunden nicht verwenden.

Eine Definition des Verbundtyps *tZeit* ist damit beispielsweise die folgende:

**type**

```
tStunde = 0..23;
tMinSek = 0..59;
```

*tZeit* = **record**

```
    h : tStunde;
    m,
    s : tMinSek
end; { tZeit }
```

Die *Bezeichner der Verbundkomponenten* müssen innerhalb der Definition eines Verbundtyps eindeutig sein, ihre Reihenfolge ist dabei unbedeutend. Als Komponenten-Typen sind beliebige Typen zulässig, also insbesondere wieder **record**-Typen.

Eindeutigkeit der  
Komponenten-  
Bezeichner

Ist ein Verbundtyp wie oben definiert, können Variable dieses Typs in der üblichen Weise deklariert werden:

**var**

```
Start,
Landung,
Flugdauer : tZeit;
```

record selector

Um auf eine Komponente eines Verbundes zuzugreifen, wird der Verbund-Name, gefolgt von einem Punkt und dem Komponenten-Bezeichner (*record selector*) angegeben:

```
Start.h := 12;
Start.m := 53;
Start.s := 0
```

selektierte Variable

Eine derart ausgewählte **record**-Komponente wird *selektierte Variable* genannt. Eine selektierte Variable kann wie eine gewöhnliche Variable verwendet werden.

Bei geschachtelten Verbunden erfolgt der Zugriff auf Komponenten von Komponenten analog (wir haben hier aus Platzgründen ausnahmsweise die Variablendeklaration und Typdefinition zusammengezogen). Nach der Deklaration

```
var
Termin : record
    Jahr : 1900..2100;
    Monat : 1..12;
    Tag : 1..31;
    Zeit : record
        h : 0..23;
        m : 0..59;
        s : 0..59
    end { Zeit }
end; { Termin }
```

können im Rumpf z.B. die folgenden Zuweisungen auftreten:

```
Termin.Jahr := 1993;
Termin.Monat := 10;
Termin.Tag := 1;
Termin.Zeit.h := 12;
Termin.Zeit.m := 53;
Termin.Zeit.s := 0
```

#### Beispiel 4.2.2.1

Zur Berechnung einer Flugdauer sollen die Start- und die Landezeit in Stunden, Minuten und Sekunden eingelesen werden, das Ergebnis Flugdauer soll im gleichen Format ausgegeben werden. Die Flugdauer liegt deutlich unter 24 Stunden, allerdings sind Nachtflüge möglich.

```
program Flugzeit (input, output);
{ berechnet die Flugdauer aus Start- und Landezeit }
```

```
const
MINSEK = 59;
STUNDE = 23;
```

```
type
tMinSek = 0..MINSEK;
tStunde = 0..STUNDE;

tZeit = record
    h : tStunde;
    m,
    s : tMinSek
end; { tZeit }
tUebertrag = 0..1;

var
Abflug,
Ankunft,
Flugdauer : tZeit;
Eingabe : integer;
Uebertrag : tUebertrag;

begin
    Flugdauer.h := 0;
    Flugdauer.m := 0;
    Flugdauer.s := 0;
    Uebertrag := 0;

    writeln ('Bitte die Abflugzeit eingeben');
    repeat
        write ('Stunde: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= STUNDE);
    Abflug.h := Eingabe;

    repeat
        write ('Minute: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= MINSEK);
    Abflug.m := Eingabe;

    repeat
        write ('Sekunde: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= MINSEK);
    Abflug.s := Eingabe;

    writeln;
    writeln ('Bitte die Ankunftszeit eingeben');
    repeat
        write ('Stunde: ');
        readln (Eingabe)
```

```
until (Eingabe >= 0) and (Eingabe <= STUNDE);
Ankunft.h := Eingabe;

repeat
  write ('Minute: ');
  readln (Eingabe)
until (Eingabe >= 0) and (Eingabe <= MINSEK);
Ankunft.m := Eingabe;

repeat
  write ('Sekunde: ');
  readln (Eingabe)
until (Eingabe >= 0) and (Eingabe <= MINSEK);
Ankunft.s := Eingabe;

{ Sekunden-Differenz berechnen }
if Ankunft.s < Abflug.s then
begin
  Uebertrag := 1;
  Flugdauer.s := MINSEK + 1 - Abflug.s + Ankunft.s
end
else
  Flugdauer.s := Ankunft.s - Abflug.s;

{ Minuten-Differenz berechnen }
if (Ankunft.m - Uebertrag) < Abflug.m then
begin
  Flugdauer.m := MINSEK + 1 - Abflug.m +
    Ankunft.m - Uebertrag;
  Uebertrag := 1
end
else
begin
  Flugdauer.m := Ankunft.m - Abflug.m - Uebertrag;
  Uebertrag := 0
end;

{ Stunden-Differenz berechnen }
if (Ankunft.h - Uebertrag) < Abflug.h then
  Flugdauer.h := STUNDE + 1 - Abflug.h +
    Ankunft.h - Uebertrag
else
  Flugdauer.h := Ankunft.h - Abflug.h - Uebertrag;

{ Ausgabe }
writeln;
writeln ('Flugdauer: ', Flugdauer.h, ':',
```



```

        Flugdauer.m, ': ', Flugdauer.s)
end. { Flugzeit }

```

```

Bitte die Abflugzeit eingeben
Stunde: 22
Minute: 53
Sekunde: 12

Bitte die Ankunftszeit eingeben
Stunde: 5
Minute: 12
Sekunde: 54

Flugdauer: 6:19:42

```



Ein *Verbundtyp* kann natürlich auch *Komponententyp eines Feldes* sein. Mit Hilfe dieser Konstruktion können wir unser Programm zur Seminarverwaltung (Aufgabe 4.2.1.5) komfortabler gestalten.

Verbundtyp als  
Komponententyp  
eines Feldes

#### Beispiel 4.2.2.2

Auf einem Seminarschein soll nicht nur die Matrikel-Nummer, sondern auch der Name des Teilnehmers ausgedruckt werden. Eine Erweiterung um die Druckausgabe des Geburtsdatums und der Adresse des Teilnehmers ist bereits vorzusehen, aber noch nicht zu benutzen. Nachdem wir den Verbundtyp kennengelernt haben, können wir nun alle Daten eines Studenten in einem Verbund zusammenzufassen, anstatt sie über mehrere Felder zu verteilen.

```

program Seminar2 (input, output);
{ zweite Programmvariante für das Seminarproblem }

```

#### **const**

```

MAXTEILNEHMER = 12;
TAGE = 31;
MONATE = 12;
MINJAHR = 1900;
MAXJAHR = 2010;

```

#### **type**

```

tTag = 1..TAGE;
tMonat = 1..MONATE;
tJahr = MINJAHR..MAXJAHR;
tNatZahlPlus = 1..maxint;
tNatZahl = 0..maxint;
tStatus = (aktiv, passiv);

```

```

tIndex = 1..MAXTEILNEHMER;
tString = string [20];
tSeminarStudent =
    record
        Name : tString;
        { Erweiterung: }
        Geburtstag :
            record
                Tag : tTag;
                Monat : tMonat;
                Jahr : tJahr
            end; { Geburtstag }
        { Ende der Erweiterung }
        MatrikelNr : tNatZahlPlus;
        Status : tStatus
    end; { tSeminarStudent }
tTeilnehmerfeld = array [tIndex] of tSeminarStudent;

var
TeilnehmerFeld : tTeilnehmerfeld;
AnzStud : tNatZahl;
i : tIndex;
Status : char; { Zeichen zum Einlesen des Studenten-
                status. Muss vom Typ char sein, um
                eingelesen werden zu koennen.
                'a' entspricht aktiv,
                'p' entspricht passiv }

begin
write ('Wie viele Studenten nahmen am Seminar teil? ');
readln (AnzStud);
if AnzStud > MAXTEILNEHMER then
begin
    writeln ('Bitte hoechstens ', MAXTEILNEHMER,
            ' Eingaben!');
    AnzStud := MAXTEILNEHMER
end;
writeln ('Geben Sie Name, Matr.Nr. und Aktivitaet ',
        'der', AnzStud:3, ' Teilnehmer ein:');

for i := 1 to AnzStud do
begin
    write ('Name: ');
    readln (TeilnehmerFeld[i].Name);
    write ('Matr.Nr. ');
    readln (TeilnehmerFeld[i].MatrikelNr);
    write ('a - aktiv, p - passiv: ');
    readln (Status);

```

```
    if Status = 'a' then
        TeilnehmerFeld[i].Status := aktiv
    else
        TeilnehmerFeld[i].Status := passiv
    end;

{ Scheine ausgeben }
writeln;
for i := 1 to AnzStud do
begin
    if TeilnehmerFeld[i].Status = aktiv then
    begin
        writeln ('Der Student ', TeilnehmerFeld[i].Name);
        write ('mit der Matrikel-Nr. ');
        writeln (TeilnehmerFeld[i].MatrikelNr);
        writeln ('hat mit Erfolg am Seminar ',
                'teilgenommen. ');
        writeln
    end
end;

writeln ('Liste aller aktiven Seminar-Teilnehmer');
writeln ('-----');
for i := 1 to AnzStud do
    if TeilnehmerFeld[i].Status = aktiv then
        writeln (TeilnehmerFeld[i].Name);
writeln;

writeln ('Liste aller Zuhörer im Seminar');
writeln ('-----');
for i := 1 to AnzStud do
    if TeilnehmerFeld[i].Status = passiv then
        writeln (TeilnehmerFeld[i].Name);
writeln
end. { Seminar2 }
```

```

Wie viele Studenten nahmen am Seminar teil? 3
Geben Sie Name, Matr.Nr. und Aktivitaet der 3 Teilneh-
mer ein:
Name: Hans Meyer
Matr.Nr. 11
a - aktiv, p - passiv: a
Name: Klaus Mueller
Matr.Nr. 10
a - aktiv, p - passiv: p
Name: Paul Schulz
Matr.Nr. 1
a - aktiv, p - passiv: a

Der Student Hans Meyer
mit der Matrikel-Nr. 11
hat mit Erfolg am Seminar teilgenommen.

Der Student Paul Schulz
mit der Matrikel-Nr. 1
hat mit Erfolg am Seminar teilgenommen.

Liste aller aktiven Seminar-Teilnehmer
-----
Hans Meyer
Paul Schulz

Liste aller Zuhoerer im Seminar
-----
Klaus Mueller

```



### 4.3 Funktionen

Wir haben bereits einige Standardfunktionen in Pascal kennengelernt, z.B. `sqrt`, `succ` oder `ord`. Im Ausdruck

```
y / sqrt (x) + 3.0
```

Funktionsaufruf

aktueller Parameter

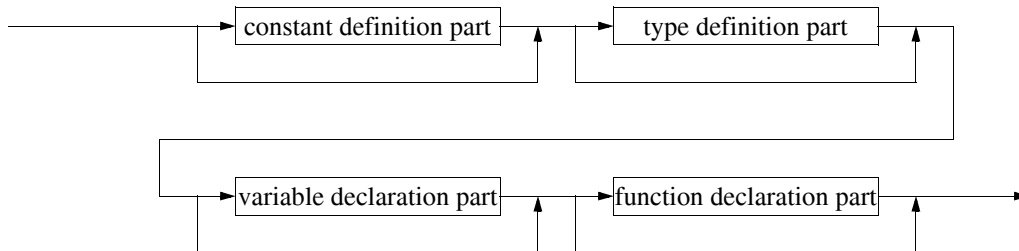
ist `sqrt (x)` ein *Funktionsaufruf*, welcher die Quadratwurzel von `x` berechnet. Zunächst wird der Wert des Funktionsaufrufs bestimmt (Wurzel von `x`) und dann der Ausdruck berechnet. Das Argument `x` bezeichnen wir als *aktuellen Parameter* des Funktionsaufrufs.

Pascal stellt nicht nur Standardfunktionen bereit, sondern bietet auch die Möglichkeit, daß der Programmierer Funktionen selbst vereinbart. In der Funktionsverein-

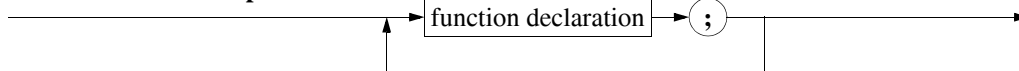
barung wird festgelegt, wie die Funktion heißt, wie sie aufgerufen wird und wie sie ihren Funktionswert berechnet. *Funktionsdeklarationen* erfolgen in Pascal im Anschluß an Konstanten-, Typen-, und Variablenvereinbarungen:

Funktions-  
deklaration

#### declaration part



#### function declaration part

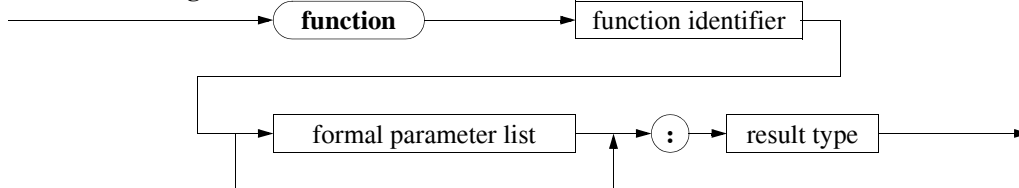


Die Deklaration einer Funktion hat die folgende ausführliche Form:

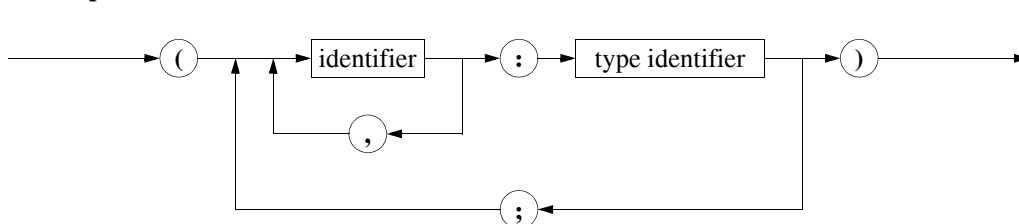
#### function declaration



#### function heading



#### formal parameter list



Den ersten Teil einer Funktionsdeklaration (function declaration) bildet der *Funktionskopf* (function heading), der beschreibt, wie die Funktion heißt und wie sie aufzurufen ist. Dabei folgt nach dem Schlüsselwort **function** der *Funktionsname* (function identifier) und in Klammern die Liste der *formalen Parameter* (formal parameter list). Der *Ergebnistyp* (result type) schließlich gibt an, von welchem Typ der Funktionswert ist.

Funktionskopf  
Funktionsname  
formale Parameter  
Ergebnistyp

Der Funktionsname wird nach den bekannten Regeln für Bezeichner gebildet. Eine selbstvereinbarte Funktion kann mehrere oder auch keinen Parameter haben. Typen werden durch Typpnamen festgelegt, d.h. Typen müssen vorher definiert sein, es sei denn, es handelt sich um Standard-Datentypen. Für den Ergebnistyp gilt die zusätzliche Einschränkung, daß er kein strukturierter Typ sein darf. Es gehört zum guten Programmierstil, in einem Kommentar nach dem Funktionskopf zu erläutern, was die Funktion leistet.

Beispiele für kommentierte Funktionsköpfe sind:

```
function istPositiv (i : integer) : boolean;
{ true, falls i > 0, und false, falls i <= 0 }

function max (x, y : real) : real;
{ bestimmt den groesseren Wert der Parameter x und y }

function Zufallszahl : real;
{ liefert eine Zufallszahl zwischen 0.0 und 1.0 }
```

Block  
Funktionsrumpf

Den zweiten Teil einer Funktionsdeklaration bildet ein *Block*, in dem aus den Parametern der Funktionswert berechnet wird. Der Block wird auch als *Funktionsrumpf* bezeichnet.

Deklarationsteil

lokale  
Vereinbarungen  
Gültigkeitsbereich

Wir wissen bereits, daß ein Block aus einem Deklarationsteil und einem Anweisungsteil besteht. Das zugehörige Syntaxdiagramm haben wir in Kapitel 3 kennengelernt. Der *Deklarationsteil* eines Funktionsrumpfs kann also Vereinbarungen von Konstanten, Typen, Variablen, Funktionen und Prozeduren enthalten. Diese Vereinbarungen bezeichnen wir als *lokal*, da sie nur innerhalb dieses Blocks gültig sind, d.h. außerhalb des Blockes sind sie unbekannt und können nicht angesprochen werden. Wir sprechen auch von dem *Gültigkeitsbereich* der vereinbarten Objekte und meinen damit den Block, in dem sie vereinbart sind.

### Beispiel 4.3.1

```
program Beispiel (input, output);
{ dient nur zu Demonstrationszwecken }
```

```
var
x,
y : integer;
```

```
function F (i : integer) : integer;
{ F quadriert i }
```

```
var
a : integer;
```

```
begin
  a := i * i;
```

```

    F := a
  end; { F }

begin
  ...
  writeln (a); { <-- Fehler: a ist ausserhalb von F
                unbekannt, da a lokal zu F ist ! }
end. { Beispiel }

```



Bei jeder Ausführung des *Anweisungsteils* einer Funktion muß dem Funktionsnamen ein Wert vom Ergebnistyp zugewiesen werden. Erfolgt keine solche Zuweisung, ist das Ergebnis des Funktionsaufrufs nicht definiert. Ansonsten wird derjenige Wert, der zuletzt dem Funktionsnamen zugewiesen wurde, als Ergebnis des Funktionsaufrufs zurückgeliefert.

Anweisungsteil  
einer Funktion

Die folgende Funktion `max` liefert als Ergebnis den größeren Wert der Parameter `x` und `y`.

```

function max (x, y : real): real;
{ bestimmt die groessere der Zahlen x und y }

begin
  if x > y then
    max := x
  else
    max := y
  end; { max }

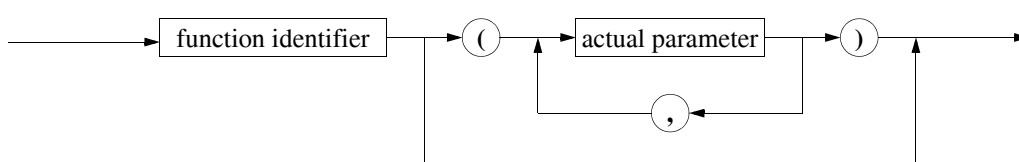
```

Im Funktionskopf sind der Bezeichner `max`, unter dem die Funktion angesprochen wird, die formalen Parameter `x` und `y` vom Typ `real` sowie der Ergebnistyp `real` spezifiziert. In dem Funktionsblock wird der größere der zwei Parameter bestimmt und dem Funktionsnamen zugewiesen. Der Deklarationsteil des Funktionsblockes ist leer.

Eine Funktion wird durch Angabe des Funktionsnamens und der Liste der aktuellen Parameter aufgerufen, so wie wir es bereits beim Aufruf von Standardfunktionen kennengelernt haben. Ein *Funktionsaufruf* ist ein Ausdruck und kann daher überall verwendet werden, wo Ausdrücke erlaubt sind. Also z.B. innerhalb eines (umfassenden) Ausdrucks oder auf der rechten Seite einer Zuweisung. Ein Funktionsaufruf hat die Form:

Funktionsaufruf

**function designator**



## Regeln für aktuelle Parameter

Ein aktueller Parameter in einem Funktionsaufruf muß ein Ausdruck sein. Die Liste der aktuellen Parameter in einem Funktionsaufruf muß mit der Liste der formalen Parameter nach folgenden Regeln übereinstimmen:

- Die Zahl der Parameter in beiden Listen muß gleich sein.
- Die Typen des formalen und entsprechenden aktuellen Parameters, der an derselben Position der Parameterliste steht, müssen identisch sein.

Der Aufruf einer Funktion bewirkt, daß die Werte der aktuellen Parameter den entsprechenden formalen Parametern zugewiesen werden. Ist ein aktueller Parameter ein zusammengesetzter Ausdruck, so wird er zunächst ausgewertet und der Wert dann dem formalen Parameter zugewiesen. Anschließend wird der Anweisungsteil der zugehörigen Funktionsdeklaration ausgeführt und damit der Funktionswert (für die aktuellen Parameter) berechnet.

Wenn  $t, u, v$  und  $w$  Variable vom Typ `real` sind, so weist

```
u := max (v, w)
```

$u$  den größeren der Werte der aktuellen Parameter  $v$  und  $w$  zu. Die Anweisung

```
u := 2.0 * max (v, 1.5)
```

weist  $u$  den größeren der Werte der aktuellen Parameter  $v$  und  $1.5$  multipliziert mit  $2.0$  zu.

```
u := max (u, max (v, w))
```

weist  $u$  den größten der Werte der aktuellen Parameter  $u$  und  $\max (v, w)$  zu. Die Auswertung des aktuellen Parameters  $\max (v, w)$  ergibt den größeren der Werte der aktuellen Parameter  $v$  und  $w$ . Insgesamt wird also  $u$  der größte der Werte von  $u, v$  und  $w$  zugewiesen. Das letzte Beispiel ist also gleichwertig mit:

```
t := max (v, w);
u := max (u, t)
```

Wir geben ein weiteres Beispiel einer Funktionsvereinbarung an. Die Existenz der Typdefinition

```
type
tNatZahl = 0..maxint;
```

wird vorausgesetzt.

```
function Potenz (
    x : real;
    n : tNatZahl): real;
{ berechnet die n-te Potenz von x fuer n >= 0 }
```



```

var
  i : tNatZahl;
  Rechnung : real;

begin
  Rechnung := 1.0;
  for i := 1 to n do
    Rechnung := Rechnung * x;
  Potenz := Rechnung
end; { Potenz }

```

Wir betonen, daß dem Funktionsbezeichner zwar mehrmals ein Wert zugewiesen werden kann, aber auf diesen Wert innerhalb der Funktion nicht lesend zugegriffen werden darf. Er kann also z.B. nicht als lokale Variable verwendet werden. Der Grund besteht darin, daß jedes Auftreten des Funktionsbezeichners als erneuter Funktionsaufruf und nicht als eine Art lokale Variable verstanden wird. Aus diesem Grund wird in obigem Beispiel die lokale Hilfsvariable `Rechnung` zur Berechnung der Potenz benutzt.

Mögliche Funktionsaufrufe der Funktion `Potenz` sind:

```

y := Potenz (6.3, 5);
if Potenz (2.0, i) > 1000.0 then
  writeln ('Grenzwert ueberschritten.')

```

Neben dem Gültigkeitsbereich von Variablen ist auch deren *Lebensdauer*, d.h. deren *Existenz*, von Wichtigkeit. In Pascal ist festgelegt, daß lokale Variablen nur während der Ausführung einer Funktion (bzw. Prozedur) existieren. Sie werden bei dem Anfang der Funktion bereitgestellt und bei deren Beendigung zerstört. Daraus folgt, daß es insbesondere keine Beziehung zwischen den Werten einer lokalen Variablen bei aufeinanderfolgenden Aufrufen einer Funktion gibt. Bei jedem Beginn der Ausführung werden die lokalen Variablen bereitgestellt, bleiben solange undefiniert, bis ihnen ein Wert zugewiesen wird, und stehen nach Ablauf der Funktion nicht mehr zur Verfügung.

Lebensdauer  
Existenz

Wir werden Fragen, die mit dem Gültigkeitsbereich und der Lebensdauer von Vereinbarungen und damit mit der Blockstruktur von Programmen zusammenhängen, in Kapitel 5 ausführlicher diskutieren.

Aus dem Kapitel 3 wissen wir bereits, daß es einige Standardfunktionen gibt, die Argumente unterschiedlicher Typen akzeptieren und abhängig davon auch unterschiedliche Werttypen haben können. Dies gilt u.a. für `sqr`, `abs`, `pred`, `succ` und `ord`. Diese Möglichkeit gibt es bei selbstdeklarierten Funktionen nicht: Für jeden Parameter einer selbstdeklarierten Funktion sowie für das Ergebnis kann nur jeweils genau ein Typ verwendet werden. Wie für alle Bezeichner gilt auch für Funktionsbezeichner, daß sie eindeutig sein müssen.

Neben einer besseren Übersichtlichkeit und Strukturierung liegt ein Hauptvorteil von Funktionen in ihrer Mehrfachverwendbarkeit, die zum Tragen kommt, wenn dieselbe Berechnungsvorschrift an verschiedenen Stellen im Programm (mit verschiedenen Werten für die aktuellen Parameter) durchzuführen ist. Dies demonstriert das folgende Programm.

#### Beispiel 4.3.2

Wir wollen ein Programm schreiben, das eine positive integer-Zahl einliest, für welche die nächstgelegene (größere oder kleinere) Primzahl gesucht wird. Sollten im gleichen Abstand zwei Primzahlen existieren, also etwa die Primzahlen 7 und 11 bei Eingabe der Zahl 9, so sollen beide ausgegeben werden. Ist die eingelesene Zahl bereits eine Primzahl, so wird die Zahl selbst ausgegeben.

Zunächst einmal überlegen wir uns einen (einfachen) Algorithmus, mit dem wir die zur Eingabezahl nächste(n) Primzahl(en) bestimmen. Wir wissen, daß eine Primzahl mit Ausnahme der 2 stets ungerade ist. Also prüfen wir (nach oben und nach unten) mit zunehmendem Abstand von der Eingabezahl alle ungeraden Zahlen, bis wir eine Primzahl gefunden haben. Unter der Annahme, daß wir bereits über eine Funktion `istPrimzahl (p)` verfügen, die den Wert `true` liefert, falls `p` Primzahl ist, und `false` sonst, erhalten wir folgendes Programmfragment (noch ohne die Deklaration von `istPrimzahl`):

```
program FindeNaechstePrimzahl (input, output);
{ bestimmt die zur Eingabezahl naechstgelegene(n)
  Primzahl(en) }

type
tNatZahlPlus = 1..maxint;

var
EinZahl,
d : tNatZahlPlus; { d ist die Schrittweite,
                  "displacement" }
gefunden : boolean;

function istPrimzahl (p : tNatZahlPlus) : boolean;
{ liefert true, falls p Primzahl ist, sonst false }
.
.
.

begin
.
.
.
end; { istPrimzahl }
```

```

begin
  writeln ('Zahl eingeben: ');
  readln (EinZahl);
  { Um das Programm zu vereinfachen, verzichten wir
    auf eine Ueberpruefung der Eingabe }
  write ('Naechste Primzahl zu ', EinZahl, ' ist ');
  if istPrimzahl (EinZahl) then
    writeln (EinZahl)
  else
    if EinZahl = 1 then
      writeln ('2':5)
    else {EinZahl <> 1 }
      begin
        gefunden := false;
        if odd (EinZahl) then
          d := 2
        else
          d := 1;
        repeat
          if istPrimzahl (EinZahl + d) then
            begin
              { Primzahl oberhalb von EinZahl gefunden }
              gefunden := true;
              write (EinZahl + d : 5)
            end;
          if istPrimzahl (EinZahl - d) then
            begin
              { Primzahl unterhalb von EinZahl gefunden }
              gefunden := true;
              write (EinZahl - d : 5)
            end;
          d := d + 2
        until gefunden;
        writeln
      end {EinZahl <> 1 }
    end. { FindeNaechstePrimzahl }

```

Wir erkennen, daß der Primzahltest an drei verschiedenen Stellen des Programms durchzuführen ist, und wenden uns nun (mit der nötigen Motivation) der Programmierung der Funktion `istPrimzahl` zu.

Per Definition ist eine positive ganze Zahl  $p$  genau dann eine Primzahl, wenn sie größer als 1 ist und es keine ganze Zahl  $q$  mit  $2 \leq q < p$  gibt, durch die  $p$  teilbar ist. Wir prüfen also für jede Zahl  $q = 2, 3, \dots, p - 1$ , ob  $p$  durch  $q$  geteilt werden kann. Dabei benutzen wir, daß eine integer-Zahl  $p$  genau dann durch eine integer-Zahl  $q$  teilbar ist, wenn  $p \bmod q = 0$  gilt.

```

function istPrimzahl (p : tNatZahlPlus) : boolean;
{ liefert true, falls p Primzahl, sonst false}

    var
        q : tNatZahlPlus;

begin
    if p < 2 then
        istPrimzahl := false
    else { p >= 2 }
        begin
            istPrimzahl := true;
            for q := 2 to p - 1 do
                if p mod q = 0 then
                    istPrimzahl := false
            end
        end;
{ istPrimzahl }

```

Fügen wir diese Funktionsdeklaration an der dafür vorgesehenen Stelle in das Programm ein, ist unser Programm lauffähig.



### Aufgabe 4.3.3

Die Funktion `istPrimzahl` läßt sich in mehrfacher Hinsicht verbessern. So brauchen wir z.B. die Zahl  $q$  nicht bis  $p - 1$  heraufzuzählen, da für  $q > \sqrt{p}$  keine Teilbarkeit von  $p$  mehr möglich ist. Wir wollen aber nicht diese mathematische Eigenschaft ausnutzen, sondern uns auf die Schleifenkonstruktion an sich konzentrieren. So können wir doch den Teilbarkeitstest abbrechen, sobald das erste Mal  $p$  von  $q$  geteilt wird. Da die Anzahl der Schleifendurchläufe im Vorhinein nicht bekannt ist, eignet sich hierfür die **for**-Schleife offensichtlich nicht, denn sie erzwingt stets die maximale Anzahl von  $p - 1$  Schleifendurchläufen.

Entwickeln Sie die Funktion `istPrimzahl` neu unter Verwendung einer geeigneteren, d.h. effizienteren, Schleife.



Das folgende Beispiel demonstriert die Verwendung eines strukturierten Parametertyps.

### Beispiel 4.3.4

Es ist ein Feld von `integer`-Zahlen aufsteigend zu sortieren. Dazu wird zunächst unter allen Feldelementen das Minimum zusammen mit seiner Feldposition bestimmt und mit dem ersten Feldelement vertauscht. Ist das erste Feldelement bereits das Minimum, so wird es (aus Gründen der programmtechnischen Einfachheit) mit sich selbst vertauscht. Mit den verbleibenden Zahlen, beginnend mit der zweiten Position, wird nun genauso verfahren, d.h. das Minimum bestimmt und mit dem

Element auf der zweiten Position vertauscht. Danach konzentrieren wir uns auf die Zahlen beginnend mit der dritten Position usw. Das Sortierverfahren wird das letzte Mal auf die beiden letzten Zahlen angewendet, dann bricht es ab.

Das Programm verwendet eine Funktion, die unter den Zahlen eines Feldes, die sich zwischen den Positionen von und bis (Grenzpositionen inklusive) befinden, die Position des Minimums bestimmt.

```
program FeldSort (input, output);
{ sortiert ein einzulesendes Feld von integer-Zahlen }

const
  FELDGROESSE = 5;

type
  tIndex = 1..FELDGROESSE;
  tFeld = array [tIndex] of integer;

var
  EingabeFeld : tFeld;
  MinPos,
  i : tIndex;
  Tausch : integer;

function FeldMinimumPos (Feld : tFeld;
                        von, bis: tIndex): tIndex;
{ bestimmt die Position des Minimums im Feld zwischen
  von und bis, 1 <= von <= bis <= FELDGROESSE }

  var
    MinimumPos,
    j : tIndex;

begin
  MinimumPos := von;
  for j:= von + 1 to bis do
    if Feld[j] < Feld[MinimumPos] then
      MinimumPos := j;
  FeldMinimumPos := MinimumPos
end; { FeldMinimumPos }

begin
  { Einlesen des Feldes }
  writeln ('Geben Sie ', FELDGROESSE, ' Werte ein: ');
  for i := 1 to FELDGROESSE do
    readln (EingabeFeld[i]);

  { Sortieren }
```

```

for i := 1 to FELDGROESSE - 1 do
begin
    MinPos := FeldMinimumPos (EingabeFeld, i,
                               FELDGROESSE);
    { Minimum gefunden, jetzt muessen wir es mit dem
      Element auf Position i vertauschen }
    Tausch := EingabeFeld[MinPos];
    EingabeFeld[MinPos] := EingabeFeld[i];
    EingabeFeld[i] := Tausch
end;

    { Ausgabe des sortierten Feldes }
for i := 1 to FELDGROESSE do
    write (EingabeFeld[i]:6);
    writeln
end. { FeldSort }

```

*Geben Sie 5 Werte ein:*

4  
7  
3  
1  
8

1          3          4          7          8



#### 4.4 Programmierstil (Teil 2)

##### Hinweise

1. Die in dem jeweiligen Kapitel neu hinzukommenden Programmierstilhinweise und -regeln sind kursiv gedruckt und am Rand markiert, damit sie leichter gefunden werden können.
2. Die nicht fortlaufende Numerierung der Muß-Regeln ist beabsichtigt. Fehlende Regeln werden in den nächsten Kapiteln ergänzt, so daß in dem abschließenden Dokument die Muß-Regeln durchnummeriert vorliegen.

Programme können, nachdem sie implementiert und getestet worden sind, in den seltensten Fällen ohne Änderung über einen längeren Zeitraum hinweg eingesetzt werden. Tatsächlich ist es meist so, daß die Anforderungen nach der Fertigstellung verändert oder erweitert werden und während des Betriebs bislang unerkannte Män-

gel oder Fehler auftreten, die zu beseitigen sind. Programme müssen während ihres Einsatzes *gewartet* werden. Zu Wartungszwecken muß der Programmtext immer wieder gelesen und verstanden werden. Die Lesbarkeit eines Programms spielt also eine große Rolle. Je größer ein Programm ist, desto wichtiger wird das Kriterium der Lesbarkeit. Dies gilt besonders für industrielle Anwendungen, bei denen häufig der bzw. die Entwickler nicht mehr verfügbar ist bzw. sind und Änderungen von Dritten vorgenommen werden müssen.

Die Lesbarkeit eines Programms hängt einerseits von der verwendeten Programmiersprache und andererseits vom *Programmierstil* ab. Der Programmierstil beeinflusst die Lesbarkeit eines Programms mehr als die verwendete Programmiersprache. Ein stilistisch gut geschriebenes C- oder COBOL-Programm kann besser lesbar sein als ein schlecht geschriebenes Pascal-Programm.

Wir bemühen uns bei unseren Beispielen um einen guten Programmierstil. Zum einen wollen wir Ihnen einen guten Programmierstil „von klein auf“ implizit vermitteln, d.h. Sie sollen nur gute und gar nicht erst schlechte Beispiele kennenlernen. Zum anderen bezwecken wir durch die erhöhte Lesbarkeit auch ein leichteres Verständnis der Konzepte, die wir durch Programme bzw. Prozeduren und Funktionen vermitteln wollen.

Programmierstil ist, wie der Name vermuten läßt, in gewissem Umfang Geschmackssache. Auch können starre Richtlinien in Einzelfällen unnatürlich wirken. Dennoch sind die Erfahrungen mit der flexiblen Handhabung solcher Richtlinien durchweg schlecht, so daß gut geführte Programmierabteilungen und Softwarehäuser einen hauseigenen Programmierstil verbindlich vorschreiben und die Einhaltung - häufig über Softwarewerkzeuge - rigoros kontrollieren.

Genau das werden wir auch in diesem Kurs tun (dies erleichtert auch die Korrektur der über tausend Einsendungen je Kurseinheit) und Regeln vorschreiben, deren Nichteinhaltung zu Punktabzügen führt. Dabei unterscheiden wir zwischen „Muß-Regeln“, die in jedem Fall befolgt werden müssen, und „Kann-Regeln“, von denen im Einzelfall abgewichen werden kann. Muß-Regeln sind durch Fettdruck und entsprechende Marginalien gekennzeichnet. **Das Nichteinhalten von Muß-Regeln wirkt sich in Punktabzügen aus.**

Wir haben uns in den Programmbeispielen nicht immer streng an die Kann-Regeln gehalten. Das hängt damit zusammen, daß die Empfehlungen zum Teil bereits für den professionellen Einsatz gedacht sind und unsere kleinen Beispiele manchmal überfrachtet hätten. So können z.B. gewisse Layout-Regeln (etwa für den Vereinbarungsteil), die bei größeren Programmen sinnvoll sind, kleinere Programme unnötig aufblähen. Außerdem nehmen wir aus Gründen der Lesbarkeit auf den Seitenumbruch Rücksicht, was sich nicht immer mit den Regeln verträgt.

Die Ausführungen zum Programmierstil werden von Kapitel zu Kapitel aktualisiert und ergänzt. Dabei werden die Regeln "kumuliert", d.h. die Programmierstilregeln von Kapitel 5 schließen die Regeln für Kapitel 3 und Kapitel 4 mit ein. Am Ende von Kapitel 6 (in Kapitel 6.3) finden Sie sämtliche kursrelevanten Regeln zusammengefasst aufgeführt.

Programmierstil

#### 4.4.1 Bezeichnerwahl

Stilistisch gute Programme werden oft als selbstdokumentierend bezeichnet. In erster Linie wird dabei auf die geschickte Wahl von Bezeichnern angespielt, die es dem Leser erlauben, die Programmlogik schnell und ohne die Hilfe zusätzlicher Kommentare zu verstehen. (Daraus folgt aber nicht, daß Kommentare überflüssig werden!) Dies wird erreicht, wenn die Bedeutung einer Anweisung aus den Namen der beteiligten Variablen, Funktionen usw. und der verwendeten Schlüsselwörter bzw. Operatoren hervorgeht.

Richtig gewählte Namen sind so kurz wie möglich, aber lang genug, um die Funktion oder das Objekt, das sie bezeichnen, so zu beschreiben, daß jemand, der mit der Aufgabenstellung vertraut ist, sich darunter etwas vorstellen kann. Selten gebrauchte Namen können etwas länger als häufig gebrauchte Namen sein.

Für die Bezeichner in Pascal-Programmen geben wir folgende Regeln vor:

- Bezeichner sind aussagekräftig (sprechend) und orientieren sich an dem Problem, welches das Programm löst.
- *Bezeichner werden zunächst grundsätzlich so geschrieben, wie es die Rechtschreibung vorgibt. Eine Ausnahme davon wird gemacht, wenn mehrere Worte (oder deren Abkürzungen) zu einem Bezeichner zusammengezogen werden. In diesem Fall werden die Anfangsbuchstaben der zusammengezogenen Worte (bzw. Abkürzungen) im Bezeichner groß geschrieben. Der erste Buchstabe des Bezeichners wird groß oder klein geschrieben, je nachdem, ob das erste Wort (bzw. dessen Abkürzung) groß oder klein geschrieben wird.*
- **Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen.**
- **Typbezeichnern wird ein `t` vorangestellt.**  
Es ist hilfreich, wenn an einem Bezeichner direkt abgelesen werden kann, ob er eine Konstante, einen Typ, eine Variable oder einen formalen Parameter bezeichnet. Dies erspart lästiges Nachschlagen der Vereinbarungen und vermindert Fehlbenutzungen. Wir erreichen eine Unterscheidung durch kurze Präfixe sowie Groß- und Kleinschreibungsregeln.
- Für Variablen bieten sich häufig Substantive als Bezeichner an, für Zustandsvariablen oder boolesche Funktionen eignen sich Adjektive oder Adverbien.

#### Abkürzungen

Bei langen Namen ist eine gewisse Vorsicht angebracht, da diese umständlich zu handhaben sind und die Lesbarkeit von Programmen vermindern, weil sie die Programmstruktur verdecken können. Unnötig lange Namen sind Fehlerquellen, der Dokumentationswert eines Namens ist nicht proportional zu seiner Länge.

Allerdings ist es oft schwierig, kurze und gleichzeitig präzise Bezeichner zu finden. In diesem Fall werden für Bezeichner Abkürzungen gewählt. Wir wollen dafür einige Regeln zusammentragen:

Muß-Regel 1

Muß-Regel 2



- Ein spezieller Kontext erleichtert das Abkürzen von Bezeichnern, ohne daß diese dadurch an Bedeutung verlieren. Allgemein gilt: Wird eine Abkürzung benutzt, sollte sie entweder im Kontext verständlich oder allgemeinverständlich sein.
- Wenn Abkürzungen des allgemeinen Sprachgebrauchs existieren, sind diese zu verwenden (z.B. Pers für Person oder Std für Stunde).
- Häufig gebrauchte Namen können etwas kürzer als selten gebrauchte Namen sein.
- Eine Abkürzung muß enträtselbar und sollte aussprechbar sein.
- Beim Verkürzen sind Wortanfänge wichtiger als Wortenden, Konsonanten sind wichtiger als Vokale.
- Es ist zu vermeiden, daß durch Verkürzen mehrere sehr ähnliche Bezeichner entstehen, die leicht verwechselt werden können.

#### 4.4.2 Programmtext-Layout

Die Lesbarkeit eines Programms hängt auch von der äußeren Form des Programms ab. Für die äußere Form von Programmen gibt es viele Vorschläge und Beispiele aus der Praxis, die sich aber oft am Einzelfall oder an einer konkreten Programmiersprache orientieren. Für unsere Zwecke geben wir die folgenden Regeln an:

- Die Definitionen und Deklarationen werden nach einem eindeutigen Schema gegliedert. Dies fordert bereits die Pascal-Syntax, aber nicht in allen Programmiersprachen ist die Reihenfolge von Definitionen und Deklarationen vorgeschrieben.
- **Jede Anweisung beginnt in einer neuen Zeile; begin und end stehen jeweils in einer eigenen Zeile.**
- Der Einsatz von redundanten Klammern und Leerzeichen in Ausdrücken kann die Lesbarkeit erhöhen. Insbesondere sind Klammern dann einzusetzen, wenn die Priorität der Operatoren unklar ist. Die Prioritäten in verschiedenen Programmiersprachen folgen eventuell unterschiedlichen Hierarchien. Aber auch hier gilt wie bei Bezeichnerlängen: Ein Zuviel hat den gegenteiligen Effekt!
- Die hervorgehobene Darstellung von Schlüsselwörtern einer Programmiersprache unterstützt die Verdeutlichung der Programmstruktur. Wir heben Schlüsselwörter durch Fettdruck hervor. In Programmtexten, die von Hand geschrieben werden, wird durch Unterstreichung hervorgehoben.
- Kommentare werden so im Programmtext eingefügt, daß sie einerseits leicht zu finden sind und ausreichende Erläuterungen bieten, andererseits aber die Programmstruktur deutlich sichtbar bleibt (vgl. auch Abschnitt "Kommentare").

Muß-Regel 3

#### Einrückungen

Die Struktur eines Programms wird durch Einrückungen hervorgehoben. Wir geben die folgenden Regeln an:

- **Anweisungsfolgen werden zwischen begin und end um eine konstante Anzahl von 2 - 4 Stellen eingerückt. begin und end stehen linksbündig unter**

Muß-Regel 4

## Muß-Regel 5

der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.

- **Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.**
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **end** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist (vgl. auch Abschnitt "Kommentare").

Beispiele für die letzten drei Regeln:

```
if <Bedingung> then
begin
  <Anweisungsfolge1>
end { if <Bedingung> }
else
begin
  <Anweisungsfolge2>
end; { else }
```

```
while <Bedingung> do
begin
  <Anweisungsfolge>
end; { while <Bedingung> }
```

```
repeat
  <Anweisungsfolge>
until <Bedingung>;
```

```
for <Variable> := <Startwert> to <Zielwert> do
begin
  <Anweisungsfolge>
end; { for <Variable> }
```

- In einem Block werden die Definitions- und Deklarationsteile ebenfalls eingerückt. Die entsprechenden Schlüsselwörter stehen eingerückt in einer eigenen Zeile. Die vereinbarten Konstanten, Typen und Variablen werden darunter linksbündig angeordnet, in jeder Zeile sollte nur ein Bezeichner stehen. **begin** und **end** des Programmblocks werden nicht eingerückt. z.B:

```
program einruecken (input, output);
{ ein Programm zur Verdeutlichung von Layoutregeln }
```

```
const
PI = 3.1415927;
ANTWORT = 42;
```

```
type
tWochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);
```

```
var
  Heute,
  Gestern : tWochentag;

begin
  write ('Die Antwort ist: ');
  writeln (ANTWORT)
end. { einruecken }
```

In Ausnahmefällen, damit z.B. ein Programm (noch) auf eine Seite paßt, kann von diesen Regeln abgewichen werden. Falls das Programm in jedem Fall durch einen Seitenumbruch getrennt wird, gelten die Layoutregeln wieder.

### Leerzeilen

Leerzeilen dienen der weiteren Gliederung des Programmtextes. Es empfiehlt sich, zusammengehörige Definitionen, Deklarationen oder Anweisungen auch optisch im Programmtext zusammenzufassen. Die einfachste Art ist die Trennung solcher Gruppen durch Leerzeilen. Es bietet sich an, einer solchen Gruppe einen zusammenfassenden Kommentar voranzustellen.

Wir schlagen den Einsatz von Leerzeilen in folgenden Fällen vor:

- Die verschiedenen Definitions- und Deklarationsteile werden durch Leerzeilen getrennt.
- *Einzelne Funktionsdeklarationen werden durch Leerzeilen voneinander getrennt.*
- Anweisungsfolgen von mehr als ca. 10 Zeilen sollten durch Leerzeilen in Gruppen unterteilt werden.

### Funktionsdeklarationen

Die Layoutregeln für Funktionsdeklarationen haben wir zusammen mit anderen Regeln im Abschnitt "Funktionen" aufgeführt.

Bei den in diesem Abschnitt "Programmtext-Layout" angesprochenen Richtlinien ist wichtig, daß selbstaufgelegte Konventionen konsequent eingehalten werden, um ein einheitliches Aussehen aller Programmbausteine zu erreichen. Um dem gesamten Programmtext ein konsistentes Aussehen zu geben, werden automatische *Programmtext-Formatierer*, sogenannte *Prettyprinter*, eingesetzt. Solche Programme formatieren einen (syntaktisch korrekten) Programmtext nachträglich gemäß fester oder in einem gewissen Rahmen variabler Regeln. Durch *syntaxgesteuerte Editoren* erreicht man vergleichbare Ergebnisse direkt bei der Programmtext-Eingabe.

Programmtext-  
Formatierer  
Prettyprinter  
  
syntaxgesteuerter  
Editor

#### 4.4.3 Kommentare

Kommentare sind nützliche und notwendige Hilfsmittel zur Verbesserung der Lesbarkeit von Programmen. Sie sind wichtige Bestandteile von Programmen, und ihre Abwesenheit ist ein Qualitätsmangel.

Die Verwendung von Kommentaren ist in jeder höheren Programmiersprache möglich. Wir unterscheiden Kommentare, welche die Historie, den aktuellen Stand (Version) oder die Funktion von Programmen beschreiben - sie werden an den Programmanfang gestellt - und solche, die Objekte, Programmteile und Anweisungen erläutern - sie werden im Deklarations- und Anweisungsteil benutzt.

Die folgenden Konventionen bleiben an einigen Stellen unscharf, wichtig ist auch hier, daß man sich konsequent an die selbst gegebenen Regeln hält.

- Kommentare werden während der Programmierung eingefügt und nicht erst nachträglich ergänzt. Selbstverständlich werden die Kommentare bei Änderungen angepaßt.
- Kommentare sind so zu plazieren, daß sie die Programmstruktur nicht verdecken, d.h. Kommentare werden mindestens so weit eingerückt wie die entsprechenden Anweisungen. Trennende Kommentare wie Sternchenreihen oder Linien werden so sparsam wie möglich eingesetzt, da sie den Programmtext leicht zerschneiden.
- **Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst. Eine Kommentierung der Ein- und Ausgabedaten gehört gewöhnlich dazu.** Bei einem schwierigen Algorithmus können ggf. allgemeine, aber auf keinen Fall detaillierte Hinweise gegeben werden.
- Jeder Funktionskopf wird ähnlich einem Programmkopf kommentiert (vgl. Abschnitt "Funktionen").
- Die Bedeutung von Konstanten, Typen und Variablen wird erläutert, wenn sie nicht offensichtlich ist.
- Komplexe Anweisungen oder Ausdrücke werden kommentiert.
- Bei Kontrollanweisungen mit längeren zusammengesetzten Anweisungen sollte das abschließende **end** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist.
- Kommentare sind prägnant zu formulieren und auf das Wesentliche zu beschränken. Sie wiederholen nicht den Code mit anderen Worten, z.B.  

$$i := i + 1; \{ \text{erhoehe } i \text{ um } 1 \},$$
sondern werden nur da eingesetzt, wo sie zusätzliche Information liefern.
- *An entscheidenden Stellen wird der Zustand der Programmausführung beschrieben, z.B.*  

$$\{ \text{Jetzt ist das Teilfeld bis Index } n \text{ sortiert} \}.$$

#### 4.4.4 Funktionen

- **Jeder Funktionskopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.**
- *Jeder Parameter steht in einer eigenen Zeile. Diese Regel haben wir aus Platzgründen nicht immer befolgt. Zumindest sollte immer gelten: In einer Zeile stehen maximal drei Parameter desselben Typs mit*

Muß-Regel 6

Muß-Regel 7

*derselben Übergabeart.*

- *Das Ende einer Funktionsdeklaration wird durch die Wiederholung des Funktionsnamens in einem Kommentar nach dem abschließenden **end** des Anweisungsteils markiert.*
- ***Das Layout von Funktionen entspricht dem von Programmen.***

Muß-Regel 10

#### 4.4.5 Sonstige Merkgeregeln

Einige weitere Regeln wollen wir kurz auflisten:

- Es werden nach Möglichkeit keine impliziten (automatischen) Typanpassungen benutzt.
- Selbstdefinierte Typen werden nicht implizit definiert.
- Bei geschachtelten **if**-Anweisungen wird nach Möglichkeit nur der **else**-Zweig geschachtelt.
- *Bietet die Programmiersprache verschiedene Schleifenkonstrukte (in Pascal: **while**, **repeat**, **for**), so ist eine dem Problem angemessene Variante zu wählen.*
- ***Die Laufvariable wird innerhalb einer **for**-Anweisung nicht manipuliert.***
- Häufig ist es aus Gründen der Lesbarkeit empfehlenswert, nicht an Variablen zu sparen und einige Hilfsvariablen mehr zu verwenden, als unbedingt nötig. So können komplizierte Berechnungen durch zusätzliche Variablen (mit aussagekräftigen Namen), die Teilergebnisse aufnehmen, wesentlich transparenter programmiert werden. Außerdem darf eine Variable innerhalb ihres Gültigkeitsbereichs nicht ihre Bedeutung wechseln. Es ist z.B. verwirrend, wenn eine Variable einmal die Durchläufe einer Schleife kontrolliert und an anderer Stelle das Ergebnis einer komplexen Berechnung speichert - "nur weil der Typ gerade paßt". Eine zusätzliche Variable ist hier von Vorteil. Auf die konsistente Bedeutung von Variablen sollte auch bei Verschachtelungen geachtet werden.

Muß-Regel 15



# Kurseinheit III





## Lernziele zum Kapitel 5

Nach diesem Kapitel sollten Sie

1. eine Prozedur deklarieren und aufrufen können,
2. die wesentlichen Vorteile von Prozeduren und Funktionen kennen,
3. die Unterschiede (in Syntax und Semantik) zwischen Prozeduren und Funktionen nennen können,
4. die Realisierung einer Aufgabe als Funktion oder Prozedur begründen können,
5. das Parameterkonzept von ParaPascal beherrschen, die Wirkung der unterschiedlichen Übergabearten (**in**, **out** und **inout**) beschreiben und problembezogen einsetzen können,
6. die Umsetzung des allgemeineren Parameterkonzeptes von ParaPascal in das Parameterkonzept von Standard-Pascal angeben können,
7. die Wirkung der Übergabearten von Standard-Pascal (Wertübergabe, Referenzübergabe) kennen und problembezogen einsetzen können,
8. das Konzept der statischen und dynamischen Blockstruktur kennen sowie die damit zusammenhängenden Begriffe (Schachtelung, Gültigkeitsbereich, globale und lokale Variablen, Lebensdauer bzw. Existenz) verstanden haben,
9. die Begriffe Namenskonflikt und Seiteneffekt kennen, deren Auswirkungen erläutern können und wissen, wie man sie vermeidet,
10. durch Einhaltung von Layoutregeln und geeignete Bezeichnerwahl Prozeduren und Funktionen übersichtlich und verständlich formulieren können,
11. das Zeigerkonzept von Pascal verstanden haben,
12. grundlegende Operationen auf linearen Listen kennen und programmtechnisch beherrschen.

## 5. Programmierkonzepte orientiert an Pascal (Teil 3)

### 5.1 Prozeduren

#### 5.1.1 Prozeduren und Parameterübergabe

Im zweiten Teil der "Programmierkonzepte orientiert an Pascal" haben wir das Funktionskonzept kennengelernt, mit dem wir einen (Funktions-) Wert abhängig von einem oder mehreren Argumenten (Parametern) berechnen können.

Das Funktionskonzept ist aus verschiedenen Gründen nicht ausreichend. Ein typisches Beispiel ist eine Berechnung, die mehr als einen Wert liefert, z.B. die Bestimmung der Lösungen einer quadratischen Gleichung

$$ax^2 + bx + c = 0.$$

Für reelle Zahlen  $a$ ,  $b$  und  $c$  mit  $a \neq 0$  können wir die Lösungen - falls sie existieren - bekanntermaßen nach der Formel

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

berechnen. Neben der Berechnung der beiden Lösungen sollte natürlich auch geprüft werden, ob überhaupt eine Lösung existiert. (Ist der Ausdruck unter der Wurzel negativ, gibt es keine reellwertige Lösung.) Nur mit Hilfe des Funktionskonzepts müßten wir die Berechnung in drei Funktionen aufteilen, etwa

```
function Loesbar (a, b, c : real): boolean;
function Loesung1 (a, b, c : real): real;
function Loesung2 (a, b, c : real): real;
```

Dieses Vorgehen ist aber nicht sinnvoll, da sich eine zusammenhängende Berechnung auch in einer zusammenhängenden Programmkonstruktion widerspiegeln sollte. Wir benötigen also einen Mechanismus, der es erlaubt, die Berechnung in einem Konstrukt zusammenzufassen, das mehrere Werte an die Aufrufumgebung zurückliefert. Ein solches Konstrukt wird in Pascal wie in allen imperativen Programmiersprachen durch das Prozedurkonzept und verschiedene Arten der Parameterübergabe realisiert.

Eine *Prozedur* (procedure) ist einer Funktion sehr ähnlich. Eine *Prozedurdeklaration* unterscheidet sich von einer Funktionsdeklaration zunächst einmal durch das Schlüsselwort **procedure** (an Stelle von **function**) und durch das Fehlen des Ergebnistyps. Dementsprechend gibt es auch keine Zuweisung eines Wertes an den Prozedurbezeichner.

Ein *Prozeduraufruf* erfolgt durch Angabe des Prozedurnamens gefolgt von der Liste der aktuellen Parameter. Er ist eine eigenständig ausführbare Anweisung und nicht Teil eines Ausdrucks wie ein Funktionsaufruf.

Prozeduraufruf

Die Übergabe von Informationen an eine Prozedur bzw. aus ihr heraus geschieht durch verschiedene Arten der *Parameterübergabe*. Um die Übergabearten zu unterscheiden, werden den formalen Parametern Schlüsselwörter vorangestellt. Im übrigen gelten bei einem Prozeduraufruf dieselben Regeln über Parameter-Reihenfolge und Typzuordnung wie bei einem Funktionsaufruf.

Parameterübergabe

Wir werden Ihnen in diesem Abschnitt drei Arten der Parameterübergabe vorstellen, die in dieser Form in Standard-Pascal nicht existieren. Wir wollen diesen fiktiven daher nicht compilierbaren Pascal-Dialekt als *ParaPascal* bezeichnen. Er ist im Hinblick auf die Parameterübergabearten angelehnt an die Programmiersprache ADA, die diese moderneren Konzepte anbietet. Die Realisierung von Teilen dieser Konzepte in Standard-Pascal lernen Sie im nächsten Kapitel kennen.

ParaPascal

*Eingangs-* bzw. *in-Parameter* sind Ihnen bereits von den Funktionen her bekannt, obwohl wir sie dort aus didaktischen Gründen nicht explizit so genannt haben. Von jetzt an werden wir Funktionsparameter allerdings stets als *in-Parameter* kennzeichnen. Ein *in-Parameter* ist an dem vorangestellten Schlüsselwort **in** zu erkennen. Wir können uns einen formalen *in-Parameter* als lokale Konstante vorstellen, die beim Prozeduraufruf durch den aktuellen Parameter ihren Wert erhält. Eine Konsequenz dieser Parameterübergabeart ist, daß der formale Parameter (= Konstante) und damit auch der aktuelle Parameter durch die Prozedur bzw. Funktion nicht verändert werden können.

Eingangsparameter  
*in-Parameter*

Mit Hilfe von *Ausgangs-* bzw. *out-Parametern* werden Informationen aus einer Prozedur nach außen an die rufende Umgebung abgeliefert. Ein *out-Parameter* ist durch das vorangestellte Schlüsselwort **out** charakterisiert. Einen formalen *out-Parameter* können wir uns vorstellen als lokale Variable, deren Wert nicht abgefragt, wohl aber verändert werden kann. Unmittelbar vor Verlassen der Prozedur wird der Wert des formalen Parameters (das "Ergebnis") dem aktuellen Parameter zugewiesen und damit der Umgebung zugänglich gemacht. Damit der aktuelle Parameter den Ergebniswert aufnehmen kann, muß er natürlich eine Variable (genauer: ein Ausdruck, der auf der linken Seite einer Zuweisung stehen kann) sein. Ein möglicher Wert des aktuellen Parameters beim Prozeduraufruf ist für die Prozedur und damit für den Wert des aktuellen Parameters nach Beendigung der Prozedur bedeutungslos.

Ausgangsparameter  
*out-Parameter*

#### Beispiel 5.1.1.1

Zur Berechnung der Lösungen obiger quadratischer Gleichung benötigen wir neben den drei Eingangsparametern für die Koeffizienten zwei Ausgangsparameter für die Lösungen und einen dritten Ausgangsparameter vom Typ `boolean`, der anzeigt, ob überhaupt eine Lösung existiert.

```

procedure LoesungBestimmen (
    in a, b, c : real;
    out Loesung1, Loesung2 : real;
    out gibtLoesung : boolean);
{ bestimmt die reellen Loesungen der Gleichung
   $a \cdot x^2 + b \cdot x + c = 0.0$ 
  unter der Vorbedingung:  $a \neq 0.0$  }

var
  d : real; { Hilfsvariable }

begin
  d := b * b - 4.0 * a * c;
  if (d < 0.0) or (a = 0.0) then
    { nicht loesbar oder Vorbedingung nicht erfuehlt }
    gibtLoesung := false
  else
    begin
      gibtLoesung := true;
      d := sqrt (d);
      Loesung1 := -(b - d) / (2.0 * a);
      Loesung2 := -(b + d) / (2.0 * a)
    end
  end; { LoesungBestimmen }

```

Unter der Voraussetzung, daß L1 und L2 als Variablen vom Typ real und exist als boolean-Variable deklariert sind, kann die Prozedur LoesungBestimmen aufgerufen werden durch

```
LoesungBestimmen (1.0, 3.0, 2.0, L1, L2, exist)
```

Im Anschluß an den Prozeduraufruf haben die Variablen die Werte:

```

L1 = -1.00,
L2 = -2.00 und
exist = true.

```

Wir geben noch ein (benutzerfreundliches) Programm an, das die Prozedur LoesungBestimmen benutzt:

```

program LoesungSuchen (input, output);
{ bestimmt die Loesungen der Gleichung
   $a \cdot x^2 + b \cdot x + c = 0.0$  fuer einzulesende a, b, c;
  Vorbedingung:  $a \neq 0.0$  }

var
  KoeffA,
  KoeffB,
  KoeffC,

```

```

L1,
L2 : real;
exist : boolean;

procedure LoesungBestimmen (
    in a, b, c : real;
    out Loesung1, Loesung2 : real;
    out gibtLoesung : boolean);
{ bestimmt die Loesungen der Gleichung
   $a*x*x + b*x + c = 0.0$ 
  unter der Vorbedingung:  $a \neq 0.0$  }

var
  d : real; { Hilfsvariable }

begin
  { Prozedurrumpf wie oben }
end; { LoesungBestimmen }

begin { LoesungSuchen }
  writeln ('Loesungen einer quadratischen Gleichung');
  writeln ('Geben Sie die Koeffizienten ein: ');
  write ('a: ');
  readln (KoeffA);
  write ('b: ');
  readln (KoeffB);
  write ('c: ');
  readln (KoeffC);
  LoesungBestimmen (KoeffA, KoeffB, KoeffC,
                    L1, L2, exist);
  if exist then
    writeln ('Loesungen: ', L1:2:2, ' und ', L2:2:2)
  else
    writeln ('KEINE Loesung!')
end. { LoesungSuchen }

```

```

Loesungen einer quadratischen Gleichung
Geben Sie die Koeffizienten ein:
a: 1.0
b: 3.0
c: 2.0
Loesungen: -1.00 und -2.00

```



Aufgabe 5.1.1.2

Ändern Sie die Funktion `FeldMinimumPos` aus Beispiel 4.3.4 ab in eine ParaPascal-Prozedur `FeldMinPosUndWert`, die nicht nur die Position, sondern auch den Wert des Minimums bestimmt. Geben Sie auch ein Beispiel für einen Aufruf dieser Prozedur an.



Ein weiterer Vorteil von `out`-Parametern gegenüber Funktionswerten ist, daß auch strukturierte Typen als Ergebnistyp zurückgegeben werden können. Wir zeigen auch dies an einem Beispiel.

Beispiel 5.1.1.3

Im Programm `FlugZeit` (Beispiel 4.2.2.1) haben wir zwei Uhrzeiten eingelesen. Wir können jetzt diese beiden Programmteile durch zwei Aufrufe einer Prozedur `ZeitLesen` ersetzen, deren `out`-Parameter vom **record**-Typ `tZeit` ist:

```
program Flugzeit2 (input, output);
{ berechnet die Flugdauer aus Start- und Landezeit }

const
  MINSEK = 59;
  STUNDE = 23;

type
  tMinSek = 0..MINSEK;
  tStunde = 0..STUNDE;
  tZeit = record
    h : tStunde;
    m,
    s : tMinSek
  end;
  tUebertrag = 0..1;

var
  Abflug,
  Ankunft,
  Flugdauer : tZeit;
  Uebertrag : tUebertrag;

procedure ZeitLesen (out Zeit : tZeit);
{ liest eine Uhrzeit in Stunden, Minuten und Sekunden
  ein }

  var
    Eingabe : integer;

begin
  repeat
```

```

        write ('Stunde: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= STUNDE);
    Zeit.h := Eingabe;

    repeat
        write ('Minute: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= MINSEK);
    Zeit.m := Eingabe;

    repeat
        write ('Sekunde: ');
        readln (Eingabe)
    until (Eingabe >= 0) and (Eingabe <= MINSEK);
    Zeit.s := Eingabe
end; { ZeitLesen }

begin { Flugzeit2 }
    writeln ('Bitte die Abflugzeit eingeben');
    ZeitLesen (Abflug);
    writeln;
    writeln ('Bitte die Ankunftszeit eingeben');
    ZeitLesen (Ankunft);
    { Flugzeit berechnen: Ab hier ist das Programm
      identisch mit dem in Beispiel 4.2.2.1 }
    ...
    writeln;
    writeln ('Flugdauer: ', Flugdauer.h, ':',
            Flugdauer.m, ':', Flugdauer.s)
end. { Flugzeit2 }

```

*Bitte die Abflugzeit eingeben*

*Stunde: 22*

*Minute: 53*

*Sekunde: 12*

*Bitte die Ankunftszeit eingeben*

*Stunde: 5*

*Minute: 12*

*Sekunde: 54*

*Flugdauer: 6:19:42*



Änderungsparameter  
inout-Parameter

Funktionen berechnen aus den vorgegebenen Parametern den zugehörigen Funktionswert. Die Parameter werden dabei nicht verändert (in-Parameter), denn wir sind nur an dem Funktionswert interessiert. Bei Prozeduren können wir über einen out-Parameter den entsprechenden aktuellen Parameter (d. h. die Variable der Aufrufumgebung) ebenfalls nicht verändern, sondern lediglich initialisieren, d. h. erstmalig einen Wert zuweisen.

Nun ist es natürlich sinnvoll, daß Prozeduraufrufe Programmvariablen verändern, wie es mit gewöhnlichen Anweisungen bzw. Anweisungsfolgen möglich ist. Betrachten wir dazu das Beispiel 4.3.4, in dem wir ein Feld sortiert haben. Ein Teil des Sortiervorgangs bestand in der Vertauschung von zwei Feldelementen:

```
Tausch := EingabeFeld[MinPos];
EingabeFeld[MinPos] := EingabeFeld[i];
EingabeFeld[i] := Tausch
```

Wir können diese Anweisungsfolge in einer Prozedur vertauschen mit den Parametern *hin* und *her* zusammenfassen. Dabei muß die Parameterinformation sowohl in die Prozedur hinein als auch nach der Vertauschung wieder hinaus transportiert werden, damit die Vertauschung auch für die Prozedurumgebung wirksam wird. Die Parameter müssen also zugleich Eingangs- und Ausgangsparameter sein. Wir wollen sie als *Änderungsparameter* oder *inout-Parameter* bezeichnen. Bei dieser dritten Art der Parameterübergabe, können wir uns den formalen Parameter als lokale Variable vorstellen, die beim Prozeduraufruf mit dem Wert des aktuellen Parameters initialisiert wird und im weiteren Verlauf der Prozedur verändert und abgefragt werden kann. Unmittelbar vor Verlassen der Prozedur wird der Wert des formalen Parameters dem aktuellen Parameter zugewiesen und damit der rufenden Umgebung zugänglich gemacht. Den formalen Parametern dieser Art werden wir in ParaPascal das Schlüsselwort **inout** voranstellen.

```
procedure vertauschen (inout hin, her : integer);
{ vertauscht die Werte der beiden Parameter }

var
  Tausch : integer;

begin
  Tausch := hin;
  hin := her;
  her := Tausch
end; { vertauschen }
```

Wir wollen die Prozedur `vertauschen` benutzen, um zwei Feldelemente zu vertauschen. Ein Aufruf sieht z.B. folgendermaßen aus:

```
vertauschen (EingabeFeld[MinPos], EingabeFeld[i]);
```



Die Werte der Feldelemente an den Positionen `MinPos` und `i` werden den formalen Parametern `hin` und `her` zugewiesen. Nach dem Vertauschen der Werte mittels der lokalen Variablen `Tausch` werden sie in die aktuellen Parameter `EingabeFeld[MinPos]` und `EingabeFeld[i]` zurückgeschrieben. Die Prozedur hat also die Werte der aktuellen Parameter vertauscht.

#### Beispiel 5.1.1.4

Mit Hilfe von `inout`-Parametern können wir auch den Programmteil, der die eigentliche Sortierung des Feldes leistet, als Prozedur zusammenfassen. Der `inout`-Parameter `SortFeld` wird bei Aufruf der Prozedur mit dem unsortierten Feld als aktuellem Parameter initialisiert, am Ende wird das sortierte Feld in den aktuellen Parameter zurückkopiert.

```
program FeldSort2 (input, output);
{ sortiert ein einzulesendes Feld von integer-Zahlen }

const
  FELDGROESSE = 5;

type
  tIndex = 1..FELDGROESSE;
  tFeld = array[tIndex] of integer;

var
  EingabeFeld : tFeld;
  idx : tIndex;

function FeldMinimumPos (
    in Feld : tFeld;
    in von, bis : tIndex) : tIndex;
{ bestimmt die Position des Minimums
  im Feld zwischen von und bis }

var
  MinimumPos,
  j : tIndex;

begin
  MinimumPos := von;
  for j := von + 1 to bis do
    if Feld[j] < Feld[MinimumPos] then
      MinimumPos := j;
  FeldMinimumPos := MinimumPos
end; { FeldMinimumPos }

procedure vertauschen (inout hin, her : integer);
{ vertauscht die Werte der beiden Parameter }
```

```

    var
      Tausch : integer;

begin
  Tausch := hin;
  hin := her;
  her := Tausch
end; { vertauschen }

procedure FeldSortieren (inout SortFeld : tFeld);
{ sortiert SortFeld aufsteigend }

    var
      MinPos,
      i : tIndex;

begin
  for i := 1 to FELDGROESSE - 1 do
    begin
      MinPos := FeldMinimumPos(SortFeld, i,
        FELDGROESSE);
      { minimum gefunden, jetzt mit dem Element
        auf Position i vertauschen }
      vertauschen (SortFeld[MinPos], SortFeld[i])
    end
  end; { FeldSortieren }

begin
  { Feld einlesen }
  writeln ('Geben Sie ', FELDGROESSE, ' Werte ein: ');
  for idx := 1 to FELDGROESSE do
    readln (EingabeFeld[idx]);

  { Feld sortieren }
  FeldSortieren (EingabeFeld);

  { sortiertes Feld ausgeben }
  for idx := 1 to FELDGROESSE do
    write (EingabeFeld[idx]:6);
  writeln
end. { FeldSort2 }

```



Beachten Sie, daß die Funktion `FeldMinimumPos` und die Prozedur `vertauschen` vor der Prozedur `FeldSortieren` deklariert sind. Das ist deshalb nötig, weil `FeldMinimumPos` und `vertauschen` in `FeldSortieren` benutzt wer-

den und in Pascal alle Bezeichner vor ihrer ersten Verwendung vereinbart sein müssen.

Alle drei Funktionen bzw. Prozeduren sind im Block des Programms vereinbart. Im Anweisungsteil des Programms wird aber lediglich die Prozedur `FeldSortieren` aufgerufen, denn `FeldMinimumPos` und `vertauschen` werden nur in `FeldSortieren` benutzt. Wir können daher im Vereinbarungsteil von `FeldSortieren` die Funktion `FeldMinimumPos` und die Prozedur `vertauschen` lokal deklarieren, d.h. deren Deklaration in die Prozedur `FeldSortieren` verlagern. Die Anweisungsteile der Blöcke von Prozeduren, Funktionen und Programm bleiben unverändert. Diese Maßnahme dokumentiert, für welche Programmbereiche die Funktion `FeldMinimumPos` und die Prozedur `vertauschen` relevant sind, und erhöht somit die Übersichtlichkeit und Lesbarkeit des Programms. Außerdem ist jetzt die Prozedur `FeldSortieren` in sich geschlossen und allein arbeitsfähig. Wir können sie nun unmittelbar als Sortierprozedur benutzen, ohne uns darum kümmern zu müssen, ob noch weitere Hilfsprozeduren bzw. -funktionen benötigt werden. Wir geben das Programm überblicksartig an.

#### Beispiel 5.1.1.5

```

program FeldSort3 (input, output);
{ sortiert ein einzulesendes Feld von integer-Zahlen }

  const
    FELDGROESSE = 5;

  type
    tIndex = 1..FELDGROESSE;
    tFeld = array[tIndex] of integer;

  var
    EingabeFeld : tFeld;
    idx : tIndex;

  procedure FeldSortieren (inout SortFeld : tFeld);

    var
      MinPos,
      i : tIndex;

    function FeldMinimumPos (
      in Feld : tFeld;
      in von, bis : tIndex) : tIndex;

      var
        MinimumPos,
        j : tIndex;

```

```

begin
    ...
end; { FeldMinimumPos }

procedure vertauschen (inout hin, her : integer);
var
    Tausch : integer;

begin
    ...
end; { vertauschen }

begin { FeldSortieren }
    ...
end; { FeldSortieren }

begin { FeldSort3 }
    ...
end. { FeldSort3 }

```



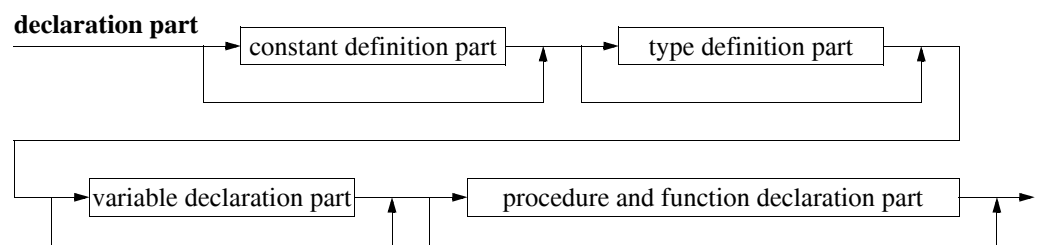
### 5.1.2 Prozeduren und Parameterübergabe in Pascal

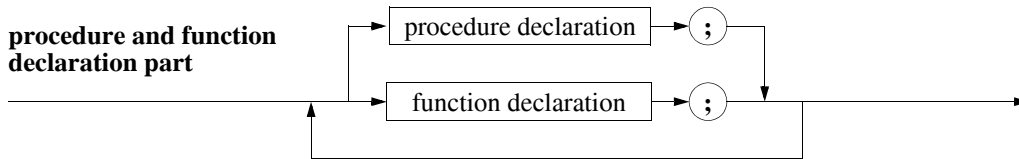
Nachdem wir im vorigen Abschnitt das Prozedurkonzept und eine moderne Sicht der Parameterübergabe vorgestellt haben, kommen wir jetzt zu dem Prozedurkonzept in Pascal. Im Unterschied zu den bisherigen Übergabearten existieren in Pascal die Übergabearten `out` und `inout` nicht, sondern es steht dafür lediglich die Referenzübergabe zur Verfügung. Die `in`-Übergabeart, in Pascal Wertübergabe genannt, haben wir nebst ihrer syntaktischen Ausprägung bereits in dem Abschnitt 4.3 über Funktionen kennengelernt.

Wir präzisieren zunächst die Prozedurdeklaration und den Prozeduraufruf durch Syntaxdiagramme in Standard-Pascal.

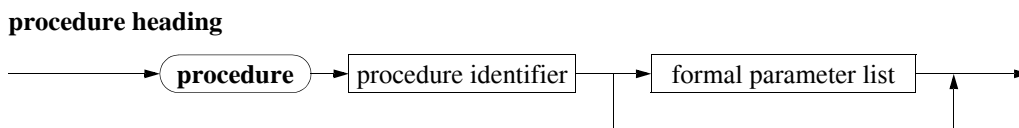
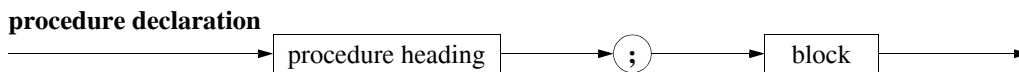
Prozedurdeklarationen müssen im Deklarationsteil eines Blocks im Anschluß an die Vereinbarungen von Konstanten, Typen und Variablen stehen. Prozedurdeklarationen können sich mit Funktionsdeklarationen in einem gemeinsamen *Deklarationsteil* abwechseln. Wir ergänzen nochmals das Syntaxdiagramm für den Deklarationsteil:

Deklarationsteil

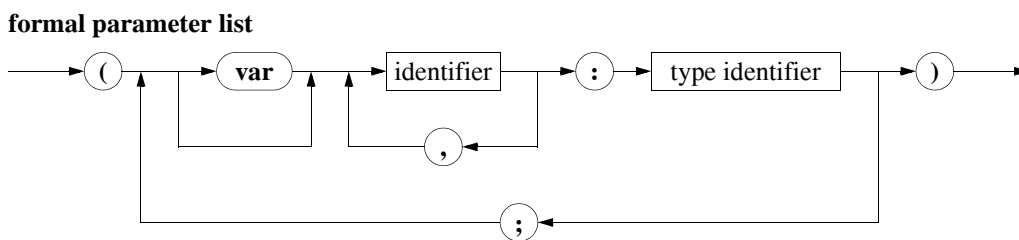




Eine *Prozedurdeklaration* ist ähnlich aufgebaut wie eine Funktionsdeklaration: Sie besteht aus *Prozedurkopf* und einem Block, der auch *Prozedurrumpf* genannt wird. Im Prozedurkopf folgt auf das Schlüsselwort **procedure** der Prozedurname (procedure identifier) sowie eine Liste von in Klammern eingeschlossenen *formalen Parametern* (formal parameter list):



Wie schon erwähnt gibt es in Standard-Pascal nur zwei Arten der Parameterübergabe: Wertübergabe (call-by-value) und Referenzübergabe (call-by-reference). *Wertübergabe* haben wir schon bei den Funktionen kennengelernt, *Wertparameter* werden durch kein Schlüsselwort gekennzeichnet. Zur Unterscheidung wird den *Referenzparametern* das Schlüsselwort **var** vorangestellt.



Die Wertübergabe in Standard-Pascal und die *in*-Übergabe in ParaPascal sind semantisch äquivalent, d.h. bedeutungsgleich. (Streng genommen ist die Aussage nicht ganz korrekt, aber die Möglichkeit, in Pascal einen Wertparameter lokal zu manipulieren, ist schlechter Programmierstil, der zu schlecht lesbaren Programmen führt, so daß wir davon keinen Gebrauch machen werden.)

*Referenzparameter* in Pascal spielen die Rolle von *inout*-Parametern in ParaPascal. Die Definition der Semantik ist zwar unterschiedlich, in der praktischen Auswirkung zeigen sich jedoch (im Normalfall) keine Unterschiede.

Prozedurdeklaration  
Prozedurkopf  
Prozedurrumpf

formale Parameter

Wertübergabe  
call-by-value  
Wertparameter

Referenzparameter

Verweis

Adresse

Referenzübergabe  
call-by-reference  
var-Parameter

Im Fall eines Referenzparameters wird nicht wie bei einem `inout`-Parameter der Wert des aktuellen Parameters an den formalen Parameter übergeben, den wir uns wiederum als (manipulierbare) lokale Variable vorstellen können, sondern es wird der *Verweis* (Referenz) auf den aktuellen Parameter an den formalen Parameter übergeben. Technisch gesehen können wir uns einen Verweis vorstellen als die *Adresse* der Speicherzelle bzw. des Speicherbereichs, in dem der aktuelle Parameter abgelegt ist. Die Konsequenz ist, daß jede Änderung des formalen Parameters zugleich eine Änderung des aktuellen Parameters bedeutet, so daß nach Abarbeitung der Prozedur der letzte Wert des formalen Parameters zugleich der Wert des aktuellen Parameters ist. Formaler und aktueller Parameter können als synonym betrachtet werden. Da - technisch gesehen - ein Verweis (Referenz) an die aufgerufene Prozedur übergeben wird, sprechen wir auch von *Referenzübergabe* (*call-by-reference*). Wegen des Schlüsselwortes **var** sprechen wir auch von *var-Parametern*.

In Standard-Pascal gibt es keine `out`-Parameter, sie müssen daher über `var`-Parameter simuliert werden. Dies hat den unschönen Nebeneffekt, daß der aktuelle Parameter vor dem Prozeduraufruf bereits einen Wert besitzen kann, obwohl doch die Initialisierung erst durch die Prozedur vorgenommen werden soll, also der "Eingangswert" des aktuellen Parameters sinnlos ist.

Wir wollen deshalb aus Gründen der besseren Lesbarkeit bzw. Verständlichkeit weiterhin drei Arten der Parameterübergabe "logisch" unterscheiden und dies dadurch dokumentieren, daß wir die Bezeichner der formalen Parameter entsprechend der logischen Übergabeart mit `in`-, `io`- oder `out`- beginnen lassen.

Wir stellen die Prozedur- und Funktionsköpfe aus den bisherigen ParaPascal-Beispielen der Standard-Pascal-Notation gegenüber. Natürlich müssen die Parameterbezeichner in den Anweisungsteilen entsprechend umbenannt werden.

#### ParaPascal:

```
procedure LoesungBestimmen (
    in    a,
        b,
        c : real;
    out  Loesung1,
        Loesung2 : real;
    out  gibtLoesung : boolean);
```

#### Standard-Pascal:

```
procedure LoesungBestimmen (
    inA,
    inB,
    inC : real;
    var outLoesung1,
        outLoesung2 : real;
    var outGibtLoesung : boolean);
```

ParaPascal:

```
procedure ZeitLesen (out Zeit : tZeit);
```

Standard-Pascal:

```
procedure ZeitLesen (var outZeit : tZeit);
```

ParaPascal:

```
procedure vertauschen (inout hin, her : integer);
```

Standard-Pascal:

```
procedure vertauschen (var ioHin, ioHer : integer);
```

ParaPascal:

```
function FeldMinimumPos (  
    in  Feld : tFeld;  
    in  von,  
    bis : tIndex): tIndex;
```

Standard-Pascal:

```
function FeldMinimumPos (  
    inFeld : tFeld;  
    inVon,  
    inBis : tIndex): tIndex;
```

ParaPascal:

```
procedure FeldSortieren (inout SortFeld : tFeld);
```

Standard-Pascal:

```
procedure FeldSortieren (var ioSortFeld : tFeld);
```

#### Aufgabe 5.1.2.1

Schreiben Sie das Programm FeldSort3 in Standard-Pascal. Erweitern Sie das Programm und die Prozedur FeldSortieren so, daß Felder unterschiedlicher Länge (bis zu einer vorgegebenen maximalen Länge) sortiert werden können.



## 5.2 Blockstrukturen: Gültigkeitsbereich und Lebensdauer

Die folgenden Ausführungen gelten für Prozeduren und Funktionen gleichermaßen. Daher werden wir den Begriff "Prozedur" verallgemeinernd auch für Funktionen benutzen.

lokal

Gültigkeitsbereich

lokale Variable

Alle Vereinbarungen von Konstanten, Typen, Variablen und Prozeduren in einem Block werden als *lokal* zu diesem Block bezeichnet. Dieser Block ist der einzige Teil des Programms, in dem diese Bezeichner angesprochen werden können. Der Block ist der *Gültigkeitsbereich* seiner lokal vereinbarten Bezeichner. Wenn wir z.B. eine Variable *a* lokal zur Prozedur *P* deklarieren, so kann *a* nur innerhalb von *P* angesprochen werden, denn *a* ist eine *lokale Variable* der Prozedur *P*, sie ist unbekannt außerhalb dieser Prozedur.

```
program BlockBeispiel1 (input, output);
{ dient lediglich zu Demonstrationszwecken }

var
  x,
  y : integer;

procedure P;

  var
    a : integer;

  begin
    a := 42;
    ...
  end; { P }

begin
  ...
  writeln (a); { <-- Fehler: a ist hier unbekannt,
  ...           da a lokal zu P ist! }
end. { BlockBeispiel1 }
```

global

globale Variable

Ein Bezeichner *n*, der in einem Block *B*<sub>1</sub> vereinbart ist, kann in jedem Block *B*<sub>2</sub> angesprochen werden, der lokal zu *B*<sub>1</sub> vereinbart ist. Ebenso können wir *n* in jedem Block *B*<sub>3</sub>, der lokal zu *B*<sub>2</sub> ist, ansprechen, usw. *n* heißt *global* für die inneren Blöcke *B*<sub>2</sub>, *B*<sub>3</sub> usw. So können z.B. die Variablen *x* und *y* des Programms *BlockBeispiel2* auch innerhalb der Prozedur *P* angesprochen werden, sie sind *globale Variablen* der Prozedur *P*.



```

program BlockBeispiel2 (input, output);
{ dient lediglich zu Demonstrationszwecken }

  var
    x,
    y : integer;

  procedure P;

    var
      a : integer;

  begin
    a := 42;
    x := a + y;    { Die Zuweisung ist zulaessig,
                    da x und y global zu P,
                    d.h. bekannt sind }
    ...
  end; { P }

begin
  ...
  y := 4;
  P;
  ...
end. { BlockBeispiel2 }

```

Betrachten wir nun noch einmal das Programm `FeldSort3` aus Beispiel 5.1.1.5. Die Prozedur `vertauschen` und die Funktion `FeldMinimumPos` werden nur innerhalb der Prozedur `FeldSortieren` verwendet und sind daher lokal vereinbart. Eine Prozedur innerhalb einer anderen Prozedur zu deklarieren, bezeichnen wir als *Schachtelung*.

### Beispiel 5.2.1

Die folgende Abbildung 5.5 zeigt die Schachtelungsstruktur des Programms `FeldSort3` (als Pascal- und nicht als ParaPascal-Programm). Um die Schachtelung graphisch darstellen zu können, haben wir den Programmtext komprimiert.

Schachtelung

```

program FeldSort3 (input, output);

    const
        FELDGROESSE = 5;

    type
        tIndex = 1..FELDGROESSE;
        tFeld = array[tIndex] of integer;

    var
        EingabeFeld : tFeld;
        idx : tIndex;

    procedure FeldSortieren(var ioSortFeld : tFeld);

        var
            MinPos,
            i : tIndex;

        function FeldMinimumPos (
            inFeld : tFeld;
            inVon,
            inBis : tIndex): tIndex;

            var
                MinimumPos,
                j : tIndex;

            begin ... end; { FeldMinimumPos }

        procedure vertauschen(
            var ioHin,
            ioHer : integer);

            var
                Tausch : integer;

            begin ... end; { vertauschen }

        begin ... end; { FeldSortieren }

    begin ... end. { FeldSort3 }

```

Abbildung 5.5 : statische Blockstruktur (Schachtelung) im Programm FeldSort3



Die Gesamtstruktur eines Pascal-Programms stellt sich also als eine geordnete Menge von Blöcken dar, die geschachtelt sein können. Wir bezeichnen dies als *statische Blockstruktur*, da die Struktur aus dem (statischen) Programmtext ablesbar ist.

statische  
Blockstruktur

Wir wollen nun die *Regeln über den Gültigkeitsbereich* von Bezeichnern in Pascal präzisieren:

Regeln über den  
Gültigkeitsbereich

1. Der Gültigkeitsbereich eines Bezeichners besteht aus dem Block, in dem er vereinbart ist, und allen in ihm geschachtelten Blöcken, jedoch unter Beachtung der Regel 2.
2. Wird ein Bezeichner in einem inneren Block  $B_i$  erneut vereinbart (Namenskonflikt), so ist die Vereinbarung aus einem äußeren Block  $B_e$  für  $B_i$  und alle inneren Blöcke von  $B_i$  nicht mehr gültig. Wird der innere Block  $B_i$  verlassen, so ist der Bezeichner aus dem äußeren Block  $B_e$  wieder gültig.
3. Standardbezeichner gelten als vereinbart in einem fiktiven Block, der das gesamte Programm umschließt. Sie haben Gültigkeit für das gesamte Programm, wenn sie nicht im Programm oder einem darin geschachtelten Block erneut vereinbart werden. (Eine solche Vereinbarung ist zwar möglich, aber schlechter Programmierstil).

Sind in einem Programm mehrere Bezeichner  $x$  vereinbart, so finden wir also folgendermaßen heraus, welcher Bezeichner  $x$  in einem Block  $B$  gültig ist:

Zunächst prüfen wir den Vereinbarungsteil des Blocks  $B$ , in dem  $x$  verwendet wird. Ist  $x$  dort vereinbart, haben wir  $x$  schon gefunden. Ist  $x$  dort nicht vereinbart, suchen wir im Vereinbarungsteil des Blocks, der  $B$  direkt umfaßt usw. Das erste  $x$ , das wir auf diesem Weg finden, ist dasjenige, das im Block  $B$  Gültigkeit hat. Finden wir so keine Vereinbarung für  $x$ , dann ist  $x$  in diesem Block kein gültiger Bezeichner.

Es kann ein und derselbe Bezeichner für unterschiedliche Zwecke in verschiedenen Blöcken vereinbart und verwendet werden. Dabei kann es zu einem *Namenskonflikt* kommen. Welcher Bezeichner im Einzelfall gültig ist, legt Regel 2 fest. Die folgende Grafik verdeutlicht die oben skizzierte Vorgehensweise zur Bestimmung der in einem Block gültigen Bezeichner.

Namenskonflikt



Beispiel 5.2.2

```

program P (input, output);
  var
    i,
    j : integer;

  procedure Q (inA : integer; var ioB :integer);
    const
      I = 16;
    var
      k : char;

    procedure R;
      var
        j : real;
    begin { R }
      { Dieser Anweisungsteil kann auf
        folgende Bezeichner zugreifen:
        lokale Variable j : real;
        formale Parameter inA und ioB von Q;
        globale Konstante I = 16;
        globale Variable k : char;
        globale Prozeduren R und Q;
        alle Standardbezeichner. }
    end; { R }

  begin { Q }
    { Dieser Anweisungsteil kann auf
      folgende Bezeichner zugreifen:
      formale Parameter inA und ioB;
      lokale Konstante I = 16;
      lokale Variable k : char;
      lokale Prozedur R;
      globale Variable j : integer;
      globale Prozedur Q;
      alle Standardbezeichner. }
  end; { Q }

begin { P }
  { Dieser Anweisungsteil kann auf
    folgende Bezeichner zugreifen:
    lokale Variablen i, j : integer;
    lokale Prozedur Q;
    alle Standardbezeichner. }
end. { P }

```

Abbildung 5.6 : statische Blockstruktur (Schachtelung) und Namenskonflikte im Programm P



dynamische  
Blockstruktur

Im Unterschied zur statischen Blockstruktur, die aus dem Programmtext ablesbar ist, ergibt sich die *dynamische Blockstruktur* aus den (dynamischen) Prozeduraufrufen während der Laufzeit. Jeder Prozeduraufruf erzeugt einen neuen inneren Block  $B_i$  (inklusive aller lokalen Vereinbarungen) in der dynamischen Blockstruktur. Der  $B_i$  unmittelbar umgebende Block in der dynamischen Blockstruktur ist der Block, aus dem heraus die Prozedur aufgerufen wird. Die dynamische Blockstruktur hängt letztlich von den Eingabedaten ab. Abhängig von der jeweiligen Eingabe ergibt sich (beispielsweise über **if**-Abfragen gesteuert), welche Prozeduren in welcher Reihenfolge aufgerufen werden. Bei einer fehlerhaften Eingabe wird etwa eine Fehlerprozedur aktiviert, bei korrekter Eingabe dagegen eine Berechnungsprozedur.

Lebensdauer  
Existenz

So wie der Begriff Gültigkeitsbereich mit der statischen Blockstruktur zusammenhängt, ist der Begriff *Lebensdauer* bzw. *Existenz* (von Variablen und Konstanten) mit der dynamischen Blockstruktur verknüpft. In Pascal ist festgelegt, daß lokale Variablen und Konstanten nur während der Ausführung der Prozedur existieren, in der sie vereinbart sind. Sie werden zu Beginn des Prozeduraufrufs bereitgestellt und nach Beendigung der Prozedur zerstört, d. h. stehen nicht mehr zur Verfügung. Daraus folgt, daß beim Aufruf einer Prozedur insbesondere der Wert einer lokalen Variablen aus dem vorausgegangenen Aufruf derselben Prozedur nicht mehr zur Verfügung steht.

Gültigkeitsbereich und Lebensdauer fallen oft nicht zusammen. Die `integer`-Variable `j` aus Beispiel 5.2.2, die lokal zum Programm `P` deklariert ist, lebt während des gesamten Programmablaufs, ist aber in der Prozedur `R` nicht gültig, da hier eine lokale `real`-Variable mit demselben Bezeichner deklariert ist.

Beispiel 5.2.3

Wir betrachten das folgende Programm D in Abbildung 5.7 mit zwei Prozeduren E und F. Zur Verdeutlichung sind die Blöcke der statischen Blockstruktur eingezeichnet.

```

program D (input, output);

  var
    i,
    j : integer;

  procedure F;

    var
      j : real;

    begin
      j := 16.12
    end; { F }

  procedure E (inA : integer; var ioB : integer);

    const
      i = 16;

    begin
      F;
      inA := i;
      ioB := ioB + inA
    end; { E }

begin { D }
  i := 3;
  j := 26;
  E (i, j);
  writeln ('i: ', i:3, ' j: ', j:3)
end. { D }

```

Abbildung 5.7 : statische Blockstruktur und Namenskonflikte im Programm D

Die in diesem Programm verwirrenden Namenskonflikte sowie die Zuweisung eines Wertes an den Wertparameter `inA` in der Prozedur E sind in Pascal zulässig, dokumentieren aber keinen guten Programmierstil und dienen lediglich der Veranschaulichung dynamischer Blockstrukturen.

Die dynamische Blockstruktur ändert sich bei jedem Aufruf oder Verlassen einer Prozedur oder Funktion. Wir geben im folgenden exemplarisch die dynamische Blockstruktur zu drei Zeitpunkten an:

1. Die dynamische Blockstruktur unmittelbar vor Aufruf der Prozedur E im Hauptprogramm stellt sich graphisch wie folgt dar:

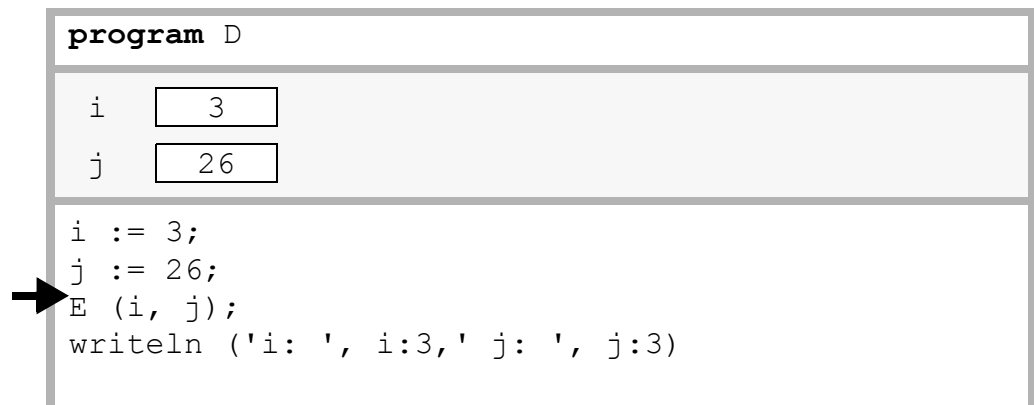


Abbildung 5.8 : dynamische Blockstruktur (1. Zeitpunkt)

Zu diesem Zeitpunkt existiert nur der Block des Programms D mit den Variablen i und j. Sie haben die Werte 3 bzw. 26, da die Zuweisungen bereits erfolgt sind.

2. Die dynamische Blockstruktur unmittelbar nach Aufruf der Prozedur F innerhalb der Prozedur E zeigt Abbildung 5.9.

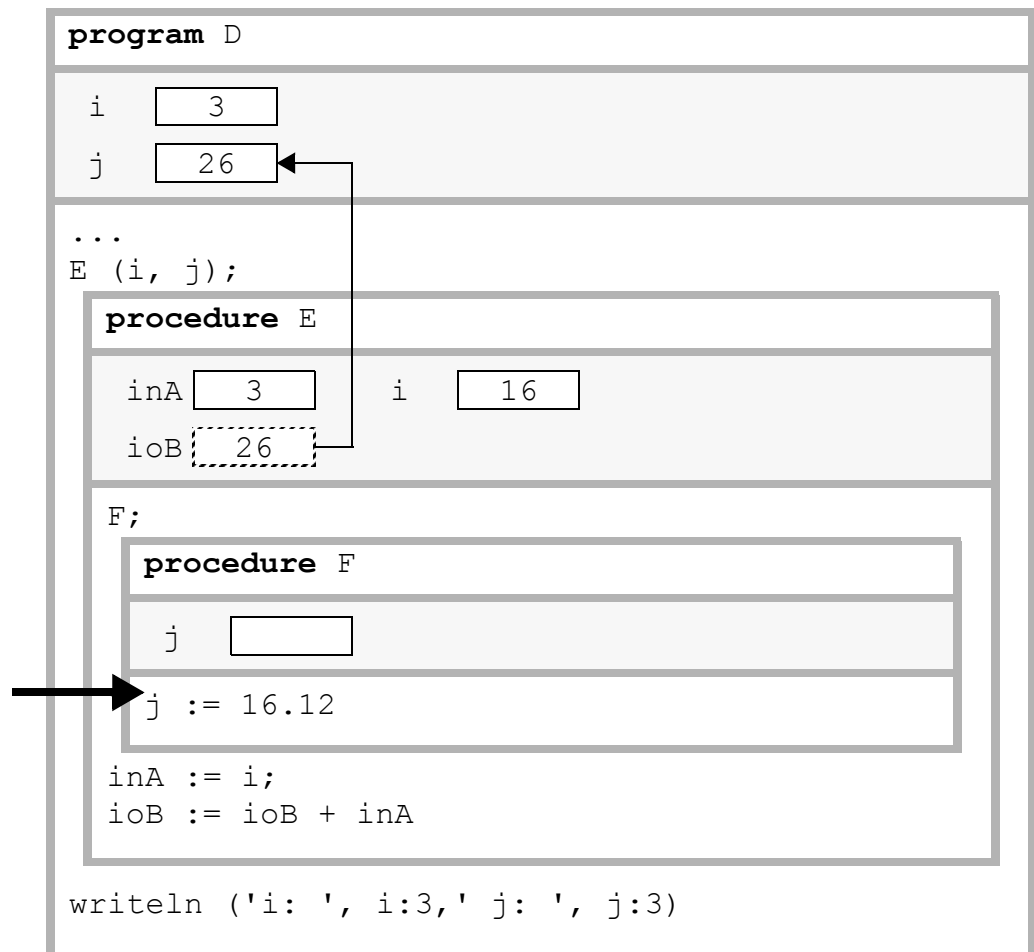


Abbildung 5.9 : dynamische Blockstruktur (2. Zeitpunkt)



Jetzt existieren drei Blöcke. Beachten Sie, daß deren Schachtelung eine andere als bei der statischen Blockstruktur ist. Bei Aufruf der Prozedur E wurde der innere Block mit den formalen Parametern `inA` und `ioB` sowie der lokalen Konstanten `i` angelegt. `inA` ist ein Wertparameter, er hat den Wert des aktuellen Parameters `i` (Variable im Programm D). `ioB` ist ein Referenzparameter mit einem Verweis auf den aktuellen Parameter `j` im Programm D.

Im Block zu E wird durch den Aufruf der Prozedur F ein weiterer Block geschachtelt. Dessen lokale Variable `j` besitzt noch keinen Wert, da die Zuweisung noch nicht erfolgt ist. In der nächsten Anweisung wird ihr der Wert `16.12` zugewiesen, bei Verlassen der Prozedur wird sie zerstört.

3. Die dynamische Blockstruktur unmittelbar vor Beendigung des Programms zeigt Abbildung 5.10:

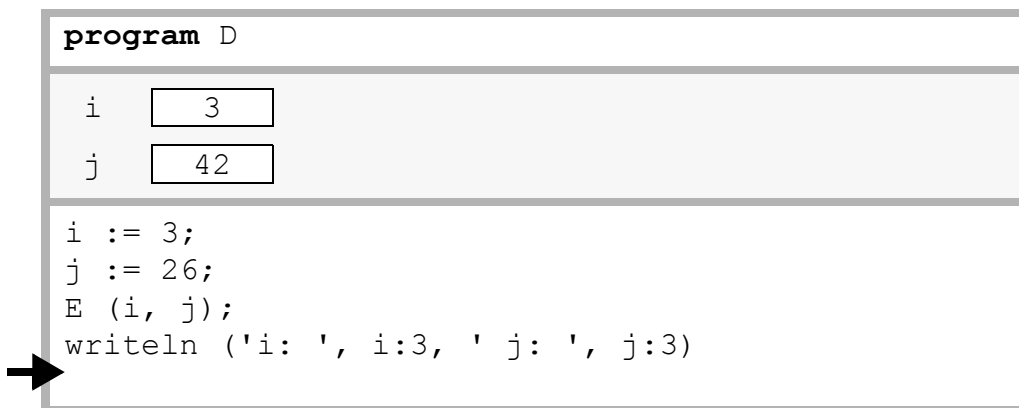


Abbildung 5.10 : dynamische Blockstruktur (3. Zeitpunkt)

Jetzt existiert nur noch der Block des Programms D. Die Variable `i` hat noch den Wert 3, da sie bei Aufruf der Prozedur E als Wertparameter übergeben wurde. Die Variable `j` wurde als Referenzparameter an die Prozedur E übergeben und hat jetzt den Wert 42. Die Auswertung des Ausdrucks `ioB + inA` in der Prozedur E ergibt 42, da dem formalen Parameter `inA` zuvor der Wert der lokalen Konstanten `i` zugewiesen wurde. Das Programm hat also die Werte 3 und 42 ausgegeben. □

Die Beschränkung der Lebensdauer von Variablen und Konstanten ermöglicht ein effizientes Speichermanagement. Der Speicher für eine lokale Variable muß nur für die Dauer der Ausführung einer Prozedur bereitgestellt werden und kann danach für lokale Variablen anderer Prozeduren verwendet werden. Das wirkt sich vorteilhaft auf den Speicherplatzbedarf eines Programms aus.

Wir möchten die Ausführungen über die statische und dynamische Blockstruktur nicht weiter vertiefen. Probleme können nur bei Namenskonflikten auftreten. Wir halten die mehrfache Vereinbarung desselben Bezeichners mit unterschiedlicher Bedeutung ohnehin für schlechten Programmierstil, da durch solche "Fallen" die Fehleranfälligkeit wächst und die Lesbarkeit leidet. Ausgenommen hiervon sind simple lokale Hilfsvariablen wie z.B. Laufvariablen oder Variablen für temporäre

Zwischenspeicherungen (z.B. Tausch in der Prozedur vertauschen), für die man sich nicht immer neue Bezeichner ausdenken möchte.

Dies wird augenfällig, wenn wir uns das Programm `FeldSort3` in Beispiel 5.2.1 (oder die Vorgängervarianten `FeldSort`, `FeldSort1` und `FeldSort2`) daraufhin anschauen. In `FeldSort3` haben wir verschiedene Bezeichner für die drei auftretenden Laufvariablen gewählt: `idx` (im Programmblock), `i` (in `FeldSortieren`) und `j` (in `FeldMinimumPos`). Haben wir es mit noch mehr Prozeduren bzw. Funktionen zu tun, gehen uns schnell die vernünftigen Bezeichner aus. Für `FeldSort3` wäre der Bezeichner `i` oder `j` für alle drei Laufvariablen sinnvoller und wahrscheinlich sogar ausdrucksstärker gewesen.

### 5.3 Hinweise zur Verwendung von Prozeduren und Funktionen

Im Verlauf des Kurses werden Sie noch viele Beispiele für Funktionen und Prozeduren kennenlernen, so daß wir unsere Ausführungen zu diesem Thema abschließen können. Allerdings wollen wir nicht versäumen, einige Hinweise zur Verwendung dieser beiden zentralen Programmierkonstrukte zu geben. Wie üblich gibt es keine "Betonregeln", die immer, d.h. unabhängig von der aktuellen Situation, gültig sind und stur angewendet werden können. Die folgenden *Merkregeln* sind also unter diesem Vorbehalt zu verstehen. Aus Vereinfachungsgründen benutzen wir wieder das Wort "Prozedur" stellvertretend für "Funktion bzw. Prozedur".

Merkregeln

Mehrfach-  
verwendung

Coderedundanz

Programmzerlegung

Schnittstelle

versteckte  
globale Variablen

1. Der Einsatz einer Prozedur empfiehlt sich mindestens dann, wenn derselbe (Teil-)Algorithmus an mehreren Stellen im Programm benötigt wird. Eine solche *Mehrfachverwendung* zeigt das Beispiel 4.3.2 aus Kapitel 4 zur Primzahlbestimmung, wo der Primzahltest an drei Stellen im Programm durchzuführen ist. Die Verwendung einer Prozedur vermeidet *Coderedundanz*, was das Programm verkürzt und die Übersichtlichkeit erhöht. Vor allem wird das Programm änderungsfreundlicher, da eine mögliche Änderung des entsprechenden Teilalgorithmus (z.B. eine effizientere Implementierung des Primzahltests) zu einer Programmänderung an nur einer einzigen Stelle führt.
2. Aber auch ohne das Argument der Mehrfachverwendung kann der Einsatz einer Prozedur vernünftig sein. Prozeduren erlauben eine wohldefinierte *Programmzerlegung*, welche den Prozeß der Programmkonstruktion erleichtert und gleichzeitig Übersichtlichkeit und Lesbarkeit verbessert. "Wohldefiniert" bedeutet dabei, daß die einzelnen Programmkomponenten (= Prozeduren) in sich abgeschlossene und selbständige Einheiten bilden, die (ausschließlich) über ihre *Schnittstellen* (= Parameterlisten) miteinander kommunizieren. Wir können also eine Prozedur wie ein selbständiges Programm betrachten, d.h. unabhängig von anderen Komponenten entwerfen und implementieren. Das Zusammenspiel der Komponenten, aus dem sich schließlich das Gesamtprogramm ergibt, wird durch die Semantik der Parameterübergabearten geregelt. Die Grundidee der unabhängigen Zerlegung funktioniert aber nur, wenn keine *"versteckte" Manipulation*, d.h. keine Manipulation an den Schnittstellen vor- bei, vorkommt. Das schließt insbesondere die Verwendung *globaler Variablen* aus. Um dies sicherzustellen, muß jede Information, die in einer Prozedur verwendet, manipuliert oder erzeugt wird, über die Prozedurschnittstelle, d.h.

explizit und sichtbar, hinein- und heraustransportiert werden (vgl. dazu auch den Abschnitt über "*Seiteneffekte*" in den Programmierstilausführungen).

Richtig angewendet unterstützen Prozeduren die Idee der unabhängigen Zerlegung und ermöglichen korrektere und besser strukturierte Programme sowie eine koordinierte Arbeitsteilung unter mehreren Programmierern. Die konsequenteste und mächtigste Form der unabhängigen Zerlegung wird durch das Programmkonstrukt *Modul* möglich. Ein Modul besteht aus einer Bündelung von Prozeduren und zugehörigen Daten und kommuniziert ebenfalls über eine wohldefinierte Schnittstelle mit anderen Komponenten. Dieses mächtigste Strukturierungskonzept imperativer Programmiersprachen existiert in Standard-Pascal nicht, wohl aber in MODULA-2, ADA und sogar (in eingeschränkter Form) in Turbo-Pascal. Wir werden in einem späteren Kapitel darauf zurückkommen.

3. Prozeduren unterstützen die *Wiederverwendung* von Programmkomponenten in verschiedenen Programmen. Die einfachsten Beispiele bilden die in Pascal vordefinierten Standardfunktionen wie z.B. `abs`, `sqr`, `succ`, `ord` sowie die Standardprozeduren `read` und `write`. Jeder Programmierer kann und sollte sich eine *Bibliothek* von Prozeduren einrichten, die er typischerweise für seinen Anwendungsbereich benötigt. (Im Fall einer Softwarefirma sollte eine einzige Bibliothek angelegt werden, die von allen Programmierern gemeinsam benutzt wird.) Diese vorgefertigten Bausteine erleichtern und beschleunigen nicht nur die Programmentwicklung, sondern unterstützen auch die Fehlerfreiheit des Programms (vorausgesetzt, die Bausteine selbst sind fehlerfrei). Eine solche Bibliothek könnte z.B. die Prozedur `FeldSortieren` aus dem Programm `FeldSort4` (vgl. Aufgabe 5.1.2.1) enthalten, mit der Felder (von `integer`-Zahlen) unterschiedlicher Länge sortiert werden können. Die Wiederverwendung solcher Bausteine funktioniert aber nur effizient, wenn sämtliche benötigte Information über die Baustein-Schnittstelle fließt und keine Annahmen über globale Variablen getroffen werden müssen. Wäre das nämlich der Fall, dann müßte der Programmierer zunächst den Programmcode des Bausteins aus der Bibliothek untersuchen, um die *global benutzten Variablen* zu finden, damit er sie in seinem Programm entsprechend deklariert. Bei Vermeidung globaler Variablen genügt ein Blick auf die Schnittstelle, um die benötigte Information zu identifizieren; das lästige, mit Fehlern behaftete Lesen des Programmcodes entfällt (vgl. dazu auch den Abschnitt über "*Seiteneffekte*" in den Programmierstilausführungen).
4. Zum Abschluß noch eine Bemerkung zur *effizienten Benutzung der Parameterübergabearten*. Bei `in`-Parametern bzw. Wertparametern werden in der Prozedur bei jedem Aufruf Kopien der aktuellen Parameter angelegt. Ist ein solcher Parameter ein Feld nichttrivialer Größe, so wird erheblicher zusätzlicher Speicherplatz für die Kopie, aber auch erheblicher zusätzlicher Zeitaufwand für das Kopieren selbst benötigt. Daher wollen wir Felder, die von der Programmlogik her als `in`- bzw. Wertparameter vorzusehen wären, zukünftig als `inout`- bzw. Referenzparameter deklarieren. Bei dieser speicher- und zeiteffizienten Maßnahme ist aber besondere Disziplin und Aufmerksamkeit angebracht, um versehentliche Manipulationen der nicht mehr geschützten aktuellen Parameter zu vermeiden. Wir werden daher in einem solchen Fall in Pascal-Prozeduren weiterhin grundsätzlich die Vorsilbe `in` vor den Bezeich-

Seiteneffekte

Modulkonzept

Wiederverwendung

Bibliothek

globale Variablen

Seiteneffekte  
effiziente Benutzung  
der Übergabearten

ner schreiben, um den eigentlichen Charakter der Übergabeart deutlich zu signalisieren.

## 5.4 Dynamische Datenstrukturen

### 5.4.1 Zeiger

Referenzkonzept,  
Zeigerkonzept  
dynamische  
Datenstrukturen

Verweis

Speicheradresse

Zeigervariable

Dereferenzierung

In diesem Abschnitt lernen wir ein weiteres wichtiges Programmiersprachenkonzept kennen, das *Referenz- oder Zeigerkonzept*. Mit Hilfe von Zeigern werden *dynamische Datenstrukturen* realisiert, die das Speichern und Bearbeiten von Objekten erlauben, deren Anzahl während der Programmausführung wachsen und schrumpfen kann. Dem Zeigerkonzept liegt die Idee zugrunde, Objekte anstatt über ihren Namen über *Verweise* (Referenzen) anzusprechen. Das Verweisprinzip ist Ihnen bereits von Referenzparametern bei Pascal-Prozeduren bekannt, aber auch als Bestandteil des täglichen Lebens geläufig. Sie kennen Verweise auf Textstellen, Aktenzeichen können wir als Verweise interpretieren, das Geländespiel „Schnitzeljagd“ benutzt Verweise usw. Technisch gesehen besteht der Verweis auf ein Objekt aus der *Speicheradresse* des Objektes. Speicheradressen werden vom Computer automatisch vergeben und sind nur intern bekannt. Wir können Verweise nicht manipulieren, wohl aber zuweisen und vergleichen. So können wir z.B. den Computer nicht anweisen, einen Verweis einzulesen oder auszugeben. Ein Objekt können wir über seinen zugehörigen Verweis sehr wohl manipulieren, und zwar über eine *Zeigervariable*, die den Verweis auf das Objekt als Wert besitzt. Einen Zugriff auf ein Objekt über eine Zeigervariable nennt man *Dereferenzierung* einer Zeigervariablen.

Abbildung 5.11 zeigt eine Zeigervariable `Zeig`, deren Wert ein Verweis auf ein Verbundobjekt ist. Das Verbundobjekt selbst ist als Rechteck dargestellt und seine Komponenten sind als innenliegende Rechtecke eingezeichnet.

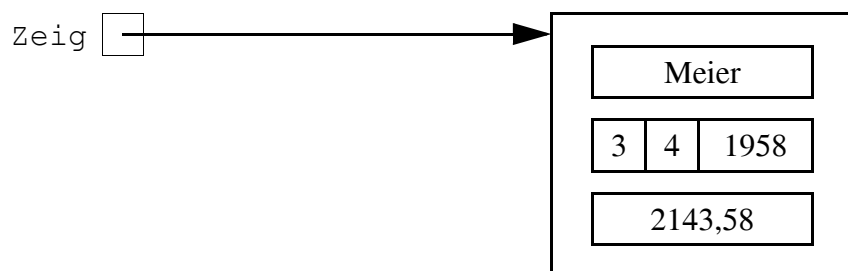


Abbildung 5.11 : Anschauliche Darstellung von Zeigervariablen

Definition eines  
Zeigertyps

Wie alle Variablen sind auch Zeigervariablen typisiert. Wir definieren einen *Zeigertyp* `tRefTyp` für den Typ `tTyp` durch

```
type
tRefTyp = ↑tTyp;
```

wobei `tTyp` ein beliebiger Datentyp ist.

Die Wertemenge von `tRefTyp` ist die Menge aller Verweise auf Objekte des Typs `tTyp`.

Sei `Zeig` eine Zeigervariable vom Typ `tRefTyp`, deklariert durch

```
var
  Zeig : tRefTyp;
```

Um das Objekt, auf das `Zeig` verweist, ansprechen und manipulieren zu können, verwenden wir die Schreibweise

`Zeig↑`

`Zeig↑` (sprich "*Zeig Dach*") ist das Objekt, auf das `Zeig` weist, und somit vom Typ `tTyp`. In Abbildung 5.11 ist `Zeig↑` gerade der Verbund, auf den `Zeig` zeigt, also:

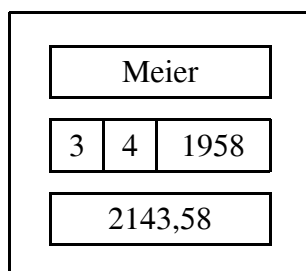


Abbildung 5.12 : Anschauliche Darstellung des Objektes `Zeig↑`

Für `Zeig↑` gelten alle "Rechte und Pflichten", die sich aus dem Wertebereich von `tTyp` und den möglichen Operationen auf `tTyp` ergeben. Ist beispielsweise `Zeig↑` ein Feld bzw. ein Verbund, so sprechen wir die Komponenten in der Form

`Zeig↑[i, j]` bzw. `Zeig↑.Name`

an (sprich "*Zeig Dach von i Komma j*" bzw. "*Zeig Dach Punkt Name*").

Da auf den meisten Computertastaturen das `↑`-Symbol nicht vorhanden ist, verwendet Pascal das sog. "Hochdach" `^` als Ersatzsymbol. Wir wollen dies auch in allen nachfolgenden Beispielen so halten.

#### Beispiel 5.4.1.1

Sei `tPerson` der folgende Verbund-Typ:

```
type
  tDatum = array [1..3] of integer;
  tPerson = record
    Name : string[10];
    GebDat : tDatum;
    Gehalt : real
  end;
```

Wir deklarieren zwei Zeigervariablen auf Objekte vom Typ `tPerson` durch:

Deklaration einer  
Zeigervariablen

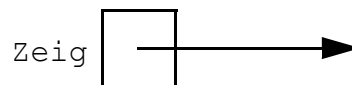
```

var
  Zeig1,
  Zeig2: ^tPerson;

```

Schlüsselwort **nil**Konstante **nil**

Nun muß ausgedrückt werden können, daß eine Zeigervariable auch (temporär) auf kein Objekt zeigen kann, also ein "leerer" Zeiger ist. Hierfür stellt Pascal das Schlüsselwort **nil** bereit. Indem wir einer Zeigervariablen **nil** zuweisen, stellen wir den leeren Zeiger her. Auf das Objekt, auf das die Zeigervariable vorher verwiesen hat, kann jetzt (über diese Zeigervariable) nicht mehr zugegriffen werden. Tatsächlich ist **nil** eine Konstante und sollte eigentlich **nil** geschrieben werden. In MODULA-2 ist dies übrigens der Fall. Die Pascal-Regeln sehen hierfür allerdings das Schlüsselwort **nil** vor. Wichtig für uns ist, daß **nil** von jedem Zeigertyp ist, also jeder beliebigen Zeigervariablen zugewiesen werden kann. Graphisch stellen wir Zeigervariablen mit Wert **nil** gemäß Abbildung 5.13 dar.

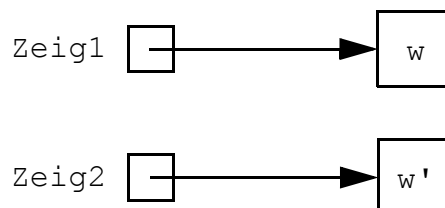
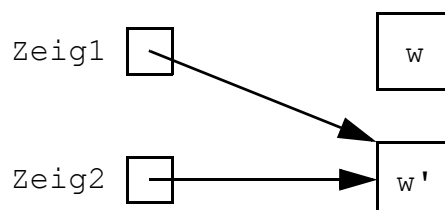
Abbildung 5.13 : Graphische Darstellung von Zeigervariablen mit Wert **nil**Operationen auf  
Zeigervariablen

Auf dem Zeigertyp  $^t\text{Typ}$  sind folgende vier Operationen zugelassen:

## a) Zuweisung.

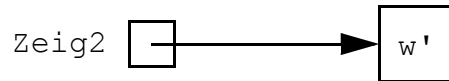
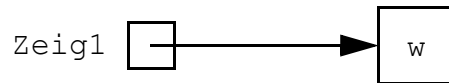
Einer Zeigervariablen **Zeig1** kann der Wert einer anderen Zeigervariablen **Zeig2** desselben Typs  $^t\text{Typ}$  oder **nil** zugewiesen werden. Abbildung 5.14 zeigt die Wirkung von **Zeig1 := Zeig2**. Beachten Sie, daß nach der Zuweisung nicht mehr von **Zeig1** (oder **Zeig2**) aus auf das Objekt mit Wert **w** zugegriffen werden kann.

vorher

nach der Zuweisung **Zeig1 := Zeig2**Abbildung 5.14 : Wirkung der Zuweisung **Zeig1 := Zeig2** bei Zeigervariablen

Mittels der Zuweisung  $\text{Zeig1}^{\wedge} := \text{Zeig2}^{\wedge}$  können wir dem Objekt  $\text{Zeig1}^{\wedge}$  den Wert von  $\text{Zeig2}^{\wedge}$  zuweisen.

vorher



nach der Zuweisung  $\text{Zeig1}^{\wedge} := \text{Zeig2}^{\wedge}$

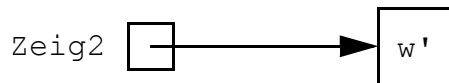
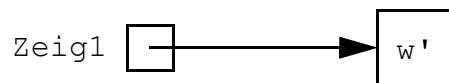


Abbildung 5.15 : Ergebnis der Zuweisung  $\text{Zeig1}^{\wedge} := \text{Zeig2}^{\wedge}$  bei Zeigervariablen

#### b) Erzeugen eines Objekts.

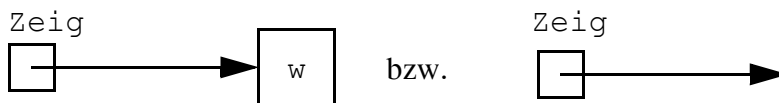
Sei  $\text{Zeig}$  eine Zeigervariable vom Typ  $\text{tTyp}$ . Der Aufruf

`new (Zeig)`

new

der Standardprozedur `new` erzeugt ein (neues) Objekt vom Typ  $\text{tTyp}$  und weist  $\text{Zeig}$  einen Zeiger auf dieses Objekt zu. Abbildung 5.16 zeigt die Situation vor und nach Ausführung von `new (Zeig)`. Der Wert von  $\text{Zeig}^{\wedge}$  ist solange undefiniert, bis  $\text{Zeig}^{\wedge}$  initialisiert wird. Wir deuten dies durch ein Fragezeichen an.

vorher



nach dem Aufruf `new (Zeig);`

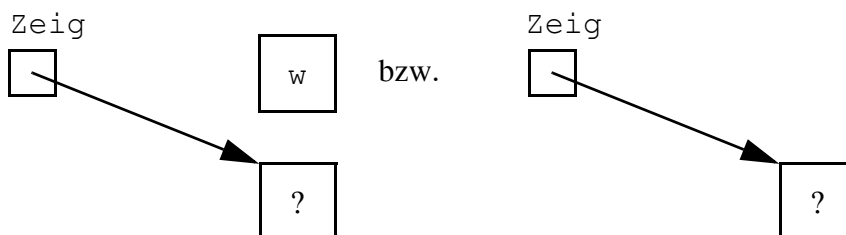


Abbildung 5.16 : Wirkung von `new (Zeig)`

dispose

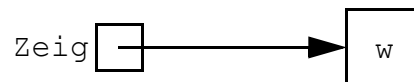
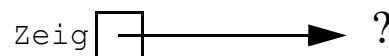
## c) Löschen eines Objekts.

Wird ein zuvor mit `new (Zeig)` erzeugtes Datenobjekt `Zeig^` nicht mehr benötigt, so kann es durch den Aufruf

```
dispose (Zeig)
```

der Standardprozedur `dispose` wieder gelöscht werden. Nach diesem Aufruf ist ein Zugriff auf das Objekt nicht mehr möglich und der Wert von `Zeig^` undefiniert. Dies gilt auch für alle anderen Zeigervariablen, die vor dem Aufruf `dispose (Zeig)` auf `Zeig^` verwiesen haben.

vorher

nach dem Aufruf `dispose (Zeig)`Abbildung 5.17: Wirkung von `dispose (Zeig)`Vergleich von  
Zeigervariablen

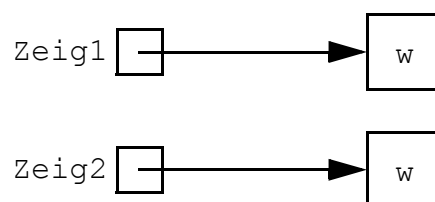
## d) Vergleich zweier Zeigervariablen.

Zwei Zeigervariablen `Zeig1` und `Zeig2` vom Typ `^tTyp` sind gleich (bzw. ungleich), d.h. der Ausdruck

```
Zeig1 = Zeig2 (bzw. Zeig1 ≠ Zeig2)
```

hat den Wert `true`, falls `Zeig1` und `Zeig2` auf dasselbe Objekt (bzw. auf verschiedene Objekte) zeigen oder beide gleich `nil` sind.

Machen Sie sich den Unterschied zwischen `Zeig1` und `Zeig1^` noch einmal anhand der Vergleichsoperationen klar: Aus `Zeig1 = Zeig2` folgt stets `Zeig1^ = Zeig2^`. Die Umkehrung gilt jedoch nicht. Wohl können die Werte der Objekte, auf die `Zeig1` und `Zeig2` zeigen, gleich sein (`Zeig1^ = Zeig2^`), es kann sich aber um verschiedene Objekte handeln (`Zeig1 ≠ Zeig2`). Abbildung 5.18 zeigt eine solche Situation:

Abbildung 5.18 : Situation, in der `Zeig1 ≠ Zeig2` und `Zeig1^ = Zeig2^ = w` gilt.Beispiel 5.4.1.2

Gegeben sei die Deklaration



```

var
  pZeig,
  qZeig,
  sZeig : ^integer;
  rZeig : ^real;

```

Die folgende zusammengesetzte Anweisung

```

begin
  { Erzeuge zwei neue Objekte }
  new (pZeig);
  new (qZeig);
  writeln (pZeig = qZeig);
  pZeig^ := 3;
  qZeig^ := 7;
  qZeig^ := qZeig^ - pZeig^ - 1;
  { jetzt hat qZeig^ den Wert 7-3-1=3 }
  writeln (pZeig = qZeig, ' ', pZeig^ = qZeig^);
  new (rZeig);
  rZeig^ := 5.0 * (qZeig^ + 2.0) / pZeig^;
  writeln (rZeig^:5:3);
  qZeig^ := 2 * qZeig^;
  { jetzt hat qZeig^ den Wert 6 }
  sZeig := qZeig;
  qZeig := pZeig;
  pZeig := sZeig;
  writeln (pZeig^, ' ', qZeig^, ' ', sZeig^);
end

```

liefert die Ausgabe

```

false
false true
8.333
6 3 6

```



### 5.4.2 Lineare Listen

Mit Hilfe von Zeigern lassen sich dynamische Datenstrukturen realisieren, in denen Objekte gespeichert werden, deren Anzahl sich während der Programmausführung ("dynamisch") verändert. Es können also Objekte während der Laufzeit erzeugt (new) und zerstört (dispose) werden. Bisher kennen wir nur "statische" Datentypen, wie z.B. das Feld oder den Verbund. Beim Feld ist die Anzahl der Elemente durch die Typdeklaration festgelegt und kann nicht mehr verändert werden. Dasselbe gilt für die Komponenten (Struktur) eines Verbundes.

Eine einfache dynamische Datenstruktur ist die lineare Liste. Betrachten wir dazu folgende Typdefinition

```

type
tRefListe = ^tListe;
tListe = record
    info : integer;
    next : tRefListe
end;

```

lineare Liste

Ein Verbundobjekt vom Typ `tListe` besteht aus einer `integer`-Zahl nebst einem Verweis `next` auf ein Objekt vom Typ `tListe`. Durch Verketteten von Objekten des Typs `tListe` erhalten wir eine *lineare Liste* (von Objekten des Typs `tListe`), wie es die folgenden Abbildungen 5.19, 5.20 und 5.21 zeigen.

Vorwärtsdefinition

Die sonst vorgeschriebene Reihenfolge von Definitionen kann und muß für Zeigertypen (und nur für diese) nicht eingehalten werden. Die Typdefinition `tRefListe = ^tListe` verwendet den Typ (bezeichner) `tListe`, der erst danach definiert (erklärt) wird. Bei einer solchen "Vorwärtsdefinition" muß aber der vorweg benutzte Typ (`tListe`) im gleichen Typdefinitionsteil definiert sein. Wir werden ihn stets unmittelbar nach seiner ersten Verwendung definieren.

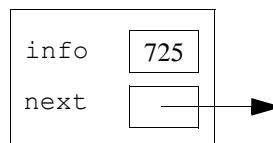


Abbildung 5.19 : lineare Liste mit einem Element

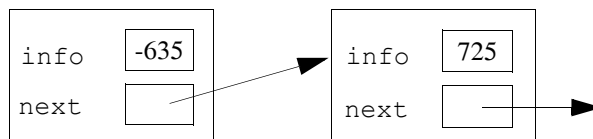


Abbildung 5.20 : lineare Liste mit zwei Elementen

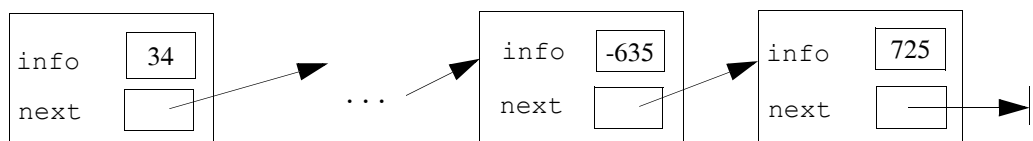


Abbildung 5.21 : Lineare Liste mit "beliebig vielen" Elementen

Um mit Listen arbeiten zu können und auch leere Listen, die kein Listenobjekt enthalten, zuzulassen, benötigen wir eine Zeigervariable `RefAnfang`, die auf das er-

ste Listenobjekt zeigt oder aber gleich **nil** ist, falls die Liste leer ist, d. h. kein Listenobjekt enthält. Wir deklarieren diesen *Anfangszeiger* oder *Anker* der Liste durch

```
var
  RefAnfang : tRefListe;
```

Abbildung 5.22 zeigt eine lineare Liste mit der Variablen RefAnfang als Anfangszeiger. Der Zugriff auf ein bestimmtes Listenelement beginnt mit dem Anfangszeiger und verläuft über das erste, zweite, dritte usw. Element bis das gewünschte Element erreicht ist.

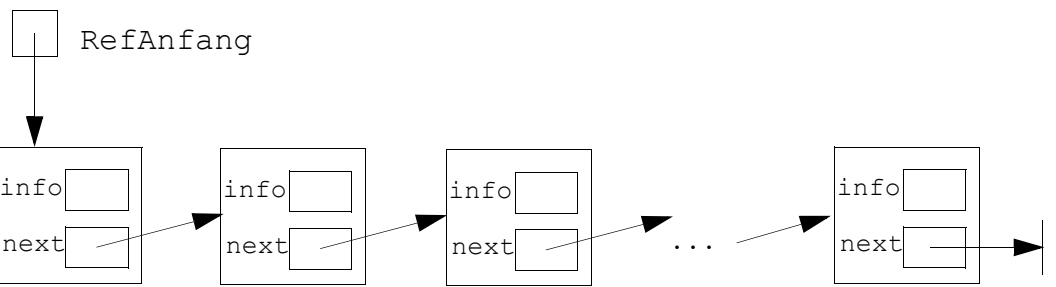


Abbildung 5.22 : lineare Liste mit Anfangszeiger RefAnfang

Für zwei typgleiche Zeigervariablen Zeigl und RefAnfang zeigt Abbildung 5.23 die Auswirkung der Zuweisung

```
Zeigl := RefAnfang
```

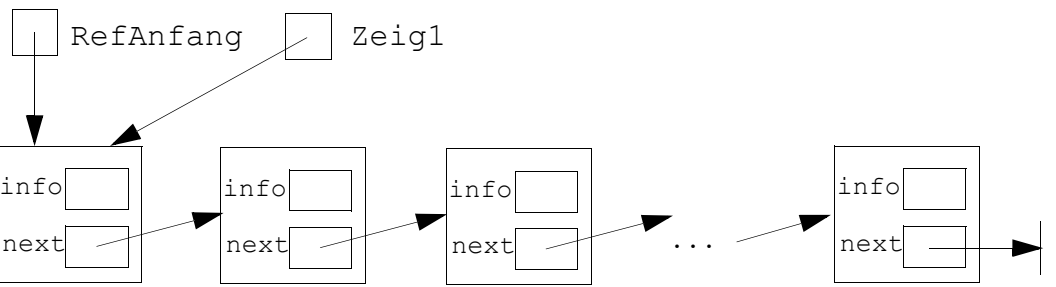


Abbildung 5.23 : Auswirkung der Zuweisung Zeigl := RefAnfang

und in Abbildung 5.24 ist die Auswirkung der Zuweisung

```
Zeigl := RefAnfang^.next^.next
```

dargestellt.

Anfangszeiger  
Anker

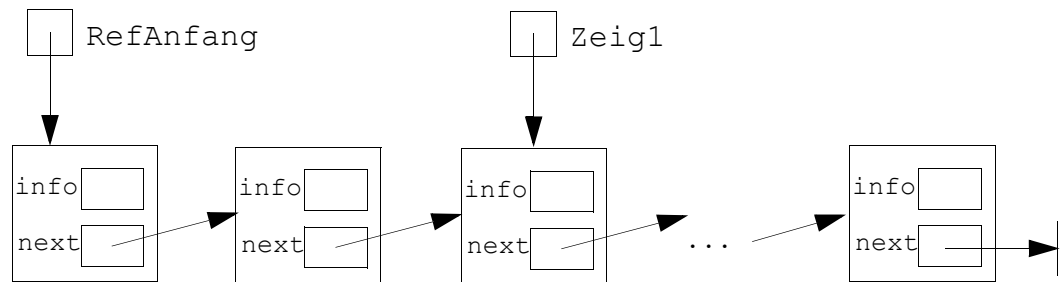


Abbildung 5.24: Auswirkung der Zuweisung  $\text{Zeigl} := \text{RefAnfang}^{\text{next}}.^{\text{next}}$

typische Operationen  
auf Listen

In den folgenden Beispielen werden wir eine Reihe *typischer Operationen auf Listen* vorstellen. Wir behandeln eine lineare Liste von *integer-Zahlen* und benutzen dazu den am Anfang des Kapitels eingeführten Datentyp `tRefListe`:

```

type
tRefListe = ^tListe;
tListe = record
    info : integer;
    next : tRefListe
end;

```

#### Beispiel 5.4.2.1

Aufbau einer  
linearen Liste

Zum Aufbau einer linearen Liste benutzen wir eine Prozedur, die von Null verschiedene *integer-Zahlen* einliest und in eine anfangs leere Liste einfügt. Das Ende der Eingabe wird durch eine Null signalisiert. Beachten Sie, daß eine neue Zahl stets am Anfang der Liste eingefügt wird und sich dadurch die Reihenfolge der Zahlen umkehrt. Dies ist typisch, da das Einfügen am Anfang besonders einfach und i.a. eine bestimmte Reihenfolge in der Liste nicht erforderlich ist (meist soll nur eine Kollektion von Objekten gespeichert werden, d.h. Wiederholungen sind erlaubt).

```

procedure ListeAufbauen(var outRefAnfang : tRefListe);
{ baut eine Liste aus einzulesenden integer-Zahlen
  auf }

var
    Zeiger : tRefListe;
    Zahl : integer;

begin
    { zunaechst outRefAnfang auf nil setzen, da mit
      der leeren Liste gestartet wird }
    outRefAnfang := nil;
    readln (Zahl);
    while Zahl <> 0 do
        begin
            new (Zeiger);

```

```

Zeiger^.info := Zahl;
Zeiger^.next := outRefAnfang;    { * }
outRefAnfang := Zeiger;          { ** }
readln (Zahl)
end { while-Schleife }
end; { ListeAufbauen }

```

Abbildung 5.25 zeigt den Anfangszeiger vor der ersten Eingabe, den Anfangszeiger und das Listenelement nach der Eingabe einer 3 und schließlich den Anfangszeiger und die zwei Listenelemente nach der Eingabe einer 5.

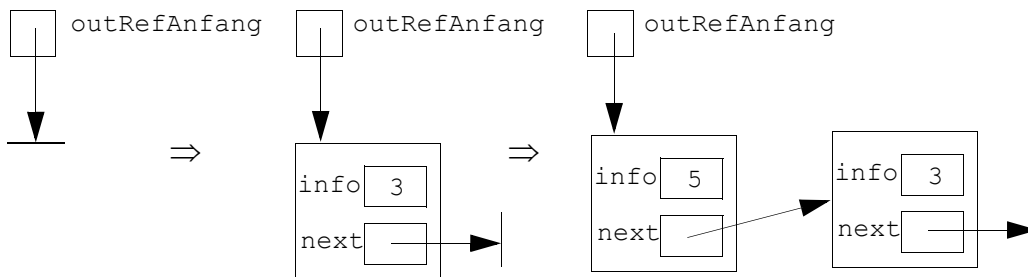


Abbildung 5.25 : Liste vor der ersten, nach der ersten und nach der zweiten Eingabe

Ein typischer Fehler ist, die Anweisungen { \* } und { \*\* } der Prozedur `ListeAufbauen` zu vertauschen. Es werden dann nämlich die erzeugten Listenelemente nicht miteinander verkettet. Diesen Effekt veranschaulicht Abbildung 5.26, welche die entstandenen Objekte für die in Abbildung 5.25 angenommene Eingabefolge zeigt.

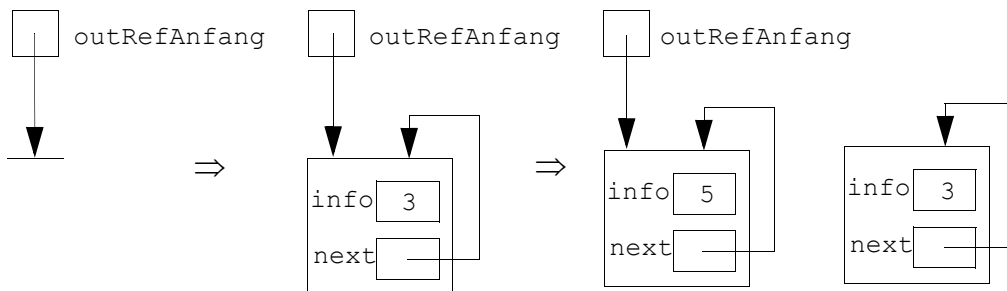


Abbildung 5.26 : Fehlerhafte Verkettung der Listenelemente



#### Beispiel 5.4.2.2

Für eine lineare Liste mit Anker `inRefAnfang` soll der Wert der `info`-Komponente jedes Elementes ausgedruckt werden. Ist die Liste leer, so gilt `inRefAnfang = nil`.

```

procedure ListeDurchlaufen (inRefAnfang : tRefListe);
{ gibt die Werte der Listenelemente aus }

```

Durchlaufen einer  
linearen Liste

```

var
  Zeiger : tRefListe;

begin
  Zeiger := inRefAnfang;
  while Zeiger <> nil do
    begin
      writeln (Zeiger^.info);
      Zeiger := Zeiger^.next
    end
  end; { ListeDurchlaufen }

```



### Beispiel 5.4.2.3

Suchen eines Elementes in einer linearen Liste

Eine Funktion soll feststellen, ob ein Element, dessen `info`-Komponente gleich `inZahl` ist, in der Liste mit Anker `inRefAnfang` vorkommt. Gibt es ein solches Element, dann wird ein Verweis auf das Element zurückgegeben, andernfalls **nil**. Gibt es mehrere solcher Elemente, dann wird das in der Reihenfolge der Verkettung erste Element bestimmt.

Variante 1:

```

function ListenElemSuchen1 (
  inRefAnfang : tRefListe;
  inZahl : integer): tRefListe;
{ bestimmt das erste Element in einer Liste, bei dem
  die info-Komponente gleich inZahl ist }

  var
    Zeiger : tRefListe;

  begin
    Zeiger := inRefAnfang;
    if Zeiger <> nil then
      { fuer Zeiger = nil ist die Liste leer und das
        gesuchte Element existiert nicht }
      while (Zeiger <> nil) and
        (Zeiger^.info <> inZahl) do
        Zeiger := Zeiger^.next;
      ListenElemSuchen1 := Zeiger
    end; { ListenElemSuchen1 }

```

Die **while**-Schleife hat die Wirkung, daß, solange `Zeiger <> nil` ist, d. h. das Listenende noch nicht erreicht ist, und `Zeiger^.info <> inZahl`, d. h. das gesuchte Element noch nicht gefunden ist, `Zeiger` auf `Zeiger^.next` weitergesetzt wird.

Variante 1 ist zwar suggestiv, aber leider in Standard-Pascal falsch. Falls nämlich `Zeiger = nil` wird, also das Listenende erreicht ist, wird weiterhin noch `Zeiger^.info <> inZahl` überprüft, und das führt zu einem Programmabsturz, da das Element `Zeiger^` für `Zeiger = nil` nicht existiert. In Turbo-Pascal würde hier übrigens, falls `Zeiger = nil`, die Auswertung des logischen Ausdrucks mit dem Wert `false` abgebrochen und somit wäre die obige Variante korrekt. Es ist aber kein guter Programmierstil, spezielle Eigenschaften von Sprachdialekten auszunutzen. Bei einer Portierung des Programms in eine andere Sprachumgebung führt dies zu lästigen und überflüssigen Änderungen.

Variante 2:

Wir erhalten im wesentlichen durch Umformulierung der Bedingung der **while**-Schleife eine korrekte Funktion.

```
function ListenElemSuchen2 (
    inRefAnfang : tRefListe;
    inZahl : integer): tRefListe;
{ bestimmt das erste Element in einer Liste, bei dem
  die info-Komponente gleich inZahl ist }

var
    Zeiger : tRefListe;

begin
    Zeiger := inRefAnfang;
    if Zeiger <> nil then
        { Liste nicht leer }
        begin
            while (Zeiger^.next <> nil) and
                (Zeiger^.info <> inZahl) do
                Zeiger := Zeiger^.next;
            if Zeiger^.info <> inZahl then
                { dann muss Zeiger^.next = nil gewesen sein,
                  d.h. Zeiger zeigt auf das letzte Element.
                  Da dessen info-Komponente <> inZahl ist,
                  kommt inZahl nicht in der Liste vor. }
                Zeiger := nil
            end;
            ListenElemSuchen2 := Zeiger
        end; { ListenElemSuchen2 }
```



#### Beispiel 5.4.2.4

Eine Liste ist (aufsteigend) sortiert, wenn die `info`-Komponenten der Listenelemente in der Reihenfolge der Verkettung (aufsteigend) sortiert sind. Abbildung 5.27 zeigt eine aufsteigend sortierte Liste:

Sortierte Liste

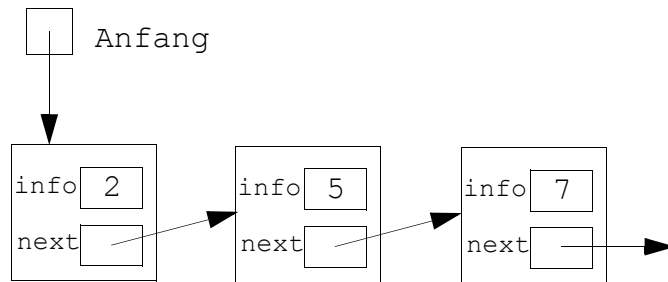


Abbildung 5.27 : Sortierte Liste

Die Prozedur `SortiertEinfuegen` fügt in eine aufsteigend sortierte Liste mit Anfangszeiger `ioRefAnfang` ein neues Element mit `info`-Komponente `inZahl` so ein, daß die entstehende Liste ebenfalls aufsteigend sortiert ist.

Anhand der Prozedur `SortiertEinfuegen` verdeutlichen wir ein typisches *Muster zur Bearbeitung linearer Listen*:

Zuerst werden die **Sonderfälle**

- leere Liste und
- Operation erfolgt am Listenanfang

behandelt, dann der **Normalfall**

- Operation erfolgt innerhalb der Liste.

Manchmal muß auch noch der Fall

- Operation erfolgt am Ende der Liste
- gesondert behandelt werden.

```

procedure SortiertEinfuegen (
    inZahl : integer;
    var ioRefAnfang : tRefListe);
{ fuegt ein neues Element fuer inZahl in eine
  sortierte Liste ein }

var
  Zeiger,
  RefNeu : tRefListe;
  gefunden : boolean;

begin
  { neues Element erzeugen }
  new (RefNeu);
  RefNeu^.info := inZahl;
  if ioRefAnfang = nil then
  { Sonderfall: Liste ist leer }
  begin
    RefNeu^.next := ioRefAnfang;
    ioRefAnfang := RefNeu
  end

```



```

else
  if ioRefAnfang^.info > inZahl then
    { Sonderfall: Einfuegen am Listenanfang }
    begin
      RefNeu^.next := ioRefAnfang;
      ioRefAnfang := RefNeu
    end
  else
    { Einfuegeposition suchen }
    begin
      gefunden := false;
      Zeiger := ioRefAnfang;
      while (Zeiger^.next <> nil) and
        (not gefunden) do
        if Zeiger^.next^.info > inZahl then
          gefunden := true
        else
          Zeiger := Zeiger^.next;
        if gefunden then
          { Normalfall: Einfuegen in die Liste }
          begin
            RefNeu^.next := Zeiger^.next;
            Zeiger^.next := RefNeu
          end
        else
          { Sonderfall: Anhaengen an das Listenende }
          begin
            Zeiger^.next := RefNeu;
            RefNeu^.next := nil
          end
        end
      end
    end; { SortiertEinfuegen }

```

Überzeugen Sie sich davon, daß doppelte Werte richtig (an das Ende der Teilliste aller Elemente mit gleichem Wert) eingefügt werden.



#### Beispiel 5.4.2.5

Es soll das Element, dessen info-Komponente gleich inZahl ist, aus der Liste entfernt werden. Gibt es mehrere solcher Elemente, dann wird das in der Reihenfolge der Verkettung erste Element entfernt.

Betrachten wir dazu zunächst isoliert den eigentlichen Entfernungsprozeß, und nehmen wir an, daß in der in Abbildung 5.28 gezeigten Liste das Element b entfernt werden soll.

Entfernen eines Elementes aus einer linearen Liste

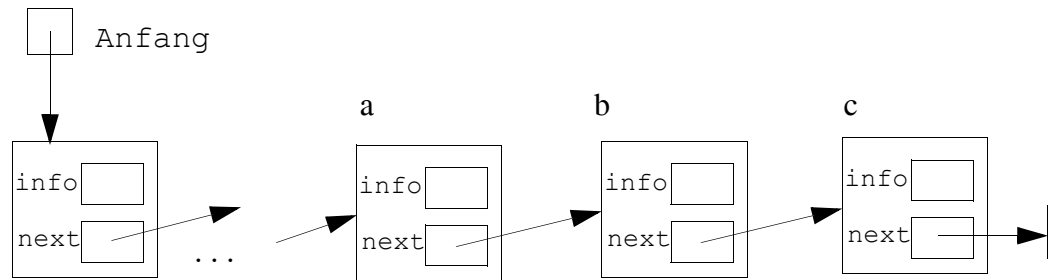


Abbildung 5.28 : lineare Liste

Dazu ändern wir den Verweis im Vorgänger a von Element b, so daß der Verweis von a anschließend auf das Element c zeigt. Wir erhalten die in Abbildung 5.28 gezeigte Liste.

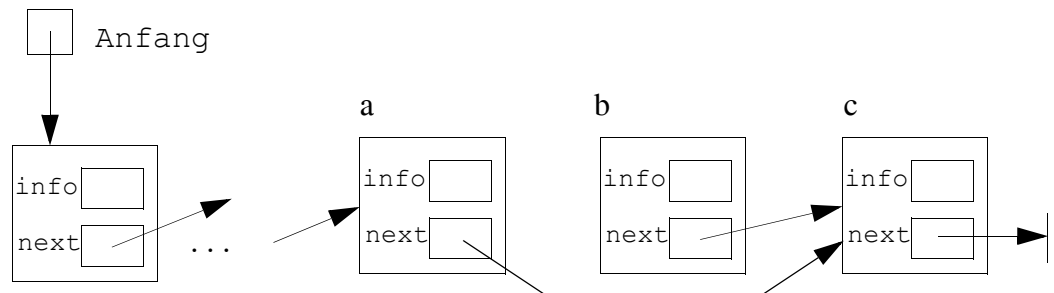


Abbildung 5.29 : lineare Liste nach dem Abhängen von b

Um b entfernen zu können, müssen wir also den Zugriff auf das Vorgängerelement a haben, um dessen Verweis abzuändern. Der erste Sonderfall liegt vor, wenn die Liste leer ist. Dann wird die Liste nicht verändert. Ist das erste Element zu entfernen, so liegt der zweite Sonderfall vor und es muß der Anfangszeiger adjustiert werden, denn er muß anschließend auf das zweite Element zeigen. Der Anfangszeiger spielt hier gewissermaßen die Rolle des Vorgängers. Da er sich ändern kann, muß er als **var**-Parameter deklariert werden. Kommt das gesuchte Element nicht in der Liste vor (dritter Sonderfall), so wird der boolesche **var**-Parameter **outGefunden** mit dem Wert **false** zurückgegeben, sonst mit dem Wert **true**. Wir benutzen eine Prozedur **abhängen**, um das Entfernen des gefundenen Elementes durch Abhängen vom Vorgängerzeiger vorzunehmen.

```
procedure ListenElemEntfernen (
    inZahl : integer;
    var ioRefAnfang : tRefListe;
    var outGefunden : boolean);
{ entfernt aus einer Liste mit Anfangszeiger
  ioRefAnfang das erste Element mit dem Wert inZahl.
  Bei erfolgreicher Entfernung wird outGefunden auf
  true gesetzt, sonst auf false }
```

```
var
Zeiger : tRefListe;
gefunden : boolean;

procedure Abhaengen (var ioZeig : tRefListe);
{ haengt das Element ab, auf das ioZeig zeigt }

    var
        HilfsZeig : tRefListe; { Hilfsvariable }

    begin
        HilfsZeig := ioZeig;
        ioZeig := ioZeig^.next;
        dispose (HilfsZeig)
    end; { Abhaengen }

begin
    if ioRefAnfang = nil then
        { Sonderfall: Liste ist leer }
        gefunden := false
    else
        if ioRefAnfang^.info = inZahl then
            begin
                { Sonderfall: erstes Element ist zu entfernen }
                Abhaengen (ioRefAnfang);
                gefunden := true
            end
        else
            begin { gesuchtes Element bestimmen }
                Zeiger := ioRefAnfang;
                gefunden := false;
                while (Zeiger^.next <> nil) and not gefunden do
                    if Zeiger^.next^.info = inZahl then
                        begin
                            Abhaengen (Zeiger^.next);
                            gefunden := true
                            { wurde das letzte Element entfernt,
                                dann gilt jetzt Zeiger^.next = nil und
                                die while-Schleife bricht ab }
                        end
                    else
                        { Sonderfall: gesuchtes Element kommt nicht
                            vor; wegen der Schleifenbedingung
                            gilt hier stets Zeiger^.next <> nil }
                        Zeiger := Zeiger^.next
                        { es ist hier Zeiger <> nil und damit
                            Zeiger^.next wohldefiniert }
                    end
                end; { Listendurchlauf }
```

```

    outGefunden := gefunden
end; { ListenElemEntfernen }

```



Beachten Sie, daß ohne den Aufruf `dispose (HilfsZeig)` das zu entfernende Element lediglich aus der Verkettung herausgenommen würde, und damit zwar nicht mehr zugreifbar, aber dennoch physisch als "Speichermüll" existent wäre. Über `dispose (HilfsZeig)` wird der Speicherplatz wieder freigegeben und kann für andere Zwecke genutzt werden. Um nicht unnötig Speicherplatz zu verschwenden, sollten daher nicht mehr zugreifbare Elemente stets zurückgegeben werden. Die Hilfsvariable `HilfsZeig` wird übrigens nur benötigt, um den ansonsten nicht mehr erreichbaren Speicherplatz zurückgeben zu können.

#### Beispiel 5.4.2.6

Es gibt verschiedene Ansätze, die erfolgreiche Suche in einer linearen Liste zu beschleunigen. Eine Möglichkeit, die sogenannte Transposition Rule, besteht darin, das in einem Suchprozeß gefundene Element mit seinem Vorgängerelement in der Liste zu vertauschen. Nach hinreichend vielen Suchprozessen etablieren sich häufig gesuchte Elemente im vorderen Bereich der Liste.

Untenstehende Funktion erweitert die klassische Suchfunktion `ListenElemSuchen2` um die nachfolgende Transposition. Da wir zum Vertauschen den Zugriff auf das Vorgängerelement des gesuchten Elements benötigen, organisieren wir das Durchlaufen wie bei der Prozedur `ListenElemEntfernen`. Wegen der schlichten Struktur der Elemente vertauschen wir nicht das gefundene Element mit seinem Vorgänger, sondern nur seine info-Komponente mit der des Vorgängers.

```

function ListenElemSuchenTransp (
    inRefAnfang: tRefListe;
    inZahl : integer): tRefListe;
{ Suchen in einer Liste und anschließende Vertausch-
  ung mit dem Vorgängerelement }

var
    Zeiger,
    ErgebnisZeiger : tRefListe;
    gefunden       : boolean;
    Hilf           : integer;

begin
    if inRefAnfang = nil then
        { Sonderfall: Liste ist leer }
        ErgebnisZeiger := nil
    else
        if inRefAnfang^.info = inZahl then
            { Sonderfall: erstes Element ist das gesuchte }
            ErgebnisZeiger := inRefAnfang
            { Es wird nicht vertauscht }

```

```

else
begin { gesuchtes Element finden }
  Zeiger := inRefAnfang;
  gefunden := false;
  while (Zeiger^.next <> nil) and not gefunden do
    if Zeiger^.next^.info = inZahl then
      begin
        { Element gefunden, info-Komponenten werden
          getauscht }
        Hilf := Zeiger^.next^.info;
        Zeiger^.next^.info := Zeiger^.info;
        Zeiger^.info := Hilf;
        ErgebnisZeiger := Zeiger;
        gefunden := true
      end
    else
      { gesuchtes Element noch nicht gefunden;
        wegen der Schleifenbedingung gilt
        hier stets Zeiger^.next <> nil }
      Zeiger := Zeiger^.next;
  { Schleife beendet. Wurde ein Element gefunden ? }
  if not gefunden then
    { Es gibt kein Element mit dem gesuchten Wert }
    ErgebnisZeiger := nil;
  end; { Listendurchlauf }
  ListenElemSuchenTransp := ErgebnisZeiger;
end; { ListenElemSuchenTransp }

```



Beachten Sie, daß wir für den Anfangszeiger der zu durchsuchenden Liste einen in-Parameter vorsehen. Dieser wird nämlich nicht verändert, er besitzt vor und nach dem Funktionsaufruf denselben Wert (nämlich denselben Adreßverweis). Ganz allgemein ändert sich die Liste laufend, aber nicht der Anfangszeiger. Hier kann man sich noch einmal den Unterschied klarmachen zwischen einem Zeiger und dem Objekt, auf das er zeigt: Das Objekt mag seine info-Komponente ändern, der Zeiger bleibt jedoch unverändert.

Die lineare Liste ist eine einfache dynamische Datenstruktur zur Verwaltung einer Kollektion von Objekten. Wir können neue Objekte zur Kollektion hinzufügen, andere entfernen sowie feststellen, ob ein bestimmtes Objekt in der Kollektion vorkommt. Der zeitliche Aufwand der Einfüge-Operation ist gering, da am Anfang der Liste eingefügt wird. Der Suchaufwand ist erheblich größer, da wir uns durch die Liste sukzessiv durchhangeln müssen, bis das gesuchte Objekt gefunden ist. Dazu muß im schlimmsten Fall die gesamte Liste durchlaufen werden. Da in der Entferne-Operation zunächst das zu entfernende Objekt gefunden werden muß, ist der

Zeigervariable und  
Parameterübergabe

Aufwand mindestens so groß wie bei der Suche. In Kapitel 6 stellen wir Ihnen eine andere dynamische Datenstruktur vor, die eine raschere Ausführung der Such- und Entferne-Operation erlaubt.

#### Aufgabe 5.4.2.7

- a) Schreiben Sie eine Pascal-Prozedur `ElementAnhaengen`, welche eine `integer-Zahl` an eine lineare Liste anhängt. Benutzen Sie die üblichen Typdefinitionen für eine lineare Liste und den folgenden Prozedurkopf für `ElementAnhaengen`. Ergänzen Sie auch die Deklaration der Prozedurparameter.

```
procedure ElementAnhaengen (
    ?? ??Zahl : integer;
    ?? ??RefAnfang : tRefListe;
    ?? ??RefEnde : tRefListe);
{ haengt ein neues Element an die Liste an, auf deren
  erstes Element ??RefAnfang und auf deren letztes
  Element ??RefEnde zeigt; der Rekordkomponenten
  info des neuen Elementes wird der Wert ??Zahl
  zugewiesen }
```

- b) Schreiben Sie ein Programm `ZahlenlisteEinAus`, das unter Benutzung der Prozedur `ElementAnhaengen` zunächst eine Folge von `integer-Zahlen` einliest. Die Eingabe einer Null, welche nicht zur Folge gehört, schließt die Eingabe ab. Danach sollen die Zahlen in der Reihenfolge ihrer Eingabe wieder auf dem Bildschirm ausgegeben werden.

### 5.5 Programmierstil (Teil 3)

#### **Hinweise**

- Die bisherige Muß-Regel 10 erhält eine allgemeinere Fassung.
- Die in dem jeweiligen Kapitel neu hinzukommenden Programmierstilhinweise und -regeln sind kursiv gedruckt und am Rand markiert, damit sie leichter gefunden werden können.

Programme können, nachdem sie implementiert und getestet worden sind, in den seltensten Fällen ohne Änderung über einen längeren Zeitraum hinweg eingesetzt werden. Tatsächlich ist es meist so, daß die Anforderungen nach der Fertigstellung verändert oder erweitert werden und während des Betriebs bislang unerkannte Mängel oder Fehler auftreten, die zu beseitigen sind. Programme müssen während ihres Einsatzes *gewartet* werden. Zu Wartungszwecken muß der Programmtext immer wieder gelesen und verstanden werden. Die Lesbarkeit eines Programms spielt also eine große Rolle. Je größer ein Programm ist, desto wichtiger wird das Kriterium der Lesbarkeit. Dies gilt besonders für industrielle Anwendungen, bei denen häufig

der bzw. die Entwickler nicht mehr verfügbar ist bzw. sind und Änderungen von Dritten vorgenommen werden müssen.

Die Lesbarkeit eines Programms hängt einerseits von der verwendeten Programmiersprache und andererseits vom *Programmierstil* ab. Der Programmierstil beeinflusst die Lesbarkeit eines Programms mehr als die verwendete Programmiersprache. Ein stilistisch gut geschriebenes C- oder COBOL-Programm kann besser lesbar sein als ein schlecht geschriebenes Pascal-Programm.

Programmierstil

Wir bemühen uns bei unseren Beispielen um einen guten Programmierstil. Zum einen wollen wir Ihnen einen guten Programmierstil "von klein auf" implizit vermitteln, d.h. Sie sollen nur gute und gar nicht erst schlechte Beispiele kennenlernen. Zum anderen bezwecken wir durch die erhöhte Lesbarkeit auch ein leichteres Verständnis der Konzepte, die wir durch Programme bzw. Prozeduren und Funktionen vermitteln wollen.

Programmierstil ist, wie der Name vermuten läßt, in gewissem Umfang Geschmackssache. Auch können starre Richtlinien in Einzelfällen unnatürlich wirken. Dennoch sind die Erfahrungen mit der flexiblen Handhabung solcher Richtlinien durchweg schlecht, so daß gut geführte Programmierabteilungen und Softwarehäuser einen hauseigenen Programmierstil verbindlich vorschreiben und die Einhaltung - häufig über Softwarewerkzeuge - rigoros kontrollieren.

Genau das werden wir auch in diesem Kurs tun (dies erleichtert auch die Korrektur der über tausend Einsendungen je Kurseinheit) und Regeln vorschreiben, deren Nichteinhaltung zu Punktabzügen führt. Dabei unterscheiden wir zwischen "Muß-Regeln", die in jedem Fall befolgt werden müssen, und "Kann-Regeln", von denen im Einzelfall abgewichen werden kann. Muß-Regeln sind durch Fettdruck und entsprechende Marginalien gekennzeichnet. **Das Nichteinhalten von Muß-Regeln wirkt sich in Punktabzügen aus.**

Wir haben uns in den Programmbeispielen nicht immer streng an die Kann-Regeln gehalten. Das hängt damit zusammen, daß die Empfehlungen zum Teil bereits für den professionellen Einsatz gedacht sind und unsere kleinen Beispiele manchmal überfrachtet hätten. So können z.B. gewisse Layout-Regeln (etwa für den Vereinbarungsteil), die bei größeren Programmen sinnvoll sind, kleinere Programme unnötig aufblähen. Außerdem nehmen wir aus Gründen der Lesbarkeit auf den Seitenumbruch Rücksicht, was sich nicht immer mit den Regeln verträgt.

Die Ausführungen zum Programmierstil werden von Kapitel zu Kapitel aktualisiert und ergänzt. Dabei werden die Regeln "kumuliert", d.h. die Programmierstilregeln von Kapitel 5 schließen die Regeln für Kapitel 3 und Kapitel 4 mit ein. Am Ende von Kapitel 6 finden Sie sämtliche kursrelevanten Regeln zusammengefasst aufgeführt.

### 5.5.1 Bezeichnerwahl

Stilistisch gute Programme werden oft als selbstdokumentierend bezeichnet. In erster Linie wird dabei auf die geschickte Wahl von Bezeichnern angespielt, die es

dem Leser erlauben, die Programmlogik schnell und ohne die Hilfe zusätzlicher Kommentare zu verstehen. (Daraus folgt aber nicht, daß Kommentare überflüssig werden!) Dies wird erreicht, wenn die Bedeutung einer Anweisung aus den Namen der beteiligten Variablen und der verwendeten Schlüsselwörter bzw. Operatoren hervorgeht.

Richtig gewählte Namen sind so kurz wie möglich, aber lang genug, um die Funktion oder das Objekt, das sie bezeichnen, so zu beschreiben, daß jemand, der mit der Aufgabenstellung vertraut ist, sich darunter etwas vorstellen kann. Selten gebrauchte Namen können etwas länger als häufig gebrauchte Namen sein.

Für die Bezeichner in Pascal-Programmen geben wir folgende Regeln vor:

- Bezeichner sind aussagekräftig (sprechend) und orientieren sich an dem Problem, welches das Programm löst.
- Bezeichner werden zunächst grundsätzlich so geschrieben, wie es die deutsche Grammatik vorgibt. Eine Ausnahme davon wird gemacht, wenn mehrere Worte (oder deren Abkürzungen) zu einem Bezeichner zusammengezogen werden. In diesem Fall werden die Anfangsbuchstaben der zusammengezogenen Worte (bzw. Abkürzungen) im Bezeichner groß geschrieben. Der erste Buchstabe des Bezeichners wird groß oder klein geschrieben, je nach dem, ob das erste Wort (bzw. dessen Abkürzung) groß oder klein geschrieben wird.
- **Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen.**
- **Typbezeichnern wird ein `t` vorangestellt.**  
*Bezeichner von Zeigertypen beginnen mit `tRef`.*  
*Bezeichner formaler Parameter beginnen mit `in`, `io` oder `out`.*  
 Es ist hilfreich, wenn an einem Bezeichner direkt abgelesen werden kann, ob er eine Konstante, einen Typ, eine Variable oder einen formalen Parameter bezeichnet. Dies erspart lästiges Nachschlagen der Vereinbarungen und vermindert Fehlbenutzungen. Wir erreichen eine Unterscheidung durch kurze Präfixe sowie Groß- und Kleinschreibungsregeln.
- Für Variablen bieten sich häufig Substantive als Bezeichner an, *für Bezeichner von Funktionen und Prozeduren werden am besten Verben (welche die Tätigkeit beschreiben) verwendet*, für Zustandsvariablen oder boolesche Funktionen eignen sich Adjektive oder Adverben.

### Abkürzungen

Bei langen Namen ist eine gewisse Vorsicht angebracht, da diese umständlich zu handhaben sind und die Lesbarkeit von Programmen vermindern, weil sie die Programmstruktur verdecken können. Unnötig lange Namen sind Fehlerquellen, der Dokumentationswert eines Namens ist nicht proportional zu seiner Länge.

Muß-Regel 1

Muß-Regel 2



Allerdings ist es oft schwierig, kurze und gleichzeitig präzise Bezeichner zu finden. In diesem Fall werden für Bezeichner Abkürzungen gewählt. Wir wollen dafür einige Regeln zusammentragen:

- Ein spezieller Kontext erleichtert das Abkürzen von Bezeichnern, ohne daß diese dadurch an Bedeutung verlieren. Allgemein gilt: Wird eine Abkürzung benutzt, sollte sie entweder im Kontext verständlich oder allgemeinverständlich sein.
- Wenn Abkürzungen des allgemeinen Sprachgebrauchs existieren, sind diese zu verwenden (z.B. Pers für Person oder Std für Stunde).
- Häufig gebrauchte Namen können etwas kürzer als selten gebrauchte Namen sein.
- Eine Abkürzung muß enträtselbar und sollte aussprechbar sein.
- Beim Verkürzen sind Wortanfänge wichtiger als Wortenden, Konsonanten sind wichtiger als Vokale.
- Es ist zu vermeiden, daß durch Verkürzen mehrere sehr ähnliche Bezeichner entstehen, die leicht verwechselt werden können.

### 5.5.2 Programmtext-Layout

Die Lesbarkeit eines Programms hängt auch von der äußeren Form des Programms ab. Für die äußere Form von Programmen gibt es viele Vorschläge und Beispiele aus der Praxis, die sich aber oft am Einzelfall oder an einer konkreten Programmiersprache orientieren. Für unsere Zwecke geben wir die folgenden Regeln an:

- Die Definitionen und Deklarationen werden nach einem eindeutigen Schema gegliedert. Dies fordert bereits die Pascal-Syntax, aber nicht in allen Programmiersprachen ist die Reihenfolge von Definitionen und Deklarationen vorgeschrieben.
- **Jede Anweisung beginnt in einer neuen Zeile; `begin` und `end` stehen jeweils in einer eigenen Zeile.**
- Der Einsatz von redundanten Klammern und Leerzeichen in Ausdrücken kann die Lesbarkeit erhöhen. Insbesondere sind Klammern dann einzusetzen, wenn die Priorität der Operatoren unklar ist. Die Prioritäten in verschiedenen Programmiersprachen folgen eventuell unterschiedlichen Hierarchien. Aber auch hier gilt wie bei Bezeichnerlängen: Ein Zuviel hat den gegenteiligen Effekt!
- Die hervorgehobene Darstellung von Schlüsselwörtern einer Programmiersprache unterstützt die Verdeutlichung der Programmstruktur. Wir heben Schlüsselwörter durch Fettdruck hervor. In Programmtexten, die von Hand geschrieben werden, wird durch Unterstreichung hervorgehoben.
- Kommentare werden so im Programmtext eingefügt, daß sie einerseits leicht zu finden sind und ausreichende Erläuterungen bieten, andererseits aber die Programmstruktur deutlich sichtbar bleibt (vgl. auch Abschnitt "Kommentare").

Muß-Regel 3

### Einrückungen

Die Struktur eines Programms wird durch Einrückungen hervorgehoben. Wir geben die folgenden Regeln an:

Muß-Regel 4

- **Anweisungsfolgen werden zwischen `begin` und `end` um eine konstante Anzahl von 2 - 4 Stellen eingerückt. `begin` und `end` stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.**

Muß-Regel 5

- **Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.**
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **`end`** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist (vgl. auch Abschnitt "Kommentare").

Beispiele für die letzten drei Regeln:

```
if <Bedingung> then
begin
    <Anweisungsfolge>
end
else { if not <Bedingung> }
begin
    <Anweisungsfolge>
end { if <Bedingung> };
```

```
while <Bedingung> do
begin
    <Anweisungsfolge>
end; { while <Bedingung> }
```

```
repeat
    <Anweisungsfolge>
until <Bedingung>;
```

```
for <Variable> := <Startwert> to <Zielwert> do
begin
    <Anweisungsfolge>
end; { for <Variable> }
```

- In einem Block werden die Definitions- und Deklarationsteile ebenfalls eingerückt. Die entsprechenden Schlüsselwörter stehen eingerückt in einer eigenen Zeile. Die vereinbarten Konstanten, Typen und Variablen werden darunter linksbündig angeordnet, in jeder Zeile sollte nur ein Bezeichner stehen. **`begin`** und **`end`** des Anweisungsteils werden nicht eingerückt. Z.B:

```
program einruecken (input, output);
{ demonstriert Einrueckungen }
```

**const**

```
PI = 3.1415927;
```

```
ANTWORT = 42;
```

**type**

```
tWochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);
```

**var**

```
Heute,
```

```
Gestern : tWochentag;
```

```
begin { einruecken }
```

```
  write ('Die Antwort ist: ');
```

```
  writeln (ANTWORT)
```

```
end. { einruecken }
```

In Ausnahmefällen, damit z.B. ein Programm (noch) auf eine Seite paßt, kann von diesen Regeln abgewichen werden. Falls das Programm in jedem Fall durch einen Seitenumbruch getrennt wird, gelten die Layoutregeln wieder.

**Leerzeilen**

Leerzeilen dienen der weiteren Gliederung des Programmtextes. Es empfiehlt sich, zusammengehörige Definitionen, Deklarationen oder Anweisungen auch optisch im Programmtext zusammenzufassen. Die einfachste Art ist die Trennung solcher Gruppen durch Leerzeilen. Es bietet sich an, einer solchen Gruppe einen zusammenfassenden Kommentar voranzustellen.

Wir schlagen den Einsatz von Leerzeilen in folgenden Fällen vor:

- Die verschiedenen Definitions- und Deklarationsteile werden durch Leerzeilen getrennt.
- Einzelne Funktions- bzw. Prozedurdeklarationen werden durch Leerzeilen voneinander getrennt.
- Anweisungsfolgen von mehr als ca. 10 Zeilen sollten durch Leerzeilen in Gruppen unterteilt werden.

**Prozedur- und Funktionsdeklarationen**

*Die Layoutregeln für Prozedur- und Funktionsdeklarationen haben wir zusammen mit anderen Regeln im Abschnitt "Prozeduren und Funktionen" aufgeführt.*

Bei den in diesem Abschnitt "Programmtext-Layout" angesprochenen Richtlinien ist wichtig, daß selbstauferlegte Konventionen konsequent eingehalten werden, um ein einheitliches Aussehen aller Programmbausteine zu erreichen. Um dem gesamten Programmtext ein konsistentes Aussehen zu geben, werden automatische *Programmtext-Formatierer*, sogenannte *Prettyprinter*, eingesetzt. Solche Programme formatieren einen (syntaktisch korrekten) Programmtext nachträglich gemäß fester

Programmtext-  
Formatierer  
Prettyprinter

syntaxgesteuerter  
Editor

oder in einem gewissen Rahmen variabler Regeln. Durch *syntaxgesteuerte Editoren* erreicht man vergleichbare Ergebnisse direkt bei der Programmtext-Eingabe.

### 5.5.3 Kommentare

Kommentare sind nützliche und notwendige Hilfsmittel zur Verbesserung der Lesbarkeit von Programmen. Sie sind wichtige Bestandteile von Programmen, und ihre Abwesenheit ist ein Qualitätsmangel.

Die Verwendung von Kommentaren ist in jeder höheren Programmiersprache möglich. Wir unterscheiden Kommentare, welche die Historie, den aktuellen Stand (Version) oder die Funktion von Programmen beschreiben - sie werden an den Programmanfang gestellt - und solche, die Objekte, Programmteile und Anweisungen erläutern - sie werden im Deklarations- und Anweisungsteil benutzt.

Die folgenden Konventionen bleiben an einigen Stellen unscharf, wichtig ist auch hier, daß man sich konsequent an die selbst gegebenen Regeln hält.

- Kommentare werden während der Programmierung eingefügt und nicht erst nachträglich ergänzt. Selbstverständlich werden die Kommentare bei Änderungen angepaßt.
- Kommentare sind so zu plazieren, daß sie die Programmstruktur nicht verdecken, d.h. Kommentare werden mindestens so weit eingerückt wie die entsprechenden Anweisungen. Trennende Kommentare wie Sternchenreihen oder Linien werden so sparsam wie möglich eingesetzt, da sie den Programmtext leicht zerschneiden.
- **Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst. Eine Kommentierung der Ein- und Ausgabedaten gehört gewöhnlich dazu.** Bei einem schwierigen Algorithmus können ggf. allgemeine, aber auf keinen Fall detaillierte Hinweise gegeben werden.
- Jeder Funktions- und Prozedurkopf wird ähnlich einem Programmkopf kommentiert (vgl. Abschnitt "Prozeduren und Funktionen").
- Die Bedeutung von Konstanten, Typen und Variablen wird erläutert, wenn sie nicht offensichtlich ist.
- Komplexe Anweisungen oder Ausdrücke werden kommentiert.
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **end** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist.
- Kommentare sind prägnant zu formulieren und auf das Wesentliche zu beschränken. Sie wiederholen nicht den Code mit anderen Worten, z.B.  

$$i := i + 1; \{ \text{erhoehe } i \text{ um } 1 \},$$
sondern werden nur da eingesetzt, wo sie zusätzliche Information liefern.
- An entscheidenden Stellen wird der Zustand der Programmausführung beschrieben, z.B. { Jetzt ist das Teilfeld bis Index n sortiert }.

Muß-Regel 6

#### 5.5.4 Prozeduren und Funktionen

- **Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.**
- *Für die Funktions- und Prozedurbezeichner werden möglichst suggestive Verben verwendet.*
- **Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgangsparameter.**
- **Die Übergabeart jedes Parameters wird durch Voranstellen von *in*, *io* oder *out* vor den Parameternamen gekennzeichnet.**
- Jeder Parameter steht in einer eigenen Zeile.  
Diese Regel haben wir aus Platzgründen nicht immer befolgt. Zumindestens sollte immer gelten: In einer Zeile stehen maximal drei Parameter desselben Typs mit derselben Übergabeart.
- Das Ende einer Funktions- bzw. Prozedurdeklaration wird durch die Wiederholung des Funktions- bzw. Prozedurnamens in einem Kommentar nach dem abschließenden *end* des Anweisungsteils markiert.
- **Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.**
- **Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert (vgl. Abschnitt "Seiteneffekte").**
- **Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix *in*, wenn das Feld nicht verändert wird.**
- **Funktionsprozeduren werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.**
- **Wertparameter werden nicht als lokale Variable mißbraucht.**  
*In Pascal ist das zwar möglich, da Wertparameter wie lokale Variable behandelt werden. Andere Programmiersprachen, z.B. Ada, verbieten Zuweisungen an Eingangsparameter. Das Verstehen der Programmstruktur wird gefördert, wenn Eingangsparameter nur gelesen werden und bei Bedarf zusätzliche lokale Variable deklariert werden.*

Muß-Regel 7

Muß-Regel 8

Muß-Regel 9

Muß-Regel 10

Muß-Regel 11

Muß-Regel 12

Muß-Regel 13

Muß-Regel 14

#### 5.5.5 Seiteneffekte

Eine Prozedur oder Funktion besitzt Seiteneffekte, wenn nicht nur die Änderungs- und Ausgangsparameter, sondern daneben weitere Programmdaten manipuliert werden. Die wichtigsten Seiteneffekte sind:

- *Modifikation globaler Variablen*
- *Änderungs- oder Ausgangsparameter in Funktionen (Diese schlechte Möglichkeit haben wir Ihnen gar nicht erst vorgestellt.)*

*Vergegenwärtigen Sie sich noch einmal die Definition von lokalen bzw. globalen Variablen aus dem Kapitel 5. Ein negativer Aspekt globaler Variablen besteht darin, daß sie durch Seiteneffekte von Prozeduren bzw. Funktionen manipuliert werden können. Wird eine Variable als Parameter übergeben, ist sie dagegen im Sinne obiger Definition lokal. Seiteneffekte sind kaum durch Kommentare zu erklären und gefährliche Fehlerquellen, wie die folgenden einfachen Beispiele zeigen:*

#### Beispiel 5.5.1

```
program Seiteneffekt1 (input, output);
{ dient der Demonstration von Seiteneffekten durch
  Manipulation globaler Variablen }

var
  a,
  b,
  c : integer;

function GlobalAendern (inX : integer): integer;
{ manipuliert die globale Variable a }

begin
  a := a + 1;
  GlobalAendern := a * inX
end; { GlobalAendern }

begin { Seiteneffekt1 }
  a := 10;
  b := 3;
  c := GlobalAendern (b) + GlobalAendern (b);
  writeln (c)
end. { Seiteneffekt1 }
```

*Auf den ersten Blick erscheint es unproblematisch, die letzte Zuweisung durch*

```
c := 2 * GlobalAendern (b);
```

*zu ersetzen. Bei der Programmausführung erhält c im Ausgangsprogramm den Wert 69, in der Variante jedoch den Wert 66 zugewiesen.*



Beispiel 5.5.2

```

program Seiteneffekt2 (input, output);
{ dient der Demonstration von Seiteneffekten durch
  Referenzparameter in Funktionen }

var
  a,
  b,
  c : integer;

function MitInoutParameter
  (   inX : integer;
    var ioY : integer): integer;
{ verwendet und manipuliert den Referenzparameter
  ioY }

begin
  ioY := ioY + inX;
  MitInoutParameter := inX * ioY
end; { MitInoutParameter }

begin { Seiteneffekt2 }
  a := 4;
  b := MitInoutParameter (5, a);
  c := MitInoutParameter (5, a);
  writeln (b, c)
end. { Seiteneffekt2 }

```

*In diesem Fall ändert die Funktion den Wert der als Referenzparameter übergebenen Variablen a. Daher werden den Variablen b und c verschiedene Werte (45 bzw. 70) zugewiesen, obwohl sich die Funktionsaufrufe von MitInoutParameter nicht sichtbar unterscheiden.*



*Seiteneffekte manipulieren Variable außerhalb des lokalen Speicherbereichs, also außerhalb des "Gesichtsfeldes" des Lesers. Gefährlich ist dies vor allem, weil anhand des Prozedurkopfes Seiteneffekte der "ersten Art" (Manipulation globaler Variablen) nicht zu erkennen sind. Als unmittelbare Konsequenz ergibt sich daraus die Forderung, globale Daten, die in der Prozedur benötigt werden, stets als Parameter zu übergeben, obwohl die Regeln über den Gültigkeitsbereich dies nicht erzwingen.*

*Wir haben Referenzparameter in Prozeduren und Funktionen zur effizienten Übergabe großer Felder zugelassen (vgl. Abschnitt "Prozeduren und Funktionen"). Daher können Seiteneffekte "der zweiten Art" (Manipulation von Referenzparametern, die semantisch Wertparameter sind) nur durch Disziplin beim Programmieren vermieden werden.*

### 5.5.6 Sonstige Merkgeregeln

Einige weitere Regeln wollen wir kurz auflisten:

- Es werden nach Möglichkeit keine impliziten (automatischen) Typanpassungen benutzt.
- Selbstdefinierte Typen werden nicht implizit definiert.
- Bei geschachtelten **if**-Anweisungen wird nach Möglichkeit nur der **else**-Zweig geschachtelt.
- Bietet die Programmiersprache verschiedene Schleifenkonstrukte (In Pascal: **while**, **repeat**, **for**), so ist eine dem Problem angemessene Variante zu wählen.
- **Die Laufvariable wird innerhalb einer for-Anweisung nicht manipuliert.**
- Häufig ist es aus Gründen der Lesbarkeit empfehlenswert, nicht an Variablen zu sparen und einige Hilfsvariablen mehr zu verwenden, als unbedingt nötig. So können komplizierte Berechnungen durch zusätzliche Variablen (mit aussagekräftigen Namen), die Teilergebnisse aufnehmen, wesentlich transparenter programmiert werden. Außerdem darf eine Variable innerhalb ihres Gültigkeitsbereichs nicht ihre Bedeutung wechseln. Es ist z.B. verwirrend, wenn eine Variable einmal die Durchläufe einer Schleife kontrolliert und an anderer Stelle das Ergebnis einer komplexen Berechnung speichert - "nur weil der Typ gerade paßt". Eine zusätzliche Variable ist hier von Vorteil. Auf die konsistente Bedeutung von Variablen sollte auch bei Verschachtelungen geachtet werden.



# Kurseinheit IV

## **Lernziele zum Kapitel 6**

Nach diesem Kapitel sollten Sie

1. binäre Suchbäume definieren können, ihre Darstellung in Pascal kennen sowie einfache Operationen auf Bäumen programmieren können,
2. den Begriff der Rekursion definieren und einfache Probleme rekursiv lösen können,
3. angeben können, wann der Einsatz der Rekursion sinnvoll ist, und wann eine iterative Lösung vorzuziehen ist.

## 6. Programmierkonzepte orientiert an Pascal (Teil 4)

### 6.1 Fortsetzung Dynamische Datenstrukturen: Binäre Bäume

Dynamische Datenstrukturen haben wir am Beispiel linearer Listen kennengelernt. Im Gegensatz zu statischen Datenstrukturen wie dem Feld oder dem Verbund, deren Größe und damit Speicherplatzbedarf bereits während der Übersetzung feststeht, wird für dynamische Datenstrukturen Speicherplatz erst während der Laufzeit auf eine Programmanforderung hin, nämlich durch einen Aufruf der Standardprozedur `new`, angelegt und dem Programm zur Verfügung gestellt. Durch einen Aufruf der Standardprozedur `dispose` wird während der Laufzeit Speicherplatz vom Programm an das Laufzeitsystem zurückgegeben. Der Speicherplatzbedarf hängt bei dynamischen Datenstrukturen von dem tatsächlichen Bedarf des aktuellen Programmlaufs ab, so dass kein Speicherplatz verschenkt wird. Dies ist ein großer Vorteil gegenüber statischen Datenstrukturen, die bei der Programmerstellung hinreichend groß vereinbart werden müssen, um für alle Eventualitäten zukünftiger Programmläufe genügend Speicherplatz anbieten zu können. Im Normalfall wird der reservierte Speicherplatz überhaupt nicht ausgeschöpft, so dass Speicherplatz verschwendet wird.

Lineare Listen bilden die einfachste Klasse dynamischer Datenstrukturen. Das Einfügen eines neuen Elementes ist einfach und schnell, denn wir können es am Anfang der Liste plazieren. Das Suchen eines Elementes ist programmtechnisch ebenfalls einfach, aber zeitaufwendig. Im *schlechtesten Fall* (*worst case*) muss die gesamte Liste durchlaufen werden, falls nämlich zufällig das gesuchte Element das letzte Element der Liste ist oder überhaupt nicht in der Liste vorkommt (*erfolglose Suche*). Im *besten Fall* (*best case*) ist das gesuchte Element das erste Listenelement, im *durchschnittlichen Fall* (*average case*) ist - wie wir uns leicht überlegen können - die Liste etwa zur Hälfte zu durchlaufen, um das gewünschte Element zu lokalisieren. Durch eine Sortierung der Elemente können wir die *erfolgreiche Suche* in der Liste - im Gegensatz zu der in einem Feld - nicht beschleunigen, da wir die Liste nur entlang der Verzeigerung durchlaufen können und eine Sortierung diesen Vorgang nicht beschleunigen kann. Eine Sortierung macht dagegen eine Einfügung teurer, da wir nicht mehr am Anfang, sondern an der richtigen Stelle einfügen müssen (die durch sequentiell „Durchhangeln“ wie in Beispiel 5.4.2.4 bestimmt wird). Das Entfernen eines Elementes aus einer linearen Liste ist ähnlich wie das Suchen zwar programmtechnisch noch relativ einfach, aber zeitaufwendig. Wir müssen nämlich das zu entfernende Element zunächst über einen Suchvorgang lokalisieren, bevor wir es entfernen können. Damit ist das Entfernen eines Elementes mindestens so teuer wie das Suchen.

Wollen wir alle drei klassischen Operationen *Suchen*, *Einfügen* und *Entfernen*, die in fast allen Anwendungen immer wieder auftreten, rascher abwickeln, dann benötigen wir „intelligenterere“ dynamische Datenstrukturen als lineare Listen. Solche „intelligenteren“ dynamischen Datenstrukturen sind zum Beispiel Bäume.

schlechtesten Fall  
worst case

bester Fall  
best case

durchschnittlicher  
Fall  
average case

Suchen  
Einfügen  
Entfernen

Baum  
Knoten  
Kanten

Ein *Baum* besteht aus einer endlichen Menge von *Knoten* (graphisch durch Kreise dargestellt), zusammen mit einer zweistelligen Relation auf der Knotenmenge (graphisch durch *Kanten* dargestellt), welche diese (hierarchisch) organisiert.

Abbildung 6.1 zeigt die graphische Darstellung eines Baumes, an der wir schon einige Eigenschaften eines Baumes erkennen können.

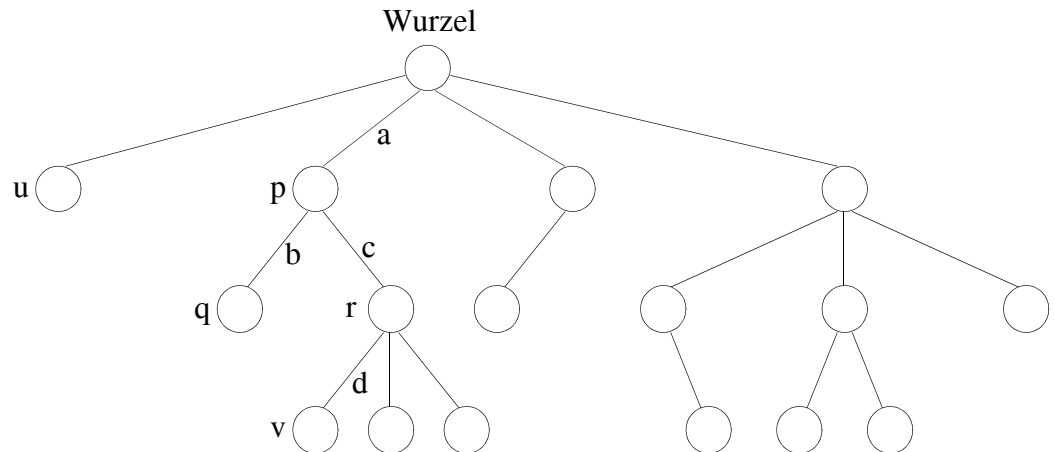


Abbildung 6.1 : Ein Baum

Wurzel  
Nachfolger  
Vorgänger

Den obersten Knoten eines Baumes mit nichtleerer Knotenmenge nennen wir *Wurzel*. Die Knoten, die mit der Wurzel durch eine Kante verbunden sind, heißen *Nachfolger*, *Söhne* oder *Kinder* der Wurzel. Die Wurzel heißt *Vorgänger* oder *Vater* ihrer Nachfolger. Diese Sprachregelung gilt nicht nur für die Wurzel, sondern für jeden Knoten im Baum. So ist der Knoten p der Vorgänger seiner Nachfolger q und r. Ein Knoten ohne Nachfolger, z.B. u oder v, heißt *Blatt*. Alle Knoten bis auf die Wurzel haben genau einen Vorgänger.

Blatt

innerer Knoten  
Teilbaum

Ein Knoten, der kein Blatt ist, heißt *innerer Knoten*. Jeden Knoten können wir als Wurzel eines *Teilbaumes* betrachten, der sich aus allen von dem Knoten aus über Kantenfolgen erreichbaren Knoten und Kanten des Ursprungsbaumes zusammensetzt. Abbildung 6.2 zeigt den Teilbaum mit Wurzel p des Ursprungsbaumes aus Abbildung 6.1.

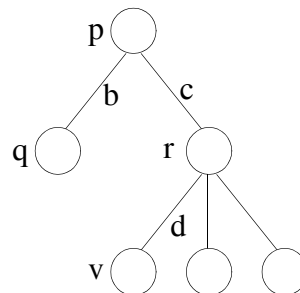


Abbildung 6.2 : Teilbaum

Baumeigenschaften

Ein Baum mit nichtleerer Knotenmenge hat zwei wichtige *Eigenschaften*:

- a) Er enthält genau einen Knoten, die Wurzel, der keinen Vorgänger hat.

- b) Jeder von der Wurzel verschiedene Knoten im Baum wird von der Wurzel durch genau eine Folge von Kanten erreicht.

So wird z.B. Knoten  $q$  durch die Kantenfolge  $a, b$ , der Knoten  $v$  durch die Kantenfolge  $a, c, d$  von der Wurzel aus erreicht, und es gibt keine andere Kantenfolge, mit der  $q$  bzw.  $v$  erreicht werden kann.

Wir betrachten in unserem Kurs nur den Spezialfall *binärer Bäume*, bei denen jeder Knoten höchstens zwei Nachfolger hat, welche wir zusätzlich als rechten und linken Nachfolger anordnen. Abbildung 6.3 zeigt einen Binärbaum in der üblichen Darstellung.

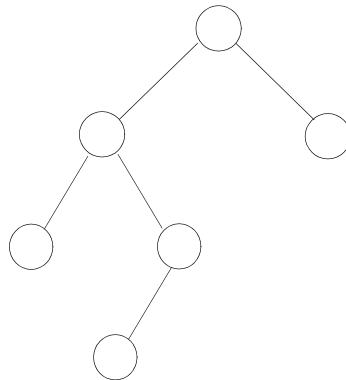


Abbildung 6.3 : Binärer Baum

Abbildung 6.4 zeigt einen Binärbaum mit dem nichtleeren linken Teilbaum  $T_1$  und dem nichtleeren rechten Teilbaum  $T_2$  in *kompakter Darstellung*.

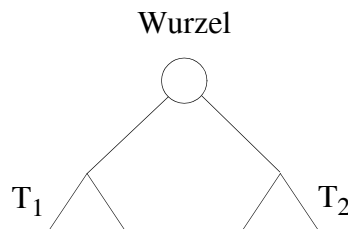


Abbildung 6.4 : kompakte Darstellung eines Binärbaums

Wir wollen nun zeigen, wie Binärbäume zur Speicherung von Objekten eingesetzt werden, so dass auf einfache Weise festgestellt werden kann, ob ein Objekt im Baum vorkommt oder nicht (Suchoperation). Wir nehmen der Einfachheit halber an, dass die Objekte paarweise verschiedene *integer*-Zahlen sind.

In einem *binären Suchbaum* werden die Zahlen in den Knoten abgelegt, wobei gerade soviel Knoten wie Zahlen vorkommen. Weiterhin gilt für jeden Knoten  $p$  des binären Suchbaumes: die Zahlen im linken Teilbaum von  $p$  sind sämtlich kleiner als die in  $p$  abgelegte Zahl, und die Zahlen im rechten Teilbaum von  $p$  sind sämtlich größer als die in  $p$  abgelegte Zahl.

Abbildung 6.5 zeigt einen binären Suchbaum für die Zahlenmenge  $\{1, 3, 14, 15, 27, 39\}$ . (Es gibt übrigens noch viele andere binäre Suchbäume für die angegebene Zahlenmenge.)

Binärbaum

kompakte  
Darstellung

binärer Suchbaum

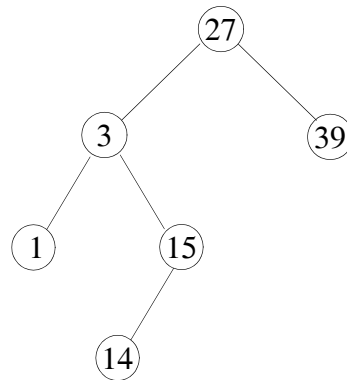


Abbildung 6.5 : Binärer Suchbaum

Wie suchen wir nun in einem Suchbaum nach der Zahl  $x$ ? Betrachten wir dazu den Suchbaum aus Abbildung 6.5. Wir vergleichen  $x$  zunächst mit der in der Wurzel gespeicherten Zahl 27. Ist  $x$  kleiner als 27, dann wird die Suche mit der Wurzel des linken Teilbaums fortgesetzt; ist  $x$  größer als 27, dann mit der Wurzel des rechten Teilbaums. Falls  $x$  weder größer noch kleiner als 27 ist, dann ist  $x$  offensichtlich gleich 27 und wir haben  $x$  im Baum gefunden.

Suchen wir also z.B. die Zahl 15, so inspizieren wir der Reihe nach die Knoten mit den Werten 27, 3, 15. Die Folge von Knoten  $p_0, p_1, \dots, p_k$ , wobei  $p_0$  die Baumwurzel und  $p_k$  der Knoten mit der gesuchten Zahl ist, bezeichnen wir als *Suchpfad* zum Knoten  $p_k$ . Die Länge des Suchpfades zu einem Knoten, gemessen in der Anzahl der Knoten auf dem Pfad, bezeichnen wir als *Tiefe* eines Knotens. Die Länge des längsten auftretenden Suchpfades im Baum bezeichnen wir als die *Höhe* des Baumes.

Suchen wir eine Zahl, die nicht im Baum vorkommt, z.B. die Zahl 20, dann endet die Suche in dem Knoten mit dem Wert 15. Die gesuchte Zahl müsste nämlich im rechten Teilbaum dieses Knotens liegen; da dieser aber nicht existiert, endet die Suche erfolglos.

Aus dem letzten Beispiel wird sofort klar, wie wir in einen Suchbaum einfügen. Wollen wir z.B. die Zahl 20 einfügen, so führt uns die Suche zunächst zum Knoten mit dem Wert 15. Dort hängen wir dann als rechten Nachfolger einen Knoten mit Wert 20 als neues Blatt an. Die Suche nach der richtigen Stelle im Baum garantiert, dass die Suchbaumeigenschaft nach dem Einfügen erhalten bleibt. Abbildung 6.6 zeigt den Suchbaum nach Einfügen eines neuen Knotens mit Wert 20.

Suchpfad

Tiefe eines Knotens  
Höhe des Baumes

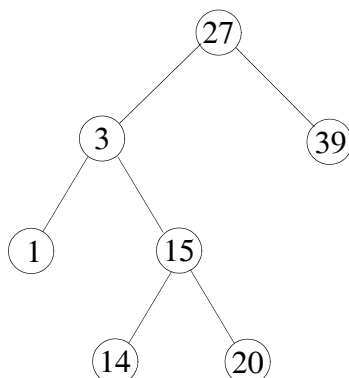


Abbildung 6.6 : Suchbaum nach dem Einfügen der Zahl 20

Für die Pascal-Implementierung eines Binärbaumes greifen wir, wie für dynamische Datenstrukturen üblich, auf Zeiger zurück. Als Knotentyp (für integer-Zahlen) bzw. als Zeigertyp vereinbaren wir

```

type
tRefBinBaum = ^tBinBaum;
tBinBaum = record
    info : integer;
    links : tRefBinBaum;
    rechts : tRefBinBaum
end;
  
```

Ein Zeiger auf den *leeren Baum* ist gleich **nil**. Abbildung 6.7 zeigt schematisch den Binärbaum von Abbildung 6.5 als Pascal-Datenstruktur. Der Zeiger `RefWurzel` vom Typ `tRefBinBaum` zeige auf die Wurzel des Baumes.

leerer Baum

Vergleichen wir lineare Listen mit Binärbäumen, so fällt auf, dass Binärbäume als verallgemeinerte lineare Listen aufgefasst werden können. Bei linearen Listen haben die Knoten maximal einen Nachfolger, während bei Binärbäumen für die Knoten maximal zwei Nachfolger zugelassen sind. Tatsächlich kann jede lineare Liste auch als Binärbaum gesehen werden, bei dem mit Ausnahme des letzten Knotens jeder Knoten genau einen leeren Teilbaum hat.

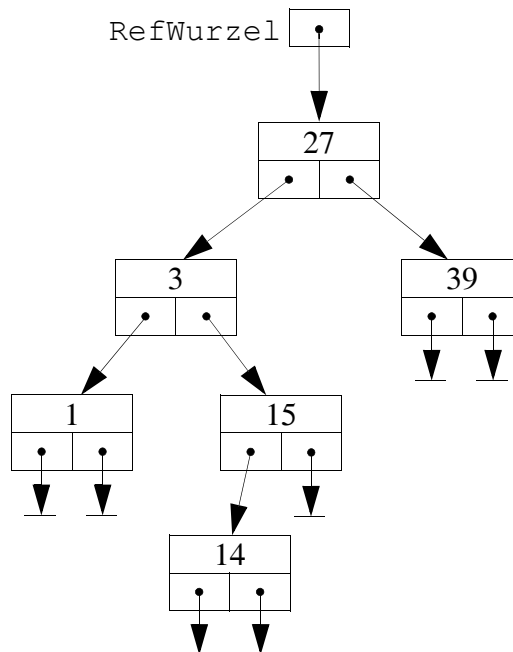


Abbildung 6.7 : Binärbaum

Beispiel 6.1.1

Suchen in einem  
Binärbaum

Die folgende Funktion BBKnotenSuchen sucht einen Knoten, dessen info-Komponente gleich inZahl ist. Ist die Suche erfolgreich, dann liefert die Funktion den Verweis auf den betreffenden Knoten, andernfalls **nil**.

```

function BBKnotenSuchen (
    inZahl : integer;
    inRefWurzel : tRefBinBaum): tRefBinBaum;
{ liefert fuer den Suchbaum, auf dessen Wurzel
  inRefWurzel zeigt, den Zeiger auf den Knoten,
  dessen Wert gleich inZahl ist }

var
  Zeiger : tRefBinBaum;
  gefunden : boolean;

begin
  Zeiger := inRefWurzel;
  gefunden := false;
  while (Zeiger <> nil) and not gefunden do
    begin
      if inZahl = Zeiger^.info then
        gefunden := true
      else { nicht gefunden, daher links oder rechts
        weitermachen }
        if inZahl < Zeiger^.info then
          Zeiger := Zeiger^.links
        else

```



```

        Zeiger := Zeiger^.rechts
    end; { while }
    { Jetzt gilt Zeiger = nil oder gefunden = true.
      Falls gefunden = true, zeigt Zeiger auf den Knoten
      mit info-Komponente = inZahl, andernfalls hat
      Zeiger den Wert nil }
    BBKnotenSuchen := Zeiger
end; { BBKnotenSuchen }

```



### Beispiel 6.1.2

In einen Suchbaum, auf dessen Wurzel der **var**-Parameter `ioRefWurzel` zeigt, soll ein neues Blatt mit `info`-Komponente `inZahl` so angefügt werden, dass die Suchbaumeigenschaft erhalten bleibt. Ist der Baum leer, so wird das neue Blatt die Wurzel. Andernfalls müssen wir, um das Blatt anhängen zu können, zunächst den Vaterknoten bestimmen. Dazu wandern wir mit zwei Hilfszeigern `RefSohn` und `RefVater` den Suchpfad entlang, bis `RefSohn` auf den leeren Baum zeigt. `RefVater` zeigt dabei auf den Vater von `RefSohn`, an den das neue Blatt angehängt wird. Kommt `inZahl` bereits im Baum vor, wird, der Definition des binären Suchbaums entsprechend, kein neues Blatt angefügt.

Einfügen in einen  
Binärbaum

```

procedure BBKnotenEinfuegen (
    inZahl : integer;
    var ioRefWurzel : tRefBinBaum);
{ fuegt in den Suchbaum, auf dessen Wurzel ioRefWurzel
  zeigt, ein Blatt mit Wert inZahl an }

var
    RefSohn,
    RefVater : tRefBinBaum;
    gefunden : boolean;

begin
    RefSohn := ioRefWurzel;
    gefunden := false;
    while (RefSohn <> nil) and (not gefunden) do
    begin
        if RefSohn^.info = inZahl then
            gefunden := true
        else
            begin
                RefVater := RefSohn;
                if inZahl < RefSohn^.info then
                    RefSohn := RefSohn^.links
                else
                    RefSohn := RefSohn^.rechts
            end
        end
    end

```

```

    end
end; { while }

if not gefunden then
{ neuen Knoten anlegen }
begin
    new (RefSohn);
    RefSohn^.info := inZahl;
    RefSohn^.links := nil;
    RefSohn^.rechts := nil;
    { neu angelegten Knoten einfuegen }
    if ioRefWurzel = nil then
        { Baum war leer, neue Wurzel zurueckgeben }
        ioRefWurzel := RefSohn
    else
        if inZahl < RefVater^.info then
            { Sohn links anhaengen }
            RefVater^.links := RefSohn
        else
            { Sohn rechts anhaengen }
            RefVater^.rechts := RefSohn
        end
    end
end; { BBKnotenEinfuegen }

```

In dem folgenden Abschnitt über Rekursion werden wir eine rekursive Einfügeprozedur vorstellen, die programmtechnisch einfacher ist.



### Beispiel 6.1.3

Wir geben nun eine Prozedur an, die von Null verschiedene integer-Zahlen einliest und mit Hilfe der Prozedur BBKnotenEinfuegen in einen anfangs leeren binären Suchbaum einfügt. Das Ende der Eingabe wird durch eine Null signalisiert. Wird eine Zahl eingegeben, die bereits im Baum vorkommt, ist nichts zu tun.

```

procedure BBAufbauen (var outRefWurzel : tRefBinBaum);
{ baut fuer eine Eingabe von integer-Zahlen <> 0 einen
  Suchbaum auf und gibt einen Zeiger auf dessen Wurzel
  in outRefWurzel zurueck }

var
    Zahl : integer;

begin
    { outRefWurzel mit leerem Baum initialisieren }
    outRefWurzel := nil;
    readln (Zahl);
    while Zahl <> 0 do
        begin

```

Aufbau eines binären Suchbaumes

```

    BBKnotenEinfuegen (Zahl, outRefWurzel);
    readln (Zahl)
  end
end; { BBAufbauen }

```



Ein durch iteriertes Einfügen einer Zahlenfolge in den anfangs leeren Baum erstellter binärer Suchbaum heißt *natürlicher Suchbaum* für diese Zahlenfolge.

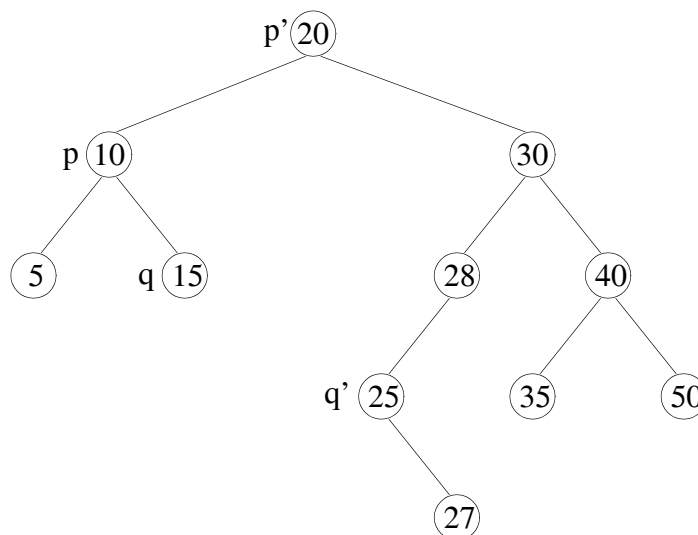
natürlicher  
Suchbaum

#### Aufgabe 6.1.4

Gegeben ist ein binärer Suchbaum mit den üblichen Typdefinitionen.

Schreiben Sie eine Pascal-Funktion, die für einen Zeiger auf einen Knoten p mit zwei Nachfolgern einen Zeiger auf denjenigen Knoten q zurückliefert, dessen Wert in der Sortierreihenfolge auf den Wert des Knotens p folgt. Diese Funktion löst übrigens ein Teilproblem bei der Entfernung eines inneren Knotens aus einem binären Suchbaum.

Beispiel:



Für den Knoten p mit Wert 10 hat der gesuchte Knoten q den Wert 15. Für den Knoten p' mit Wert 20 hat der gesuchte Knoten q' den Wert 25.



Für eine Menge von n verschiedenen Zahlen  $\{z_1, z_2, \dots, z_n\}$  gibt es  $n!$  verschiedene Eingabereihenfolgen (Permutationen). Je nach Eingabereihenfolge können sich verschiedene natürliche Suchbäume ergeben. (Allerdings gibt es viel weniger verschiedene natürliche Suchbäume als Eingabereihenfolgen). Im Extremfall kann ein zu einer Liste *degenerierter Suchbaum* der Höhe n entstehen, wenn etwa n Zahlen in aufsteigend sortierter Reihenfolge eingefügt werden. Es kann aber auch ein *vollständiger Suchbaum* mit minimal möglicher Höhe  $\lceil \log_2 (n+1) \rceil$  entstehen, bei dem sich die Tiefe sämtlicher Blätter um höchstens eins unterscheidet<sup>1</sup>.

degenerierter  
Suchbaum  
vollständiger  
Suchbaum

1. Für eine reelle Zahl x bezeichnet der Ausdruck  $\lceil x \rceil$  die kleinste ganze Zahl größer oder gleich x.

Abbildung 6.8 zeigt diese beiden Extremfälle eines Suchbaumes für die Menge  $\{1, 3, 14, 15, 27, 39\}$ . Es sind die natürlichen Suchbäume für die Folgen  $(15, 3, 39, 1, 14, 27)$  und  $(1, 3, 14, 15, 27, 39)$  dargestellt.

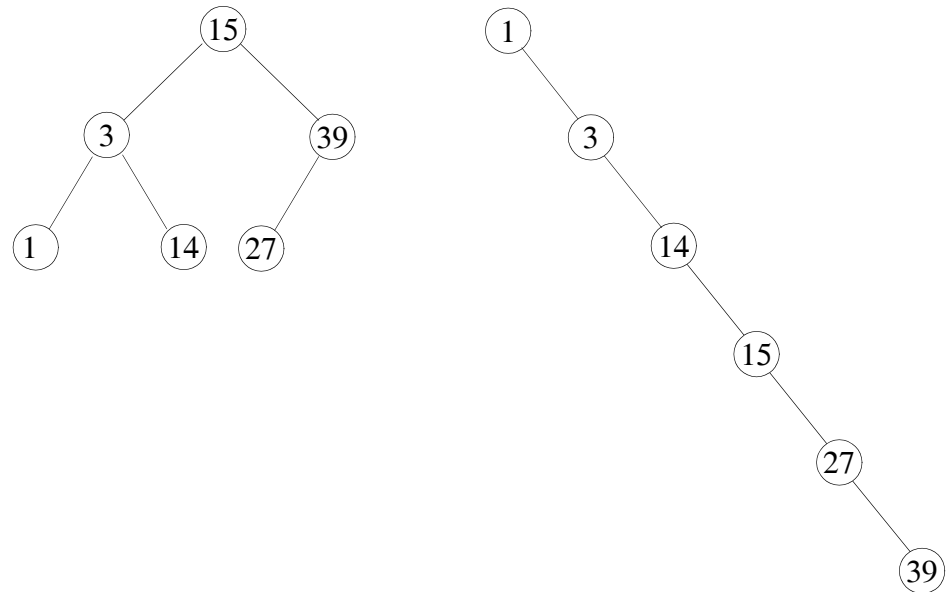


Abbildung 6.8 : zwei natürliche Suchbäume für die Zahlenmenge  $\{1, 3, 14, 15, 27, 39\}$

Eine wichtige Frage ist, ob die ausgeglichenen niedrigen Bäume oder die hohen, zu Listen degenerierten Bäume häufiger auftreten, wenn man die den möglichen Anordnungen von  $n$  Zahlen entsprechenden natürlichen Suchbäume erzeugt. Eine Antwort auf diese Frage finden Sie im Kurs „Datenstrukturen“. Hier sei nur soviel gesagt, dass die durchschnittliche Höhe aller unterschiedlichen Suchbäume für eine Menge von  $n$  Zahlen proportional zu  $\log_2(n+1)$ , also sehr niedrig ist. So ist  $\log_2 1000 \approx 10$  und  $\log_2 1000000 \approx 20$ . Selbst für 1 Million Knoten ist die durchschnittliche Höhe eines Binärbaumes also nicht viel größer als 20. Da für einen Suchbaum der (Zeit-)Aufwand zum Suchen und Einfügen (gemessen in Knotenbesuchen) durch die Baumhöhe beschränkt ist, können Suchen und Einfügen im Durchschnitt sehr rasch durchgeführt werden. Für einen binären Suchbaum mit 1 Million Knoten erfordern die Operationen Suchen und Einfügen im Durchschnitt wenig mehr als 20 Knotenbesuche (zum Vergleichen der Knotenwerte mit der gesuchten bzw. einzufügenden Zahl). Dasselbe gilt für das Entfernen eines Knotens aus einem Suchbaum. Der zugehörige Algorithmus bzw. die zugehörige Prozedur soll aber dem Kurs „Datenstrukturen“ vorbehalten bleiben. Dennoch sollte Ihnen klar geworden sein, dass Suchbäume die Operationen *suchen*, *einfügen* und *entfernen* weitaus besser unterstützen als lineare Listen. Die Operation *einfügen* ist zwar etwas langsamer als für lineare Listen, aber praktisch so schnell wie die Baum-Suche und damit effizient genug.

Ein Problem der natürlichen Bäume bleibt aber in Form ihrer Entartung zu linearen Listen bei unglücklichen Einfügereihenfolgen. In diesem Fall sind alle drei Operationen sehr ineffizient. Im Kurs „Datenstrukturen“ werden Sie *balancierte Binär-*

*bäume* kennenlernen, die für keine Eingabereihenfolge mehr entarten und für alle drei Operationen auch im schlechtesten Fall sehr effizient sind.

## 6.2 Rekursion

Die Rekursion ist ein fundamentales Prinzip in der Mathematik und Informatik. In der Mathematik spricht man von der rekursiven Definition einer Funktion, wenn diese „durch sich selbst definiert wird“, d.h. wenn in der Definition der Funktion selbst wieder auf diese Funktion zurückgegriffen wird.

### Beispiel 6.2.1

Wir geben eine rekursive Definition der Fakultätsfunktion für nicht-negative ganze Zahlen an:

$$f(x) = \begin{cases} 1 & \text{für } x = 0 \\ x \cdot f(x-1) & \text{für } x > 0. \end{cases}$$

Für  $x = 2$  erhalten wir also

$$f(2) = 2 \cdot f(1).$$

Um  $f(2)$  zu berechnen, benötigen wir  $f(1)$ . Nun ist

$$f(1) = 1 \cdot f(0).$$

Da lt. Definition

$$f(0) = 1$$

ist (Abbruchbedingung), ergibt sich

$$f(1) = 1 \cdot 1 = 1$$

und damit

$$f(2) = 2 \cdot f(1) = 2 \cdot 1 = 2.$$



In der Informatik spricht man analog von einer *rekursiven Prozedur* (bzw. *rekursiven Funktion*), wenn diese sich innerhalb ihres Anweisungsteils wieder selbst aufruft. Ebenso wie eine endlose Schleife vermieden werden muss, darf auch eine rekursive Prozedur sich nicht unendlich oft selbst aufrufen, wenn das Programm terminieren soll. Vielmehr muss nach endlich vielen Aufrufen eine Situation erreicht werden, in der kein *rekursiver Aufruf* mehr erfolgt. Daher muss sich eine rekursive Prozedur für immer kleiner werdende Teilprobleme aufrufen, bis schließlich ein so kleines Teilproblem erreicht ist, dass seine Lösung ohne rekursiven Aufruf angegeben werden kann. Für die Fallunterscheidung, ob bereits ein solch kleines Teilproblem vorliegt oder noch Rekursionsschritte erfolgen müssen, benötigt man außerdem eine Abbruchbedingung. Zwei wesentliche Merkmale rekursiver Prozeduren sind daher stetige *Problemverkleinerung* sowie eine *Abbruchbedingung*.

Fakultät

rekursive Prozedur  
rekursive Funktion

rekursiver Aufruf

Problemverkleinerung  
Abbruchbedingung

Das Ziel dieses Abschnittes besteht darin, die Rekursion als praktisch verwertbare Methode der Programmierung zu untersuchen. Unter Praktikern hat die Rekursion oftmals einen schlechten Ruf, da sie für ineffizient bzgl. Laufzeit- und Speicherplatzbedarf gehalten wird. Dass dieser Eindruck vor allem aus ungeeigneten Anwendungen der Rekursion resultiert und deshalb ungerechtfertigt ist, wollen wir in diesem Abschnitt verdeutlichen.

Wir werden mit einfachen Beispielen beginnen, um Sie mit dem Prinzip der Rekursion vertraut zu machen. Diese einfachen, leicht zu durchschauenden Beispiele haben allerdings eines gemeinsam: Sie zeigen größtenteils ungeeignete Anwendungen der Rekursion. Es schließen sich deshalb sinnvollere, aber meist kompliziertere Beispiele an. Grundsätzlich werden wir jedes Beispiel daraufhin untersuchen, ob es eine sinnvolle Anwendung der Rekursion darstellt oder nicht, und zum Schluss dieses Abschnitts eine Merkregel für den sinnvollen Einsatz der Rekursion angeben.

### Beispiel 6.2.2

Mit Blick auf die Definition der Fakultätsfunktion aus Beispiel 6.2.1 geben wir im Folgenden die Pascal-Funktion `FakultaetRek` zur Berechnung der Fakultätsfunktion für  $\text{inZahl} \geq 0$  an, wobei wir die Definition

```
type
  tNatZahl = 0..maxint;
```

voraussetzen.

```
1  function FakultaetRek (
           inZahl : tNatZahl) : tNatZahl;
   { berechnet die Fakultät von inZahl >= 0 }
2  begin
3    if inZahl = 0 then { Abbruchbedingung erfüllt }
4      FakultaetRek := 1
5    else
6      FakultaetRek :=
           inZahl * FakultaetRek (inZahl - 1)
7  end; { FakultaetRek }
```

Die Funktion `FakultaetRek` ruft sich in Zeile 6 selbst auf, ist also eine rekursive Funktion. Wir erkennen deutlich die Problemverkleinerung (der rekursive Aufruf erfolgt mit einem kleineren Argument) und die Abbruchbedingung (in Zeile 3), bei deren Erreichen kein rekursiver Aufruf mehr erfolgt.

Abbildung 6.9 stellt schematisch dar, wie der Aufruf von `FakultaetRek` mit  $\text{inZahl} = 3$  abgewickelt wird. Hierbei geben  $t_1, \dots, t_{22}$  die Zeitpunkte an, zu denen jeweils die zugehörige Programmzeile ausgeführt wird. Die Zeile 6 wird beispielsweise erstmalig zum Zeitpunkt  $t_4$  ausgeführt, (rekursiver Aufruf mit dem Wert 2) und letztmalig zum Zeitpunkt  $t_{21}$  (das Ergebnis des rekursiven Aufrufs wird mit

inZahl multipliziert und das Produkt als Funktionsergebnis festgelegt). Ein Pfeil  $\longrightarrow$  bedeutet Eintritt in die Funktion FakultaetRek, ein Pfeil  $\longleftarrow$  Rückkehr aus der Funktion.



Abbildung 6.9 : Abarbeitung von FakultaetRek (3)

Aus der Abbildung 6.9 geht hervor, dass bei der Abarbeitung von FakultaetRek(3) die Funktion FakultaetRek insgesamt viermal mit den Werten inZahl = 3, 2, 1, 0 aufgerufen wird. Beim vierten Aufruf existieren vier verschiedene Instanzen der Fakultätsfunktion, wobei aber nur die vierte aktiv ist, während die ersten drei eingefroren sind, denn die dritte muss warten, bis die vierte ihr Ergebnis abliefert, während die zweite auf die dritte und die erste auf die zweite war-

ten muss. Zu den Zeitpunkten  $t_4$ ,  $t_8$  und  $t_{12}$  wird jeweils die Abarbeitung des Funktionsaufrufs unterbrochen und die dazugehörige „Umgebung“ eingefroren. Die unterbrochenen Abarbeitungen werden dann zu den Zeitpunkten  $t_{17}$ ,  $t_{19}$  und  $t_{21}$  fortgesetzt und schließlich beendet.

Die Tabelle 6.1 zeigt noch einmal in anderer Darstellung die Abarbeitung der Funktionsaufrufe.

Zeilennr.	Programmteil	Status
2	1.Aufruf von FakultaetRek	wird ausgeführt
3	1.Aufruf von FakultaetRek	wird ausgeführt
5	1.Aufruf von FakultaetRek	wird ausgeführt
6	1.Aufruf von FakultaetRek	wird unterbrochen
2	2.Aufruf von FakultaetRek	wird ausgeführt
3	2.Aufruf von FakultaetRek	wird ausgeführt
5	2.Aufruf von FakultaetRek	wird ausgeführt
6	2.Aufruf von FakultaetRek	wird unterbrochen
2	3.Aufruf von FakultaetRek	wird ausgeführt
3	3.Aufruf von FakultaetRek	wird ausgeführt
5	3.Aufruf von FakultaetRek	wird ausgeführt
6	3.Aufruf von FakultaetRek	wird unterbrochen
2	4.Aufruf von FakultaetRek	wird ausgeführt
3	4.Aufruf von FakultaetRek	wird ausgeführt
4	4.Aufruf von FakultaetRek	wird ausgeführt
7	4.Aufruf von FakultaetRek	wird beendet
6	3.Aufruf von FakultaetRek	wird ausgeführt
7	3.Aufruf von FakultaetRek	wird beendet
6	2.Aufruf von FakultaetRek	wird ausgeführt
7	2.Aufruf von FakultaetRek	wird beendet
6	1.Aufruf von FakultaetRek	wird ausgeführt
7	1.Aufruf von FakultaetRek	wird beendet

Tabelle 6.1: Abarbeitung der Funktionsaufrufe von FakultaetRek (3)

Angesichts der rekursiven Definition der Fakultätsfunktion aus Beispiel 6.2.1 liegt eine entsprechende Umsetzung als rekursive Pascal-Funktion nahe: Wir haben im Wesentlichen die mathematische Definition nur in der Sprache Pascal neu formuliert. Statt der rekursiven Funktion FakultaetRek können wir jedoch eine ebenso einfache iterative Funktion FakultaetIt angeben:

```
function FakultaetIt (  
    inZahl : tNatZahl): tNatZahl;  
{ berechnet die Fakultät von inZahl >= 0 }
```



```

var
  i : integer;
  temp : tNatZahl;

begin
  temp := 1;
  for i := 2 to inZahl do
    temp := i * temp;
  FakultaeIt := temp
end; { FakultaeIt }

```

Da die iterative Funktion nicht komplizierter als die rekursive ist, ziehen wir sie vor, denn sie ist geringfügig effizienter. Grundsätzlich wird nämlich bei einem Prozedur- bzw. Funktionsaufruf (unabhängig, ob rekursiv oder nicht) die aktuelle Situation zum Zeitpunkt des Aufrufs „eingefroren“ und festgehalten, mit welcher Anweisung nach der Abarbeitung der Prozedur bzw. Funktion fortzufahren ist. Dann werden Parameter übergeben und die lokalen Variablen angelegt. Diese zur Verwaltung der dynamischen Blockstruktur benötigten Informationen werden vom Compiler in einer impliziten Hilfsdatenstruktur, dem *Laufzeitstack* (stack ist der englische Begriff für Stapel) nachgehalten. Bei der Ausführung der Funktion *FakultaetRek* (*inZahl*) gibt es neben dem initialen Aufruf noch *inZahl* rekursive Aufrufe, und die maximale *Rekursionstiefe*, d.h. die maximale Anzahl gleichzeitig nicht abgeschlossener rekursiver Aufrufe, beträgt ebenfalls *inZahl*. In Konsequenz benötigt die rekursive Variante umso mehr Speicher für den Laufzeitstack, je größer die Eingabe *inZahl* ist, während die iterative Variante unabhängig von der Eingabe immer gleich viel Speicher (für die Variablen *i* und *temp*) benötigt. Da die Fakultätsfunktion sehr stark wächst (13! ist bereits größer als 6 Milliarden) und damit *inZahl* in der Praxis nur kleine Werte annehmen kann<sup>1</sup>, fällt der Nachteil der Rekursion in diesem Fall allerdings nicht sehr stark ins Gewicht.



Laufzeitstack

Rekursionstiefe

### Aufgabe 6.2.3

Die Folge der so genannten Fibonacci-Zahlen beginnt mit der Zahl 0, gefolgt von der Zahl 1. Jede weitere Fibonacci-Zahl ergibt sich aus der Summe der beiden ihr vorangehenden Zahlen:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, ...

Für  $n \geq 0$  können wir die  $n$ -te Fibonacci-Zahl auch rekursiv definieren:

$$f(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ f(n-1) + f(n-2) & \text{für } n > 1. \end{cases}$$

1. Bei einer Integer-Größe von z.B. 16 Bit, also  $\text{maxint}=32787$ , funktioniert *FakultaetRek* (ebenso wie *FakultaetIt*) nur für  $0 \leq \text{inZahl} \leq 7$ , bei 32 Bit für  $0 \leq \text{inZahl} \leq 12$ , bei 64 Bit für  $0 \leq \text{inZahl} \leq 20$ .

Schreiben Sie eine rekursive Pascal-Funktion `FibonacciRek`, die eine natürliche Zahl  $n$  (des Typs `tNatZahl` aus Beispiel 6.2.2) entgegennimmt und dazu entsprechend obiger rekursiver Definition die  $n$ -te Fibonacci-Zahl (ebenfalls vom Typ `tNatZahl`) zurückgibt.



Das Beispiel der Fibonacci-Zahlen zeigt übrigens, dass durchaus mehr als ein rekursiver Aufruf in einer rekursiven Funktion (bzw. Prozedur) vorkommen kann. Die Musterlösung demonstriert außerdem eindrucksvoll, um wieviel ineffizienter eine elegante rekursive Lösung gegenüber einer iterativen sein kann – was natürlich, wie wir im Folgenden noch sehen werden, keineswegs immer gelten muss. Vorab können wir aber schon festhalten, dass die Vorteilhaftigkeit einer rekursiven Problemlösung nur im Vergleich mit einem iterativen Lösungsansatz zu beurteilen ist, wobei neben der *Programmkomplexität* (im Sinne der Komplexität des Programmtexts, also z.B. Anzahl und Schachtelungsstruktur von Fallunterscheidungen und Schleifen oder Bedarf an Hilfsdatenstrukturen) auch die *Effizienz* (d.h. der Laufzeit- und Speicherbedarf) zu berücksichtigen ist.

#### Beispiel 6.2.4

Eine Zeichenfolge unbekannter Länge, die durch einen Punkt abgeschlossen wird, soll in umgekehrter Reihenfolge, d.h. *gespiegelt*, auf dem Bildschirm ausgegeben werden. Wir nehmen der Einfachheit halber an, dass innerhalb der Zeichenfolge kein Punkt auftritt.

Wir schlagen eine *rekursive Lösung* vor und stellen dazu zunächst folgende Überlegung an:

Für jedes Zeichen der Zeichenfolge muss doch folgende Eigenschaft gelten: Nach seinem Einlesen müssen erst alle nachfolgenden Zeichen eingelesen und ausgegeben werden, bevor das Zeichen selbst ausgegeben werden kann.

Ausgehend von dieser Überlegung sehen wir also eine rekursive Prozedur vor, die zunächst ein Zeichen einliest, dann durch einen rekursiven Aufruf die restliche Zeichenfolge einliest und gespiegelt ausgibt, um anschließend das eingelesene Zeichen selbst auszugeben. Beachten Sie, dass hier die Eigenschaften dynamischer Blockstrukturen zum Tragen kommen.

Programmkomplexität

Effizienz

```
procedure ZeichenDrehen;  
{ liest eine Folge von Zeichen ein und gibt sie in  
  umgekehrter Reihenfolge wieder aus }  
  
  var  
    Zeichen : char;  
  
  begin  
    read (Zeichen);  
    if Zeichen = '.' then { Rekursionsabbruch }  
      writeln  
    else  
      begin  
        { restliche Zeichen bearbeiten }  
        ZeichenDrehen;  
        { nach Einlesen und Ausgeben der nachfolgenden  
          Zeichen kann jetzt das Zeichen selbst ausgegeben  
          werden }  
        write (Zeichen)  
      end  
    end; { ZeichenDrehen }
```

Für die Zeichenkette 'abc.' können wir uns den Programmablauf wie in der folgenden Abbildung 6.10 veranschaulichen. Ein Pfeil  $\longrightarrow$  bedeutet Eintritt in die Prozedur ZeichenDrehen, ein Pfeil  $\longleftarrow$  Rückkehr aus der Prozedur.

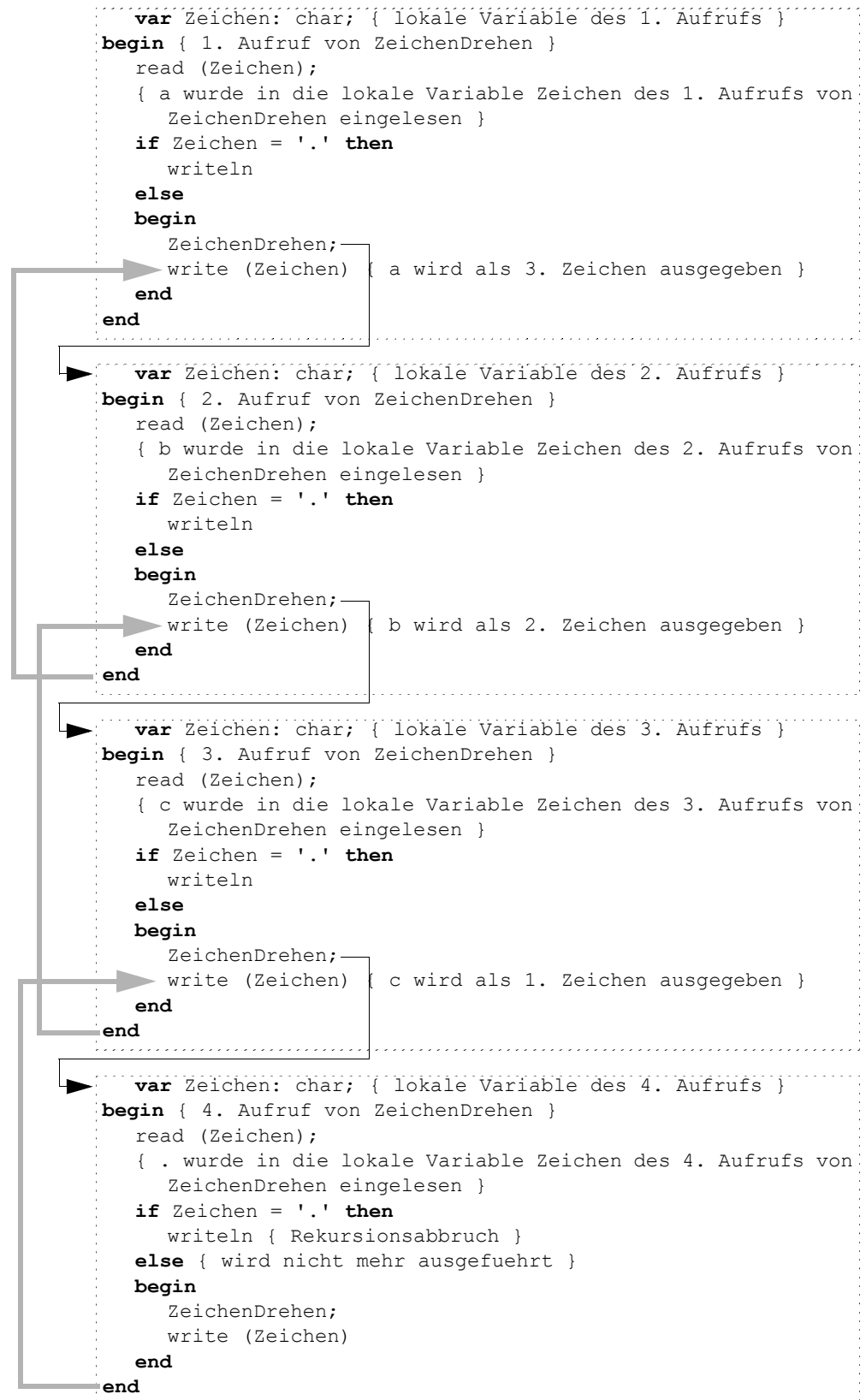


Abbildung 6.10 : Abarbeitung von ZeichenDrehen für die Zeichenfolge 'abc.'

Eine alternative *iterative Lösung* des Problems könnte z.B. die Zeichen solange am Anfang einer Liste einfügen, bis der Punkt eingelesen wird. Dadurch stehen die Zeichen in umgekehrter Reihenfolge in der Liste. Anschließend wird solange das jeweils erste Zeichen der Liste ausgegeben und dann entfernt, bis die Liste leer ist. Die Liste implementiert eine so genannte LIFO-Organisation („last in, first out“), auch *Stapel* (engl. stack) genannt. Ein Stapel ist eine lineare Datenstruktur, bei der die Bearbeitung auf den Anfang beschränkt ist: Einfügen und Entfernen ist nur am Anfang erlaubt, ebenso darf nur auf das erste Element zugegriffen werden.

Stapel  
Stack

Die rekursive Prozedur ist dagegen besonders einfach aufgebaut, weil wir keinen Stapel zum Umdrehen der Zeichen programmieren müssen, sondern den vom Compiler angelegten impliziten Laufzeitstack ausnutzen. Auf diesem werden vom Laufzeitsystem die Zustände der „eingefrorenen“ Prozedurabarbeitungen, also insbesondere deren lokale Variablen mit den bereits eingegebenen Zeichen, abgelegt.

Nicht zuletzt wegen der geringeren Programmkomplexität kann man die rekursive Variante durchaus als sinnvolle Anwendung der Rekursion betrachten – sofern die Eingabefolgen nicht zu lang werden. Letztere Einschränkung machen wir insbesondere, da die Rekursionstiefe gleich der Anzahl der eingegebenen Zeichen ist, der Laufzeitstack also relativ stark (linear) wächst. Zwar benötigt der von der iterativen Lösung angelegte Stapel ebenfalls linearen Speicherplatz, allerdings wird vom System häufig für den Laufzeitstack deutlich weniger Speicher zur Verfügung gestellt als für vom Programm angelegte Datenstrukturen. Insbesondere viele Turbo-Pascal-Versionen stellen standardmäßig nur einen kleinen Laufzeitstack zur Verfügung, so dass die Rekursionstiefe bei der Standardeinstellung recht eingeschränkt sein kann.



Lineare Listen und Bäume werden auch als *rekursive Datenstrukturen* bezeichnet, da sie rekursiv definiert werden können. Eine lineare Liste ist nämlich entweder leer oder besteht aus einem Knoten (dem Anfangsknoten), der auf eine Liste zeigt. Die analoge rekursive Definition für einen Binärbaum lautet: Ein Binärbaum ist entweder leer oder besteht aus einem Knoten (der Wurzel), der auf zwei Binärbäume zeigt. Zur Bearbeitung rekursiver Datenstrukturen bieten sich rekursive Algorithmen oft intuitiv an – sind jedoch nicht grundsätzlich sinnvoll, wie wir an den folgenden Beispielen sehen werden.

rekursive  
Datenstruktur

#### Beispiel 6.2.5

Wir suchen eine Zahl `inZahl` in einer linearen Liste. Die Situation ist die gleiche wie in Beispiel 5.4.2.3, aber jetzt wird die Liste rekursiv durchsucht.

rekursives Suchen in  
einer linearen Liste

```
function ListenElemSuchen (
    inRefAnfang : tRefListe;
    inZahl : integer) : tRefListe;
{ bestimmt das erste Element in einer Liste mit
```

```
Anfangszeiger inRefAnfang, bei dem die info-
Komponente gleich inZahl ist }
```

```
var
Zeiger : tRefListe;
```

```
begin
  Zeiger := inRefAnfang;
  if inRefAnfang <> nil then
    if inRefAnfang^.info <> inZahl then
      { nicht gefunden, suche weiter in der Restliste }
      Zeiger :=
        ListenElemSuchen (inRefAnfang^.next, inZahl);
  ListenElemSuchen := Zeiger
end; { ListenElemSuchen }
```

Machen Sie sich die Arbeitsweise der Funktion klar, am besten vielleicht mit einer Liste von 3 Elementen, und probieren Sie die Fälle aus, dass `inZahl` das erste, zweite, dritte Element ist bzw. gar nicht vorkommt. Es ist sehr wichtig, dass Sie an diesem simplen Beispiel das rekursive „Durchhangeln“ durch eine dynamische Datenstruktur verstehen.

Die rekursive Lösung ist einfach und ästhetisch, aber dennoch nicht zu empfehlen, da die Rekursionstiefe und der Laufzeitstack so groß wie die Anzahl der Listenelemente werden können (z.B. bei erfolgloser Suche). Die iterative Variante kommt dagegen ohne zusätzliche Hilfsdatenstruktur (und ohne Laufzeitstack) und daher mit konstantem Speicherbedarf aus.



rekursives Entfernen  
in einer linearen Liste

### Beispiel 6.2.6

Wir geben eine rekursive Prozedur für das Entfernen eines Elementes aus einer Liste an. Die entsprechende iterative Prozedur finden Sie als Beispiel 5.4.2.5. Die folgende Prozedur zeigt eindrucksvoll, wie eine rekursive Lösung die Programmkomplexität drastisch reduzieren kann.

```
procedure ListenElemEntfernen (
    inZahl : integer;
    var ioRefAnfang : tRefListe;
    var outGefunden : boolean);
{ entfernt aus einer Liste mit Anfangszeiger
  ioRefAnfang das Element mit dem Wert inZahl, bei
  erfolgreicher Entfernung wird outGefunden auf true
  gesetzt, sonst auf false }

var
  Zeiger : tRefListe;
  gefunden : boolean;
```

```

begin
  if ioRefAnfang = nil then
    gefunden := false
  else
    if ioRefAnfang^.info = inZahl then
      { Element gefunden, also entfernen }
      begin
        Zeiger := ioRefAnfang;
        ioRefAnfang := ioRefAnfang^.next;
        dispose (Zeiger);
        gefunden := true
      end
    else
      { Element noch nicht gefunden, es folgt das
        rekursive Durchsuchen des Listenrestes }
      ListenElemEntfernen
        (inZahl, ioRefAnfang^.next, gefunden);
    outGefunden := gefunden
  end; { ListenElemEntfernen }

```

Beachten Sie, dass `ioRefAnfang` nur verändert wird, falls das erste Listenelement entfernt wird. Bei dem rekursiven Aufruf (und bei den folgenden rekursiven Aufrufen) wird `ioRefAnfang` nicht verändert, denn es wird mit `ioRefAnfang^.next` und nicht mit `ioRefAnfang` aufgerufen.

Die erheblich verringerte Programmkomplexität der rekursiven gegenüber der iterativen Version ist schon eklatant. Versuchen Sie wiederum, für eine Liste mit 3 Elementen die Prozedur zu verstehen. Wenn Sie diese Prozedur wirklich verstanden haben, sind Sie für (fast) alle Eventualitäten der rekursiven Welt bestens gerüstet. Ebenso wie das rekursive Durchsuchen einer Liste ist wegen der möglichen hohen Rekursionstiefe die rekursive Variante der Entferne-Prozedur aus Effizienzgründen leider wieder nicht zu empfehlen.



#### Aufgabe 6.2.7

- a) Schreiben Sie eine rekursive Pascal-Funktion, welche die Position des größten Elementes in einem `integer`-Feld zurückgibt. Die Feldelemente seien paarweise verschieden.

Dabei soll wie folgt vorgegangen werden:

Es wird beginnend ab dem zweiten Feldelement rekursiv die Position des größten Elementes unter den restlichen Elementen bestimmt. Das so gefundene Maximum wird mit dem ersten Feldelement verglichen. Ist es größer als das erste Element, dann wird seine Position, andernfalls die Position des ersten Elementes zurückgegeben.

Als Konstanten- und Typdefinitionen sind zu benutzen

```

const
  FELDGROESSE = 10;

```

```

type
  tIndex = 1..FELDGROESSE;
  tFeld = array [tIndex] of integer;

```

sowie der Funktionskopf

```

function rekFeldMax (
    inMaxPos : tIndex;
    var inFeld : tFeld) : tIndex;
{ bestimmt rekursiv in inFeld die Position
  des Maximums }

```

Durch den Aufruf `rekFeldMax (1, Feld)` wird die Position des Maximums von `Feld` zurückgeliefert. Überlegen Sie sich zunächst die Abbruchbedingung der Rekursion.

- b) Handelt es sich um eine sinnvolle Anwendung der Rekursion?  
Begründen Sie Ihre Aussage.



Nach der rekursiven Bearbeitung von Listen und Feldern betrachten wir in den folgenden Beispielen die rekursive Implementierung von Operationen auf binären Bäumen.

### Beispiel 6.2.8

Wir geben zunächst eine rekursive Variante für die Funktion `BBKnotenSuchen` aus Beispiel 6.1.1 an.

```

function BBKnotenSuchen (
    inZahl : integer;
    inRefWurzel : tRefBinBaum) : tRefBinBaum;
{ liefert in dem Suchbaum, auf dessen Wurzel
  inRefWurzel zeigt, den Zeiger auf den Knoten,
  dessen Wert gleich inZahl ist }

var
  Zeiger : tRefBinBaum;

begin
  Zeiger := inRefWurzel;
  if inRefWurzel <> nil then
    if inZahl < inRefWurzel^.info then
      Zeiger := BBKnotenSuchen
        (inZahl, inRefWurzel^.links)
    else
      if inZahl > inRefWurzel^.info then
        Zeiger := BBKnotenSuchen
          (inZahl, inRefWurzel^.rechts);

```

rekursives Suchen in  
einem binären Such-  
baum



```

    BBKnotenSuchen := Zeiger
end; { BBKnotenSuchen }

```

Falls es sich um einen balancierten Suchbaum handelt, bei dem die Höhe, d.h. die Anzahl der Knoten auf dem längsten Suchpfad, „logarithmisch beschränkt“ ist, so ist die Rekursionstiefe auch „logarithmisch beschränkt“. Dann ist die rekursive Variante noch akzeptabel effizient und könnte als Alternative zur iterativen Variante in Betracht gezogen werden. Falls allerdings nicht sichergestellt ist, dass die eingegebenen Suchbäume balanciert sind und im schlimmsten Fall sogar zu einer Liste entarten können, so ist auch die Rekursionstiefe – und damit der Speicherbedarf für den Laufzeitstack – schlimmstenfalls linear. Dagegen kommt die iterative Lösung aus Beispiel 6.1.1 völlig ohne eine dynamische Hilfsdatenstruktur (und ohne Laufzeitstack) aus, beansprucht also unabhängig von Baumgröße und Balancierung immer konstant viel Speicher. Da die iterative Lösung auch nur unwesentlich komplexer als die rekursive ist, geben wir ihr hier den Vorzug. □

#### Beispiel 6.2.9

Nun folgt eine rekursive Prozedur zum Einfügen einer Zahl `inZahl` in einen binären Suchbaum mit Wurzelzeiger `ioRefWurzel`.

```

procedure BBKnotenEinfuegen (
    inZahl : integer;
    var ioRefWurzel : tRefBinBaum;
    var outGefunden: boolean);
{ fuegt in den Suchbaum, auf dessen Wurzel ioRefWurzel
  zeigt, einen Knoten mit Wert inZahl ein }

var
    gefunden : boolean;

begin
    if ioRefWurzel = nil then
    { neuen Knoten mit info-Komponente = inZahl
      einfuegen }
    begin
        new (ioRefWurzel);
        ioRefWurzel^.info := inZahl;
        ioRefWurzel^.links := nil;
        ioRefWurzel^.rechts := nil;
        gefunden := false
    end
    else { ioRefWurzel <> nil }
    begin
        if inZahl < ioRefWurzel^.info then
            BBKnotenEinfuegen
                (inZahl, ioRefWurzel^.links, gefunden)
        else

```

rekursives Einfügen in  
einen binären Such-  
baum

```

if inZahl > ioRefWurzel^.info then
  BBKnotenEinfuegen
    (inZahl, ioRefWurzel^.rechts, gefunden)
else
  { es gibt inZahl schon im Baum }
  gefunden := true;
end; { ioRefWurzel <> nil }
outGefunden := gefunden
end; { BBKnotenEinfuegen }

```

Hier ist die Programmkomplexität deutlich geringer als bei der iterativen Variante in Beispiel 6.1.2. Unter der Prämisse, dass die Baumhöhe logarithmisch beschränkt bleibt, ist die rekursive Version ebenfalls akzeptabel effizient. Jedoch gilt auch hier, dass die iterative Variante keine dynamische Hilfsdatenstruktur (und natürlich auch keinen Laufzeitstack) benötigt, und insbesondere für beliebig große und unbalancierte Bäume doch vorzuziehen ist.



#### Aufgabe 6.2.10

Gegeben sei ein Binärbaum mit den üblichen Typdefinitionen:

```

type
tRefBinBaum = ^tBinBaum;
tBinBaum = record
  info : integer;
  links : tRefBinBaum;
  rechts : tRefBinBaum
end;

```

Implementieren Sie eine Funktion, die einen Zeiger auf das „erste Blatt von links“ in diesem Binärbaum zurückliefert. Beispielsweise soll für den in Abbildung 6.11 gezeigten Binärbaum ein Zeiger auf den Knoten mit dem Wert 2 zurückgeliefert werden.

- Implementieren Sie eine rekursive Funktion `BlattSuchenRek`.
- Implementieren Sie eine iterative Funktion `BlattSuchenIt`.
- Handelt es sich bei `BlattSuchenRek` um eine sinnvolle Anwendung der Rekursion?

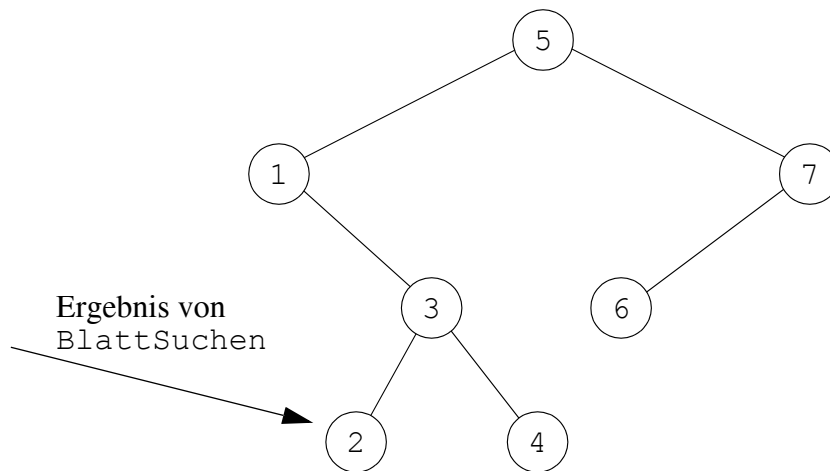


Abbildung 6.11 : Binärbaum



Betrachten wir nun eine *sinnvolle* Anwendung der Rekursion. Dazu kommen wir auf einen interessanten Aspekt zurück, der sich bereits bei der Prozedur `ZeichenDrehen` aus Beispiel 6.2.4 zeigte: Die iterative Variante mancher Algorithmen muss eine nichttriviale Hilfsdatenstruktur (einen Stapel) aufbauen und verwalten, während die rekursive Variante eine solche nicht benötigt. Die Rekursion nutzt stattdessen den vom Compiler verwalteten Laufzeitstack aus, um den der Programmierer sich nicht kümmern muss. Wir wollen diesen großen Vorteil rekursiver Algorithmen an dem folgenden Beispiel genauer demonstrieren.

### Beispiel 6.2.11

Wir betrachten das Durchlaufen aller Knoten eines Binärbaumes in einer bestimmten Reihenfolge. Algorithmen zum Durchlaufen aller Knoten eines Baumes bilden das weitgehend problemunabhängige Gerüst für spezifische Aufgaben. Dazu gehören beispielsweise das Ausdrucken, Markieren oder Kopieren aller auftretenden Knoten(werte) in bestimmter (z.B. sortierter) Reihenfolge, die Berechnung der Summe, des Durchschnitts, der Anzahl usw. aller Knotenwerte, die Ermittlung der Höhe des Baumes, die Klärung der Frage, ob alle Blätter die gleiche Tiefe haben usw.

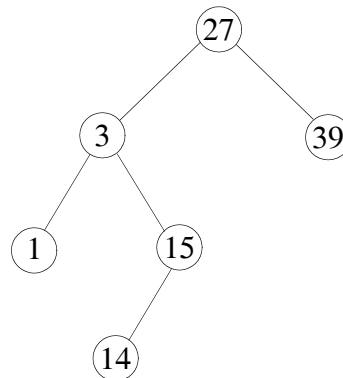
Bei einer linearen Liste ist das Durchlaufen trivial: wir betrachten zunächst das erste Element, dann das Nachfolgerelement usw., bis das Listenende erreicht ist. Bei einem Binärbaum können wir ein Durchlaufen, bei dem jeder Knoten genau einmal betrachtet wird, nicht so offensichtlich durchführen. Doch hier kommt uns die rekursive Natur von Bäumen zu Hilfe. Wir können nämlich z.B. das Durchlaufen eines Binärbaumes in sogenannter *Hauptreihenfolge* (*preorder*) wie folgt rekursiv definieren:

Hauptreihenfolge  
preorder

Hauptreihenfolge:

1. Ist der Baum leer, dann STOP. Sonst:
2. Betrachte die Wurzel.
3. Durchlaufe den linken Teilbaum der Wurzel in Hauptreihenfolge.
4. Durchlaufe den rechten Teilbaum der Wurzel in Hauptreihenfolge.

Für den binären Suchbaum



sind die Knotenwerte in Hauptreihenfolge 27, 3, 1, 15, 14, 39.

Es gibt verschiedene Reihenfolgen des Durchlaufens durch einen Binärbaum. Eine weitere ist die *symmetrische Reihenfolge (inorder)*, bei der die Knoten „von links nach rechts“ besucht werden. Im Fall eines binären Suchbaumes werden die Knoten dann in sortierter Reihenfolge besucht.

symmetrische  
Reihenfolge  
inorder

Symmetrische Reihenfolge:

1. Ist der Baum leer, dann STOP. Sonst:
2. Durchlaufe den linken Teilbaum der Wurzel in symmetrischer Reihenfolge.
3. Betrachte die Wurzel.
4. Durchlaufe den rechten Teilbaum der Wurzel in symmetrischer Reihenfolge.

Für den oben gezeigten binären Suchbaum erhalten wir 1, 3, 14, 15, 27, 39.

Eine *rekursive Prozedur* z.B. für die Hauptreihenfolge anzugeben, ist aufgrund der rekursiven Definition trivial:

```

procedure BBHauptreihenfolge (
    inRefWurzel : tRefBinBaum);
{ durchlaeuft rekursiv in Hauptreihenfolge die Knoten
  des Binaerbaumes, auf dessen Wurzel inRefWurzel
  zeigt }

begin
  if inRefWurzel <> nil then
    begin
      { betrachte die Wurzel, d.h. fuehre die
        problemabhaengigen Arbeitsschritte fuer
        die Wurzel aus }
    
```

```

    BBHauptreihenfolge (inRefWurzel^.links);
    BBHauptreihenfolge (inRefWurzel^.rechts)
  end
end; { BBHauptreihenfolge }

```

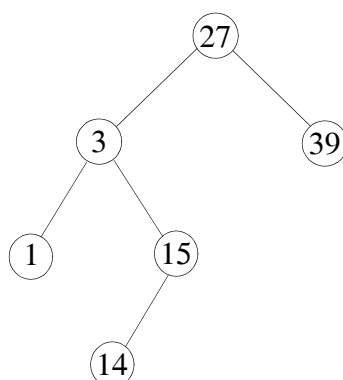
Ein *iterativer Algorithmus* für das Durchlaufen eines Binärbaumes in Hauptreihenfolge ist dagegen überhaupt nicht trivial. Wir benötigen nämlich eine zusätzliche Hilfsdatenstruktur, in der wir die Wurzeln derjenigen Teilbäume ablegen, an denen wir vorbeigelaufen sind, ohne sie betrachtet zu haben. Da wir nach dem Besuch eines Knotens mit zwei Teilbäumen stets zunächst in seinen linken Teilbaum absteigen, müssen wir uns also die Wurzel des jeweiligen rechten Teilbaumes merken.

Da es eine rekursive Lösung für die Hauptreihenfolge gibt, bei der der Laufzeitstack die Rolle der Hilfsdatenstruktur übernimmt, wählen wir als Hilfsdatenstruktur für die iterative Lösung auch einen Stapel. Man kann sich aber auch anschaulich überlegen, dass die Knoten aus der Hilfsdatenstruktur in einer Reihenfolge ausgelesen werden müssen, die gerade umgekehrt zur Reihenfolge ihres Abspeicherns ist, was wiederum bedeutet, dass die Hilfsdatenstruktur ein Stapel sein muss.

#### Grundidee des iterativen Algorithmus:

Schleife: Solange (*Teil-*) Baum ist nicht leer tue folgendes  
 Beginn  
*Betrachte die Wurzel;*  
 Falls *rechter Teilbaum ist nicht leer* dann  
   *speichere dessen Wurzel in einem Stapel ST;*  
*Fahre mit dem linken Teilbaum fort*  
 Ende; { von "solange" }  
 {hier ist der linke Teilbaum leer}  
 Falls *ST ist nicht leer* dann  
   *entferne die zuletzt gespeicherte Wurzel aus ST*  
   *und fahre mit dem zugehörigen Teilbaum bei Schleife fort.*  
 sonst STOP. {ST ist leer}

Wir betrachten wieder unseren Beispielbaum



und geben unter Verwendung des iterativen Algorithmus die Knotenwerte in der Hauptreihenfolge aus.

Ausgabe	Wurzel des rechten Teilbaumes	in ST gespeicherte Wurzeln
27	③⑨	③⑨
3	①⑤	①⑤, ③⑨
1	—	①⑤, ③⑨

Tabelle 6.2

Jetzt müssen wir auf die zuletzt in ST gespeicherte Wurzel zurückgreifen und weitermachen:

Ausgabe	Wurzel des rechten Teilbaumes	in ST gespeicherte Wurzeln
15	—	③⑨
14	—	③⑨

Tabelle 6.3

Jetzt müssen wir auf die zuletzt in ST gespeicherte Wurzel zurückgreifen und weitermachen:

Ausgabe	Wurzel des rechten Teilbaumes	in ST gespeicherte Wurzeln
39	—	—

Tabelle 6.4

Jetzt müssen wir auf die zuletzt in ST gespeicherte Wurzel zurückgreifen. Da der Stapel leer ist, STOP.

Für eine Implementierung müssen wir uns noch eine adäquate Implementierung des Stapels überlegen. Zunächst einmal benötigen wir eine dynamische Datenstruktur, da die Anzahl der gleichzeitig gespeicherten Wurzeln von Programmlauf zu Programmlauf variiert und die auftretende maximale Anzahl von der Struktur des aktuellen Baumes abhängt. Für einen Binärbaum mit  $n$  Knoten müssen im schlimmsten Fall bis zu  $n/2$  Wurzeln gleichzeitig gespeichert werden (vgl. dazu die Diskussion am Ende des Beispiels). Soviel Platz soll aber nicht von vornherein re-

serviert werden. Als Realisierung des Stapels wählen wir eine lineare Liste, die nur am Anfang manipuliert wird.

Um eine einfachere Schleife zu erhalten, weichen wir etwas von dem oben skizzierten Vorgehen ab und stapeln grundsätzlich zu jedem Knoten die Wurzeln seiner beiden Teilbäume (aber nur, wenn sie nicht leer sind), wobei wir zuerst die Wurzel des rechten und dann die des linken Teilbaumes stapeln. Wir nehmen dann die zuletzt gestapelte Wurzel (also die des linken Teilbaumes) vom Stapel herunter, geben deren Wert aus und stapeln wieder ihre beiden Nachfolger usw. Als erstes stapeln wir die Wurzel des Baumes.

Wir erhalten die folgende Prozedur BBHauptIter:

```
procedure BBHauptIter (inRefWurzel : tRefBinBaum);
{ durchläuft iterativ in Hauptreihenfolge die Knoten
  des Binaerbaumes, auf dessen Wurzel inRefWurzel
  zeigt }

{ zuerst die Typdefinitionen und Operationen fuer
  den Stapel, in dem Zeiger auf Knoten des Baumes
  gestapelt werden }

type
tRefStack = ^tStack;
tStack = record
    Zeiger : tRefBinBaum;
    next : tRefStack
end;

var
RefStackAnf : tRefStack; { Zeiger auf Stapelanfang }
Zeiger : tRefBinBaum;

procedure push (
    inZeiger : tRefBinBaum;
    var ioRefStackAnf : tRefStack);
{ legt den Knotenzeiger inZeiger auf dem Stapel mit
  Anfangszeiger ioRefStackAnf ab }

var
RefNeu : tRefStack;

begin
    new (RefNeu);
    RefNeu^.Zeiger := inZeiger;
    RefNeu^.next := ioRefStackAnf;
    ioRefStackAnf := RefNeu
end; { push }
```

```

function top (
    inRefStackAnf : tRefStack) : tRefBinBaum;
{ liefert das oberste Stapелеlement eines nicht-
  leeren Stapels mit Anfangszeiger inRefStackAnf }

begin
    top := inRefStackAnf^.Zeiger
end; { top }

procedure pop (var ioRefStackAnf : tRefStack);
{ entfernt aus einem nichtleeren Stapel mit
  Anfangszeiger ioRefStackAnf das oberste Element }

var
    RefLoesch : tRefStack;

begin
    RefLoesch := ioRefStackAnf;
    ioRefStackAnf := ioRefStackAnf^.next;
    dispose (RefLoesch)
end; { pop }

function isempty (
    inRefStackAnf : tRefStack) : boolean;
{ liefert true, falls der Stapel mit Anfangszeiger
  inRefStackAnf leer ist, sonst false }

begin
    isempty := (inRefStackAnf = nil)
end; { isempty }

begin
    RefStackAnf := nil; { leeren Stapel initialisieren }
    if inRefWurzel <> nil then
        { Baum ist nicht leer; Wurzel stapeln }
        push (inRefWurzel, RefStackAnf);
    while not isempty (RefStackAnf) do
        begin { naechste Wurzel aus dem Stapel holen und
            entfernen }
            Zeiger := top (RefStackAnf);
            pop (RefStackAnf);
            { jetzt koennte der Knotenwert z.B. durch
              write (Zeiger^.info, ' ');
              ausgegeben werden }
            if Zeiger^.rechts <> nil then
                { rechten Nachfolger stapeln }
                push (Zeiger^.rechts, RefStackAnf);
            if Zeiger^.links <> nil then

```



```

    { linken Nachfolger stapeln }
    push (Zeiger^.links, RefStackAnf)
  end
end; { BBHauptIter }

```



Das Beispiel des Baumdurchlaufens demonstriert eindrucksvoll (wenn nicht dramatisch) den Vorteil richtig angewandter rekursiver Programmierung. Die rekursive Variante besteht aus einer **if**-Anweisung ohne **else**-Zweig, deren Rumpf zwei rekursive Aufrufe und z.B. eine **write**-Anweisung enthält. Die iterative Variante definiert und verwaltet einen Hilfsstapel und weist nicht zuletzt deswegen eine höhere Programmkomplexität auf.

Auf dem impliziten Stapel der rekursiven Variante werden bei jedem (rekursiven) Prozeduraufruf u. a. die aktuelle Situation und die (Adresse der) dem Aufruf folgende(n) Anweisung abgelegt. Die Rekursionstiefe (und damit die Höhe dieses Stapels) und ist gleich der Höhe des Binärbaumes – 1. Die minimale Stapelhöhe der rekursiven Version liegt bei einem vollständigen Binärbaum vor (dessen Höhe  $\lceil \log_2 (n + 1) \rceil$  für  $n$  Knoten beträgt).

Interessant ist übrigens, dass die Stapelhöhe für die iterative Variante gerade minimal ist, wenn sie für die rekursive Variante maximal ist, nämlich wenn der Binärbaum zu einer Liste entartet ist, also jeder Knoten höchstens einen Nachfolger hat. Die maximale Stapelhöhe der iterativen Version ergibt sich für einen Baum wie in Abbildung 6.12 gezeigt. Die Anzahl  $n$  der Knoten dieses Baumes ist gerade so dass die Stapelhöhe  $n/2$  eine ganze Zahl ist. Für jeden betrachteten Knoten – außer dem letzten, der keinen linken Nachfolger hat – werden beide Nachfolger auf dem Stapel abgelegt, von denen allerdings der linke sofort wieder aus dem Stapel entfernt wird.

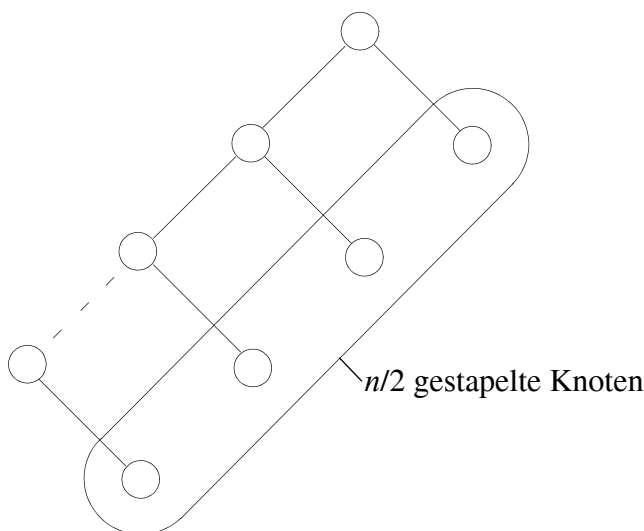


Abbildung 6.12 : Binärbaum mit  $n$  Knoten, für den die Stapelhöhe des iterativen Verfahrens maximal ist.

Ist die Baumhöhe logarithmisch beschränkt (wie z.B. bei balancierten Bäumen), so ist auch die Stapelhöhe für beide Varianten logarithmisch beschränkt.



Zum Abschluss dieses Abschnitts wollen wir unsere Erfahrungen mit rekursiven Prozeduren bzw. Funktionen zu einer Merkregel zusammenfassen. Wir möchten betonen, dass es sich nicht um eine Muss-Regel handelt, sondern im Einzelfall auch anders entschieden werden kann. Jedes Abweichen sollte jedoch stets kritisch hinterfragt und begründet werden.

Merkregel Rekursion:

- a) Benötigt eine iterative Lösung für das betrachtete Problem einen vom Programm verwalteten Stapel, während eine rekursive Lösung hierfür den Laufzeitstack ausnutzen kann, dann ist eine rekursive Lösung vorzuziehen.
- b) Andernfalls ist eine iterative Lösung vorzuziehen.

Zu a) Als Grund für die Empfehlung ist die geringere Programmkomplexität der rekursiven Lösung zu nennen, die eine größere Übersichtlichkeit und geringere Fehleranfälligkeit impliziert. In diesem Fall ist die rekursive Variante sogar gewöhnlich schneller, da der Laufzeitstack, zu dessen Verwaltung die technischen Möglichkeiten des Computers voll ausgenutzt werden können, effizienter verwaltet wird als ein vom Programm verwalteter Stapel.

Zu b) Benötigt eine iterative Lösung keinen vom Programm verwalteten Stapel, dann handelt es sich oft um ein „lineares Problem“ (z.B. Fakultätsfunktion, sequentielle Suche in einer linearen Datenstruktur usw.). Eine rekursive Lösung führt hier gewöhnlich zu linearer Rekursionstiefe und damit zu einem Laufzeitstack linearer Größe. Diese Nachteile kennt die iterative Lösung nicht. Zudem ist ihre Programmkomplexität meist vergleichbar mit der rekursiven Lösung, da kein Stapel vom Programm verwaltet werden muss.

Für viele Problemklassen stellen rekursive Algorithmen elegante und effiziente Lösungen dar. Dazu gehören z.B. Probleme, bei denen eine große Zahl möglicher Lösungen geprüft werden muss oder eine komplizierte Suche innerhalb einer Menge von Möglichkeiten (z.B. Labyrinthsuche) zu organisieren ist. Im Kurs „Datenstrukturen“ wird Ihnen die Rekursion als Lösungsmethodik in vielen Anwendungen wiederbegegnen.

### 6.3 Programmierstil (Gesamtübersicht)

#### Hinweise

- Im letzten Kapitel sind keine neuen Programmierstilhinweise oder -regeln hinzugekommen.
- In den folgenden Abschnitten finden Sie die endgültige Übersicht aller Muss-regeln.

Programme können, nachdem sie implementiert und getestet worden sind, in den seltensten Fällen ohne Änderung über einen längeren Zeitraum hinweg eingesetzt werden. Tatsächlich ist es meist so, dass die Anforderungen nach der Fertigstellung verändert oder erweitert werden und während des Betriebs bislang unerkannte Mängel oder Fehler auftreten, die zu beseitigen sind. Programme müssen während ihres Einsatzes *gewartet* werden. Zu Wartungszwecken muss der Programmtext immer wieder gelesen und verstanden werden. Die Lesbarkeit eines Programms spielt also eine große Rolle. Je größer ein Programm ist, desto wichtiger wird das Kriterium der Lesbarkeit. Dies gilt besonders für industrielle Anwendungen, bei denen häufig der bzw. die Entwickler nicht mehr verfügbar ist bzw. sind und Änderungen von Dritten vorgenommen werden müssen.

Die Lesbarkeit eines Programms hängt einerseits von der verwendeten Programmiersprache und andererseits vom *Programmierstil* ab. Der Programmierstil beeinflusst die Lesbarkeit eines Programms mehr als die verwendete Programmiersprache. Ein stilistisch gut geschriebenes C- oder COBOL-Programm kann besser lesbar sein als ein schlecht geschriebenes Pascal-Programm.

Wir bemühen uns bei unseren Beispielen um einen guten Programmierstil. Zum einen wollen wir Ihnen einen guten Programmierstil „von klein auf“ implizit vermitteln, d.h. Sie sollen nur gute und gar nicht erst schlechte Beispiele kennenlernen. Zum anderen bezwecken wir durch die erhöhte Lesbarkeit auch ein leichteres Verständnis der Konzepte, die wir durch Programme bzw. Prozeduren und Funktionen vermitteln wollen.

Programmierstil ist, wie der Name vermuten lässt, in gewissem Umfang Geschmackssache. Auch können starre Richtlinien in Einzelfällen unnatürlich wirken. Dennoch sind die Erfahrungen mit der flexiblen Handhabung solcher Richtlinien durchweg schlecht, so dass gut geführte Programmierabteilungen und Softwarehäuser einen hauseigenen Programmierstil verbindlich vorschreiben und die Einhaltung - häufig über Softwarewerkzeuge - rigoros kontrollieren.

Genau das werden wir auch in diesem Kurs tun (dies erleichtert auch die Korrektur der über tausend Einsendungen je Kurseinheit) und Regeln vorschreiben, deren Nichteinhaltung zu Punktabzügen führt. Dabei unterscheiden wir zwischen „Muss-Regeln“, die in jedem Fall befolgt werden müssen, und „Kann-Regeln“, von denen im Einzelfall abgewichen werden kann. Muss-Regeln sind durch Fettdruck und ent-

Programmierstil

sprechende Marginalien gekennzeichnet. **Das Nichteinhalten von Muss-Regeln wirkt sich in Punktabzügen aus.**

Wir haben uns in den Programmbeispielen nicht immer streng an die Kann-Regeln gehalten. Das hängt damit zusammen, dass die Empfehlungen zum Teil bereits für den professionellen Einsatz gedacht sind und unsere kleinen Beispiele manchmal überfrachtet hätten. So können z.B. gewisse Layout-Regeln (etwa für den Vereinbarungsteil), die bei größeren Programmen sinnvoll sind, kleinere Programme unnötig aufblähen. Außerdem nehmen wir aus Gründen der Lesbarkeit auf den Seitenumbruch Rücksicht, was sich nicht immer mit den Regeln verträgt.

Die Ausführungen zum Programmierstil wurden von Kapitel zu Kapitel aktualisiert und ergänzt. Dabei wurden die Regeln „kumuliert“, d.h. die Programmierstilregeln von Kapitel 5 schließen die Regeln für Kapitel 3 und Kapitel 4 mit ein. Am Ende des Kurses finden Sie nun sämtliche kursrelevanten Regeln aufgeführt.

### 6.3.1 Bezeichnerwahl

Stilistisch gute Programme werden oft als selbstdokumentierend bezeichnet. In erster Linie wird dabei auf die geschickte Wahl von Bezeichnern angespielt, die es dem Leser erlauben, die Programmlogik schnell und ohne die Hilfe zusätzlicher Kommentare zu verstehen. (Daraus folgt aber nicht, dass Kommentare überflüssig werden!) Dies wird erreicht, wenn die Bedeutung einer Anweisung aus den Namen der beteiligten Variablen, Funktionen usw. und der verwendeten Schlüsselwörter bzw. Operatoren hervorgeht.

Richtig gewählte Namen sind so kurz wie möglich, aber lang genug, um die Funktion oder das Objekt, das sie bezeichnen, so zu beschreiben, dass jemand, der mit der Aufgabenstellung vertraut ist, sich darunter etwas vorstellen kann. Selten gebrauchte Namen können etwas länger als häufig gebrauchte Namen sein.

Für die Bezeichner in Pascal-Programmen geben wir folgende Regeln vor:

- Bezeichner sind aussagekräftig (sprechend) und orientieren sich an dem Problem, welches das Programm löst.
- Bezeichner werden zunächst grundsätzlich so geschrieben, wie es die Rechtschreibung vorgibt. Eine Ausnahme davon wird gemacht, wenn mehrere Worte (oder deren Abkürzungen) zu einem Bezeichner zusammengezogen werden. In diesem Fall werden die Anfangsbuchstaben der zusammengezogenen Worte (bzw. Abkürzungen) im Bezeichner groß geschrieben. Der erste Buchstabe des Bezeichners wird groß oder klein geschrieben, je nachdem, ob das erste Wort (bzw. dessen Abkürzung) groß oder klein geschrieben wird.
- **Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen.**

- **Typbezeichnern wird ein `t` vorangestellt.**

Bezeichner von Zeigertypen beginnen mit `tRef`.

Bezeichner formaler Parameter beginnen mit `in`, `io` oder `out`.

Es ist hilfreich, wenn an einem Bezeichner direkt abgelesen werden kann, ob er eine Konstante, einen Typ, eine Variable oder einen formalen Parameter bezeichnet. Dies erspart lästiges Nachschlagen der Vereinbarungen und vermindert Fehlbenutzungen. Wir erreichen eine Unterscheidung durch kurze Präfixe sowie Groß- und Kleinschreibungsregeln.

- Für Variablen bieten sich häufig Substantive als Bezeichner an, für Bezeichner von Funktionen und Prozeduren werden am besten Verben (welche die Tätigkeit beschreiben) verwendet, für Zustandsvariablen oder boolesche Funktionen eignen sich Adjektive oder Adverbien.

### Abkürzungen

Bei langen Namen ist eine gewisse Vorsicht angebracht, da diese umständlich zu handhaben sind und die Lesbarkeit von Programmen vermindern, weil sie die Programmstruktur verdecken können. Unnötig lange Namen sind Fehlerquellen, der Dokumentationswert eines Namens ist nicht proportional zu seiner Länge.

Allerdings ist es oft schwierig, kurze und gleichzeitig präzise Bezeichner zu finden. In diesem Fall werden für Bezeichner Abkürzungen gewählt. Wir wollen dafür einige Regeln zusammentragen:

- Ein spezieller Kontext erleichtert das Abkürzen von Bezeichnern, ohne dass diese dadurch an Bedeutung verlieren. Allgemein gilt: Wird eine Abkürzung benutzt, sollte sie entweder im Kontext verständlich oder allgemeinverständlich sein.
- Wenn Abkürzungen des allgemeinen Sprachgebrauchs existieren, sind diese zu verwenden (z.B. `Pers` für Person oder `Std` für Stunde).
- Häufig gebrauchte Namen können etwas kürzer als selten gebrauchte Namen sein.
- Eine Abkürzung muss enträtselbar und sollte aussprechbar sein.
- Beim Verkürzen sind Wortanfänge wichtiger als Wortenden, Konsonanten sind wichtiger als Vokale.
- Es ist zu vermeiden, dass durch Verkürzen mehrere sehr ähnliche Bezeichner entstehen, die leicht verwechselt werden können.

### 6.3.2 Programmtext-Layout

Die Lesbarkeit eines Programms hängt auch von der äußeren Form des Programms ab. Für die äußere Form von Programmen gibt es viele Vorschläge und Beispiele aus der Praxis, die sich aber oft am Einzelfall oder an einer konkreten Programmiersprache orientieren. Für unsere Zwecke geben wir die folgenden Regeln an:

Muß-Regel 3

- Die Definitionen und Deklarationen werden nach einem eindeutigen Schema gegliedert. Dies fordert bereits die Pascal-Syntax, aber nicht in allen Programmiersprachen ist die Reihenfolge von Definitionen und Deklarationen vorgeschrieben.
- **Jede Anweisung beginnt in einer neuen Zeile; `begin` und `end` stehen jeweils in einer eigenen Zeile.**
- Der Einsatz von redundanten Klammern und Leerzeichen in Ausdrücken kann die Lesbarkeit erhöhen. Insbesondere sind Klammern dann einzusetzen, wenn die Priorität der Operatoren unklar ist. Die Prioritäten in verschiedenen Programmiersprachen folgen eventuell unterschiedlichen Hierarchien. Aber auch hier gilt wie bei Bezeichnerlängen: Ein Zuviel hat den gegenteiligen Effekt!
- Die hervorgehobene Darstellung von Schlüsselwörtern einer Programmiersprache unterstützt die Verdeutlichung der Programmstruktur. Wir heben Schlüsselwörter durch Fettdruck hervor. In Programmtexten, die von Hand geschrieben werden, wird durch Unterstreichung hervorgehoben.
- Kommentare werden so im Programmtext eingefügt, dass sie einerseits leicht zu finden sind und ausreichende Erläuterungen bieten, andererseits aber die Programmstruktur deutlich sichtbar bleibt (vgl. auch Abschnitt "Kommentare").

### Einrückungen

Die Struktur eines Programms wird durch Einrückungen hervorgehoben. Wir geben die folgenden Regeln an:

Muß-Regel 4

- **Anweisungsfolgen werden zwischen `begin` und `end` um eine konstante Anzahl von 2 - 4 Stellen eingerückt. `begin` und `end` stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.**

Muß-Regel 5

- **Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.**
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **`end`** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist (vgl. auch Abschnitt "Kommentare").

Beispiele für die letzten drei Regeln:

```

if <Bedingung> then
begin
    <Anweisungsfolge>
end
else { if not <Bedingung> }
begin
    <Anweisungsfolge>
end { if <Bedingung> };

while <Bedingung> do
begin

```

```

    <Anweisungsfolge>
end; { while <Bedingung> }

repeat
    <Anweisungsfolge>
until <Bedingung>;

for <Variable> := <Startwert> to <Zielwert> do
begin
    <Anweisungsfolge>
end; { for <Variable> }

```

- In einem Block werden die Definitions- und Deklarationsteile ebenfalls eingerückt. Die entsprechenden Schlüsselwörter stehen eingerückt in einer eigenen Zeile. Die vereinbarten Konstanten, Typen und Variablen werden darunter linksbündig angeordnet, in jeder Zeile sollte nur ein Bezeichner stehen. **begin** und **end** des Programmblocks werden nicht eingerückt. Z.B.:

```

program einruecken (input, output);
{ ein Programm zur Verdeutlichung von Layoutregeln }

const
PI = 3.1415927;
ANTWORT = 42;

type
tWochentag = (Mon, Die, Mit, Don, Fre, Sam, Son);

var
Heute,
Gestern : tWochentag;

begin
    write ('Die Antwort ist: ');
    writeln (ANTWORT)
end. { einruecken }

```

In Ausnahmefällen, damit z.B. ein Programm (noch) auf eine Seite passt, kann von diesen Regeln abgewichen werden. Falls das Programm in jedem Fall durch einen Seitenumbruch getrennt wird, gelten die Layoutregeln wieder.

### Leerzeilen

Leerzeilen dienen der weiteren Gliederung des Programmtextes. Es empfiehlt sich, zusammengehörige Definitionen, Deklarationen oder Anweisungen auch optisch im Programmtext zusammenzufassen. Die einfachste Art ist die Trennung solcher Gruppen durch Leerzeilen. Es bietet sich an, einer solchen Gruppe einen zusammenfassenden Kommentar voranzustellen.

Wir schlagen den Einsatz von Leerzeilen in folgenden Fällen vor:

- Die verschiedenen Definitions- und Deklarationsteile werden durch Leerzeilen getrennt.
- Einzelne Funktions- bzw. Prozedurdeklarationen werden durch Leerzeilen voneinander getrennt.
- Anweisungsfolgen von mehr als ca. 10 Zeilen sollten durch Leerzeilen in Gruppen unterteilt werden.

### Prozedur- und Funktionsdeklarationen

Die Layoutregeln für Prozedur- und Funktionsdeklarationen haben wir zusammen mit anderen Regeln im Abschnitt "Prozeduren und Funktionen" aufgeführt.

Bei den in diesem Abschnitt "Programmtext-Layout" angesprochenen Richtlinien ist wichtig, dass selbstauferlegte Konventionen konsequent eingehalten werden, um ein einheitliches Aussehen aller Programmbausteine zu erreichen. Um dem gesamten Programmtext ein konsistentes Aussehen zu geben, werden automatische *Programmtext-Formater*, sogenannte *Prettyprinter*, eingesetzt. Solche Programme formatieren einen (syntaktisch korrekten) Programmtext nachträglich gemäß fester oder in einem gewissen Rahmen variabler Regeln. Durch *syntaxgesteuerte Editoren* erreicht man vergleichbare Ergebnisse direkt bei der Programmtext-Eingabe.

### 6.3.3 Kommentare

Kommentare sind nützliche und notwendige Hilfsmittel zur Verbesserung der Lesbarkeit von Programmen. Sie sind wichtige Bestandteile von Programmen, und ihre Abwesenheit ist ein Qualitätsmangel.

Die Verwendung von Kommentaren ist in jeder höheren Programmiersprache möglich. Wir unterscheiden Kommentare, welche die Historie, den aktuellen Stand (Version) oder die Funktion von Programmen beschreiben - sie werden an den Programmanfang gestellt - und solche, die Objekte, Programmteile und Anweisungen erläutern - sie werden im Deklarations- und Anweisungsteil benutzt.

Die folgenden Konventionen bleiben an einigen Stellen unscharf, wichtig ist auch hier, dass man sich konsequent an die selbst gegebenen Regeln hält.

- Kommentare werden während der Programmierung eingefügt und nicht erst nachträglich ergänzt. Selbstverständlich werden die Kommentare bei Änderungen angepasst.
- Kommentare sind so zu plazieren, dass sie die Programmstruktur nicht verdecken, d.h. Kommentare werden mindestens so weit eingerückt wie die entsprechenden Anweisungen. Trennende Kommentare wie Sternchenreihen oder Linien werden so sparsam wie möglich eingesetzt, da sie den Programmtext leicht zerschneiden.



- **Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst. Eine Kommentierung der Ein- und Ausgabedaten gehört gewöhnlich dazu.** Bei einem schwierigen Algorithmus können ggf. allgemeine, aber auf keinen Fall detaillierte Hinweise gegeben werden.
- Jeder Funktions- und Prozedurkopf wird ähnlich einem Programmkopf kommentiert (vgl. Abschnitt "Prozeduren und Funktionen").
- Die Bedeutung von Konstanten, Typen und Variablen wird erläutert, wenn sie nicht offensichtlich ist.
- Komplexe Anweisungen oder Ausdrücke werden kommentiert.
- Bei Kontrollanweisungen mit längeren Anweisungsfolgen sollte das abschließende **end** mit einem Kommentar versehen werden, der auf die zugehörige Kontrollstruktur verweist.
- Kommentare sind prägnant zu formulieren und auf das Wesentliche zu beschränken. Sie wiederholen nicht den Code mit anderen Worten, z.B.  
`i := i + 1; {erhoehe i um 1 },`  
sondern werden nur da eingesetzt, wo sie zusätzliche Information liefern.
- An entscheidenden Stellen wird der Zustand der Programmausführung beschrieben, z.B. `{ jetzt ist das Teilfeld bis Index n sortiert }.`

Muß-Regel 6

#### 6.3.4 Prozeduren und Funktionen

- **Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.**
- Für die Funktions- und Prozedurbezeichner werden möglichst suggestive Verben verwendet.
- **Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgangsparameter.**
- **Die Übergabeart jedes Parameters wird durch Voranstellen von `in`, `io` oder `out` vor den Parameternamen gekennzeichnet.**
- Jeder Parameter steht in einer eigenen Zeile.  
Diese Regel haben wir aus Platzgründen nicht immer befolgt. Zumindestens sollte immer gelten: In einer Zeile stehen maximal drei Parameter desselben Typs mit derselben Übergabeart.
- Das Ende einer Funktions- bzw. Prozedurdeklaration wird durch die Wiederholung des Funktions- bzw. Prozedurnamens in einem Kommentar nach dem abschließenden **end** des Anweisungsteils markiert.
- **Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.**
- **Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert (vgl. Abschnitt "Seiteneffekte").**

Muß-Regel 7

Muß-Regel 8

Muß-Regel 9

Muß-Regel 10

Muß-Regel 11

Muß-Regel 12

- **Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix `in`, wenn das Feld nicht verändert wird.**

Muß-Regel 13

- **Funktionsprozeduren werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.**

Muß-Regel 14

- **Wertparameter werden nicht als lokale Variable missbraucht.**

In Pascal ist das zwar möglich, da Wertparameter wie lokale Variable behandelt werden. Andere Programmiersprachen, z.B. Ada, verbieten Zuweisungen an Eingangsparameter. Das Verstehen der Programmstruktur wird gefördert, wenn Eingangsparameter nur gelesen werden und bei Bedarf zusätzliche lokale Variable deklariert werden.

### 6.3.5 Seiteneffekte

Eine Prozedur oder Funktion besitzt Seiteneffekte, wenn nicht nur die Änderungs- und Ausgangsparameter, sondern daneben weitere Programmdaten manipuliert werden. Die wichtigsten Seiteneffekte sind:

- Modifikation globaler Variablen
- Änderungs- oder Ausgangsparameter in Funktionen (Diese schlechte Möglichkeit haben wir Ihnen gar nicht erst vorgestellt.)

Vergegenwärtigen Sie sich noch einmal die Definition von lokalen bzw. globalen Variablen aus Kapitel 5. Ein negativer Aspekt globaler Variablen besteht darin, dass sie durch Seiteneffekte von Prozeduren bzw. Funktionen manipuliert werden können. Wird eine Variable als Parameter übergeben, ist sie dagegen im Sinne obiger Definition lokal. Seiteneffekte sind kaum durch Kommentare zu erklären und gefährliche Fehlerquellen, wie die folgenden einfachen Beispiele zeigen:

#### Beispiel 6.3.1

```
program Seiteneffekt1 (input, output);
{ dient der Demonstration von Seiteneffekten durch
  Manipulation globaler Variablen }

var
  a,
  b,
  c : integer;

function GlobalAendern (inX : integer): integer;
{ manipuliert die globale Variable a }
```

```

begin
  a := a + 1;
  GlobalAendern := a * inX
end; { GlobalAendern }

begin { Seiteneffekt1 }
  a := 10;
  b := 3;
  c := GlobalAendern (b) + GlobalAendern(b);
  writeln (c)
end. { Seiteneffekt1 }

```

Auf den ersten Blick erscheint es unproblematisch, die letzte Zuweisung durch

```
c := 2 * GlobalAendern (b);
```

zu ersetzen. Bei der Programmausführung erhält c im Ausgangsprogramm den Wert 69, in der Variante jedoch den Wert 66 zugewiesen.



### Beispiel 6.3.2

```

program Seiteneffekt2 (input, output);
{ dient der Demonstration von Seiteneffekten durch
  Referenzparameter in Funktionen }

var
  a,
  b,
  c : integer;

function MitInoutParameter (
  inX : integer;
  var ioY : integer) : integer;
{ verwendet und manipuliert den Referenzparameter
  ioY }

begin
  ioY := ioY + inX;
  MitInoutParameter := inX * ioY
end; { MitInoutParameter }

begin
  a := 4;
  b := MitInoutParameter (5, a);
  c := MitInoutParameter (5, a);
  writeln (b, c)
end. { Seiteneffekt2 }

```

In diesem Fall ändert die Funktion den Wert der als Referenzparameter übergebenen Variablen `a`. Daher werden den Variablen `b` und `c` verschiedene Werte (45 bzw. 70) zugewiesen, obwohl sich die Funktionsaufrufe von `MitInOutParameter` nicht sichtbar unterscheiden.



Seiteneffekte manipulieren Variable außerhalb des lokalen Speicherbereichs, also außerhalb des “Gesichtsfeldes” des Lesers. Gefährlich ist dies vor allem, weil anhand des Prozedurkopfes Seiteneffekte der „ersten Art“ (Manipulation globaler Variablen) nicht zu erkennen sind. Als unmittelbare Konsequenz ergibt sich daraus die Forderung, globale Daten, die in der Prozedur benötigt werden, stets als Parameter zu übergeben, obwohl die Regeln über den Gültigkeitsbereich dies nicht erzwingen.

Wir haben Referenzparameter in Prozeduren und Funktionen zur effizienten Übergabe großer Felder zugelassen (vgl. Abschnitt “Prozeduren und Funktionen”). Daher können Seiteneffekte “der zweiten Art” (Manipulation von Referenzparametern, die semantisch Wertparameter sind) nur durch Disziplin beim Programmieren vermieden werden.

### 6.3.6 Sonstige Merkgeln

Einige weitere Regeln wollen wir kurz auflisten:

- Es werden nach Möglichkeit keine impliziten (automatischen) Typanpassungen benutzt.
- Selbstdefinierte Typen werden nicht implizit definiert.
- Bei geschachtelten **if**-Anweisungen wird nach Möglichkeit nur der **else**-Zweig geschachtelt.
- Bietet die Programmiersprache verschiedene Schleifenkonstrukte (in Pascal: **while**, **repeat**, **for**), so ist eine dem Problem angemessene Variante zu wählen.
- **Die Laufvariable wird innerhalb einer for-Anweisung nicht manipuliert.**
- Häufig ist es aus Gründen der Lesbarkeit empfehlenswert, nicht an Variablen zu sparen und einige Hilfsvariablen mehr zu verwenden, als unbedingt nötig. So können komplizierte Berechnungen durch zusätzliche Variablen (mit aussagekräftigen Namen), die Teilergebnisse aufnehmen, wesentlich transparenter programmiert werden. Außerdem darf eine Variable innerhalb ihres Gültigkeitsbereichs nicht ihre Bedeutung wechseln. Es ist z.B. verwirrend, wenn eine Variable einmal die Durchläufe einer Schleife kontrolliert und an anderer Stelle das Ergebnis einer komplexen Berechnung speichert - “nur weil der Typ gerade passt”. Eine zusätzliche Variable ist hier von Vorteil. Auf die konsistente Bedeutung von Variablen sollte auch bei Verschachtelungen geachtet werden.

### 6.3.7 Strukturierte Programmierung

Zum Verständnis des berühmten Schlagwortes “Strukturierte Programmierung” fehlt Ihnen noch die Erläuterung von Sprunganweisungen und Marken. Wir haben

Ihnen dieses Pascal-Konstrukt bisher vorenthalten - aus gutem Grund, wie Sie sehen werden. Wir werden deshalb nur kurz darauf eingehen.

Eine Anweisung kann in Pascal markiert werden, indem eine *Marke* (**label**) vorangestellt wird. Eine Marke ist eine vorzeichenlose `integer`-Konstante, gefolgt von einem Doppelpunkt. Einige Pascal-Compiler beschränken Marken auf höchstens vier Stellen.

Marke

Die Markierung von Anweisungen schafft die Möglichkeit, sich an einer anderen Stelle des Programms auf diese Anweisung zu beziehen, und zwar in einer *Sprunganweisung* (**goto**-statement). Sie besteht aus dem Schlüsselwort **goto**, gefolgt von einer Marke, die das Sprungziel angibt.

Sprunganweisung

Wenn die Ausführung eines Programms zu einer Sprunganweisung gelangt, geht die Ausführung bei derjenigen Anweisung weiter, die mit der angegebenen Marke versehen ist, und nicht bei der, die unmittelbar auf die Sprunganweisung folgt. Natürlich können **goto**-statements auch Teile von zusammengesetzten Anweisungen sein. Als Anweisungsteil einer **if**-Anweisung sind beispielsweise bedingte Sprünge möglich. Auch kann eine Schleife unter einer anderen Bedingung als der Schleifenabbruchbedingung verlassen werden.

Der historische Begriff der *strukturierten Programmierung* geht auf eine Arbeit von Dijkstra in den späten 60-er Jahren zurück, der erkannte, dass die unkontrollierte Verwendung der **goto**-Anweisung die Hauptursache für unzuverlässige und fehlerhafte Programme (sogenannte Spaghetti-Programme) war, und als Konsequenz daraus ihre Verwendung bei der Programmierung streng reglementierte. Zur Steuerung des Kontrollflusses sind in der strukturierten Programmierung als Kontrollkonstrukte neben der Hintereinanderausführung (Sequenz) nur noch Iteration und Selektion erlaubt. In imperativen Sprachen mit entsprechenden Sprachkonstrukten (Schleifen, **if**-Anweisung) entspricht diese Restriktion schlicht dem Verbot von **goto**-Anweisungen.

Die Vorschläge von Dijkstra trafen anfangs auf große Skepsis. Konnte es zu jedem komplizierten "Spaghetti-Programm" ein semantisch äquivalentes "strukturiertes" Programm geben? Ein grundlegendes Resultat der Berechenbarkeitstheorie bestätigt, dass zu jeder berechenbaren Funktion ein Algorithmus existiert, der ausschließlich aus den Konstrukten Sequenz, Selektion und Iteration von Einzelberechnungen aufgebaut ist.

Ursprünglich mit dem Ziel entwickelt, die Korrektheitsbeweise von Programmen zu vereinfachen, war die strukturierte Programmierung der erste Schritt weg von dem Chaos früher Programme hin zu einer disziplinierten Programmentwicklung. Das Vermeiden von **gotos** offenbarte schnell einige weitere Vorteile, die bis heute Gültigkeit besitzen:

Vorteile der strukturierten Programmierung

- Programmierer sind gezwungen, sorgfältig über ihr Programm nachzudenken, bevor sie mit der Codierung beginnen können. Weniger logische Fehler in der Programmentwicklung sind die Folge.

- Strukturierte Programme gliedern sich in geschachtelte Anweisungsfolgen mit genau einem Ein- und Ausgang. Sie können sequentiell gelesen werden und sind damit leichter verständlich und überprüfbar.
- Abbruchbedingungen von Schleifen sind stets explizit, d.h. die einzige Bedingung, unter der die Schleife verlassen werden kann, ist die Abbruchbedingung im Bedingungsteil der Schleife. Eine implizite Abbruchbedingung wäre ein bedingter Sprung aus dem Anweisungsteil der Schleife.

Welche Bedeutung die strukturierte Programmierung heute besitzt, zeigt sich z.B. an den Definitionen moderner Programmiersprachen. Einige imperative und objektorientierte Sprachen wie z.B. Modula-2 oder Eiffel besitzen überhaupt keine **goto**-Anweisung mehr, und in anderen wie z.B. Ada wird ihre Existenz nur noch mit Anforderungen aus der automatischen Programmgenerierung (Übertragung von Spezifikationen oder Software, die in einer anderen Sprache geschrieben ist) gerechtfertigt und vor unkontrollierter Benutzung ausdrücklich gewarnt.

Muß-Regel 16

- **Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen. Dadurch stimmen die (statische) Programmstruktur und der (dynamische) Kontrollfluss bei der Ausführung überein und der Programmtext kann linear, d.h. ohne Sprünge gelesen werden.**

### 6.3.8 Zusammenfassung der Muss-Regeln

Muß-Regel 1

- Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen

Muß-Regel 2

- Typbezeichnern wird ein `t` vorangestellt.  
Bezeichner von Zeigertypen beginnen mit `tRef`.  
Bezeichner formaler Parameter beginnen mit `in`, `io` oder `out`.

Muß-Regel 3

- Jede Anweisung beginnt in einer neuen Zeile;  
**begin** und **end** stehen jeweils in einer eigenen Zeile

Muß-Regel 4

- Anweisungsfolgen werden zwischen **begin** und **end** um eine konstante Anzahl von 2 - 4 Stellen eingerückt. **begin** und **end** stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.

Muß-Regel 5

- Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.

Muß-Regel 6

- Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst.

Muß-Regel 7

- Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.

Muß-Regel 8

- Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgangsparameter.

Muß-Regel 9

- Die Übergabeart jedes Parameters wird durch Voranstellen von `in`, `io` oder `out` vor den Parameternamen gekennzeichnet.

Muß-Regel 10

- Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.

Muß-Regel 11

- Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert.

- |   |              |
|---|--------------|
| • Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix <code>in</code> , wenn das Feld nicht verändert wird. | Muß-Regel 12 |
| • Funktionsprozeduren werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.   | Muß-Regel 13 |
| • Wertparameter werden nicht als lokale Variable missbraucht.   | Muß-Regel 14 |
| • Die Laufvariable wird innerhalb einer <code>for</code> -Anweisung nicht manipuliert.  | Muß-Regel 15 |
| • Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen.   | Muß-Regel 16 |





## 7. Exkurs: Prozedurale Programmentwicklung

Dieser Exkurs über prozedurale Programmentwicklung gehört nicht zum Pflichtpensum des Kurses 1613. Zu diesem Kapitel gibt es weder Einsendeaufgaben noch Aufgaben in den abschließenden Klausuren. Die Ausführungen dienen lediglich der Abrundung des Wissens interessierter Kursteilnehmer und sollen Ihnen vor Augen führen, daß wir lediglich einen ersten Eindruck von der (imperativen) Programmierung vermittelt haben.

Wir haben in diesem Kurs bisher kleine Programme oder einzelne Prozeduren vorgestellt, die mit etwas Übung „direkt“ hingeschrieben werden können. Dieses Vorgehen wird bei komplizierteren Programmen nicht mehr ausreichen. In diesem Abschnitt wollen wir eine Möglichkeit zeigen, wie mit Hilfe der *prozeduralen Zerlegung* Programme, die eine starke algorithmische Komponente und eine geringe Größe von wenigen hundert Zeilen besitzen, sinnvoll entwickelt werden können.

Die prozedurale Programmentwicklung stellt selbst auch wieder nur einen kleinen Ausschnitt aus der Welt der Entwicklung komplexer Software dar. Die Entwicklung größerer Softwaresysteme gehört in den Bereich des *Software Engineering* und wird in den entsprechenden Kursen ausführlich behandelt. Dort spielt die *objektorientierte Programmentwicklung* eine große Rolle.

### 7.1 Prozedurale Zerlegung und Verfeinerung

In Kapitel 5 haben wir bereits auf die Bedeutung von Prozeduren für die Programmentwicklung hingewiesen. Neben der Mehrfachverwendung und der Wiederverwendung erlauben Prozeduren eine Programmzerlegung, welche die Übersichtlichkeit und Lesbarkeit eines Programms erhöht. Daneben hilft die prozedurale Zerlegung aber auch bei der systematischen Entwicklung kleinerer Programme.

Ein *Grundgerüst* für ein Programm zur Lösung relativ einfacher Aufgaben läßt sich leicht angeben. Es gehorcht gewöhnlich folgendem Schema:

```
program Grundgeruest (input, output);
...
begin
  EingabeOderInitialisierung (...);
  Verarbeitung (...);
  Ausgabe (...);
end. { Grundgeruest }
```

Die Reihenfolge der Teilaufgaben Eingabe, Verarbeitung und Ausgabe ist für viele Programme (mit nur wenig Benutzerinteraktion) charakteristisch und spiegelt sich oft sogar in der Architektur von Rechenanlagen wider, weshalb hierfür die Bezeichnung EVA-Prinzip geprägt wurde.

prozedurale  
Zerlegung

Software Engineering  
objektorientierte  
Programmentwick-  
lung

Grundgerüst

EVA-Prinzip

Die drei Prozeduraufrufe machen die Programmstruktur deutlich. Jede Prozedur erfüllt eine (typische) Teilaufgabe, die bei einem konkreten Beispiel noch genauer spezifiziert werden muß. Eine solche Teilaufgabe kann in den meisten Fällen nicht sofort als Pascal-Code notiert werden, weil sie noch zu kompliziert ist. In diesem Fall zerlegen wir die Prozeduren weiter, d.h. „verfeinern“ sie, bis elementare Teilaufgaben erreicht sind, die unmittelbar codiert werden können.

Wirth formuliert dieses Vorgehen als allgemeines Prinzip:

1. Zerlege die Aufgabe in Teilaufgaben.
2. Betrachte jede Teilaufgabe für sich, und zerlege sie wieder in Teilaufgaben, bis diese so einfach geworden sind, daß sie programmtechnisch formuliert werden können.

Methode der schrittweisen Verfeinerung

Dieses Prinzip nennt Wirth die *Methode der schrittweisen Verfeinerung*. Das Prinzip besagt, daß wir uns bei einer komplexen Aufgabe nicht sofort auf die Behandlung der Einzelheiten stürzen, sondern erst einmal versuchen sollen, die Aufgabe in ihre großen Bestandteile zu zerlegen. Die daraus resultierenden Teilaufgaben sind in jedem Fall von geringerer Komplexität als die Gesamtaufgabe und daher auch einfacher zu lösen. Die fortschreitende Zerlegung führt zu immer kleineren Teilaufgaben bei gleichzeitig fortschreitender Präzisierung. Der Prozeß ist zu Ende, wenn die Teilaufgaben trivial bzw. alle Details hinreichend präzisiert sind.

funktionsorientiertes Vorgehen  
datenstrukturorientiertes Vorgehen

Bei der Verfeinerung gehen wir *funktionsorientiert* (operations-, kontrollflußorientiert) vor und nicht *datenstrukturorientiert*, denn wir verfeinern bzw. präzisieren Prozeduren und Funktionen und keine Datenstrukturen. Dies ist typisch für die prozedurale Programmentwicklung, während ein datenstrukturorientiertes Vorgehen in der modularen und objektorientierten Programmentwicklung eine wichtige Rolle spielt.

iterative Vorgehensweise

Im Verlauf der Verfeinerung kann sich aber auch herausstellen, daß die gewählte Zerlegung ungeschickt war und zu Schwierigkeiten bei der weiteren Verfeinerung führt. In diesem Fall wird ein Zerlegungsschritt zurückgenommen oder im schlimmsten Fall wieder von vorn begonnen. Die Vorgehensweise ist also *iterativ*.

Prinzip der Abstraktion

Die Verfeinerung geht vom Allgemeinen zum Speziellen, d.h. von einer abstrakten Formulierung hin zu einer genauen Spezifikation. Dieses Vorgehen folgt einem der wichtigsten Prinzipien zur Beherrschung von Komplexität, nämlich dem *Prinzip der Abstraktion*. Dieses Prinzip besagt, dass man sich bei der Betrachtung eines Problems auf wesentliche Aspekte konzentriert und Detailprobleme ausklammert. Bei der Verfeinerung geht man von einem hohen Abstraktionsniveau (wenige fundamentale Aspekte und keine Details) schrittweise über immer niedrigere, d.h. immer mehr Details einbeziehende, Abstraktionsstufen bis zu der Stufe mit der feinsten Granularität, auf der alle ProblemDetails ausspezifiziert sind.

Top-Down-Entwicklung

Die schrittweise Verfeinerung ist eine *Top-Down-Entwicklung*, da ausgehend von der Aufgabenstellung eine Aufgabe solange top down (von oben nach unten) in Teilaufgaben zerlegt wird, bis Operationen vorliegen, die leicht als Prozeduren oder Funktionen formuliert werden können.

## 7.2 Ein Beispiel

Wir wollen die prozedurale Zerlegung und Verfeinerung am Beispiel des sogenannten Acht-Damen-Problems demonstrieren.

### 7.2.1 Aufgabenstellung

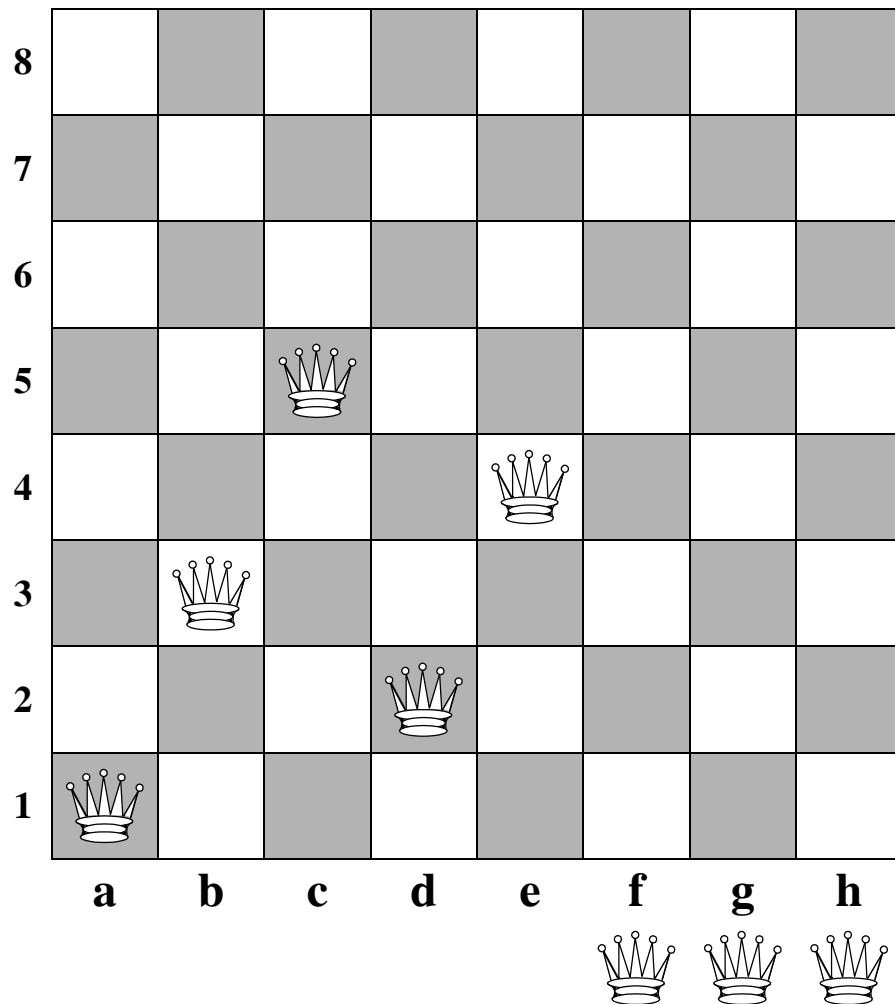
Acht Damen sind auf einem Schachbrett so aufzustellen, daß sie sich gegenseitig nicht schlagen können, d.h. keine zwei Damen dürfen in derselben Zeile, Spalte oder Diagonalen stehen. Da in jeder Spalte eine Dame stehen muß, soll das Programm die einzelnen Lösungen in der Form

1 5 8 6 3 7 2 4

ausgeben, d.h. die a-Dame steht in Zeile 1, die b-Dame in Zeile 5 usw.

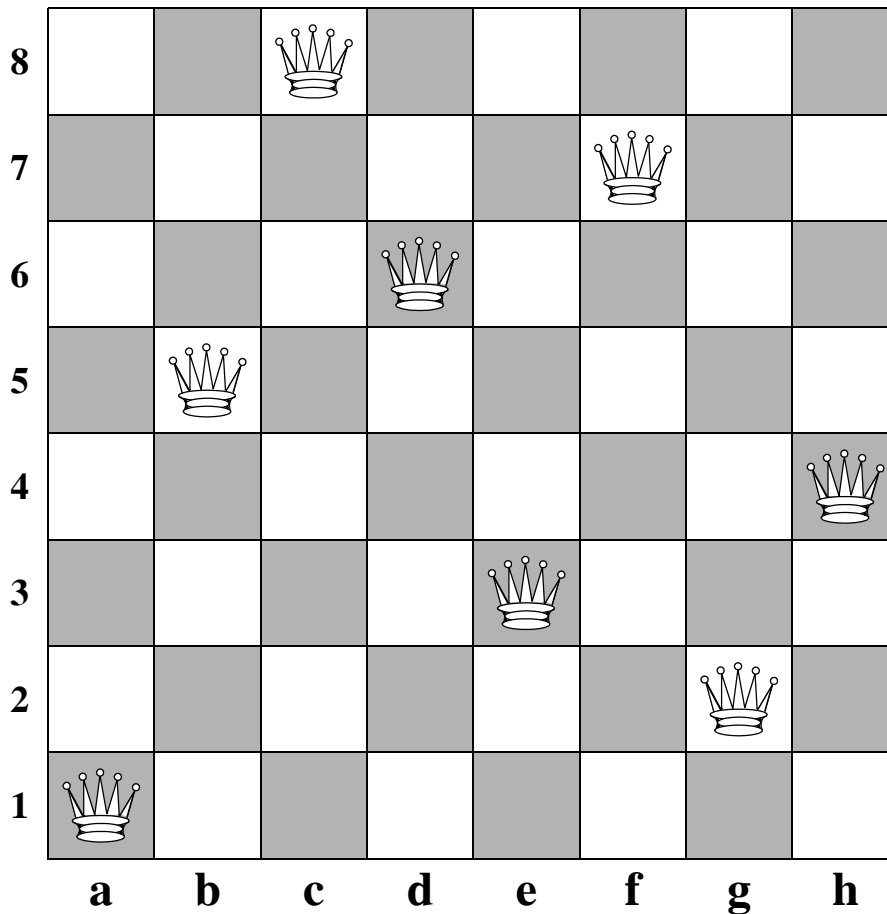
### 7.2.2 Lösungsstrategie

Zu Beginn setzen wir alle Damen vor das Schachbrett. Dann setzen wir die a-Dame auf die 1. Zeile. Danach lassen wir die b-Dame solange vorrücken, bis sie in Zeile 3 das erste unbedrohte Feld gefunden hat. Mit den Damen der folgenden Spalten verfahren wir nacheinander entsprechend. Wie die folgende Abbildung zeigt, sind nun für die f-Dame in ihrer Spalte alle möglichen Felder von den übrigen Damen bedroht:



Wir versuchen nun, (in umgekehrter Reihenfolge) die bereits gesetzten Damen auf andere unbedrohte Felder vorrücken zu lassen, um dadurch die Situation der noch nicht gesetzten Damen zu verbessern. Die Verschiebung der e-Dame auf die einzige unbedrohte Zeile 8 bringt allerdings für die f-Dame keine Vorteile. Für die e-Dame gibt es nun kein weiteres unbedrohtes Feld, deshalb wird sie wieder vor das Schachbrett gesetzt und die d-Dame rückt auf die unbedrohte 7. Zeile. Nun versuchen wir wieder, bei dieser neuen Ausgangslage wie oben beschrieben die übrigen Damen zu setzen.

Wir wiederholen dieses Verfahren solange, bis wir die erste Lösung gefunden haben:



Nach der Ausgabe der Lösung wird, um die nächste Lösung zu finden, die h-Dame wieder vor das Brett gesetzt, da sie keine weitere unbedrohte Zeile finden kann. Die g-Dame beginnt jetzt vorzurücken usw. Damit erreichen wir, daß schließlich alle Lösungen ausgegeben werden, denn auch die a-Dame wird auf diese Weise bis zur 8. Zeile vorrücken. Müßte sie darüber hinausrücken, ist keine weitere Lösung vorhanden und der Algorithmus wird beendet.

Derartige Algorithmen bezeichnet man als *Backtracking-Verfahren*. Ein anderer bekannter Backtracking-Algorithmus ist die Suche nach dem Weg aus einem Labyrinth. An jeder Weggabelung probieren wir zunächst den linken Abzweig, bis wir in eine Sackgasse geraten. Dann gehen wir zur letzten Gabelung zurück und untersuchen den nächsten Abzweig usw. Führen alle Abzweigungen von einer Gabelung in Sackgassen, gehen wir zur davor liegenden Gabelung zurück, um dort den nächsten Abzweig zu untersuchen usw.

Backtracking-  
Verfahren

### 7.2.3 Prozedurale Zerlegung und Verfeinerung

Wir werden nun aus dem Grundgerüst durch prozedurale Zerlegung und Verfeinerung ein lauffähiges Pascal-Programm entwickeln, das alle möglichen Plazierungen der acht Damen in einfacher Form auf dem Bildschirm ausgibt.

## Pseudo-Code

Auf dem Weg zum endgültigen Programm mischen sich in den Zwischenschritten der Zerlegung, d.h. auf den verschiedenen Abstraktionsebenen, Passagen, die bereits der Pascal-Syntax genügen, und noch umgangssprachlich formulierte Teile (*Pseudo-Code*), die noch nicht vollständig verfeinerte Aufgaben beschreiben. Zur besseren Unterscheidung sind die umgangssprachlichen Teile kursiv gedruckt und zusätzlich am Beginn der Zeile mit einem Pfeil markiert.

### Das Grundgerüst

Zunächst schaffen wir mit dem Grundgerüst den Rahmen für die weitere Arbeit.

```
program AchtDamenProblem (input, output);
{ bestimmt alle bedrohungsfreien Stellungen des Acht-Damen-Problems
  und gibt sie in der Form 1 5 8 6 3 7 2 4 (a-Dame in Zeile 1,
    b-Dame in Zeile 5, ...) auf dem Bildschirm aus }
```

-> *Definitionen*

-> *Deklarationen*

**begin**

-> *Eingabe bzw. Initialisierung;*

-> *Verarbeitung;*

-> *Ausgabe*

**end.** { AchtDamenProblem }

### Der erste Verfeinerungsschritt

In diesem Schritt stehen drei Aspekte im Mittelpunkt: die Auswahl der Datenstrukturen und ihre Initialisierung, die Formulierung des Lösungsalgorithmus in Pseudo-Code sowie die Spezifikation der Ausgabe.

- Die Plazierungen der Damen auf dem Schachbrett stellen wir in Form einer Abbildung dar. Jeder Dame (die Damen werden nach ihrer Spalte benannt, d.h. a-Dame, b-Dame usw.) wird die Zeile auf dem Schachbrett zugeordnet, in der sie gerade steht. Eine Damenposition vor bzw. hinter dem Brett entspricht der „Zeile“ 0 (Konstante DAVOR) bzw. 9 (Konstante DAHINTER). Zu Beginn stehen alle Damen vor dem Schachbrett, also in „Zeile“ 0.  
In den „Spalten“ links und rechts vom Brett stehen zwar keine Damen, dennoch nehmen wir sie aus programmtechnischen Gründen in den Aufzählungstyp `tSpalte` auf. (Auf diese sinnvolle Erweiterung des Typs wird man gewöhnlich erst in einem späteren Verfeinerungsschritt stoßen.)
- Den in der Lösungsmethode beschriebenen Backtracking-Algorithmus präzisieren wir in Pseudo-Code in der Prozedur `DameSetzen`.
- Jede Lösung des Acht-Damen-Problems stellen wir unmittelbar auf dem Bildschirm dar, um uns die Verwaltung der Stellungen in einer Ausgabedatenstruktur zu ersparen. Die Ausgabe der Lösungen erfolgt also einzeln und wird jeweils von der Prozedur `DameSetzen` angestoßen, wenn eine Lösung gefunden ist.

Wir haben übrigens mehrfach in geschachtelten **if**-Anweisungen den **then**-Zweig geschachtelt. Dieses Vorgehen spiegelt den intuitiven Algorithmus natürli-

cher wider, so daß wir uns dazu entschlossen haben. Wir sehen, daß einzelne Programmierstilempfehlungen kein Dogma darstellen. Im übrigen sind Umformungen, um nur noch geschachtelte **else**-Zweige zu erreichen, einfach vorzunehmen.

```

program AchtDamenProblem (input, output);
{ bestimmt alle bedrohungsfreien Stellungen des Acht-Damen-Problems
  und gibt sie in der Form 1 5 8 6 3 7 2 4 (a-Dame in Zeile 1,
  b-Dame in Zeile 5, ...) auf dem Bildschirm aus }

const
DAVOR = 0;          { Konstanten fuer die erste }
DAHINTER = 9;      { und letzte "Zeile" }

type
tSpalte = (links, a, b, c, d, e, f, g, h, rechts);
          { Acht Spalten und zwei fiktive Randspalten }
tZeile = DAVOR..DAHINTER;
tPosInSpalte = array [tSpalte] of tZeile;

var
DamenPos : tPosInSpalte; { Zeilenpositionen der acht Damen }

procedure StellungInitialisieren (
    var outInitDamenPos : tPosInSpalte);
{ Eingabe bzw. Initialisierung }

begin
->   Alle Damen werden vor das Schachbrett gesetzt
end; { StellungInitialisieren }

procedure LoesungAusgeben (
    inLoesDamenPos: tPosInSpalte);

begin
->   Ausgabe einer einzelnen Loesung des Acht-Damen-Problems
end; { LoesungAusgeben }

procedure DameSetzen (
    inSpalte : tSpalte;
    var ioSetzDamenPos : tPosInSpalte);
{ Berechnen und Ausgeben aller Loesungen mittels Backtracking,
  inSpalte ist die Spalte der aktuell zu setzenden Dame.
  Vorbedingung: Alle Damen von der a-Dame bis zur Dame
  auf der Vorgaengerspalte von inSpalte sind auf dem Brett
  platziert und bilden eine bedrohungsfreie Stellung. }

begin { DameSetzen }
  if inSpalte < a then
    { DameSetzen wurde fuer inSpalte = links aufgerufen, d.h.
      die a-Dame steht nicht mehr auf dem Brett,
      alle Loesungen sind gefunden,
      die Rekursion bricht ab.
      Es gibt keine Anweisungen, d.h. der then-Zweig ist leer. }
  else { inSpalte >= a }
  begin

```

```

        if inSpalte = rechts then
        { Alle Damen bedrohungsfrei platziert }
        begin
            LoesungAusgeben (ioSetzDamenPos);
->         h-Dame wieder vor das Brett setzen;
->         g-Dame weitersetzen { Rekursiver Aufruf von DameSetzen }
        end
        else { noch Damen zu setzen }
        begin
->         Dame in inSpalte auf unbedrohtes Feld vorruecken;
->         if Dame steht noch auf dem Brett
            { Dame fand ein unbedrohtes Feld } then
->             Nächste Dame setzen { Rekursiver Aufruf von DameSetzen }
            else { Dame in inSpalte fand kein unbedrohtes Feld }
            begin
->                 Dame in inSpalte wieder vor das Brett setzen;
->                 Dame der Vorgaengerspalte weitersetzen
                    { Rekursiver Aufruf von DameSetzen }
            end
        end { else - noch Damen zu setzen }
    end { else - inSpalte >= a }
end; { DameSetzen }

begin { AchtDamenProblem }
    StellungInitialisieren (DamenPos);
    DameSetzen (a, DamenPos)
end. { AchtDamenProblem }

```

### Der zweite Verfeinerungsschritt

Die Aufgaben der Prozeduren StellungInitialisieren, DameSetzen und LoesungAusgeben werden nacheinander präzisiert. Es treten zwei Fälle auf:

- Die in Frage kommende (Teil-)Aufgabe ist wenig umfangreich bzw. komplex. Die umgangssprachliche Beschreibung kann dann unmittelbar in Pascal-Anweisungen oder -Ausdrücke umgesetzt werden. Der Pseudo-Code bleibt als Kommentar erhalten.
- Bei komplizierteren Aufgaben wird die Präzisierung in Prozeduren verlagert und der Pseudo-Code durch einen Aufruf der entsprechenden Prozedur ersetzt. Die Beschreibungen sollten die Wahl des Prozedurnamens bestimmen und in die Kommentierung der neuen Prozedur übernommen werden. Den Aufbau der Prozeduren spezifizieren wir in der gleichen Weise wie die Prozedur DameSetzen, d.h. kontrollflußorientiert mit inhaltlichen Beschreibungen in Pseudo-Code.

In unserem Beispiel separieren wir die Prozedur DameVorruecken, durch die eine Dame von ihrem aktuellen Platz auf den nächsten unbedrohten Platz vorge-rückt wird. Die übrigen Teile können direkt, meist durch rekursive Aufrufe von DameSetzen oder einfache Operationen auf dem Parameter ioSetzDamenPos als Quelltext formuliert werden.



```

program AchtDamenProblem (input, output);
{ bestimmt alle bedrohungsfreien Stellungen des Acht-Damen-Problems
  und gibt sie in der Form 1 5 8 6 3 7 2 4 (a-Dame in Zeile 1,
  b-Dame in Zeile 5, ...) auf dem Bildschirm aus }

const
DAVOR = 0;      { Konstanten fuer die erste }
DAHINTER = 9;   { und letzte "Zeile" }

type
tSpalte = (links, a, b, c, d, e, f, g, h, rechts);
          { Acht Spalten und zwei fiktive Randspalten }
tZeile = DAVOR..DAHINTER;
tPosInSpalte = array [tSpalte] of tZeile;

var
DamenPos: tPosInSpalte; { Zeilenpositionen der acht Damen }

procedure StellungInitialisieren (
    var outInitDamenPos: tPosInSpalte);
{ Eingabe bzw. Initialisierung }
{ Alle Damen werden vor das Schachbrett gesetzt }

    var
    Spalte: tSpalte;

begin
    for Spalte := a to h do
        outInitDamenPos[Spalte] := DAVOR
end; { StellungInitialisieren }

procedure LoesungAusgeben (
    inLoesDamenPos: tPosInSpalte);
{ eine Loesung des Acht-Damen-Problems ausgeben }

    var
    Spalte: tSpalte;

begin
    for Spalte := a to h do
        write(inLoesDamenPos[Spalte]:2);
        writeln
end; { LoesungAusgeben }

procedure DameSetzen (
    inSpalte : tSpalte;
    var ioSetzDamenPos : tPosInSpalte);
{ Berechnen und Ausgeben aller Loesungen mittels Backtracking,
  inSpalte ist die Spalte der aktuell zu setzenden Dame.
  Vorbedingung: Alle Damen von der a-Dame bis zur Dame
  auf der Vorgaengerspalte von inSpalte sind auf dem Brett
  platziert und bilden eine bedrohungsfreie Stellung. }

    var
    AufDemBrett: boolean;

```

```

procedure DameVorruecken
    (
        inSpalte      : tSpalte;
        var ioRueckDamenPos : tPosInSpalte;
        var outAufDemBrett  : boolean);
{ Vorruecken der Dame in inSpalte auf ein unbedrohtes Feld
  oder hinter das Brett }

begin { DameVorruecken }
    repeat
->      Weiterschieben der Dame in inSpalte
    until
->      Keine der bereits platzierten Damen ist bedroht,
->      oder Dame steht hinter dem Brett;
->      outAufDemBrett := true: Dame steht auf unbedrohtem Feld
->      outAufDemBrett := false: Dame steht hinter dem Brett
    end; { DameVorruecken }

begin { DameSetzen }
    if inSpalte < a then
        { DameSetzen wurde fuer inSpalte = links als Vorgaenger
          von a aufgerufen, d.h. die a-Dame steht nicht mehr auf
          dem Brett, alle Loesungen sind gefunden,
          die Rekursion bricht ab.
          Es gibt keine Anweisungen, d.h. der then-Zweig ist leer. }
    else { inSpalte >= a }
    begin
        if inSpalte = rechts then
            { Alle Damen bedrohungsfrei platziert }
            begin
                LoesungAusgeben(ioSetzDamenPos);
                ioSetzDamenPos[h] := DAVOR;
                DameSetzen(g, ioSetzDamenPos)
            end
        else { noch Damen zu setzen }
        begin
            DameVorruecken(inSpalte, ioSetzDamenPos, AufDemBrett);
            if AufDemBrett { Dame fand unbedrohtes Feld } then
                DameSetzen(succ(inSpalte), ioSetzDamenPos)
            else { Dame fand kein unbedrohtes Feld }
            begin
                ioSetzDamenPos[inSpalte] := DAVOR;
                DameSetzen(pred(inSpalte), ioSetzDamenPos)
            end
        end { else - noch Damen zu setzen }
    end { else - inSpalte >= a }
end; { DameSetzen }

begin { AchtDamenProblem }
    StellungInitialisieren (DamenPos);
    DameSetzen (a, DamenPos)
end. { AchtDamenProblem }

```

### Der dritte Verfeinerungsschritt

Wir gehen analog zum zweiten Verfeinerungsschritt vor und verlagern einen Teil der Aufgaben aus DameVorruecken in die Prozedur DameUnbedroht. Dort

wird überprüft, ob eine gerade gezogene Dame von den bisherigen bedrohungsfrei auf dem Spielfeld stehenden Damen angegriffen wird. Wir beschränken uns bei der Darstellung auf diese beiden Prozeduren.

```

program AchtDamenProblem (input, output);

...

procedure DameVorruecken (
    inSpalte : tSpalte;
    var ioRueckDamenPos : tPosInSpalte;
    var outAufDemBrett : boolean);
{ Vorruecken der Dame in inSpalte auf ein unbedrohtes Feld
  oder hinter das Brett
  outAufDemBrett -> true: Dame steht auf unbedrohtem Feld
  outAufDemBrett -> false: Dame steht hinter dem Brett }

function DameUnbedroht (
    inSpalte : tSpalte;
    inPruefDamenPos : tPosInSpalte): boolean;
{ prueft, ob Dame in inSpalte von anderen Damen auf dem
  Brett bedroht wird.
  Vorbedingung: Alle Damen von der a-Dame bis zur Dame
  auf der Vorgaengerspalte von inSpalte sind auf dem Brett
  platziert und bilden eine bedrohungsfreie Stellung }

begin { DameUnbedroht }
->   Pruefen der Dame in inSpalte auf Zeilenbedrohung;
->   Pruefen der Dame in inSpalte auf Bedrohung in steigender Diagonale;
->   Pruefen der Dame in inSpalte auf Bedrohung in fallender Diagonale;
->   Rueckkehrwert true: Dame ist unbedroht
->   Rueckkehrwert false: Dame ist bedroht
end; { DameUnbedroht }

begin { DameVorruecken }
  repeat
    { Dame in inSpalte weiterschieben}
    ioRueckDamenPos[inSpalte] := ioRueckDamenPos[inSpalte] + 1
  until DameUnbedroht(inSpalte, ioRueckDamenPos)
    or (ioRueckDamenPos[inSpalte] >= DAHINTER);
  outAufDemBrett := ioRueckDamenPos[inSpalte] < DAHINTER
end; { DameVorruecken }

...

begin { AchtDamenProblem }
  StellungInitialisieren(DamenPos);
  DameSetzen(a, DamenPos)
end. { AchtDamenProblem }

```

### Das fertige Programm

Mit dem Ausprogrammieren der Prozedur DameUnbedroht wird die Implementierung abgeschlossen. In der Prozedur DameSetzen haben wir die äußere **if**-Anweisung äquivalent umgeformt, indem wir die Bedingung negiert haben, so daß

an die Stelle des **then**-Zweiges der **else**-Zweig tritt und der **then**-Zweig entfällt.

```

program AchtDamenProblem (input, output);
{ bestimmt alle bedrohungsfreien Stellungen des Acht-Damen-Problems
  und gibt sie in der Form 1 5 8 6 3 7 2 4 (a-Dame in Zeile 1,
  b-Dame in Zeile 5, ...) auf dem Bildschirm aus }

const
DAVOR = 0;          { Konstanten fuer die erste }
DAHINTER = 9;      { und letzte "Zeile" }

type
tSpalte = (links, a, b, c, d, e, f, g, h, rechts);
          { Acht Spalten und zwei fiktive Randspalten }
tZeile = DAVOR..DAHINTER;
tPosInSpalte = array [tSpalte] of tZeile;

var
DamenPos: tPosInSpalte; { Zeilenpositionen der acht Damen }

procedure StellungInitialisieren (
    var outInitDamenPos: tPosInSpalte);
{ Eingabe bzw. Initialisierung,
  alle Damen werden vor das Schachbrett gesetzt }

    var
    Spalte: tSpalte;

begin
    for Spalte := a to h do
        outInitDamenPos[Spalte] := DAVOR
    end; { StellungInitialisieren }

procedure LoesungAusgeben (inLoesDamenPos: tPosInSpalte);
{ eine Loesung des Acht-Damen-Problems ausgeben }

    var
    Spalte: tSpalte;

begin
    for Spalte := a to h do
        write(inLoesDamenPos[Spalte]:2);
        writeln
    end; { LoesungAusgeben }

procedure DameSetzen (
    inSpalte : tSpalte;
    var ioSetzDamenPos: tPosInSpalte);
{ Berechnen und Ausgeben aller Loesungen mittels Backtracking
  inSpalte ist die Spalte der aktuell zu setzenden Dame
  Vorbedingung: Alle Damen von der a-Dame bis zur Dame
  auf der Vorgaengerspalte von inSpalte sind auf dem Brett
  plaziert und bilden eine bedrohungsfreie Stellung }

```

```

var
AufDemBrett: boolean;

procedure DameVorruecken (
    inSpalte : tSpalte;
    var ioRueckDamenPos : tPosInSpalte;
    var outAufDemBrett : boolean);
{ Vorruecken der Dame in inSpalte auf ein unbedrohtes Feld
  oder hinter das Brett.
  outAufDemBrett -> true: Dame steht auf unbedrohtem Feld
  outAufDemBrett -> false: Dame steht hinter dem Brett }

function DameUnbedroht (
    inSpalte : tSpalte;
    inPruefDamenPos : tPosInSpalte) : boolean;
{ Prueft, ob Dame in inSpalte von anderen Damen auf dem
  Brett bedroht wird.
  Vorbedingung: Alle Damen von der a-Dame bis zur Dame
  auf der Vorgaengerspalte von inSpalte sind auf dem Brett
  plaziert und bilden eine bedrohungsfreie Stellung.
  Rueckkehrwert true: Dame ist unbedroht
  Rueckkehrwert false: Dame ist bedroht }

var
andereSpalte      : tSpalte;
SpaltenAbstand    : integer;
unbedroht         : boolean;

begin { DameUnbedroht }
  unbedroht := true;
  andereSpalte := a;
  { fuer die bisher gesetzten Damen wird jeweils geprueft,
    ob sie die Dame in inSpalte in der Zeile, der steigenden
    oder fallenden Diagonale bedrohen (diese Reihenfolge) }
  for SpaltenAbstand := ord(inSpalte) - ord(a) downto 1 do
  begin
    unbedroht :=
      unbedroht
      and
      (inPruefDamenPos[andereSpalte]
        <> inPruefDamenPos[inSpalte])
      and
      (inPruefDamenPos[andereSpalte] + SpaltenAbstand <>
        inPruefDamenPos[inSpalte])
      and
      (inPruefDamenPos[andereSpalte] - SpaltenAbstand <>
        inPruefDamenPos[inSpalte]);
    andereSpalte := succ (andereSpalte)
  end; { for SpaltenAbstand }
  DameUnbedroht := unbedroht
end; { DameUnbedroht }

begin { DameVorruecken }
  repeat
    { Weiterschieben der Dame in inSpalte }
    ioRueckDamenPos[inSpalte] := ioRueckDamenPos[inSpalte] + 1

```

```

        until DameUnbedroht(inSpalte, ioRueckDamenPos)
            or (ioRueckDamenPos[inSpalte] >= DAHINTER);
        outAufDemBrett := ioRueckDamenPos[inSpalte] < DAHINTER
    end; { DameVorruecken }

begin { DameSetzen }
    if inSpalte >= a then
        { es sind noch weitere Loesungen zu bestimmen }
    begin
        if inSpalte = rechts then
            { alle Damen bedrohungsfrei plaziert }
        begin
            LoesungAusgeben (ioSetzDamenPos);
            ioSetzDamenPos[h] := DAVOR;
            DameSetzen (g, ioSetzDamenPos)
        end
        else { noch Damen zu setzen }
        begin
            DameVorruecken(inSpalte, ioSetzDamenPos, AufDemBrett);
            if AufDemBrett { Dame fand unbedrohtes Feld } then
                DameSetzen(succ(inSpalte), ioSetzDamenPos)
            else { Dame fand kein unbedrohtes Feld }
            begin
                ioSetzDamenPos[inSpalte] := DAVOR;
                DameSetzen(pred(inSpalte), ioSetzDamenPos)
            end
        end { else - noch Damen zu setzen }
    end { if inSpalte >= a }
end; { DameSetzen }

begin { AchtDamenProblem }
    StellungInitialisieren(DamenPos);
    DameSetzen(a, DamenPos)
end. { AchtDamenProblem }

```

Fassen wir die wichtigsten Punkte noch einmal zusammen: Das Umsetzen der Lösungsmethode erfolgt in mehreren Schritten durch Zerlegung nach funktionalen Gesichtspunkten. Ausgehend von einem (nach dem EVA-Prinzip strukturierten) Hauptprogramm werden top down Teilaufgaben geringerer funktionaler Komplexität und kleineren Umfangs extrahiert und durch Prozeduren realisiert. Der Verfeinerungsprozeß bricht ab, wenn einfach lösbare Probleme entstanden sind, die direkt in Quelltext umgesetzt werden können. Orientiert am Kontrollfluß des Algorithmus stützen sich Prozeduren auf einer hohen Abstraktionsebene (z.B. DameSetzen) auf Prozeduren mit einem niedrigen Abstraktionsniveau ab (z.B. DameVorruecken). Die unterste Abstraktionsebene bilden elementare Prozeduren, die übergeordneten Prozeduren als Hilfsmittel bei der Realisierung von Teilaufgaben dienen und selbst keine Unterprozeduren mehr in Anspruch nehmen (z.B. DameUnbedroht). Um die hierarchische Struktur des Programms deutlich zu machen, haben wir die lokal ausgelagerten Teilaufgaben einer Prozedur jeweils in einer Unterprozedur realisiert.

Das Programm AchtDamenProblem hat einen entscheidenden Nachteil: Die Rekursionstiefe beträgt 3712. Dies liegt daran, daß ein rekursiver Aufruf bei „jeder

sich bietenden Möglichkeit“ erfolgt. Das ist erkennbar an den rekursiven Aufrufen von `DameSetzen` sowohl für die Nachfolger- als auch für die Vorgänger-Dame.

Ein sinnvollerer Einsatz der Rekursion ergibt sich, wenn nur für Nachfolger-Damen ein rekursiver Aufruf erfolgt. Ist bei diesem Vorgehen kein solcher Aufruf mehr möglich, dann setzen wir die Vorgängerdame auf ein neues unbedrohtes Feld (falls nicht vorhanden, deren Vorgängerdame usw.) und rufen `DameSetzen` rekursiv für die jetzigen Nachfolgerdamen auf. Dann ergibt sich eine maximale Rekursionstiefe von acht.

**Ende des Exkurses.**





# Kurseinheit V

## Lernziele zu Kurseinheit V

Nach dieser Kurseinheit sollten Sie

1. verschiedene Verfahren der analytischen Qualitätssicherung kennen und einordnen können,
2. die Begriffe Testfall und Testdatum unterscheiden und im Rahmen der verschiedenen dynamischen Testverfahren anwenden können,
3. zu einem gegebenen Programm, einer Prozedur oder Funktion einen kompakten Kontrollflussgraphen erstellen und kontrollflussorientierte Testverfahren anwenden können,
4. die verschiedenen Überdeckungsmaße wie insbesondere Anweisungsüberdeckung, Zweigüberdeckung, minimale Mehrfach-Bedingungsüberdeckung sowie den Boundary-Interior-Pfadtest kennen, voneinander abgrenzen und anwenden können,
5. Black-Box-Testverfahren anwenden und insbesondere aus einer gegebenen Spezifikation funktionale Ein- und Ausgabeäquivalenzklassen ableiten können,
6. Bedeutung und Besonderheiten des Regressionstests erläutern können,
7. Inspektion und (strukturierten) Walk-Through erklären und voneinander abgrenzen können.

## 8. Einführung in die Analytische Qualitätssicherung

### 8.1 Motivation und grundlegende Definitionen

Mit dem vorigen Kapitel sind unsere Ausführungen über grundlegende imperative Programmierkonzepte beendet. Während es bisher darum ging, wie man einfache, wohlstrukturierte Programme erstellt, beschäftigen wir uns im letzten Teil des Kurses mit der *analytischen Qualitätssicherung*, die Verfahren zur Sicherstellung hoher Softwarequalität und insbesondere der Korrektheit von Programmen zum Gegenstand hat. Neben der Korrektheit können auch noch andere Qualitätsanforderungen an Software gestellt werden, wie Effizienz, Robustheit, einfache Benutzbarkeit u.s.w. Wir werden uns im Folgenden aber auf die Betrachtung der Korrektheit beschränken.

Man sollte natürlich schon während der Programmentwicklung Maßnahmen ergreifen, um die Fehlerwahrscheinlichkeit (oder allgemeiner die Wahrscheinlichkeit der Entstehung von Qualitätsmängeln) zu reduzieren, also Fehler möglichst zu vermeiden. Solche Maßnahmen werden unter dem Begriff *konstruktive Qualitätssicherung* zusammengefasst. Dazu gehört z.B. im Kleinen schon die Einführung und Befolgung von Programmierstil-Richtlinien (wie wir es im Kurs getan haben), eventuell die Verwendung einer Entwicklungsumgebung mit Funktionen, die den Programmierer unterstützen und helfen, Fehler zu vermeiden (oder zumindest frühzeitig zu erkennen), sowie nicht zuletzt eine bewährte Entwicklungsmethodik. Gerade für die Softwareentwicklung im Großen kommen noch eine Reihe von Aspekten hinzu, z.B. in Hinblick auf das methodische Vorgehen in Anforderungsermittlung und Softwareentwurf – Themen, die Sie in den *Software Engineering*-Kursen noch kennenlernen werden.

konstruktive  
Qualitätssicherung

Software  
Engineering

Konstruktive Qualitätssicherungsmaßnahmen sind zwar in der Lage, z.B. Anzahl und Ausmaß von Fehlern zu verringern und auch die Wartbarkeit und Korrekturfreundlichkeit zu steigern, jedoch wäre es naiv zu glauben, dass sie die Korrektheit von Programmen so gut wie sicherstellen. Die Fehlerquelle ist der Mensch selbst, und so können jederzeit und bei jeder menschlichen Tätigkeit im Rahmen der Softwareentwicklung Fehler gemacht werden.

Hieraus resultiert unsere *erste Faustregel*:

erste Faustregel

Jedes Programm ist als fehlerhaft anzusehen, es sei denn, wir haben uns vom Gegenteil überzeugt.

Welche Folgerungen müssen wir daraus ziehen? Zunächst vor allem eine: Die Funktionsfähigkeit von praxistauglichen Programmen darf sich nicht auf Vermutungen stützen, da ein Fehlverhalten oft schwerwiegende Folgen haben kann. Hier setzen die Verfahren der *analytischen Qualitätssicherung* an, die sich mit der *Überprüfung der Qualität*, insbesondere dem Finden von Fehlern oder dem Nachweis der Korrektheit befassen.

analytische  
Qualitätssicherung

Korrektheit	<p>Wie können wir nun die Korrektheit eines Programms sicherstellen? Dazu wäre zunächst zu klären, was <i>Korrektheit</i> eigentlich ist. Salopp gesagt ist ein Programm korrekt, wenn es genau das tut, was es tun soll. Um also beurteilen zu können, ob ein Programm korrekt oder fehlerhaft ist, muss zunächst bekannt sein, was es tun soll. Dazu erinnern wir uns an die <i>Problemspezifikation</i> in Abschnitt 2.1 und formulieren etwas präziser: Ein Programm ist <i>korrekt bezüglich einer Spezifikation</i>, wenn es für <i>jede</i> Eingabe, welche die <i>Vorbedingung</i> der Spezifikation erfüllt, eine Ausgabe erzeugt, welche die <i>Nachbedingung</i> der Spezifikation erfüllt.</p>
Problemspezifikation	
Verifikation	<p>Wird die Korrektheit eines Programms formal bewiesen, sprechen wir von der <i>Verifikation</i> eines Programms. Formale Verifikationsverfahren sind mathematisch kompliziert und sehr aufwendig durchzuführen. Desweiteren sind sie nur anwendbar, wenn eine <i>formale</i> Problemspezifikation vorliegt und auch die Semantik (d.h. Bedeutung) jeder einzelnen Anweisung der verwendeten Programmiersprache <i>formal</i> definiert ist. Diese Einschränkungen machen Verifikationsmethoden nur in bestimmten Anwendungsbereichen sinnvoll einsetzbar. Für die meisten administrativen und kaufmännischen Programme stehen formale Verifikationen in keinem vernünftigen Kosten-Nutzen-Verhältnis. Ganz anders sieht es aber in Bereichen aus, in denen inkorrekte Programme extrem teure Schäden hervorrufen oder sogar Menschenleben fordern können. Zu diesen sicherheitskritischen Programmen gehören z.B. Steuerungs- und Überwachungsprogramme von (Kern-)Kraftwerken, Verkehrsleitsysteme, Flugüberwachungssysteme (denken Sie an den Frankfurter Flughafen), Programme, die Flugzeuge steuern und überwachen (würden Sie noch in den Airbus steigen, wenn die korrekte Funktion seiner sicherheitskritischen Softwarekomponenten nicht sichergestellt wäre?) oder Programme für NASA-Weltraumprojekte.</p>
Testen	<p>Ist ein vollständiger formaler Nachweis der Korrektheit nicht durchführbar, müssen wir uns nach einfacher durchzuführenden (aber auch schwächeren) Methoden umsehen. Eine solche Methode ist das Testen. Unter <i>Testen</i> verstehen wir jede Tätigkeit, die ein Programm mit dem Ziel untersucht, Fehler aufzufinden.</p>
Fehlerfreiheit	<p>Da Tests gezielt auf Fehler fokussieren, verwenden wir in deren Kontext vornehmlich den Begriff <i>Fehlerfreiheit</i> an Stelle des Begriffs Korrektheit. Alle noch so ausgeklügelten Testverfahren können jedoch nur die <i>Anwesenheit von Fehlern</i>, aber niemals deren <i>Abwesenheit</i> feststellen. Man spricht daher auch von <i>Falsifikation</i> – im Gegensatz zur Verifikation, die ja die Fehlerfreiheit nachweist. Ein Programm, das nach dem Testen keine Fehler mehr zeigt, muss noch lange nicht korrekt sein. Wir müssen also daran interessiert sein, so viele Fehler wie möglich zu finden, um die Wahrscheinlichkeit für seine Fehlerfreiheit zu erhöhen. Daher sollte der Programmtest – außer in Spezialfällen – nicht (nur) vom Autor des Programms selbst durchgeführt werden, da die Gefahr einer Betriebsblindheit zu groß ist. Besser ist es, das Testen von geschulten Testern durchführen zu lassen, die außer ihrer Objektivität auch über entsprechendes Spezialwissen verfügen. Hinzu kommt, dass für einen Tester das Finden möglichst vieler Fehler eine Bestätigung für seine gute und gründliche Arbeit darstellt, während der Programmierer sich eher dann bestätigt fühlt, wenn möglichst wenig Fehler gefunden werden.</p>
Falsifikation	

Damit erhalten wir eine *zweite Faustregel*:

Der Programmtest sollte – außer in Spezialfällen – nicht (nur) vom Programmautor durchgeführt werden.

Zur Diskussion von Testverfahren ist es hilfreich, zwischen Fehlerursache und Fehlersymptom zu differenzieren:

#### Definition 8.1.1 (Fehlerursache, Fehlersymptom)

Eine *Fehlerursache* ist ein inkorrektter Programmteil, der (bei der Ausführung) ein fehlerhaftes Verhalten des Programms bewirkt. Der Benutzer bzw. Tester kann nur das fehlerhafte Verhalten erkennen, das wir auch als *Fehlersymptom* bezeichnen. Eine einzige Fehlerursache kann mehrere Fehlersymptome zeigen.

Den Begriff *Fehler* verwenden wir als Oberbegriff über Fehlerursache und Fehlersymptom.



Um ein Fehlersymptom erkennen zu können, benötigen wir, wie schon erwähnt, eine *Spezifikation* des erwarteten, korrekten Verhaltens, allerdings muss diese – anders als bei der Verifikation – nicht notwendig formal sein. Ein *Fehlersymptom* liegt z.B. vor, wenn die Ausgabe eines Programms nicht der Nachbedingung der Spezifikation genügt, das Programm in eine Endlosschleife gerät oder „abstürzt“, *obwohl* die Eingaben der Vorbedingung der Spezifikation genügen. Eine *Fehlerursache* kann z.B. eine falsche Anweisung, eine vergessene Initialisierung oder eine falsche Schleifenbedingung sein.

#### Definition 8.1.2 (Testen, Debuggen)

Als *Testen* bezeichnen wir das gezielte Suchen nach Fehlern<sup>1</sup>. Tests, bei denen das zu prüfende Programm ausgeführt wird (sog. *dynamische Tests*), können aber lediglich Fehlersymptome feststellen. Die darauf folgende Tätigkeit des Aufspürens der Fehlerursachen und deren Korrektur, also die Fehlerbehebung, bezeichnen wir dagegen als *Debuggen* (engl. to debug: entwanzen).



Für den Rest dieses Kapitels wollen wir die Begriffe Programm und Prozedur/Funktion synonym verwenden, solange keine Missverständnisse möglich sind. Ein zum Test vorliegendes Programm bezeichnen wir als *Prüfling*. Handelt es sich bei dem Prüfling um eine Prozedur oder Funktion, so wird zur Testausführung noch ein Hauptprogramm benötigt. Ein lediglich zum Ausführen und Steuern von Tests geschriebenes Programm wird auch als *Testtreiber* bezeichnet.

Jeder Programmierer wird das, was er gerade geschrieben hat, typischerweise ad hoc testen, indem er den Prüfling ausführt, ihn spontan mit *zulässigen Eingaben*, d.h. Eingaben, die der Spezifikation genügen, „füttert“ und überprüft, ob das ge-

zweite Faustregel

Fehlerursache

Fehlersymptom

Fehler

(Problem-) Spezifikation

Testen

Debuggen

Prüfling

Testtreiber

zulässige Eingaben

1. Es gibt streng genommen auch noch andere Tests, die nicht auf die Fehlerfreiheit des Prüflings, sondern auf Kriterien wie z.B. Robustheit, Benutzungsfreundlichkeit oder Effizienz abzielen. Diese werden wir hier jedoch nicht behandeln.

Testlauf

wünschte Ergebnis herauskommt. Nur wird er solche *Testläufe* i.d.R. eher unsystematisch und nicht in hinreichendem Umfang durchführen. Sicher gibt es Fehler, deren Symptome ein unsystematischer Test mit einiger Wahrscheinlichkeit aufdecken wird – da sie bei allen oder zumindest den meisten möglichen Eingaben auftreten werden. Andererseits gibt es aber meist noch genügend „versteckte“ Fehler, deren Symptome nur in bestimmten Fällen auftreten und bei Ad-hoc-Tests oft verborgen bleiben. Um möglichst viele davon zu finden, strebt man eine systematische Wahl der beim Testlauf zu verwendenden Eingabedaten an. Bevor wir diesen Gedanken vertiefen, definieren wir mit Testdatum und Testfall zwei weitere im Folgenden verwendete Grundbegriffe<sup>1</sup>.

### Definition 8.1.3 (Testdatum, Testfall)

Testdatum

Ein *Testdatum* ist ein Tupel bestehend aus zulässigen Eingabedaten und den dazu erwarteten Ausgabedaten, d.h. dem laut Spezifikation erwarteten Ergebnis. Unter einem *Testfall* verstehen wir eine aus der Spezifikation oder dem Programmtext abgeleitete Menge von Testdaten, für die man vom Prüfling ein gleichartiges Verhalten erwartet.

Testfall



Ein Testdatum spezifiziert genau einen Testlauf: Es gibt an, welche Daten der Tester bei diesem Testlauf eingeben soll, und es gibt auch an, welches Ergebnis der Prüfling zu diesen Eingaben liefern muss. Der Tester muss nur das tatsächliche Ergebnis mit dem Soll-Ergebnis aus dem Testdatum vergleichen, um zu entscheiden, ob ein Fehler(-symptom) gefunden wurde. Wenn solche Testdaten erst einmal vorliegen, ist die eigentliche Testdurchführung eine eher schematische Aktivität, die prinzipiell sogar automatisiert werden kann. Der interessante Aspekt, mit dem wir uns im Folgenden ausführlicher beschäftigen werden, ist daher die systematische Ermittlung sinnvoller Testdaten – mit dem Ziel, möglichst viele (auch „versteckte“) Fehler zu finden. Dabei bestimmt man i.d.R. nicht direkt Testdaten, sondern bildet zunächst Testfälle. Aus jedem Testfall wird dann mindestens ein Testdatum als *Repräsentant* für die Testausführung ausgewählt. Hierzu werden wir in Kapitel 10 noch verschiedene Auswahlstrategien vorstellen.

### Beispiel 8.1.4

Wir betrachten noch einmal die bekannte Fakultätsfunktion

$$f(x) = \begin{cases} 1 & \text{für } x = 0 \\ x \cdot f(x-1) & \text{für } x > 0. \end{cases}$$

und folgende mögliche Implementierung<sup>2</sup>:

**type**

```
tDefBereich = 0..12;
tNatZahl = 0..maxint;
```

- 
1. In der Literatur sind häufig auch abweichende Definitionen anzutreffen.
  2. Die obere Grenze des Definitionsbereichs wurde so gewählt, dass bei einer Integer-Größe von 16 Bit (oder mehr) stets  $\text{inZahl!} < \text{maxint}$  gilt.

```

function Fakultaet (
    inZahl : tDefBereich): tNatZahl;
{ berechnet die Fakultaet von inZahl >= 0 }

    var
    i : integer;
    temp : tNatZahl;

begin
    temp := 1;
    for i := 2 to inZahl do
        temp := i * temp;
    Fakultaet := temp
end; { Fakultaet }

```

Die Definition der Fakultätsfunktion aus Beispiel 6.2.1, die hier die Rolle der Spezifikation einnimmt, unterscheidet zwei Fälle ( $x > 0$  und  $x = 0$ ). Analog dazu können wir in Abhängigkeit vom Parameter `inZahl` zwei *Testfälle* bilden (einen für den Fall `inZahl > 0` und einen für den Fall `inZahl = 0`):

Testfall 1:  $\{(n, n!) \mid n \in \mathbb{N}, n > 0\}$ <sup>1</sup>

Testfall 2:  $\{(0, 1)\}$ .

Diese beiden Testfälle decken alle möglichen Eingaben ab. Negative Eingaben für `inZahl` sind z.B. laut Spezifikation keine gültigen Eingaben – und auch bei der hier vorliegenden Implementierung nicht möglich (das garantiert das Pascal-Typsystem).

Als nächstes wählen wir *Testdaten* als Repräsentanten der oben gewählten Testfälle. Sinnvolle Eingabedaten für den ersten Testfall sind die untere und obere Grenze seines definierten Bereichs sowie eine oder mehrere Zahlen aus dem Bereichsinnen. Der zweite Testfall besteht nur aus einem Testdatum, dessen Eingabedatum die untere Bereichsgrenze ist. Als Testdaten wählen wir z.B. folgende Zahlenpaare aus Eingabedatum (für `inZahl`) und richtigem Ergebnis:

Testdaten 1: (1, 1), (2, 2), (10, 3628800), (12, 479001600)

Testdaten 2: (0, 1).




---

1. Streng genommen gibt es bei Beispielen wie diesen eine kleine Diskrepanz zwischen Spezifikation und Implementierung: Die Spezifikation der Fakultätsfunktion legt z.B. das Ergebnis für eine beliebige natürliche Zahl fest, die Implementierung nimmt aber aufgrund technischer Grenzen (z.B. `maxint`) keine beliebig großen Eingaben entgegen. Da diese Grenzen plattformabhängig sind, werden wir sie nicht mit in die Problemspezifikation aufnehmen. Auch werden wir sie in den Testfällen nicht mit angeben, wohl aber bei der Auswahl konkreter Testdaten berücksichtigen.

dynamische  
Verfahren

Ausgehend von einer Ad-hoc-Herangehensweise an den Softwaretest haben wir bisher mit dem Ausführen des Prüflings unter Eingabe von Testdaten (möglichst aus vorher systematisch ermittelten Testfällen) eine bestimmte Klasse von Testverfahren betrachtet. Solche sogenannten *dynamischen Verfahren* werden den Schwerpunkt dieser Kurseinheit bilden, wobei es sich bei verschiedenen vorgestellten dynamischen Testverfahren im Wesentlichen um unterschiedliche Strategien zur Testfallbildung handelt. Bevor wir aber dazu kommen, wollen wir zunächst einen Überblick über Verfahren der analytischen Qualitätssicherung geben, unter denen Tests im Allgemeinen und dynamische Testverfahren im Speziellen jeweils nur einen Teilbereich bilden.

## 8.2 Klassifikation der Verfahren der analytischen Qualitätssicherung

analytische  
Qualitätssicherung

Die verschiedenen Ansätze, welche die Untersuchung von Programmen bezüglich der Einhaltung gewisser Qualitätskriterien zum Gegenstand haben – wobei wir uns hier auf die Fehlerfreiheit beschränken –, werden unter dem Begriff *analytische Qualitätssicherung* zusammengefasst. Dazu gehören jedoch nicht nur die Verfahren der Verifikation, Analyse und Testen, sondern auch die zugehörigen organisatorischen Maßnahmen, also z.B. die Planung und Kontrolle von Testaktivitäten. Ein wichtiger Gesichtspunkt der analytischen Qualitätssicherung ist, dass sie eine die Programmentwicklung *begleitende Tätigkeit* darstellt und nicht erst am Ende der Entwicklung, z.B. in Form von Testen, einsetzt. Diesen Aspekt werden wir hier jedoch nicht weiter verfolgen.

verifizierende  
Verfahren

Die Verfahren der analytischen Qualitätssicherung unterscheiden wir zunächst aufgrund ihrer unterschiedlichen Zielsetzungen. Verifizierende Verfahren beweisen die Korrektheit eines Programms, analysierende Verfahren quantifizieren bestimmte Programmeigenschaften und/oder stellen diese dar und testende Verfahren versuchen, Fehler aufzudecken.

*Verifizierende Verfahren* beweisen die Korrektheit eines Programms bezüglich seiner Spezifikation. Enthält der Beweis selbst keine Fehler, so ist dies die höchste Qualitätsstufe, die überhaupt erreichbar ist. Allerdings erfordern verifizierende Verfahren, dass die Spezifikation *formal* vorliegt und auch die Semantik der verwendeten Anweisungen der Programmiersprache *formal* definiert ist. Verifizierende Verfahren sind sehr komplex und selbst kleine Beweise sind ohne Werkzeugunterstützung kaum möglich. Für sicherheitskritische Anwendungen spielt allerdings die Verifikation wichtiger Programmteile eine wichtige Rolle.

analysierende  
Verfahren  
zyklomatische Zahl  
Halstead-Metrik

*Analysierende Verfahren* erlauben weder Korrektheitsbeweise noch dienen sie direkt der Fehlersuche, sondern sind eher – wertvolle, weil leicht automatisierbare – Hilfstechniken. Metriken wie die *zyklomatische Zahl* oder die *Halstead-Metriken* erlauben die Quantifizierung von Eigenschaften wie strukturelle Komplexität, Programmlänge oder Kommentierungsgrad und ermöglichen so insbesondere vergleichende Beurteilungen von Dokumenten (z.B. Programmlistings). Die Überschreitung einer kritischen, aus den Erfahrungen früherer Projekte abgeleiteten



Maßzahl kann dabei ein Hinweis auf eine erhöhte Fehlerwahrscheinlichkeit sein und zu einer genaueren Untersuchung Anlass geben.

*Testverfahren* suchen gezielt nach Fehlern, können jedoch die Fehlerfreiheit nicht nachweisen. Sie gliedern sich in statische und dynamische Testverfahren.

*Statische Verfahren* zeichnen sich insbesondere durch folgende Merkmale aus (angelehnt an [Lig02]):

- Das Programm wird nicht ausgeführt, sondern analysiert.
- Eine statische Analyse kann manuell (ohne Computerunterstützung) durchgeführt werden.
- Statische Verfahren können nicht nur Fehlersymptome, sondern auch Fehlerursachen direkt aufdecken.
- Die Korrektheit des Programms (bzgl. seiner Spezifikation) wird nicht bewiesen (daher die Einordnung als Testverfahren).

Ein wesentlicher Vorteil statischer Verfahren ist, dass ihre Anwendung nicht auf ausführbare Programme beschränkt bleibt (es werden insbesondere auch keine Testtreiber benötigt), sondern dass sie im Grunde für alle Arten von Softwaredokumenten geeignet sind. Die wichtigste statische Technik ist der *Review*, in dem ein Softwaredokument von einer Gruppe von Personen mit bestimmter Zielsetzung „gelesen“ wird. Einige der in Reviews entdeckten Fehler können auch durch automatische Werkzeuge wie *statische Programmanalysatoren* erkannt werden.

Im Gegensatz zu statischen Testverfahren setzen *dynamische Testverfahren* zur Fehlererkennung die Ausführbarkeit des Programms voraus und zeichnen sich durch folgende Merkmale aus (angelehnt an [Lig02]):

- Der Prüfling wird mit konkreten Testdaten ausgeführt,
- er kann in der realen Umgebung getestet werden,
- dabei werden nur Fehlersymptome, nicht die Fehlerursachen gefunden<sup>1</sup>.
- Dynamische Verfahren sind Stichprobenverfahren.
- Die Korrektheit des Programms (bzgl. seiner Spezifikation) wird nicht bewiesen.

Dynamische Testverfahren sind also systematische Ausprägungen dessen, was wir intuitiv unter Testen verstehen: Das „Füttern“ eines ausführbaren Programms mit – mehr oder weniger systematisch gewählten – Eingabedaten.

Insgesamt bilden die dynamischen Verfahren die mit Abstand stärkste Gruppe unter den Testverfahren. Die Unterschiede zwischen den verschiedenen dynamischen Verfahren liegen im Wesentlichen in der Systematik zur Bildung von Testfällen. Verfahren, bei denen die Testfälle ausschließlich unter Verwendung der Problem-

statische Verfahren

Review

statische Programm-  
analysatoren

dynamische  
Verfahren

1. Wie wir jedoch in den noch folgenden Beispielen sehen werden, erfordern manche dynamische Testtechniken eine eingehende Testvorbereitung, im Rahmen derer gewisse Fehlerursachen gefunden werden können, noch bevor ein Testlauf ausgeführt wird.

Black-Box- und  
White-Box-Testver-  
fahren

kontrollfluss- und  
datenflussorientie  
Testverfahren

strukturorientierte  
Testverfahren

spezifikation und unabhängig von der Realisierung des Prüflings gebildet werden, bezeichnen wir als *Black-Box-Testverfahren*. Diesen stehen die so genannten *White-Box-Testverfahren* gegenüber, die zur Testfallbildung neben der Spezifikation auf Informationen über die interne Struktur des Prüflings zurückgreifen. Nach der Art dieser Informationen kann man verschiedene Gruppen von White-Box-Verfahren unterscheiden. So analysieren *kontrollflussorientierte Testverfahren* mögliche Anweisungsabfolgen des Prüflings, während *datenflussorientierte Testverfahren* den Austausch von Daten zwischen Programmteilen betrachten, z.B. an welchen Stellen im Prüfling Daten, die in einer Variable abgelegt wurden, wieder Verwendung finden. Kontrollfluss- und datenflussorientierte Verfahren werden zusammenfassend häufig auch als *strukturorientierte Testverfahren* bezeichnet.

Die Testfallbildung, die wir in Beispiel 8.1.4 vorgenommen haben, war unabhängig von der Realisierung und basierte ausschließlich auf der Spezifikation. Es handelt sich also um ein Black-Box-Verfahren.

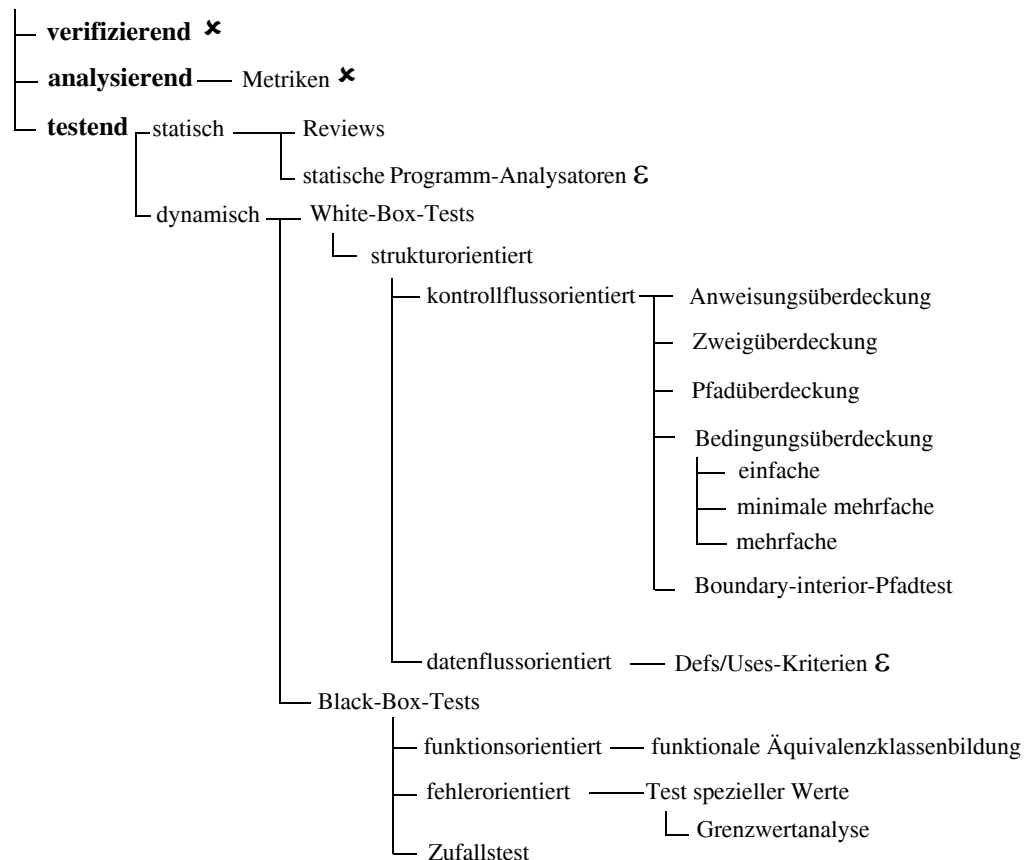


Abbildung 8.1 : Klassifikation von Verfahren der analytischen Qualitätssicherung.

Eine mögliche Klassifikation der Verfahren zur analytischen Qualitätssicherung ist in Abbildung 8.1 schematisch dargestellt, wobei exemplarisch einige wichtige Vertreter der einzelnen Klassen aufgeführt sind. Verfahren, die in dieser Kurseinheit nicht behandelt werden, sind in Abbildung 8.1 mit einem „✖“ markiert, „&“ kennzeichnet ein in einem Exkurs beschriebenes Verfahren.

Bisher haben wir Tests nach der verwendeten *Prüftechnik* klassifiziert. Man kann Tests aber auch danach unterscheiden, welcher *Stufe der Softwareentwicklung* sie zugeordnet sind. Die unterste Stufe besteht aus dem Test einzelner kleiner Programmeinheiten (die dabei durchaus Teile von sehr komplexer Software sein können) und wird auch als *Modultest* (engl. *unit test*) bezeichnet. Er wird häufig durch den Entwickler selbst ausgeführt. Der anschließende *Integrationstest* umfasst das Zusammenführen voneinander abhängiger Module zu immer komplexeren Teilsystemen und den Test ihrer Zusammenarbeit. Beim *Systemtest* wird das gesamte System getestet. Üblicherweise wird der Systemtest von dem mit der Entwicklung betrauten Softwarehaus auf einer Testumgebung durchgeführt. Das letzte Glied der Kette bildet der *Abnahmetest*. Er ist dem Systemtest ähnlich, allerdings testet hier der Kunde das System in seiner Umgebung, die eine Testumgebung oder auch die Produktivumgebung sein kann. Der Abnahmetest ist oft Voraussetzung für die Rechnungsstellung.

Im Folgenden werden wir verschiedene White-Box- und Black-Box-Testverfahren vorstellen, die wir zum Modultest, hier zum Test von Prozeduren, Funktionen und ggf. kleinen unmodularisierten Programmen verwenden, so wie wir sie in den vorhergehenden Kapiteln entwickelt haben. Insbesondere die strukturorientierten Testverfahren sind vornehmlich für den Modultest geeignet und weniger für den Systemtest größerer Software<sup>1</sup>. Auch die funktionale Äquivalenzklassenbildung als hier vorgestellter Vertreter funktionsorientierter Testverfahren bietet sich eher für den Modultest an. Systemtests insbesondere komplexer interaktiver Software, wie beispielsweise einer Textverarbeitung oder einer Webanwendung für einen Online-Shop, wird man anders angehen.

Modultest  
Integrationstest  
  
Systemtest  
  
Abnahmetest

---

1. Die vorgestellten Überdeckungskriterien lassen sich jedoch – losgelöst von der Code-Ebene – z.B. auf (Graphen zu) Verhaltensmodellen übertragen, wie sie auf wesentlich abstrakterem Niveau unter anderem zur Spezifikation möglicher Benutzer-System-Interaktionen Anwendung finden.

## 9. White-Box-Testverfahren

### 9.1 Kontrollflussorientierte Testverfahren

kontrollflussorien-  
tierte Verfahren

*Kontrollflussorientierte Testverfahren* sind White-Box-Verfahren. Testfälle werden in erster Linie anhand der (statischen) Kontrollstruktur des Prüflings gebildet. Die Spezifikation wird „nur“ herangezogen, um die Testfälle auf die laut Vorbedingung zulässigen Eingaben einzuschränken und die erwarteten Ausgaben zu den möglichen Eingaben zu bestimmen. Mit letzteren werden dann die tatsächlichen Ausgaben des Prüflings verglichen.

#### 9.1.1 Kontrollflussgraphen

Der Kontrollfluss strukturierter Programme wird durch Anweisungsfolgen, Selektion und Iteration gesteuert. Für kontrollflussorientierte Tests bereitet man den Quelltext üblicherweise graphisch auf, um den Kontrollfluss zu visualisieren. Häufig verwendet wird neben dem *Programmablaufplan* der *Kontrollflussgraph*, der eine abstraktere Sichtweise ermöglicht. Für die Beschreibung der meisten kontrollflussorientierten Verfahren werden wir im Folgenden Kontrollflussgraphen verwenden. Vorbereitend für deren Beschreibung definieren wir zunächst den Begriff des gerichteten Graphen.

##### Definition 9.1.1.1 (gerichteter Graph)

gerichteter Graph

Ein *gerichteter Graph*  $G = (N, E)$  ist ein Paar aus einer Menge  $N$  von *Knoten* (Nodes) und einer Menge  $E \subseteq N \times N$  von gerichteten *Kanten* (Edges) zwischen Knoten. Für eine Kante  $(n_1, n_2) \in E$  bezeichnen wir  $n_1$  als einen *Vorgänger* von  $n_2$  und  $n_2$  als einen *Nachfolger* von  $n_1$ .

□

Beim Zeichnen eines gerichteten Graphen wird ein Knoten meist als Kästchen oder Kreis, eine gerichtete Kante als Pfeil von ihrem Quell- zu ihrem Zielknoten dargestellt.

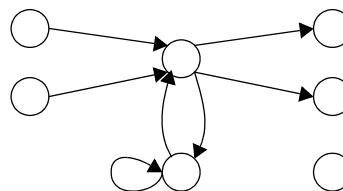


Abbildung 9.1 : nicht zusammenhängender gerichteter Graph mit zwei Zyklen

Abbildung 9.1 stellt beispielhaft einen gerichteten Graphen dar und demonstriert insbesondere, dass ein Knoten keine, einen oder mehrere Nachfolger oder Vorgänger haben kann. Weiterhin sind Zyklen möglich, ein Knoten kann sogar Nachfolger und Vorgänger von sich selbst sein. Sind alle Knoten des Graphen über Kanten miteinander verbunden, so bezeichnen wir den Graphen als *zusammenhängend*. Der

zusammenhängen-  
der Graph

Graph in Abbildung 9.1 enthält z.B. zwei Zyklen und ist nicht zusammenhängend (da der Knoten rechts unten nicht mit anderen Knoten verbunden ist).

Mit Listen und Bäumen haben Sie bereits Spezialisierungen von gerichteten Graphen kennengelernt. So sind Listen z.B. zusammenhängende gerichtete Graphen, bei denen jeder Knoten maximal einen Vorgänger und maximal einen Nachfolger hat. Bäume sind zusammenhängende zyklensfreie gerichtete Graphen mit hierarchischem Aufbau, die (mit Ausnahme des leeren Baums) genau einen Knoten ohne Vorgänger haben (die Wurzel) und bei denen jeder andere Knoten genau einen Vorgänger hat. Weiterhin sind die Nachfolger jedes Knotens paarweise verschieden. Binärbäume begrenzen zusätzlich die Zahl der Nachfolger auf maximal zwei<sup>1</sup>. Auch den Kontrollflussgraphen definieren wir als Spezialisierung eines gerichteten Graphen:

**Definition 9.1.1.2** (Kontrollflussgraph, Pfad, kompakter Kontrollflussgraph)

Der *Kontrollflussgraph*  $G = (N, E, n_{start}, n_{final})$  eines Programms  $P$  ist ein zusammenhängender gerichteter Graph  $(N, E)$  mit einem ausgezeichneten Startknoten  $n_{start} \in N$ , der als einziger Knoten aus  $N$  keinen Vorgänger besitzt, und einem ausgezeichneten Endknoten  $n_{final} \in N$ , der als einziger Knoten aus  $N$  keinen Nachfolger besitzt.

Kontrollflussgraph

Der Startknoten  $n_{start}$  dokumentiert den Eintrittspunkt des Programms, d.h. den Beginn seines Anweisungsteils, und der Endknoten  $n_{final}$  repräsentiert alle Austrittspunkte. Knoten ungleich  $n_{start}$  und  $n_{final}$  stellen Programmzeilen (des Anweisungsteils) dar, wobei wir unterstellen, dass der Programmtext den im Kurs vorgeschriebenen Layoutregeln genügt. Insbesondere beginnt jede Anweisung in einer neuen Zeile. Ein Knoten aus  $N \setminus \{n_{start}, n_{final}\}$  repräsentiert dann im einfachsten Fall genau eine Zeile. Eine gerichtete Kante  $(n_i, n_j) \in E$  drückt aus, dass es einen statischen Kontrollfluss von  $n_i$  nach  $n_j$  im Programm gibt.

Eine mit  $n_{start}$  beginnende und mit  $n_{final}$  endende Folge aus Knoten, deren aufeinanderfolgende Elemente im Kontrollflussgraphen jeweils durch eine Kante verbunden sind, heißt *Pfad* (durch den Kontrollflussgraphen). Formal definiert ist ein Pfad durch einen Kontrollflussgraphen  $G = (N, E, n_{start}, n_{final})$  eine Folge  $(n_1, \dots, n_k)$ , für die gilt:

Pfad

- $n_1, \dots, n_k \in N$
- $n_1 = n_{start}$  und  $n_k = n_{final}$
- Für alle  $i \in \{2, \dots, k\}$  gilt  $(n_{i-1}, n_i) \in E$

Entsprechende Knotenfolgen, die nicht mit  $n_{start}$  beginnen oder nicht mit  $n_{final}$  enden, bezeichnen wir als *Teilpfade*.

Teilpfad

Ein *Block* (eines Kontrollflussgraphen) ist eine Folge von Knoten  $n_1, \dots, n_k \in N \setminus \{n_{start}, n_{final}\}$ ,  $k \geq 1$ , für deren zugehörige Folge von Programmzeilen  $z(n_1), \dots, z(n_k)$  gilt:

Block eines Kontrollflussgraphen

1. Oft definieren Bäume noch zusätzlich eine Ordnung auf den Nachfolgern eines Knotens (so unterscheiden Binärbäume einen linken und einen rechten Nachfolger), was mit einem gerichteten Graphen allein nicht auszudrücken ist.

- a) Die Programmzeilen werden stets in der Reihenfolge  $z(n_1), z(n_2), \dots, z(n_k)$  genau einmal ausgeführt.
- b) Die Anzahl  $k$  der Programmzeilen ist bezüglich Eigenschaft a) maximal (d.h. ein Block kann nicht eine Teilfolge eines Blocks sein).

Ein Block besteht also aus einer größtmöglichen Knotenfolge, die ausschließlich zusammenhängend („en bloc“) ausgeführt wird. Zwischen den Knoten eines Blocks kann es insbesondere keine Kontrollflussverzweigungen geben, d.h. jeder Knoten eines Blocks bis auf den letzten (jeder Knoten aus  $\{n_1, \dots, n_{k-1}\}$ ) hat genau einen Nachfolger und jeder Knoten eines Blocks mit Ausnahme des ersten (jeder Knoten aus  $\{n_2, \dots, n_k\}$ ) hat genau einen Vorgänger.

Blöcke sind ein Hilfsmittel, um zu einer kompakteren Form des Kontrollflussgraphen zu gelangen, da alle Knoten eines Blocks zu einem einzigen Knoten reduziert werden können. Eine weitere Reduktion der Anzahl der Knoten erhalten wir, indem wir für Programmzeilen, die nur aus Schlüsselwörtern (z.B. **begin**, **end**, **repeat**, **else**, ...) bestehen, keinen eigenen Knoten vorsehen, sondern sie einem Nachbarknoten zuordnen. Außerdem werden Leerzeilen und Kommentarzeilen nicht berücksichtigt.

kompakter Kontrollflussgraph

Durch iterative Anwendung oben genannter Möglichkeiten zur Reduktion der Knotenzahl entsteht ein *kompakter Kontrollflussgraph*. Dieser enthält keine Blöcke aus mehr als einem Knoten sowie keine Knoten ohne Anweisungen mehr.



assoziierter Testfall

Kontrollflussorientierte Verfahren zielen darauf ab, eine bestimmte Menge von Pfaden beim Test zu durchlaufen. Zu jedem Pfad erhalten wir einen (evtl. leeren) *assozierten Testfall*, indem wir die zulässigen Eingabedaten auf genau solche einschränken, welche die Ausführung der dem Pfad zugeordneten Anweisungsfolge – und nur dieser – bewirken.

Ein kontrollflussorientierter Test kann nun wie folgt ablaufen:

1. Ermittlung des (kompakten) Kontrollflussgraphen
2. Ermittlung aller mindestens zu durchlaufenden Pfade anhand eines (im Folgenden noch zu betrachtenden) Überdeckungskriteriums<sup>1</sup>
3. Bilden des assoziierten Testfalls zu jedem der ermittelten Pfade (anhand der Spezifikation)
4. Auswahl konkreter Testdaten aus den Testfällen (mindestens eines pro Testfall)

---

1. Der Vollständigkeit halber sei erwähnt, dass es auch Werkzeuge gibt, die eine dynamische Testausführung überwachen und zu (mindestens) einem Überdeckungskriterium den mit allen ausgeführten Testläufen erreichten Überdeckungsgrad berechnen. Diese ermöglichen alternativ z.B. auch eine Testdurchführung mit zufälligen Eingabedaten und liefern dann ein *Testendekriterium*: Bei 100% Überdeckungsgrad kann der Zufallstest beendet werden. Solche Tests ohne systematische Testfallbildung vor der Durchführung benötigen jedoch – neben der Werkzeugunterstützung – i.d.R. deutlich mehr Testläufe.

### 5. Testdurchführung: Ausführung des Prüflings mit jedem der ausgewählten Testdaten und Vergleich von Soll- und Ist-Ergebnis

Die einzelnen Verfahren unterscheiden sich jeweils in dem in Schritt 2. zur Anwendung kommenden Überdeckungskriterium. Die wichtigsten Überdeckungskriterien sind:

Die vollständige *Anweisungsüberdeckung* ist das einfachste Kriterium. Die Testfälle werden so gewählt, dass alle Knoten des Kontrollflussgraphen mindestens einmal besucht werden.

Anweisungsüberdeckung

Die vollständige *Zweigüberdeckung* ist ein etwas strengeres Kriterium. Die Testfälle werden so gewählt, dass alle Kanten des Kontrollflussgraphen (hier *Zweige* genannt) mindestens einmal passiert werden.

Zweigüberdeckung  
Zweig

Für eine vollständige *Pfadüberdeckung* sind die Testfälle so zu wählen, dass jeder mögliche Pfad durch den Kontrollflussgraphen getestet wird. Die vollständige Pfadüberdeckung stellt das umfassendste kontrollflussorientierte Kriterium dar, ist jedoch aufgrund der bereits in kleinen Programmen unpraktikabel hohen Anzahl von Testfällen kaum erreichbar. Aus diesem Grund existieren Testverfahren, die sich der vollständigen Pfadüberdeckung auf unterschiedliche Arten annähern.

Pfadüberdeckung

Die *Bedingungsüberdeckung* zieht zur Definition von Testfällen die Bedingungen heran, die über Verzweigungen im Kontrollfluss entscheiden (z.B. Schleifenabbruchbedingungen oder **if**-Bedingungen). Man unterscheidet verschiedene Bedingungsüberdeckungskriterien, die sich im Umgang mit zusammengesetzten Bedingungen unterscheiden.

Bedingungsüberdeckung

Normalerweise strebt man eine Testfallauswahl an, die das jeweilige Überdeckungskriterium vollständig erfüllt. Das ist jedoch nicht immer möglich. So können z.B. Anweisungen unerreichbar sein wie in folgendem Codeausschnitt:

```
if a > b then
    if a < b then
        DieseAnweisungIstNichtErreichbar;
```

In diesem Fall wäre bereits eine vollständige Anweisungsüberdeckung unmöglich. Das ist im Allgemeinen ein Indiz für einen Fehler im Prüfling, wenn auch kein Nachweis für einen Fehler: Es besteht immerhin die Möglichkeit, dass der Prüfling dennoch für jede zulässige Eingabe das korrekte Ergebnis liefert, der unerreichbare Code also zur Lösung des Problems gar nicht benötigt wird. Eine solche Konstellation ist zwar vergleichsweise selten, in Ausnahmefällen aber durchaus anzutreffen (z.B. nach einer größeren Programmänderung, im Zuge derer vormals benötigter Code sowohl überflüssig als auch unerreichbar wurde und vergessen wurde, ihn zu entfernen). Derartige Fehler werden übrigens nicht erst bei der Testdurchführung, sondern bereits bei der Testfallkonstruktion entdeckt.

Im Folgenden werden wir Tests bezüglich verschiedener Überdeckungskriterien genauer betrachten.

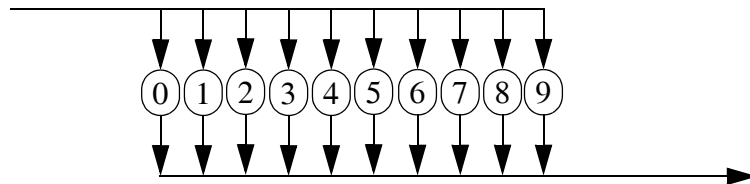
begleitendes Beispiel

Beispiel 9.1.1.3

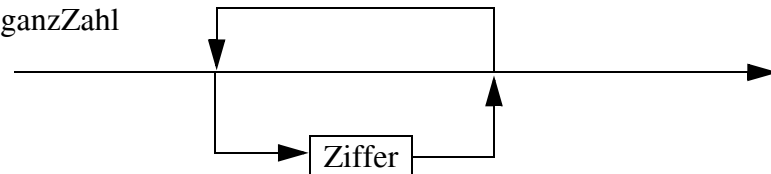
Als *begleitendes Beispiel* dient uns eine Pascal-Funktion, der folgende Problembeschreibung zugrunde liegt: Gesucht ist eine Funktion, der als Parameter ein String übergeben wird, die den Wert der dadurch definierten (positiven) Zahl errechnet und diesen zurückliefert. Der Ergebnistyp ist `real`.

Zur Definition eines Ziffernstrings geben wir folgende Syntaxdiagramme an:

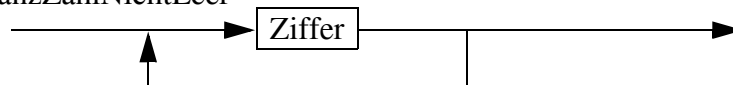
Ziffer



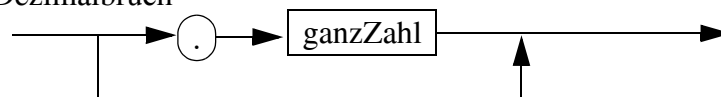
ganzZahl



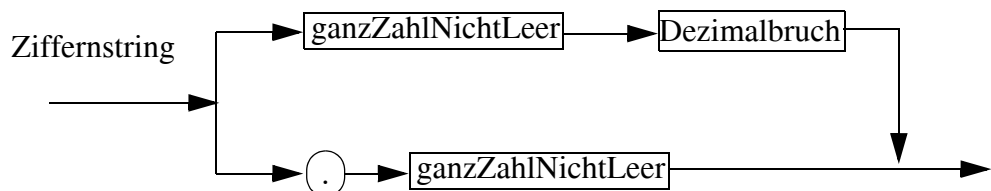
ganzZahlNichtLeer



Dezimalbruch



Ziffernstring



Beispiele für zulässige Ziffernstrings sind:

17.25    .25    17    17.

Beispiele für unzulässige Ziffernstrings sind:

leere Eingabe    .    A.25    17.A    1.2.3



Nur ein Ziffernstring, der gemäß den obigen Syntaxdiagrammen aufgebaut ist, wird als korrekte Eingabe anerkannt und sein Wert berechnet. Jeder String, der nicht dieser Spezifikation entspricht, führt zur Rückgabe eines Fehlercodes.

Zur Bearbeitung des übergebenen Ziffernstrings werden die Funktionen `length`, `liesZeichen` und `pruefeZeichen` benutzt. `length` bestimmt die Länge einer Zeichenkette, `liesZeichen` liefert das Zeichen an der angegebenen Position einer Zeichenkette zurück und `pruefeZeichen` überprüft, ob das Zeichen an der angegebenen Position aus der Menge {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'} stammt. Außerdem setzen wir die Existenz der Funktion `konvertiere` voraus, die zur Konvertierung einer Ziffer (Typ `char`) in die zugehörige `real`-Zahl eingesetzt wird. Alle vier Funktionen sind bereits getestet bzw. verifiziert, so dass wir ihre Fehlerfreiheit voraussetzen können.

In der untenstehenden entsprechenden Pascal-Funktion `werteZiffernfolgeAus` haben wir auf Kommentare aus Platzgründen weitgehend verzichtet.<sup>1</sup>

```

const
FEHLERCODE = -1.0;

function werteZiffernfolgeAus (
    inZiffernString : string) : real;
{ liefert den Wert des uebergebenen Ziffernstrings als
  real-Zahl zurück, benutzt die Funktionen length,
  liesZeichen, pruefeZeichen und konvertiere }

type
tWoBinIch = (vorDemKomma, nachDemKomma);
tNatZahl = 0..maxint;

var
Zeichen : char;
Wert,
Genauigkeit : real;
Position : tNatZahl;
woBinIch : tWoBinIch;
fehlerfrei : boolean;

begin
  Wert := 0.0;
  Genauigkeit := 1.0;
  woBinIch := vorDemKomma;
  fehlerfrei := true;
  Position := 1;

```

---

1. Die Funktion lässt sich programmtechnisch verbessern, wodurch sich auch die Übersichtlichkeit erhöht. Wir haben diese Version gewählt, um die verschiedenen Testverfahren besser erläutern zu können.

```
while (Position <= length (inZiffernString))  
    and fehlerfrei do  
begin  
    Zeichen := liesZeichen (inZiffernString,  
        Position);  
    if pruefeZeichen (Zeichen) then  
        begin  
            if woBinIch = nachDemKomma then  
                Genauigkeit := Genauigkeit / 10.0;  
            Wert := 10.0 * Wert + konvertiere (Zeichen)  
        end { if }  
        else  
            if (Zeichen = '.') and  
                (woBinIch = vorDemKomma) then  
                woBinIch := nachDemKomma  
            else  
                fehlerfrei := false;  
            Position := Position + 1  
        end; { while }  
    if not fehlerfrei  
        or (length (inZiffernString) = 0)  
        or ((woBinIch = nachDemKomma)  
            and (length (inZiffernString) = 1)) then  
            { illegale Zeichenfolge }  
            werteZiffernfolgeAus := FEHLERCODE  
        else  
            werteZiffernfolgeAus := Wert * Genauigkeit  
        end; { werteZiffernfolgeAus }
```

Den kompakten Kontrollflussgraphen der Funktion `werteZiffernfolgeAus` zeigt Abbildung 9.2. Die gepunktet gezeichneten Teile deuten dabei an, wo Blöcke des Ursprungsgraphen zu einem einzigen Knoten zusammengefasst sind.

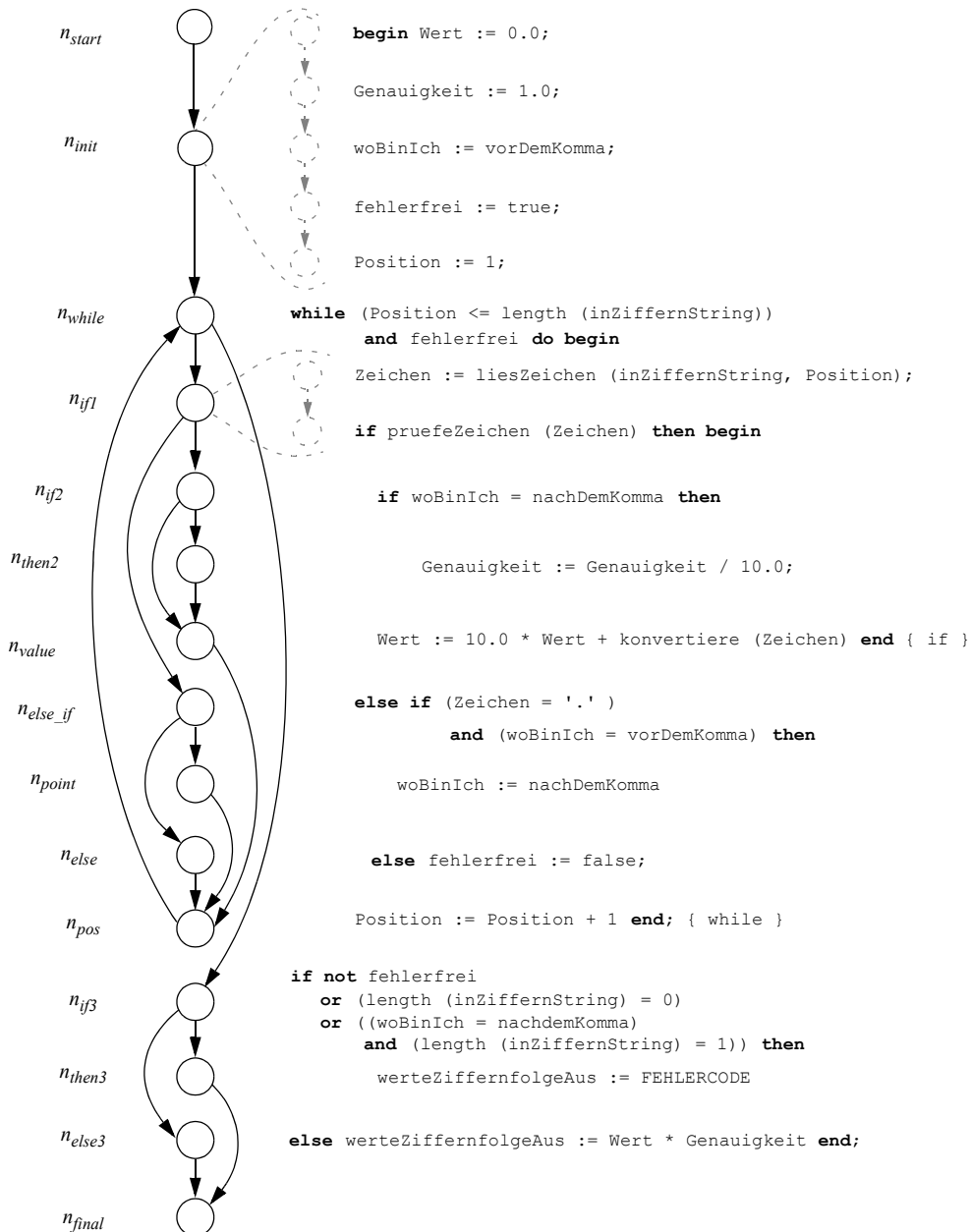


Abbildung 9.2 : Kompakter Kontrollflussgraph der Funktion  
`werteZiffernfolgeAus`



Wir betrachten im Folgenden nur noch kompakte Kontrollflussgraphen und werden diese oft, wenn keine Missverständnisse möglich sind, vereinfachend als Kontrollflussgraphen bezeichnen.

Anweisungsüberde-  
ckungstest,  
C<sub>0</sub>-Test

Anweisungsüberde-  
ckungsgrad

### 9.1.2 Anweisungsüberdeckung

Der *Anweisungsüberdeckungs-* oder *C<sub>0</sub>-Test* ist das einfachste kontrollflussorientierte Testverfahren. Ziel ist es, alle Knoten des Kontrollflussgraphen mindestens einmal zu besuchen, so dass alle Anweisungen des Programms mindestens einmal (in mindestens einem Testlauf) ausgeführt werden.<sup>1</sup>

Als Testmaß wird der *Anweisungsüberdeckungsgrad* definiert. Er ist das Verhältnis der besuchten Knoten zu der Gesamtanzahl der Knoten im Kontrollflussgraphen:

$$C_{\text{Anweisung}} \stackrel{\text{def}}{=} \frac{\text{Anzahl der besuchten Knoten}}{\text{Anzahl aller Knoten}}$$

Eine vollständige Anweisungsüberdeckung bedeutet also  $C_{\text{Anweisung}} = 1$ .

#### Beispiel 9.1.2.1

Wir wählen eine Menge von Pfaden im Kontrollflussgraphen von Abbildung 9.2, so dass alle Knoten überdeckt werden. Zu jedem Pfad geben wir den assoziierten Testfall an und wählen daraus ein Testdatum aus.

Zuerst wählen wir Pfad 1 aus Abbildung 9.3 (a)

$$(n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if1}}, n_{\text{else\_if}}, n_{\text{point}}, n_{\text{pos}}, n_{\text{while}}, n_{\text{if1}}, n_{\text{else\_if}}, n_{\text{else}}, n_{\text{pos}}, n_{\text{while}}, n_{\text{if3}}, n_{\text{then3}}, n_{\text{final}}).$$

Dieser Pfad wird für jede Eingabe durchlaufen, deren erstes Zeichen ein Punkt und deren zweites Zeichen keine Ziffer ist. Erwarteter Rückgabewert ist FEHLERCODE. Der assoziierte Testfall lautet also:

$\{ ('.' + z + s, \text{FEHLERCODE}) \mid z \notin \{ '0', '1', \dots, '9' \}, s \text{ ist beliebiger String} \}$ <sup>2</sup>

Ein mögliches Testdatum ist z.B.  $('..', \text{FEHLERCODE})$ .

Der Pfad enthält 11 verschiedene Knoten, so dass wir bei einer Knotenmenge von 15 Knoten bereits eine Anweisungsüberdeckung von 73% erreichen.

Die restlichen Knoten enthält Pfad 2 aus Abbildung 9.3 (b)

$$(n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if1}}, n_{\text{else\_if}}, n_{\text{point}}, n_{\text{pos}}, n_{\text{while}}, n_{\text{if1}}, n_{\text{if2}}, n_{\text{then2}}, n_{\text{value}}, n_{\text{pos}}, n_{\text{while}}, n_{\text{if3}}, n_{\text{else3}}, n_{\text{final}}),$$

der für jede Eingabe einer Ziffer mit vorangestelltem Punkt durchlaufen wird. Erwartetes Ergebnis ist ein Zehntel des Werts der Ziffer. Der assoziierte Testfall lautet also:  $\{ ('.' + z, \text{konvertiere}(z) / 10) \mid z \in \{ '0', '1', \dots, '9' \} \}$ , als Testdatum wählen wir  $('.9', 0,9)$ .

- 
1. Man müsste deshalb eigentlich von „Knotenüberdeckung“ sprechen. Jedoch ist „Anweisungsüberdeckung“ der in der Literatur üblicherweise benutzte Begriff.
  2. Der „+“-Operator steht im Kontext von Strings für die Konkatination (Verkettung).

Wir erreichen bereits mit diesen zwei Pfaden und – bei Wahl je eines Testdatums pro assoziiertem Testfall – mit zwei Testläufen eine vollständige Anweisungsüberdeckung.

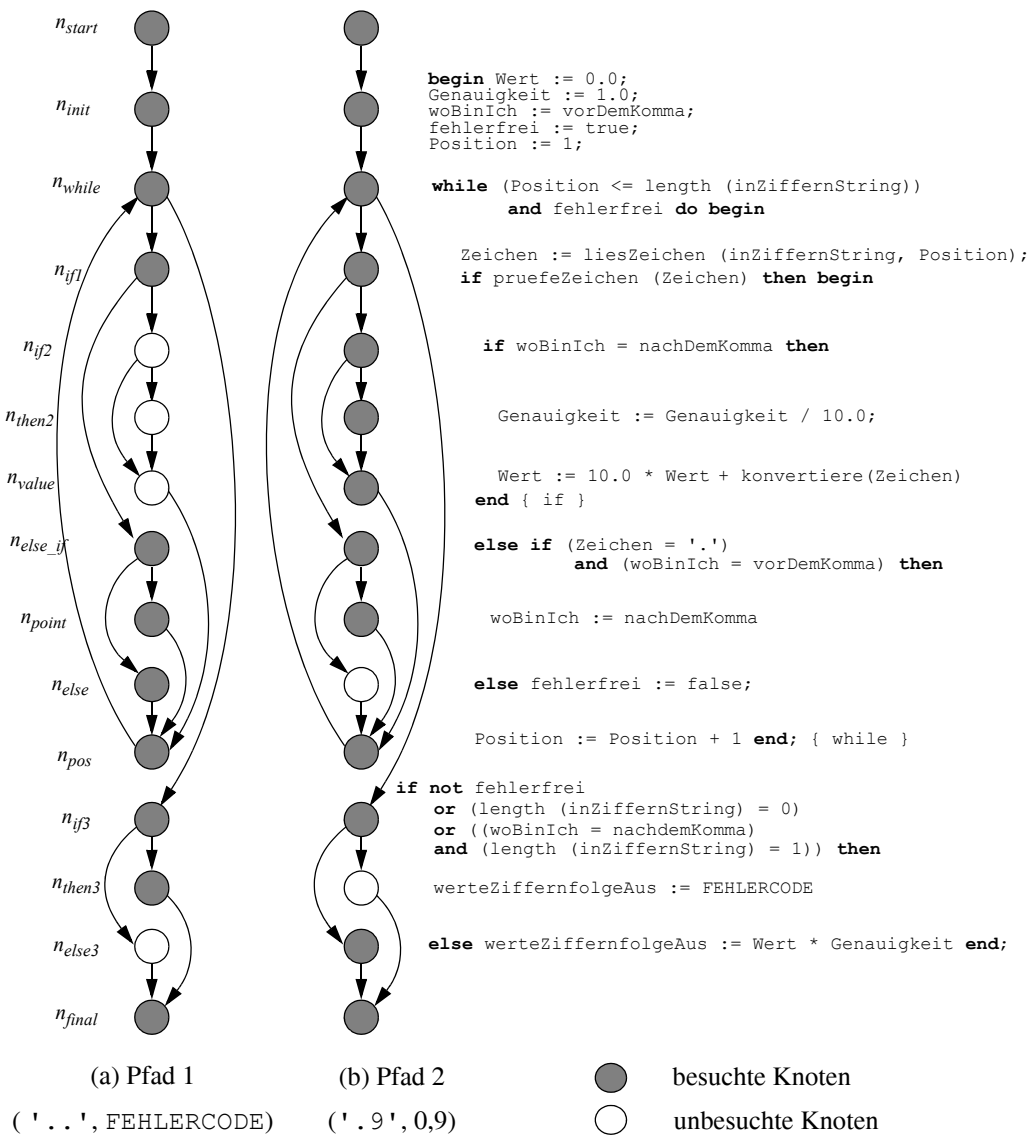


Abbildung 9.3 : Vollständige Anweisungsüberdeckung der Funktion *werteZiffernfolgeAus*



Die geringe Anzahl von Testfällen sowie ihre Einfachheit sind Vorteile dieser Testtechnik, werden aber durch eine mangelhafte Testqualität erkaufte. Das sollte an diesem Beispiel unmittelbar einleuchten, denn immerhin haben wir für große Klassen von Eingabedaten, z.B. ganze Zahlen oder Fließkommazahlen  $\geq 1$ , gar keine Tests vorgesehen. Empirischen Untersuchungen zufolge werden selbst bei einem Überdeckungsgrad von 100% durchschnittlich weniger als 20% der Programmfehler identifiziert. Die Systematik des Tests garantiert nur, dass nicht-ausführbare Programmteile erkannt werden. Sonstige Fehler werden nahezu nur zufällig gefunden.

Zweigüberdeckungs-  
test,  
 $C_1$ -Test

Zweigüberdeckungs-  
grad

### 9.1.3 Zweigüberdeckung

Eine 100%-ige Anweisungsüberdeckung unserer Beispielfunktion können wir schon erreichen, ohne dass wir eine Ziffernfolge eingeben müssen, die einem Wert größer als 1 entspricht. Das liegt daran, dass der **else**-Zweig der **if**-Anweisung des Knotens  $n_{if2}$  keine Anweisungen enthält und daher für dieses Kriterium nicht relevant ist. Wir konzentrieren uns daher auf die Zweige des Kontrollflussgraphen, das entsprechende Testverfahren heißt *Zweigüberdeckungstest* ( $C_1$ -Test).

Als Maß für den *Zweigüberdeckungsgrad* wird das Verhältnis der Anzahl der besuchten Zweige zu der Gesamtanzahl der Zweige des Kontrollflussgraphen definiert. Offensichtlich umfasst der Zweigüberdeckungstest den Anweisungsüberdeckungstest in dem Sinn, dass eine vollständige Zweigüberdeckung eine vollständige Anweisungsüberdeckung impliziert.

$$C_{\text{Zweig}} \stackrel{\text{def}}{=} \frac{\text{Anzahl der besuchten Zweige}}{\text{Gesamtanzahl der Zweige}}$$

Eine vollständige Zweigüberdeckung bedeutet also  $C_{\text{Zweig}} = 1$ .

#### Beispiel 9.1.3.1

Wir betrachten die beiden Pfade aus Beispiel 9.1.2.1 und stellen fest, dass nur der Zweig ( $n_{if2}$ ,  $n_{\text{value}}$ ) nicht durchlaufen wird, vgl. Abbildung 9.4 (a), (b). Pfad 3 aus Abbildung 9.4 (c)

$$(n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{if1}, n_{if2}, n_{\text{value}}, n_{\text{pos}}, n_{\text{while}}, n_{if3}, n_{\text{else3}}, n_{\text{final}})$$

enthält diesen fehlenden Zweig. Die Zweigüberdeckung erhöht sich dadurch von 95% auf die gewünschten 100%. Der Pfad wird bei Eingabe einer einzigen Ziffer durchlaufen, deren Wert die erwartete Rückgabe bildet. Der assoziierte Testfall lautet also:  $\{(z, \text{konvertiere}(z)) \mid z \in \{'0', '1', \dots, '9'\}\}$ . Als Testdatum eignet sich z.B. ('9', 9).

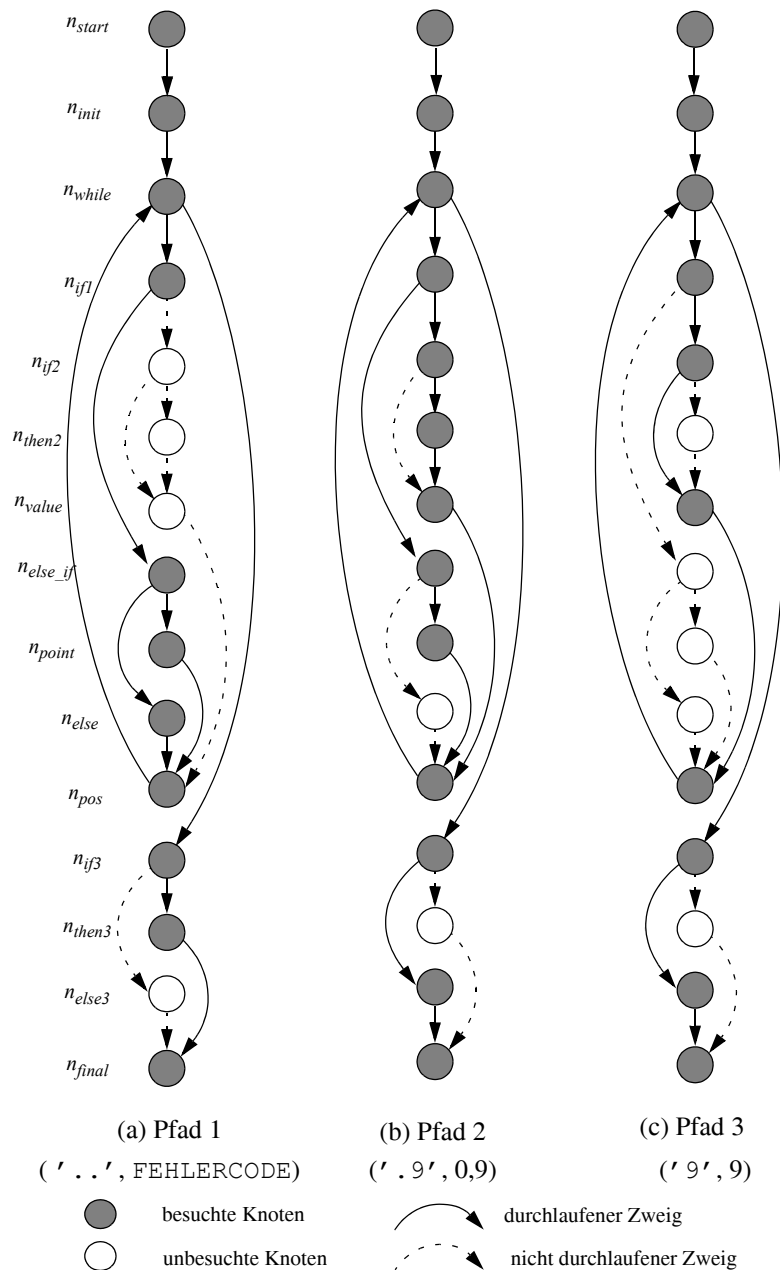


Abbildung 9.4 : Vollständige Zweigüberdeckung der Funktion *werteZiffernfolgeAus*



Der Zweigüberdeckungstest spürt systematisch nicht-ausführbare Programmzweige auf, denn eine vollständige Zweigüberdeckung ist nur erreichbar, wenn jeder Programmzweig auch ausführbar ist. Empirische Studien zeigen, dass durch einen  $C_1$ -Test durchschnittlich ein Viertel aller Fehler entdeckt werden — in der Mehrzahl fehlerhafte Schleifen und **if**-Bedingungen und nur wenige Berechnungsfehler, wie z.B. fehlerhafte Ausdrücke in Zuweisungen oder Fehler durch implizite Typkonvertierungen.

## Entscheidung

Zum Erzielen einer vollständigen Zweigüberdeckung muss jeder von einem Knoten des Kontrollflussgraphen ausgehende Zweig im Test passiert werden. Hat ein Knoten mehrere Nachfolger, so bedeutet das, dass hier eine *Entscheidung* über den weiteren Kontrollfluss stattfindet (der Knoten enthält dann z.B. eine **if**-Anweisung oder eine Entscheidung über einen Schleifenabbruch). Um in diesem Fall sämtliche ausgehenden Zweige zu überdecken, muss die entsprechende Entscheidung einmal positiv und einmal negativ ausfallen. Die Zweigüberdeckung entspricht also einer Überdeckung aller Programmentscheidungen – was wir als Minimalanforderung an einen gründlichen Softwaretest ansehen.

Aufgabe 9.1.3.2

Wir betrachten die Problemstellung, eine natürliche Zahl<sup>1</sup>  $n$  „nach oben durch eine Zehnerpotenz abzuschätzen“, d.h. die kleinste Zehnerpotenz  $10^x$  zu finden, die größer oder gleich  $n$  ist. Für  $n = 0$  sei das Ergebnis ebenfalls 0.

Eine formale Problemspezifikation lautet wie folgt:

Eingabe:  $n \in \mathbb{IN}$   
 Vorbedingung: —  
Ausgabe:  $m \in \mathbb{IN}$   
 Nachbedingung:  $m = 0$ , falls  $n = 0$ ,  
 $m = 10$ , falls  $n = 1$ ,  
 $m = 10^x$  mit  $10^{x-1} < n \leq 10^x$ , falls  $n > 1$

Wir greifen auf die Typdefinition

```
type
tNatZahl = 0..maxint;
```

zurück und bieten als Lösung folgende zu testende (möglicherweise fehlerhafte) Funktion an:

```
function ObereSchranke (inZahl: tNatZahl): tNatZahl;
{ bestimmt die kleinste Zehnerpotenz  $\geq$  inZahl. }
var
  Zahl,
  Schranke : tNatZahl;

begin
  if inZahl = 0 then
    Schranke := 0
  else
    begin
      Zahl := inZahl;
      Schranke := 1;
    repeat
```

---

1. Wir zählen dabei auch die 0 mit zu den natürlichen Zahlen



```

    Zahl := Zahl div 10;
    Schranke := Schranke * 10
until Zahl = 0
end; { if }
    ObereSchranke := Schranke
end; { ObereSchranke }

```

- Zeichnen Sie den kompakten Kontrollflussgraphen zu ObereSchranke.
- Geben Sie zwei Pfade an, mit denen insgesamt eine vollständige Zweigüberdeckung (d.h. ein Zweigüberdeckungsgrad von 100%) erreicht wird.
- Geben Sie zu jedem der beiden Pfade den assoziierten Testfall an.
- Wählen Sie aus jedem Testfall mindestens ein Testdatum als Repräsentanten und prüfen Sie, ob der Prüfling für diese Testdaten jeweils korrekt funktioniert oder ob ein Fehler aufgedeckt wird.



Abschließend weisen wir auf eine Eigenschaft des Zweigüberdeckungsmaßes hin, die leicht zu einer Überschätzung der durchgeführten Testaktivitäten führt: In Beispiel 9.1.3.1 brachten bereits die ersten beiden Testfälle eine Zweigüberdeckung von fast 95%, während der dritte und genauso wichtige Testfall gerade noch 5% zur vollständigen Überdeckung beitrug. Allgemein ist zu beobachten, dass schon mit wenigen Testfällen ein sehr hoher Überdeckungsgrad erreicht wird und dieser sich unter Hinzunahme weiterer Testfälle in immer kleineren Schritten dem maximalen Wert 1 nähert, vgl. Abbildung 9.5.

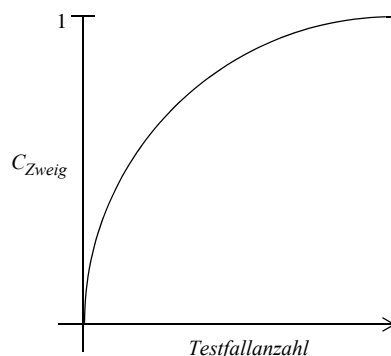


Abbildung 9.5 : Zweigüberdeckungsgrad als Funktion der Testfallanzahl

Gleiches gilt auch für ähnlich definierte Maße, insbesondere für  $C_{\text{Anweisung}}$ .

#### 9.1.4 Minimale Mehrfach-Bedingungsüberdeckung

Wie wir im letzten Abschnitt festgestellt haben, entspricht die Zweigüberdeckung einer Überdeckung aller Programmentscheidungen. Letztere erfolgen anhand von *Bedingungen* (*Prädikaten*). Für eine vollständige Zweigüberdeckung muss jede solche Bedingung, die über eine Verzweigung im Kontrollfluss entscheidet (also z.B.

Prädikate

eine **if**-, **while**- oder **until**-Bedingung), in mindestens einem Testlauf zu `true` und in mindestens einem Testlauf zu `false` ausgewertet werden.

Betrachten wir z.B. die folgende bedingte Anweisung:

```
if (i<0) or (j<0) then ...
```

atomare Prädikate  
(Atome)

Hier fällt die Entscheidung über die Ausführung des **then**- oder **else**-Zweiges anhand der Bedingung "`(i<0) or (j<0)`". Diese ist zusammengesetzt, und zwar aus den beiden *atomaren* Bedingungen (kurz: *Atome*) "`i<0`" sowie "`j<0`". Nehmen wir nun beispielsweise zwei Testläufe an: Im ersten Testlauf gelte sowohl `i ≥ 0` als auch `j ≥ 0`. Im zweiten Testlauf gelte zwar `i < 0`, jedoch `j ≥ 0`. Mit diesen beiden Testläufen erreichen wir eine vollständige Zweigüberdeckung (bezogen auf obige Beispiel-**if**-Anweisung) – obwohl wir gar nicht getestet haben, wie sich der Prüfling im vom Programmierer explizit berücksichtigten Fall `j < 0` verhält. Dieses Beispiel demonstriert den Nachteil der Zweigüberdeckung, dass sie lediglich die gesamte Bedingung einer Entscheidung berücksichtigt, unabhängig davon wie komplex diese auch zusammengesetzt sein mag. *Bedingungsüberdeckungsmaße* versuchen, diesem Nachteil zu begegnen, indem sie auch Teilbedingungen einzeln betrachten.

Bedingungsüber-  
deckungsmaße

einfache Bedin-  
gungsüberdeckung

Das schwächste Bedingungsüberdeckungsmaß ist die so genannte *einfache Bedingungsüberdeckung*, die ausschließlich die einfachen (d.h. atomaren) Teilbedingungen überdeckt, also für jedes in einer Entscheidung vorkommende Atom verlangt, dass es in mindestens einem Testlauf den Wert `true` und in mindestens einem Testlauf den Wert `false` annehmen muss. Betrachten wir wieder obiges Beispiel, so würde zur einfachen Bedingungsüberdeckung ein Testlauf, in dem zum Zeitpunkt der Entscheidung `i < 0` und `j ≥ 0` gilt, sowie ein zweiter Testlauf mit `i ≥ 0` und `j < 0` völlig genügen. Die **if**-Bedingung würde dann jedoch in beiden Testläufen zu `true` ausgewertet, d.h. der **else**-Zweig würde nicht getestet! Die einfache Bedingungsüberdeckung ist folglich ein Kriterium, das nicht einmal die Zweigüberdeckung umfasst. Ein *einfacher Bedingungsüberdeckungstest* (auch *C<sub>2</sub>-Test* genannt) sollte daher zumindest nicht isoliert, sondern höchstens als Ergänzung zu einem Zweigüberdeckungstest durchgeführt werden. Diese Kombination wird auch als *Bedingungs-/Entscheidungsüberdeckungstest* bezeichnet ([Lig02]).

C<sub>2</sub>-Test

Bedingungs-/  
Entscheidungsüber-  
deckungstest

Mehrfach-Bedin-  
gungsüberdeckung

Das andere Extrem der Bedingungsüberdeckungsmaße ist die so genannte *Mehrfach-Bedingungsüberdeckung*, welche für jede Entscheidung die Überdeckung aller Wahrheitswertkombinationen der atomaren Teilprädikate fordert. Das ist das umfassendste Bedingungsüberdeckungskriterium, hat allerdings das Problem, dass die mögliche Anzahl dieser Wertekombinationen – und damit die Anzahl der Testfälle – exponentiell mit der Anzahl der in einer Bedingung enthaltenen Atome wächst. Bereits für obige Beispiel-Bedingung aus nur zwei Atomen ergeben sich demnach 4 Testfälle, nehmen wir aber z.B. eine etwas komplexere Entscheidung aus 5 Atomen an, so ergäben sich allein zu deren Abdeckung bereits  $2^5 = 32$  Testfälle – und ggf. weitere im Prüfling auftretende Entscheidungen sind dabei noch gar nicht berücksichtigt. Ein *Mehrfach-Bedingungsüberdeckungstest* (auch *C<sub>3</sub>-Test* genannt)

C<sub>3</sub>-Test

ist daher für Prüflinge mit vielen oder komplexen Entscheidungen schnell kaum noch handhabbar.

Einen guten Kompromiss stellt die *minimale Mehrfach-Bedingungsüberdeckung* dar, die wir im Folgenden genauer betrachten wollen. Statt – wie die Mehrfach-Bedingungsüberdeckung – alle beliebigen Wertekombinationen der atomaren Teilbedingungen zu betrachten, orientiert sie sich an deren Zusammensetzung: Für jede Entscheidung ist die Gesamtbedingung sowie jede ihrer Teilbedingungen (egal ob atomar oder zusammengesetzt) zu überdecken, d.h. sowohl zu `true` als auch zu `false` auszuwerten. Betrachten wir z.B. die zusammengesetzte Bedingung "`((i<0) or (j<0)) and (i<>j)`": Dann müssen für eine minimale Mehrfach-Bedingungsüberdeckung neben der Gesamtbedingung auch noch die Teilbedingungen "`(i<0) or (j<0)`", "`i<0`", "`j<0`" sowie "`i<>j`" jeweils überdeckt werden. Dadurch wird – anders als bei der Bedingungs-/Entscheidungsüberdeckung – die hierarchische Struktur von Bedingungen berücksichtigt und jeder Schachtelungsstufe die gleiche Aufmerksamkeit gewidmet.

minimale Mehrfach-Bedingungsüberdeckung

Bezeichnen wir die Menge der bereits mindestens einmal zu `true` ausgewerteten Prädikate mit  $\text{Prädikate}_{\text{true}}$  und die der mindestens einmal zu `false` ausgewerteten mit  $\text{Prädikate}_{\text{false}}$ , so erhalten wir als Maß für die Überdeckung

$$C_{\text{MinMehrfach}} \stackrel{\text{def}}{=} \frac{\text{Anzahl Prädikate}_{\text{true}} + \text{Anzahl Prädikate}_{\text{false}}}{2 * \text{Anzahl der Prädikate}}$$

#### Beispiel 9.1.4.1

Die Funktion `werteZiffernfolgeAus` aus unserem begleitenden Beispiel 9.1.1.3 enthält fünf Entscheidungen (Verzweigungen im Kontrollfluss), deren Bedingungen wir untersuchen müssen: vier **if**-Anweisungen und eine **while**-Schleife. In Tabelle 9.1 haben wir alle zehn Atome und die vier zusammengesetzten Prädikate in der Reihenfolge ihres Auftretens aufgelistet. Beachten Sie, dass jedes aufgelistete Prädikat zu einer Entscheidung gehört, die an einer bestimmten Stelle im Kontrollfluss (Verzweigungsknoten im Kontrollflussgraphen) getroffen wird. Die Zeilen E und K in Tabelle 9.1 enthalten z.B. zwar das gleiche Prädikat, gehören jedoch jeweils zu einer anderen Entscheidung: Zeile E enthält die Bedingung zur zweiten **if**-Entscheidung, Zeile K eine Teilbedingung zur vierten **if**-Entscheidung. Da diese Entscheidungen zu anderen Zeitpunkten in der Ausführung und damit ggf. unter anderen Variablenbelegungen getroffen werden, müssen beide Vorkommen des Prädikats separat untersucht werden.

Bei der Festlegung der Testfälle gehen wir nun iterativ vor, indem wir die Prädikate der Liste nacheinander abarbeiten, bis der geforderte Überdeckungsgrad erreicht ist. Der erste Testfall ist die Menge aller Testdaten, unter deren Eingabe das Prädikat A (`Position <= length (inZiffernString)`) zu `true` ausgewertet wird – das sind die Testdaten zu allen Eingaben außer dem leeren String<sup>1</sup>. Wir wählen daraus das Testdatum ('9', 9) und führen die Funktion damit aus. Dabei notieren wir in der Tabelle, welche anderen Prädikate während des Testlaufs noch ausgewertet werden und welche Wahrheitswerte sie annehmen. (Beachten Sie, dass durch die

Schleife manche Prädikate während des Testlaufs mehr als einmal ausgewertet werden.) Der Überdeckungsgrad beträgt nach diesem Testfall 46% und wir können die Untersuchung der Prädikate A und C bereits abschließen, da diese sowohl zu `true` als auch zu `false` ausgewertet worden sind. Wir suchen nun in der Tabelle nach dem ersten Prädikat, das noch nicht vollständig überdeckt ist – sei es, weil es von dem untersuchten Testfall noch nicht betroffen war oder bisher nur zu einem der beiden Wahrheitswerte ausgewertet wurde. Das ist in diesem Fall Prädikat B (fehlerfrei), das bisher nur den Wert `true` angenommen hat. Der nächste Testfall umfasst daher alle Testdaten, die das Prädikat `fehlerfrei` (aus der **while**-Bedingung) zu `false` auswerten. Wir wählen als Testdatum ('z', FEHLERCODE), führen das Programm aus und protokollieren erneut die Auswirkungen auf die anderen noch nicht überdeckten Prädikate. Der Überdeckungsgrad steigt durch dieses Eingabedatum auf 71%. Auf diese Weise fahren wir fort, bis letztendlich alle Prädikate überdeckt sind.

Prädikate		Eingabedaten					
		'9'	'z'	'9'	'..'	'.'	'.'
A	Position <= length(inZiffernString)	T,F					
B	fehlerfrei	T	T,F				
C	A and B (while-Bedingung)	T,F					
D	pruefeZeichen (Zeichen)	T	F				
E	woBinIch = nachDemKomma	F	-	T			
F	Zeichen = '.'	—	F	T			
G	woBinIch = vorDemKomma	—	T	T	T,F		
H	F and G (3. if-Bedingung)	—	F	T			
I	not fehlerfrei	F	T				
J	length(inZiffernString)=0	F	F	F	F	T	
K	woBinIch = nachDemKomma	F	F	T			
L	length(inZiffernString)=1	T	T	F			
M	K and L	F	F	F	F	F	T
N	I or J or M (4. if-Bedingung)	F	T				

Tabelle 9.1 Minimale Mehrfach-Bedingungsüberdeckung zu `werteZiffernfolgeAus`

1. In diesem Beispiel verzichten wir auf formale Angaben der Testfälle. Erstens wären diese aufgrund der Kriterien, nach denen wir hier die Testfälle bilden, nur sehr kompliziert formal auszudrücken, zweitens beschränken wir uns im weiteren Vorgehen ohnehin auf jeweils ein Testdatum daraus.

Wir haben uns in der Betrachtung des Verfahrens auf das Überdeckungsmaß konzentriert. Wir dürfen aber natürlich nicht das eigentliche Ziel des dynamischen Tests vergessen, nach jedem Testlauf auch das tatsächliche Ergebnis mit dem erwarteten Ergebnis zu vergleichen. Zum Protokollieren der Testergebnisse kann man z.B. obige Tabelle um zwei Zeilen „erwartetes Ergebnis“ und „tatsächliches Ergebnis“ ergänzen.



#### Aufgabe 9.1.4.2

Wir geben im Folgenden eine (möglicherweise fehlerhafte) Funktion an, die bestimmen soll, ob ein Feld aufsteigend sortiert ist und für aufsteigend sortierte Felder zusätzlich unterscheidet, ob darin gleiche Werte aufeinander folgen oder das Feld duplikatfrei ist, also jeder Wert im Feld echt größer als sein Vorgängerwert im Feld ist. Da diese Implementierung mehrere zusammengesetzte Bedingungen enthält, soll sie einem minimalen Mehrfach-Bedingungsüberdeckungstest unterzogen werden:

```

const
FELDMAX=5;

type
tBereich = 1..FELDMAX;
tFeld = Array[tBereich] of integer;
tRueckgabe =
    (unsortiert, sortiert, sortiertDuplikatfrei);

function pruefeFeld(inFeld:tFeld): tRueckgabe;
{Prüft, ob inFeld aufsteigend sortiert ist.
Rueckgabe:
-sortiert, falls das Feld aufsteigend sortiert ist,
  also inFeld[i] >= inFeld[i-1] für 2 <= i <= FELDMAX
-sortiertDuplikatfrei, falls das Feld aufsteigend
  sortiert und duplikatfrei ist, also
  inFeld[i] > inFeld[i-1] für 2 <= i <= FELDMAX
-unsortiert sonst}

var
i: tBereich;
dupl: boolean;

begin
  i:=2;
  while (i < FELDMAX) and (inFeld[i-1] <= inFeld[i]) do
    begin
      if inFeld[i-1] = inFeld[i] then
        dupl:=true;
      i := i+1;
    end;

```

```

if (i < FELDMAX) or
    ((i = FELDMAX) and (inFeld[i-1] > inFeld[i])) then
    pruefeFeld := unsortiert
else
    if (inFeld[i-1] < inFeld[i]) and not dupl then
        pruefeFeld := sortiertDuplikatfrei
    else
        pruefeFeld := sortiert;
end;

```

- a) Ermitteln Sie zunächst alle zu überprüfenden Prädikate für den minimalen Mehrfach-Bedingungsüberdeckungstest.
- b) Führen Sie den Test durch, indem Sie sukzessive Testdaten wählen und die Überdeckung der unter a) ermittelten Prädikate protokollieren, bis alle Prädikaten überdeckt wurden. Die Eingabe eines Testdatums, also die Ausprägung eines Feldes vom Typ `tFeld`, geben Sie einfach als Aufzählung von 5 Zahlen an. Geben Sie zu jedem gewählten Eingabedatum neben dem erwarteten Ergebnis auch das tatsächliche Ergebnis an (zu dessen Ermittlung könnte es sinnvoll sein, wenn Sie den Prüfling tatsächlich dynamisch auf einem Computer testen.)

Wir empfehlen, für die Durchführung dieser Aufgabe eine Tabelle nach folgendem Muster anzulegen (mit entsprechend weiteren Zeilen und Spalten):

<i>Prädikate</i>		inFeld														
A	erstes Prädikat															
B	...															
	letztes Prädikat															
<i>erwartete Ausgabe</i>																
<i>tatsächliche Ausgabe</i>																

Tragen Sie dort also in der linken Spalte die in Teil a) ermittelten Prädikate ein, oben über jeder weiteren Spalte die fünf Zahlen, mit denen das einzugebende Feld belegt sein soll. In den letzten beiden Tabellenzeilen vermerken Sie das erwartete Ergebnis zum gewählten Eingabedatum sowie das tatsächliche Testergebnis.

- c) *Ergänzungsaufgabe* (keine Übung zur minimalen Mehrfach-Bedingungsüberdeckung):

Der Prüfling enthält einen Fehler, den Ihre Tests aufgedeckt haben sollten (allerdings gibt es auch Compiler, die obigen Prüfling in ein tatsächlich fehlerfrei funktionierendes Maschinenprogramm übersetzen, mit denen also beim dynamischen Test kein Fehlersymptom auftritt).

Debuggen Sie den Prüfling, d.h. finden Sie die Fehlerursache und schlagen Sie eine Korrektur vor.



### 9.1.5 Boundary-interior-Pfadtest

Schleifen werden von den bisher betrachteten Verfahren noch nicht systematisch getestet. Selbst vollständige Zweig- und Bedingungsüberdeckungstests kommen oft mit nur wenigen und willkürlich gewählten Schleifenwiederholungen aus. Man könnte versuchen, mit einer *vollständigen Pfadüberdeckung* zum Ziel zu kommen. Diese umfasst alle Pfade, zu denen es Programmausführungen gibt, also auch alle möglichen Schleifendurchläufe. Leider ist die Anzahl der Pfade (und der mit ihnen assoziierten Testfälle) schnell nicht mehr handhabbar. Die Anzahl der Pfade, die zu einem Programm mit einer einzigen, nicht geschachtelten Schleife gehören, steigt bereits linear in der Anzahl der möglichen Schleifeniterationen, bei mehreren möglichen Pfaden durch eine Schleife wächst die Zahl der zu testenden Pfade bereits exponentiell, womit ein *Pfadüberdeckungstest* (auch *C<sub>4</sub>-Test* genannt) für viele Probleme praktisch undurchführbar wird. Daher wird es notwendig, die Zahl der Testfälle einzuschränken, indem nicht mehr alle durch eine Schleife führenden Pfade, sondern nur noch ausgewählte Pfade getestet werden.

vollständige Pfad-  
überdeckung

C<sub>4</sub>-Test

Der *Boundary-interior-Pfadtest* gehört zu einer Gruppe von Verfahren, die gezielt die Schleifen von Programmen testen, dabei aber die Anzahl der Testfälle klein halten. Der Einfachheit halber gehen wir im Folgenden von einem Prüfling aus, der genau eine Schleife enthält (andernfalls sind für jede weitere Schleife analog die zu testenden Pfade zu bestimmen). Die durch den Test auszuführenden Pfade werden in drei Klassen aufgeteilt: Die erste Klasse enthält alle Pfade, bei denen der Schleifenrumpf überhaupt nicht ausgeführt wird. Bei **while**-Schleifen sind das z.B. Pfade, für welche die Schleifenbedingung unmittelbar zu `false` ausgewertet wird. Die zweite Klasse enthält alle Pfade, bei denen der Schleifenrumpf genau einmal ausgeführt wird (*Grenz- oder Boundary-Test*). Die dritte Klasse umfasst alle Pfade, bei denen der Schleifenrumpf genau  $n$  mal ( $n > 1$ ) ausgeführt wird (*Interior-Test*). Aus Effizienzgesichtspunkten heraus sollte  $n$  möglichst klein gewählt werden.<sup>1</sup> Statt alle möglichen Pfade zu überdecken, beschränkt man sich nun auf die Pfade aus diesen drei Pfadklassen.

Boundary-interi-  
or-Pfadtest

Boundary-Test  
Interior-Test

#### Beispiel 9.1.5.1

Wir betrachten zunächst die Klasse der *die Schleife nicht durchlaufenden Pfade*. Für die Funktion `werteZiffernfolgeAus` aus Beispiel 9.1.1.3 enthält diese Klasse genau zwei Pfade:

1.  $(n_{start}, n_{init}, n_{while}, n_{if3}, n_{then3}, n_{final})$
2.  $(n_{start}, n_{init}, n_{while}, n_{if3}, n_{else3}, n_{final})$

---

1. In früheren Kursversionen umfasste die dritte Klasse alle Pfade, bei denen der Schleifenrumpf genau zweimal ausgeführt wird, also stets  $n=2$  gewählt wird. Wir haben dies geändert, um konform zur gängigen Literatur zu sein.

Der leere String ist die einzige Eingabe, die zum Durchlauf des ersten Pfades führt, der assoziierte Testfall lautet also: { (' ', FEHLERCODE) }. Der zweite Pfad ist gar nicht ausführbar, der assoziierte Testfall ist also die leere Menge. Da zu nicht ausführbaren Pfaden keine Tests durchgeführt werden können, werden wir sie im Folgenden nicht mehr alle explizit mit aufführen.

Zur Bestimmung der übrigen Testfälle greifen wir auf Abbildung 9.6 zurück, die alle möglichen *Teilpfade* innerhalb der **while**-Schleife zeigt.

Wir betrachten als nächstes den *Boundary-Test*. Ein Blick auf Abbildung 9.6 zeigt, dass für den Boundary-Test der Schleife theoretisch vier zu testende Teilpfade zu durchlaufen sind. Praktisch kann jedoch der Fall (d) niemals eintreten: Das Prädikat `woBinIch = nachDemKomma` der **if**-Anweisung des Knotens  $n_{if2}$  kann beim ersten Schleifen-Durchlauf nur zu `false` ausgewertet werden, da `woBinIch` zu Beginn mit `vorDemKomma` initialisiert wird und bis zu diesem Zeitpunkt nicht verändert wird. Die Pfadklasse für den Boundary-Test umfasst also, wenn wir uns auf die tatsächlich ausführbaren Pfade beschränken, alle Pfade (von  $n_{start}$  nach  $n_{final}$ ), welche die Schleife auf einem der Teilpfade (a), (b) oder (c) durchlaufen. Aufgrund der Verzweigung in Knoten  $n_{if3}$  lassen sich zwei Pfade pro Teilpfad bilden, jedoch ist davon wiederum nur jeweils einer ausführbar. Es verbleiben also insgesamt drei ausführbare Pfade in der Boundary-Klasse:

1. ( $n_{start}, n_{init}, n_{while}, n_{if1}, n_{else\_if}, n_{else}, n_{pos}, n_{while}, n_{if3}, n_{then3}, n_{final}$ )  
 Assoziierter Testfall:  
 { ( $z+s$ , FEHLERCODE) |  $z \notin \{ ' . ', ' 0 ', ' 1 ', \dots, ' 9 ' \}$ ,  $s$  beliebiger String }  
 Mögliches Testdatum: (' x ', FEHLERCODE)
2. ( $n_{start}, n_{init}, n_{while}, n_{if1}, n_{else\_if}, n_{point}, n_{pos}, n_{while}, n_{if3}, n_{then3}, n_{final}$ )  
 Assoziierter Testfall: { (' . ', FEHLERCODE) }  
 Einziges Testdatum: (' . ', FEHLERCODE)
3. ( $n_{start}, n_{init}, n_{while}, n_{if1}, n_{if2}, n_{value}, n_{pos}, n_{while}, n_{if3}, n_{else3}, n_{final}$ )  
 Assoziierter Testfall: { ( $z$ , konvertiere( $z$ )) |  $z \in \{ ' 0 ', ' 1 ', \dots, ' 9 ' \}$  }  
 Mögliches Testdatum: (' 9 ', 9)



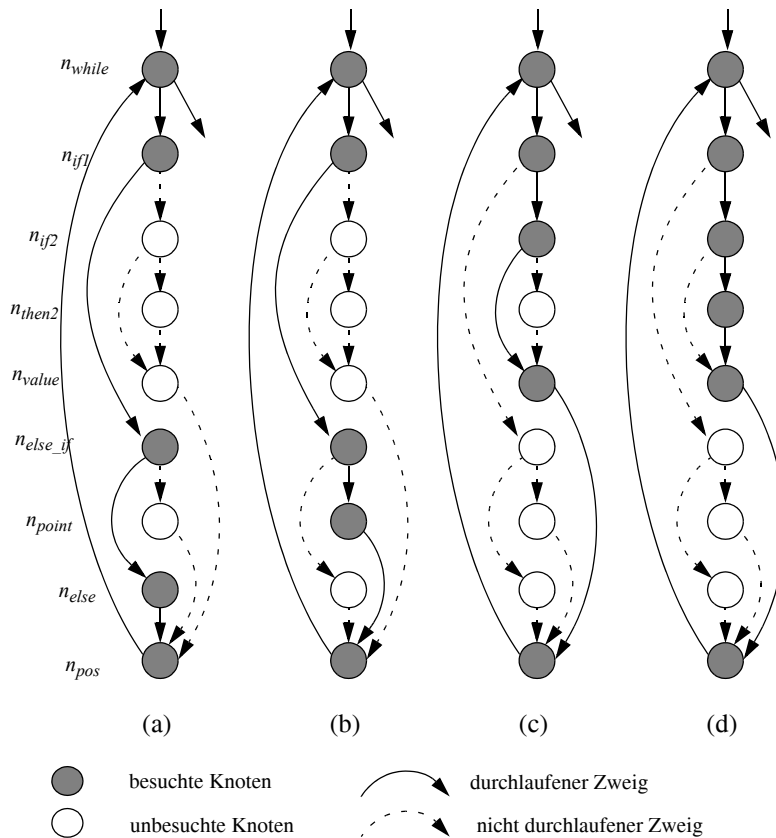


Abbildung 9.6 : Teilpfade im Schleifenrumpf der Funktion *werteZiffernfolgeAus*

Wir wenden uns jetzt dem *Interior-Test* zu und wählen  $n = 2$ . Die Schleife wird genau zweimal durchlaufen, wenn je zwei der in Abbildung 9.6 gezeigten Teilpfade hintereinander ausgeführt werden. (Es kann auch derselbe Teilpfad zweimal ausgeführt werden.) Damit ergeben sich prinzipiell 16 Teilpfade durch die Schleife. Bei näherem Hinschauen sind jedoch nur fünf Kandidaten ausführbar<sup>1</sup>. Wir geben im Folgenden die fünf ausführbaren Pfade der Interior-Klasse an, wobei wir, um die Pfadangaben abzukürzen, die einzelnen Teilpfade durch den Schleifenblock jeweils durch ihren Buchstaben gemäß Abbildung 9.6 ersetzen:

1.  $(n_{start}, n_{init}, (b), (a), n_{if3}, n_{else3}, n_{final})$   
 Assoziierter Testfall:  
 $\{ ('.' + z + s, FEHLERCODE) \mid z \notin \{ '0', '1', \dots, '9' \}, s \text{ beliebiger String} \}$   
 Mögliches Testdatum:  $('.xy', FEHLERCODE)$
2.  $(n_{start}, n_{init}, (b), (d), n_{if3}, n_{then3}, n_{final})$   
 Assoziierter Testfall:  
 $\{ ('.' + z, \text{konvertiere}(z) / 10) \mid z \in \{ '0', '1', \dots, '9' \} \}$   
 Mögliches Testdatum:  $('.9', 0,9)$

1. So wird z.B. die **while**-Schleife nach Durchlaufen des Teilpfades (a) wegen `fehlerfrei = false` stets abgebrochen, so dass kein anderer Pfad mehr angehängt werden kann. Oder Teilpfad (d) kann z.B. ausschließlich auf Teilpfad (b) folgen.

3.  $(n_{start}, n_{init}, (c), (a), n_{if3}, n_{else3}, n_{final})$   
 Assoziierter Testfall:  
 $\{ (y+z+s, FEHLERCODE) \mid y \in \{ '0', \dots, '9' \}, z \notin \{ ' ', '0', \dots, '9' \}, s \text{ beliebiger String} \}$   
 Mögliches Testdatum:  $( '8*', FEHLERCODE )$
4.  $(n_{start}, n_{init}, (c), (b), n_{if3}, n_{then3}, n_{final})$   
 Assoziierter Testfall:  
 $\{ (z+' ', konvertiere(z)) \mid z \in \{ '0', \dots, '9' \} \}$   
 Mögliches Testdatum:  $( '9.', 9 )$
5.  $(n_{start}, n_{init}, (c), (c), n_{if3}, n_{then3}, n_{final})$   
 Assoziierter Testfall:  
 $\{ (y+z, 10 \cdot konvertiere(y) + konvertiere(z)) \mid y, z \in \{ '0', \dots, '9' \} \}$   
 Mögliches Testdatum:  $( '89', 89 )$

Erinnern wir uns an die Spezifikation der Funktion `werteZiffernfolgeAus`, so stellen wir fest, dass die Testfälle des Boundary-interior-Tests die Zeichenfolgen der Längen 1 und 2 für den Eingangsparameter `inZiffernString` abdecken.  $\square$

geschachtelte  
Schleifen

Bei *geschachtelten Schleifen* gehen wir von innen nach außen vor. Wir testen zunächst die innerste Schleife, testen dann deren umschließende Schleife, wobei wir die innere als Black Box interpretieren, usw., bis wir schließlich die äußerste Schleife testen.

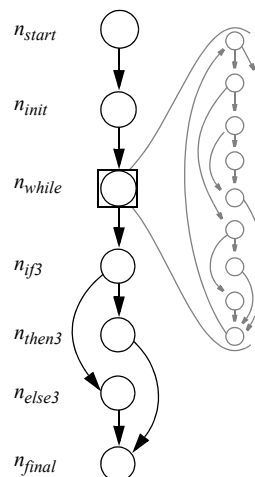


Abbildung 9.7 : Die **while**-Schleife der Funktion `werteZiffernfolgeAus` als Black Box

Die Effektivität des Boundary-interior-Pfadtests liegt empirischen Untersuchungen zufolge um 100% über dem Zweigüberdeckungstest. Es werden also durchschnittlich etwa doppelt so viele Fehler entdeckt.

#### Beispiel 9.1.5.2

Wir betrachten die folgende Problemspezifikation für die ganzzahlige Division zweier natürlicher Zahlen mit Rest:

Eingabe: zwei Zahlen  $a$  und  $b \in \mathbb{N}$

Ausgabe: zwei Zahlen  $p$  und  $q \in \mathbb{N}$

Nachbedingung: 
$$\begin{cases} p = \lfloor a / b \rfloor \wedge q = a - \lfloor a / b \rfloor \cdot b & \text{falls } b \neq 0 \\ p = q = 0 & \text{falls } b = 0 \end{cases}$$

( $\lfloor a / b \rfloor$  entspricht „ $a \text{ div } b$ “ und  $a - \lfloor a / b \rfloor \cdot b$  der Operation „ $a \text{ mod } b$ “ in Pascal.)

Als Lösung bieten wir die folgende Prozedur `Division` an:

```

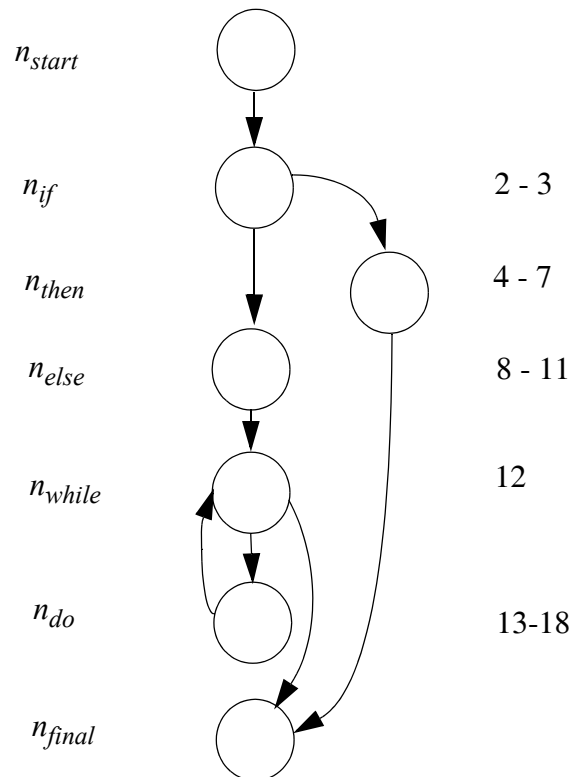
type
  tNatZahl = 0..maxint;

1  procedure Division (
        inDividend,
        inDivisor: tNatZahl;
        var outErg,
        outRest: tNatZahl);
    { ganzzahlige Division zweier natuerlicher Zahlen
      mit Rest }

2  begin
3    if inDivisor = 0 then
4      begin
5        outRest := 0;
6        outErg := 0
7      end
8    else
9      begin
10     outErg := 0;
11     outRest := inDividend;
12     while outRest >= inDivisor do
13       begin
14         outRest := inDividend - inDivisor;
15         outErg := outErg + 1
16       end { while }
17     end { if }
18 end; { Division }

```

Bei dem Kontrollflussgraphen zu `Division` in der folgenden Abbildung haben wir statt der Programmzeilen deren Zeilennummern neben den Knoten notiert):



Wir wollen jetzt einen Boundary-interior-Test durchführen und beginnen mit dem Aufstellen der drei Klassen von Pfaden.

Klasse 0 der Pfade, die die Schleife nicht durchlaufen<sup>1</sup>, umfasst die Pfade  $(n_{start}, n_{if}, n_{then}, n_{final})$  und  $(n_{start}, n_{if}, n_{else}, n_{while}, n_{final})$ .

Klasse 1 (Boundary-Klasse) umfasst genau den Pfad  $(n_{start}, n_{if}, n_{else}, n_{while}, n_{do}, n_{while}, n_{final})$ .

Klasse 2 (Interior-Klasse für  $n = 2$ ) umfasst den genau den Pfad  $(n_{start}, n_{if}, n_{else}, n_{while}, n_{do}, n_{while}, n_{do}, n_{while}, n_{final})$ .

Danach versuchen wir, die assoziierten Testfälle zu bestimmen<sup>2</sup>:

Klasse 0, 1. Pfad:

$$T_0 = \{(a, 0; 0, 0) \mid a \in \mathbb{N}\}$$

- 
1. Da wir neben dem Test der Schleife auch an einer vollständigen Zweigüberdeckung als minimalem Testkriterium interessiert sind, rechnen wir hier zu dieser Klasse nicht nur den „abweisenden Pfad“ für 0-maligen Schleifendurchlauf, sondern auch den Pfad durch den **then**-Zweig, der die Schleife gar nicht erst erreicht.
  2. Testdaten notieren wir hier in der Form  $(a, b; p, q)$ , wobei wir zur besseren Lesbarkeit ein Semikolon als Trenner zwischen Ein- und Ausgabedaten verwenden.

Klasse 0, 2. Pfad:

$$T_1 = \{(a, b; p, q) \mid a, b \in \mathbb{N}, 0 \leq a < b, p = \lfloor a/b \rfloor, q = a - \lfloor a/b \rfloor \cdot b\}$$

Klasse 1:

$$T_2 = \{(a, b; p, q) \mid a, b \in \mathbb{N}, 0 < b \leq a < 2b \text{ und } p = \lfloor a/b \rfloor, q = a - \lfloor a/b \rfloor \cdot b\}$$

Klasse 2:

$$T_3 = \{ \}$$

Es ist nicht möglich, Testdaten für einen zweimaligen Durchlauf der Schleife zu finden,  $T_{4,3}$  ist also leer. Das gilt ebenfalls, wenn wir für den Interior-Test  $n > 2$  Schleifendurchläufe wählen. Das liegt daran, dass, nachdem die Schleife das zweite Mal betreten wird, sich der Wert von `outRest` nicht mehr ändert, also eine Endlosschleife vorliegt (während  $n$  – egal wie groß gewählt – stets endlich ist). Um eine funktionierende Schleife zu erhalten, hätten wir statt der Zuweisung

```
outRest := inDividend - inDivisor
```

die Zuweisung

```
outRest := outRest - inDivisor
```

schreiben müssen.

In diesem Beispiel finden wir die Endlosschleife bei dem Versuch, Testdaten für die Klasse 3 zu finden, also schon vor der Programmausführung. Oft ist es aber so, dass man glaubt, geeignete Testdaten für den Boundary- oder Interior-Test gefunden zu haben und eine Endlosschleife sich erst bei der Programmausführung herausstellt.

Übrigens ist es auch möglich, die Endlosschleife in unserem Beispiel mit einem vollständigen Zweigüberdeckungstest zu finden, falls man nämlich einen Pfad wählt, der die Schleife mehrfach durchläuft.



### Aufgabe 9.1.5.3

Betrachten Sie noch einmal die Schleife aus Aufgabe 9.1.3.2, die nun zusätzlich zum bereits durchgeführten Zweigüberdeckungstest mit dem Boundary-Interior-Test noch genauer untersucht werden soll. Da wir uns hier auf die Schleife konzentrieren, betrachten wir nur Pfade, die den **else**-Zweige durchlaufen, da nur diese die **repeat**-Schleife überhaupt erreichen. Geben Sie die drei Pfadklassen des Boundary-Interior-Tests an, ordnen Sie den den **else**-Zweig durchlaufenden Pfad aus Aufgabe 9.1.3.2 Teil b) der korrekten Pfadklasse zu und geben Sie die Pfade zu den beiden anderen Pfadklassen samt assoziiertem Testfall an. Nehmen Sie für den Interior-Test  $n = 2$  an.



Aufgabe 9.1.5.4

Als weiteres, etwas komplexeres Beispiel zum Boundary-Interior-Test betrachten wir das Problem, das Maximum einer linearen Liste von Integer-Zahlen zu bestimmen. Dabei beschränken wir uns auf nicht-leere Listen, da sich für leere Listen kein Maximum definieren lässt. Genauere Problemspezifikation:

Eingabe: Eine Liste ganzer Zahlen

Vorbedingung: Liste ist nicht leer (enthält min. ein Element)  
(andernfalls wäre das Maximum nicht definiert)

Ausgabe: ganze Zahl

Nachbedingung: Die ausgegebene Zahl ist die größte Zahl in der Liste.

Für die Liste verwenden wir folgende Typen:

```
type
tRefElement = ^tElement;
tElement = record
    info:integer;
    next:tRefElement;
end;
```

Wir bieten folgende Implementierung an:

```
1  function maximum(inRefAnfang:tRefElement): integer;
   { Bestimmt das Maximum der Liste, deren Anfang von
     inRefAnfang referenziert wird.
     Vorbedingung: Übergebene Liste ist nicht leer
     (inRefAnfang <> nil) }
2  var
3      max: integer;
4      lauf: tRefElement;
5  begin
6      max := inRefAnfang^.info;
7      lauf := inRefAnfang^.next;
8      while lauf <> nil do
9          begin
10         if lauf^.info > max then
11             max := lauf^.info;
12             lauf := lauf^.next;
13         end;
14     maximum := max;
15 end;
```

- a) Erstellen Sie zunächst den kompakten Kontrollflussgraphen der Funktion maximum. Sie können dabei, analog zu Beispiel 9.1.5.2, statt der Programmzeilen einfach deren Nummern neben die Knoten schreiben.

- b) Bestimmen Sie (möglichst wenige) Pfade für einen vollständigen Boundary-Interior-Pfadtest.
- c) Geben Sie zu jedem der Pfade den assoziierten Testfall an.  
 Zur Notation von Listen als Eingabedaten können Sie folgende Kurzschreibweise verwenden: Zählen Sie einfach die Zahlen in der Reihenfolge, in der sie in der Liste stehen sollen, auf, durch Kommas getrennt, und fassen Sie diese Aufzählung zur Kennzeichnung einer Liste in eckige Klammern ein. So würde die leere Liste z.B. als „[]“ notiert, die Liste der Zahlen Eins, Zwei und Drei als „[1, 2, 3]“.
- Als weitere Hilfe geben wir hier einen der Testfälle beispielhaft vor, Sie müssen ihn nur noch dem entsprechenden Pfad zuordnen:
- $$\{ ([x, y], x) \mid x \geq y \}^1$$
- d) Wählen Sie zu jedem Testfall ein Testdatum als Repräsentant aus.
- e) Die Implementierung ist korrekt. Im Folgenden beschreiben wir drei typische Fehler, die dem Programmierer hätten unterlaufen können. Überlegen Sie zu jedem davon, ob er schon bei der Testfallbildung (zum Boundary-Interior-Test) aufgedeckt würde oder möglicherweise erst bei einem dynamischen Testlauf auffiele:
- e1) Zeile 8: **while** `lauf^.next` <> **nil** **do**
- e2) Zeile 12 fehlt (wurde vergessen)
- e3) Zeile 14 fehlt (wurde vergessen)



## 9.2 Exkurs: Datenflussorientierte Testverfahren

Jedes Programm besteht neben Kontrollstrukturen auch aus Datenstrukturen. Entsprechend können Programme neben Kontrollflussfehlern auch Datenfehler enthalten. Typische *Datenfehler* sind z.B. Variablen, die definiert, aber nicht benutzt werden, oder wenn versucht wird, nicht initialisierte Variablen auszuwerten. *Datenflussorientierte Testverfahren* verwenden die Zugriffe auf Variablen zur Erzeugung von Testfällen.

Datenfehler

datenflussorientierte  
Testverfahren

### 9.2.1 Kontrollflussgraphen in Datenflussdarstellung

Bevor wir den Kontrollflussgraphen um Datenflussinformationen erweitern können, müssen wir die Begriffe *Definition* und *Benutzung* von Variablen einführen. Wir unterscheiden Zugriffe auf Variablen

Variablendefinition  
Variablenbenutzung

- zur Wertzuweisung, d.h. zur *Definition* einer Variablen,
- zur Berechnung von Werten innerhalb eines Ausdrucks, wobei ein Ausdruck sowohl arithmetischer Natur, als auch ein Zugriff auf die Komponente einer zusammengesetzten Datenstruktur sein kann (*computational-use*, kurz *c-use*, *berechnende Benutzung*), und

berechnende  
Benutzung

1. Dass  $x$  und  $y$  ganze Zahlen sein müssen, geben wir in den Testfällen aus Gründen der Übersichtlichkeit nicht extra mit an. Das gilt ohnehin testfallunabhängig.

prädikative  
Benutzung

- zur Auswertung von Prädikaten in bedingten Anweisungen und Schleifenbedingungen (*predicate-use*, kurz *p-use*, *prädikative Benutzung*).

In einer Anweisung der Form  $v := f(v_1, \dots, v_n)$  wird beispielsweise die Variable  $v$  durch berechnende Benutzung der Variablen  $v_1, \dots, v_n$  definiert. Eine Eingabeanweisung entspricht einer Definition der eingelesenen Variablen, die Ausgabe einer Variablen einer berechnenden Benutzung.

lokale Definition

lokale Benutzung

globale Benutzung

Eine Variable kann in einem Knoten, der einen Block repräsentiert, definiert und anschließend in demselben Block auch berechnend benutzt werden. Eine derartige Definition und Benutzung heißen *lokal*. Eine Variablenbenutzung ist dagegen *global*, wenn die letzte vorangehende Definition der Variablen außerhalb des Blocks erfolgte.

globale Definition

Die letzte Definition einer Variablen in einem Block heißt *global*. Eine global definierte Variable kann auch gleichzeitig lokal definiert sein, wenn nämlich nach der globalen, das heißt der letzten, Definition noch eine lokale Benutzung erfolgt. Global definierte Variablen sind für Zugriffe in nachfolgenden Blöcken wichtig. Ist eine Definition weder lokal noch global, so haben wir es mit einer überflüssigen Anweisung zu tun.

Datenflussdarstellung

Die *Datenflussdarstellung* eines Kontrollflussgraphen entsteht durch Hinzufügen von Informationen über die Definition und Benutzung von Variablen. Definitionen und berechnende Benutzungen werden dem Knoten zugeordnet, der die entsprechende Anweisung enthält. Prädikative Benutzungen treten ausschließlich in der letzten Anweisung von Knoten auf, da danach der Kontrollfluss verzweigt. Sie werden an alle von diesem Knoten ausgehenden Kanten geheftet.

Wir definieren also drei Abbildungen, die den Knoten bzw. Kanten des Graphen Mengen von Variablen zuordnen.

def

1. Die Abbildung *def* bildet jeden Knoten  $n_i$  auf die Menge der in  $n_i$  global definierten Variablen ab.

c-use

2. Durch *c-use* wird jedem Knoten die Menge der Variablen zugeordnet, für die in ihm eine globale berechnende Benutzung existiert.

p-use

3. Die Abbildung *p-use* weist jeder Kante die Menge der Variablen zu, für die in ihr eine prädikative Benutzung erfolgt.

Handelt es sich bei dem Prüfling um eine Prozedur/Funktion, so wird der Datenimport bzw. -export über Parameter (oder globale Variablen) mit Hilfe der Knoten  $n_{start}$  und  $n_{final}$  dargestellt. Ein Datenimport, der nur bei der Übergabe einer Variablen als IN- oder INOUT-Parameter vorliegt, entspricht der Definition dieser Variablen. Diese wird dem Knoten  $n_{start}$  zugeordnet. Ein Datenexport, der nur bei INOUT- oder OUT-Parametern auftritt, entspricht einer berechnenden Benutzung. Diese Information wird dem Knoten  $n_{final}$  zugeordnet. Wird der Datenexport über eine Funktion vorgenommen, wird sie wie ein OUT-Parameter behandelt.



Wie behandeln wir nun die Situation, wenn innerhalb des Prüflings Prozeduren/Funktionen aufgerufen werden? Zunächst einmal gehen wir stets davon aus (bzw. sorgen dafür), dass die aufgerufene Prozedur/Funktion verifiziert oder getestet ist. Unter dieser Voraussetzung liegt beim Aufruf für einen **IN**-Parameter eine berechnende Benutzung und für einen **OUT**-Parameter eine Definition vor. Im Falle eines **INOUT**-Parameters liegt sowohl eine Definition als auch eine berechnende Benutzung vor. Tritt ein Funktionsaufruf in einem Prädikat auf, so werden alle seine Parameter stets prädikativ benutzt. Dabei unterstellen wir unseren guten Programmierstil und benutzen ausschließlich **IN**-Parameter.

### Beispiel 9.2.1.1

Abbildung 9.8 zeigt den Kontrollflussgraphen der Funktion `werteZiffernfolgeAus` in Datenflussdarstellung.

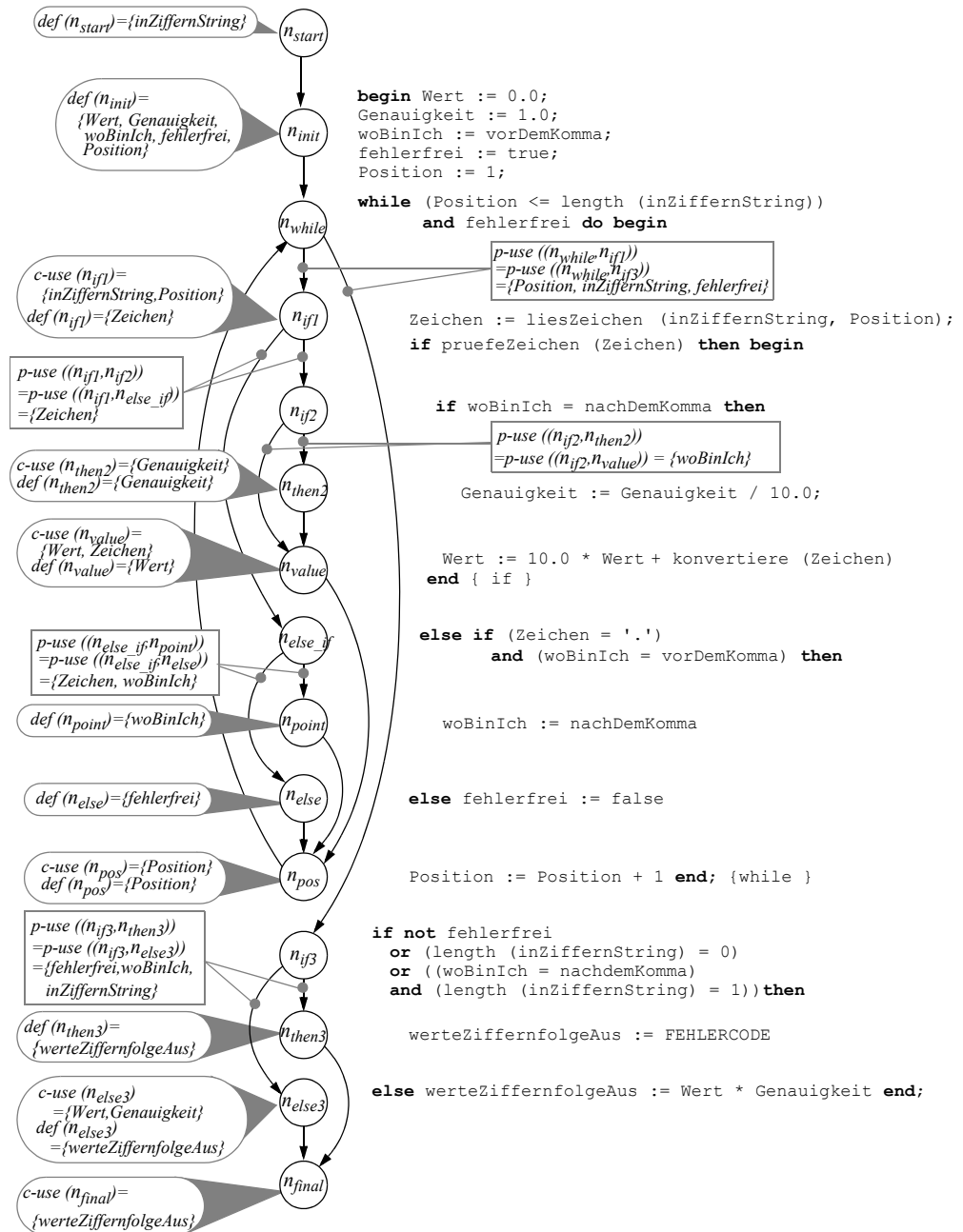


Abbildung 9.8 : Der Kontrollflussgraph der Funktion `werteZiffernfolgeAus` in Datenflussdarstellung.

### Beispiel 9.2.1.2

Tabelle 9.2 zeigt die Abbildungen *def* und *c-use* und Tabelle 9.3 die Abbildung *p-use* für die Funktion *werteZiffernfolgeAus*.

Knoten $n_i$	$def(n_i)$	$c-use(n_i)$
$n_{start}$	{inZiffernString}	----
$n_{init}$	{Wert, Genauigkeit, woBinIch, fehlerfrei, Position}	$\emptyset$
$n_{while}$	$\emptyset$	$\emptyset$
$n_{if1}$	{Zeichen}	{inZiffernString, Position}
$n_{if2}$	$\emptyset$	$\emptyset$
$n_{then2}$	{Genauigkeit}	{Genauigkeit}
$n_{value}$	{Wert}	{Wert, Zeichen}
$n_{else\_if}$	$\emptyset$	$\emptyset$
$n_{point}$	{woBinIch}	$\emptyset$
$n_{else}$	{fehlerfrei}	$\emptyset$
$n_{pos}$	{Position}	{Position}
$n_{if3}$	$\emptyset$	$\emptyset$
$n_{then3}$	{werteZiffernfolgeAus}	$\emptyset$
$n_{else3}$	{werteZiffernfolgeAus}	{Wert, Genauigkeit}
$n_{final}$	----	{werteZiffernfolgeAus}

Tabelle 9.2: Die *def*s und *c-uses* der Knoten der Funktion *werteZiffernfolgeAus*.

Kanten $(n_i, n_j)$	$p-use((n_i, n_j))$
$(n_{while}, n_{if1})$ und $(n_{while}, n_{if3})$	{Position, inZiffernString, fehlerfrei}
$(n_{if1}, n_{if2})$ und $(n_{if1}, n_{else\_if})$	{Zeichen}
$(n_{if2}, n_{then2})$ und $(n_{if2}, n_{value})$	{woBinIch}
$(n_{else\_if}, n_{point})$ und $(n_{else\_if}, n_{else})$	{Zeichen, woBinIch}
$(n_{if3}, n_{then3})$ und $(n_{if3}, n_{else3})$	{fehlerfrei, woBinIch, inZiffernString}

Tabelle 9.3: Die *p-uses* der Kanten der Funktion *werteZiffernfolgeAus*.



all uses-Kriterium

### 9.2.2 Der Test nach dem *all uses*-Kriterium

Wichtig für eine Analyse des Datenflusses ist es, zu wissen, wo die Definition einer Variablen benutzt wird. Die Definition einer Variablen  $x$  in einem Knoten  $n_i$  *erreicht* eine berechnende Benutzung von  $x$  in  $n_j$ , wenn ein Pfad von  $n_i$  nach  $n_j$  im Graphen existiert, der keine weitere Definition von  $x$  enthält. Analoges gilt für prädikative Benutzungen. Das *all uses-Kriterium* fordert nun für jede Definition einer Variablen den vollständigen Test der von ihr erreichbaren prädikativen und berechnenden Variablenzugriffe.

Wir präzisieren zunächst für einen Knoten  $n_i$  des Kontrollflussgraphen und eine Variable  $v$ , die in  $n_i$  global definiert wird, die Menge  $du(v, n_i)$ :

- $du(v, n_i)$  enthält alle Knoten  $n_j$  mit  $v \in c\text{-use}(n_j)$  und
- alle Kanten  $(n_k, n_l)$  mit  $v \in p\text{-use}((n_k, n_l))$ ,
- wobei ein Teilpfad von  $n_i$  nach  $n_j$  bzw. ein  $(n_k, n_l)$  enthaltener Teilpfad von  $n_i$  nach  $n_l$  existiert, auf dem  $v$  nicht (neu) definiert wird.

Ein dynamischer Test nach dem *all uses*-Kriterium bedeutet dann, dass für jeden Knoten  $n_i$  und jede Variable  $v \in \text{def}(n_i)$  mindestens ein definitionsfreier (Teil-)Pfad von  $n_i$  zu allen Elementen in  $du(v, n_i)$  ausgeführt wird. (Unter einem (Teil-)Pfad von  $n_i$  zu einer Kante  $(n_k, n_l) \in du(v, n_i)$  verstehen wir den Teilpfad von  $n_i$  nach  $n_l$ .)

#### Beispiel 9.2.2.1

Exemplarisch geben wir für einige Variablen und Knoten die  $du$ -Mengen an:

$$\begin{aligned}
 du(\text{inZiffernString}, n_{\text{start}}) &= \{n_{\text{if1}}, (n_{\text{while}}, n_{\text{if1}}), (n_{\text{while}}, n_{\text{if3}}), \\
 &\quad (n_{\text{if3}}, n_{\text{then3}}), (n_{\text{if3}}, n_{\text{else3}})\} \\
 du(\text{Wert}, n_{\text{init}}) &= \{n_{\text{value}}, n_{\text{else3}}\} \\
 du(\text{Zeichen}, n_{\text{if1}}) &= \{n_{\text{value}}, (n_{\text{if1}}, n_{\text{if2}}), (n_{\text{if1}}, n_{\text{else\_if}}), \\
 &\quad (n_{\text{else\_if}}, n_{\text{point}}), (n_{\text{else\_if}}, n_{\text{else}})\} \\
 du(\text{woBinIch}, n_{\text{point}}) &= \{(n_{\text{if2}}, n_{\text{then2}}), (n_{\text{if2}}, n_{\text{value}}), \\
 &\quad (n_{\text{else\_if}}, n_{\text{point}}), (n_{\text{else\_if}}, n_{\text{else}}), \\
 &\quad (n_{\text{if3}}, n_{\text{then3}}), (n_{\text{if3}}, n_{\text{else3}})\}
 \end{aligned}$$

Mit Hilfe der  $du$ -Mengen formulieren wir nun die Testfälle, indem wir zu jedem Element einer  $du$ -Menge einen definitionsfreien Teilpfad von dem Definitionsknoten  $n_i$  zu der Benutzung der Variablen  $v$  bestimmen. Für obige Beispielmengen ergeben sich folgende Teilpfade:

$$\begin{aligned}
 n_{\text{start}} &\leftarrow & n_{\text{if1}}: & (n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if1}}) \\
 & & (n_{\text{while}}, n_{\text{if1}}): & (n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if1}}) \\
 & & (n_{\text{while}}, n_{\text{if3}}): & (n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if3}}) \\
 & & (n_{\text{if3}}, n_{\text{then3}}): & (n_{\text{start}}, n_{\text{init}}, n_{\text{while}}, n_{\text{if3}}, n_{\text{then3}})
 \end{aligned}$$

$$\begin{aligned}
 (n_{if3}, n_{else3}): & (n_{start}, n_{init}, n_{while}, n_{if1}, n_{if2}, n_{value}, n_{pos}, n_{while}, \\
 & n_{if3}, n_{else3}) \\
 n_{init} \leftarrow & \quad n_{value}: (n_{init}, n_{while}, n_{if1}, n_{if2}, n_{value}) \\
 & \quad n_{else3}: (n_{init}, n_{while}, n_{if3}, n_{else3}) \\
 n_{if1} \leftarrow & \quad n_{value}: (n_{if1}, n_{if2}, n_{value}) \\
 & \quad (n_{if1}, n_{if2}): (n_{if1}, n_{if2}) \\
 & \quad (n_{if1}, n_{else\_if}): (n_{if1}, n_{else\_if}) \\
 & \quad (n_{else\_if}, n_{point}): (n_{if1}, n_{else\_if}, n_{point}) \\
 & \quad (n_{else\_if}, n_{else}): (n_{if1}, n_{else\_if}, n_{else}) \\
 n_{point} \leftarrow & \quad (n_{if2}, n_{then2}): (n_{point}, n_{pos}, n_{while}, n_{if1}, n_{if2}, n_{then2}) \\
 & \quad (n_{if2}, n_{value}): (n_{point}, n_{pos}, n_{while}, n_{if1}, n_{if2}, n_{value}) \\
 & \quad (n_{else\_if}, n_{point}): (n_{point}, n_{pos}, n_{while}, n_{if1}, n_{else\_if}, n_{point}) \\
 & \quad (n_{else\_if}, n_{else}): (n_{point}, n_{pos}, n_{while}, n_{if1}, n_{else\_if}, n_{else}) \\
 & \quad (n_{if3}, n_{then3}): (n_{point}, n_{pos}, n_{while}, n_{if1}, n_{else\_if}, n_{else}, n_{pos}, \\
 & \quad n_{while}, n_{if3}, n_{then3}) \\
 & \quad (n_{if3}, n_{else3}): (n_{point}, n_{pos}, n_{while}, n_{if3}, n_{else3})
 \end{aligned}$$

Die Testdaten sind nun so zu wählen, dass jeder Teilpfad mindestens einmal durchlaufen wird. Dabei darf ein Testdatum mehrere Teilpfade abdecken. Es ist durchaus nicht ungewöhnlich, wenn zu einigen Teilpfaden — wie z.B. dem Pfad  $n_{init} \leftarrow n_{else3}$  — aufgrund der Programmlogik keine Testdaten existieren.

Die Eingabe der Zeichenfolgen ' . ' , ' . . ' , ' z ' , ' ' , ' 9 ' , ' . 9 ' , ' 9 . ' stellt eine von vielen Möglichkeiten dar, eine vollständige Überdeckung zu erreichen. Auf eine detaillierte Herleitung der Testdaten verzichten wir an dieser Stelle und empfehlen sie als Übungsaufgabe.



Mit dem *all uses*-Test können sowohl Kontrollflussfehler als auch Berechnungsfehler identifiziert werden. Durch die Zuordnung von prädikativen Benutzungen zu den Kanten des Kontrollflussgraphen umfasst der *all uses*-Test den Zweigüberdeckungstest. Die Betrachtung der berechnenden Benutzungen kontrolliert die Auswertung und Modifikation von Daten in arithmetischen Ausdrücken. Analoges gilt für prädikative Benutzungen und boolesche Ausdrücke. Empirischen Untersuchungen zufolge werden mit Hilfe des *all uses*-Kriteriums etwa 70% aller Programmfehler gefunden. Wie kontrollflussorientierte Verfahren berücksichtigen auch datenflussorientierte Verfahren die Spezifikation des zu untersuchenden Programms nur zur Definition der Testdaten. Ein Programm, das erfolgreich kontroll- und datenflussorientierte Tests bestanden hat, kann dabei immer noch funktionale Fehler besitzen und z.B. eine bestimmte geforderte Funktionalität überhaupt nicht bereitstellen.

Diskussion des  
all uses-Kriteriums

Aufgabe 9.2.2.2

Wir greifen die Multiplikation zweier positiver, ganzer Zahlen nochmals auf und geben als Lösung das Programm `Multiplikation3` an:

```
program Multiplikation3 (input, output);
{ Multiplikation durch fortgesetzte Addition }

type
tNatZahlPlus = 1..maxint;

var
i,
Produkt,
Faktor1,
Faktor2 : tNatZahlPlus;

begin
  write ('Bitte den ersten Faktor (groesser Null) ',
        'eingeben: ');
  readln (Faktor1);
  write ('Bitte den zweiten Faktor (groesser Null) ',
        'eingeben: ');
  readln (Faktor2);
  Produkt := Faktor1;
  i := Faktor2;
  while i > 1 do
    begin
      Produkt := Produkt + Faktor1;
      i := i - 1
    end;
  writeln ('Produkt: ', Produkt)
end. { Multiplikation3 }
```

- a) Geben Sie zum Programm `Multiplikation3` den Kontrollflussgraphen in Datenflussdarstellung an.
- b) Geben Sie die *du*-Mengen an.
- c) Ermitteln Sie die Testfälle für einen Test nach dem *all uses*-Kriterium. Bestimmen Sie dazu zunächst die zu den *du*-Mengen gehörenden Teilpfade im Kontrollflussgraphen. Geben Sie ausführbare Pfade an, in denen diese Teilpfade enthalten sind, und leiten Sie daraus die assoziierten Testfälle ab. Wählen Sie für jeden Testfall ein Eingabedatum aus und halten Sie das Ergebnis des Tests fest.

## 10. Black-Box-Testverfahren

Nachdem wir verschiedene White-Box-Testverfahren ausführlich dargestellt haben, widmen wir uns nun den *Black-Box-Testverfahren*. Diese ignorieren jegliche Informationen über die interne Struktur des Prüflings und bilden Testfälle lediglich auf der Basis der Problemspezifikation. Obwohl somit einem Black-Box-Verfahren weniger Informationen für die Bildung der Testfälle zur Verfügung stehen, können Black-Box-Tests aufgrund einer anderen Systematik Fehler aufdecken, die bei einem White-Box-Tests möglicherweise unerkannt bleiben. Hierzu zunächst ein Beispiel:

### Beispiel 10.0.1

Wir betrachten das Problem, zu entscheiden, ob eine ganze Zahl  $t$  eine ganze Zahl  $g$  teilt oder nicht. Die Problemspezifikation notieren wir wie folgt:

Eingabe:  $g, t \in \mathbb{Z}$

Ausgabe: true, falls  $t \mid g$ , andernfalls false

Dabei gelte  $t \mid g$ , genau dann wenn  $g$  ein Vielfaches von  $t$  ist:

$t \mid g \Leftrightarrow$  Es gibt ein  $x \in \mathbb{Z}$  mit  $g = x \cdot t$

Zur Lösung des Problems bieten wir folgende Funktion `istTeiler` an:

```
function istTeiler (inDividend,
                    inDivisor: integer) : boolean;
{ antwortet mit true, falls inDivisor ein Teiler von
  inDividend ist, andernfalls mit false. }

begin
  if (inDividend mod inDivisor) = 0 then
    istTeiler := true
  else
    istTeiler := false
end; { istTeiler }
```

Für unser Beispiel liefern die Kriterien für Zweig- und Pfadüberdeckungstest die gleichen Testfälle:

$$T_1 = \{ (g, t, \text{true}) \mid g, t \in \mathbb{Z}, g \bmod t = 0 \}$$

$$T_2 = \{ (g, t, \text{false}) \mid g, t \in \mathbb{Z}, g \bmod t \neq 0 \}$$

Wir wählen aus dem ersten Testfall das Testdatum (4, 2, true) und aus dem zweiten Testfall das Testdatum (4, 3, false). Die tatsächlichen Ausgaben entsprechen dabei jeweils dem erwarteten Ergebnis, die beiden kontrollflussorientierten Testverfahren decken also keinen Fehler auf.

Dennoch ist die Pascal-Funktion nicht korrekt. Wir haben nämlich einen Fall außer Acht gelassen: Ist der Divisor  $t = 0$ , so ist der Ausdruck

Black-Box-Testver-  
fahren

„Dividend **mod** Divisor“ nicht definiert, da eine Division durch Null nicht definiert ist. Das Programm `Teiler` bricht mit einem Laufzeitfehler ab, obwohl laut Spezifikation die Eingabe zulässig ist und ein Boolean-Ergebnis erwartet wird. Dieser Fehler kann mit keinen Testdaten aus unseren mit kontrollflussorientierten Verfahren ermittelten Testfällen aufgezeigt werden, denn Testdaten mit der Eingabe  $t = 0$  gehören weder zum Testfall  $T_1$  noch zu  $T_2$ .

□

Das Beispiel zeigt eine Schwäche von White-Box-Verfahren auf, die ja ihre Testfälle in erster Linie auf Basis der inneren Struktur des Prüflings bilden und dabei die Spezifikation mitunter unzureichend berücksichtigen: Die Spezifikation im Beispiel ist korrekt, der Prüfling jedoch ist fehlerhaft, da er den Sonderfall der Division durch Null nicht wie spezifiziert behandelt. Die Testfälle wurden auf Basis der fehlerhaften Implementierung ermittelt, mit der Konsequenz, dass kein Testfall gebildet wurde, der den Sonderfall  $t = 0$  abdeckt. Die Testfälle enthalten ausschließlich Testdaten für den Normalfall, in dem der Prüfling auch tatsächlich korrekt funktioniert. Die Spezifikation wird durch solche White-Box-Testverfahren lediglich zur Einschränkung der Eingabedaten der Testfälle auf laut Spezifikation zulässige Werte (im Beispiel auf ganze Zahlen) sowie zur Ermittlung der erwarteten Ausgaben herangezogen. Zum Vergleich werden wir in Beispiel 10.1.1 denselben Prüfling einem Black-Box-Test unterziehen, der den Fehler im Prüfling aufdeckt.

Black-Box-Tests sind besonders wirkungsvoll für Programme, die präzise spezifiziert sind. Nur aus exakten Spezifikationen sind Black-Box-Testfälle und Testdaten eindeutig und vollständig ableitbar. Weniger präzise Spezifikationen besitzen ein gewisses Maß an Interpretierbarkeit, was ein unterschiedliches Verständnis der Spezifikation durch den Autor der Spezifikation, Programmierer und Tester nach sich ziehen und damit zu Problemen führen kann. Das ist natürlich auch schon bei der Testfallbildung zu White-Box-Tests ein Problem, beim Black-Box-Test ist die Problematik aufgrund der zentralen Rolle der Spezifikation aber potentiell größer. Es sei übrigens – egal ob bei White-Box- oder Black-Box-Tests – davon abgeraten, zur Klärung von Unklarheiten bezüglich der Spezifikation einen Blick in die Implementierung zu werfen, denn die könnte fehlerhaft sein – sie soll ja gerade getestet werden.

funktionsorientierte  
Verfahren

Eine wichtige Klasse der Black-Box-Testverfahren sind *funktionsorientierte Testverfahren*. Diese überprüfen gezielt die vom Prüfling laut Spezifikation zu erbringende Funktionalität. Die Grundidee funktionsorientierter Testverfahren ist dabei naheliegend: Die Spezifikation legt genau fest, was der Prüfling leisten soll, und der funktionsorientierte Test überprüft nun die Funktionalität gemäß dieser Spezifikation. Wie auch bei den White-Box-Verfahren werden systematisch Testfälle gebildet, aus denen Testdaten als Repräsentanten für die Testläufe herangezogen werden, nur erfolgt beim funktionsorientierten Test auch die Konstruktion der Testfälle ausschließlich auf Grundlage der Spezifikation, der Quelltext des Prüflings wird überhaupt nicht berücksichtigt.

Als einen wichtigen Vertreter funktionsorientierter Verfahren betrachten wir im folgenden Abschnitt die funktionale Äquivalenzklassenbildung. Im Anschluss behan-



deln wir mit dem *Zufallstest*, dem *Test spezieller Werte* und der *Grenzwertanalyse* drei klassische nicht funktionsorientierte Black-Box-Testverfahren. Die in diesen Testverfahren verwendeten *Auswahlstrategien* für Testdaten lassen sich vielfach auch in anderen Testverfahren einsetzen, insbesondere in Kombination mit der funktionalen Äquivalenzklassenbildung.

Auswahlstrategie

## 10.1 Funktionsorientierter Test: Funktionale Äquivalenzklassen

Das Ziel funktionsorientierter Testverfahren, und damit insbesondere der funktionalen Äquivalenzklassenbildung, ist wieder die Ermittlung von Testfällen, also Mengen von Testdaten, unter deren Eingaben sich der Prüfling bezüglich gewisser Kriterien gleichartig verhält. Bei den strukturorientierten Verfahren war ein solches Kriterium meist das Durchlaufen desselben Pfades durch den Kontrollflussgraphen. Bei funktionsorientierten Tests dagegen werden solche Kriterien aus der Spezifikation abgeleitet.

In Beispiel 8.1.4 hatten wir ad hoc zwei Testfälle unmittelbar aus der Fallunterscheidung in der Spezifikation der Fakultätsfunktion abgeleitet. In anderen Worten haben wir die Eingabedaten in zwei Klassen aufgeteilt, für die wir ein unterschiedliches Verhalten erwarten. Dieser Ansatz findet sich in der Bildung *funktionaler Eingabeäquivalenzklassen* wieder: Die Menge aller laut Spezifikation zulässigen Eingaben wird hierbei in Äquivalenzklassen *zerlegt*<sup>1</sup>, wobei alle Eingaben aus einer Klasse nach der Spezifikation ein gleichartiges funktionales Verhalten auslösen. Anschaulich testet man also mit allen Eingaben aus einer Klasse dieselbe Teilfunktionalität des Prüflings. Eine erste Zerlegung kann sich, wie in Beispiel 8.1.4, an expliziten Fallunterscheidungen in der Spezifikation orientieren. Eine solche Zerlegung ist allerdings oft noch zu grob und muss noch nach problemabhängigen Kriterien weiter verfeinert werden, indem einzelne Klassen weiter zerlegt werden, so dass man im Endeffekt eine hierarchische Zerlegung erhält.

funktionale Eingabe-  
äquivalenzklassen

Eingabeäquivalenzklassen lassen sich zu kompletten *Testfällen* (gemäß Definition 8.1.3) erweitern, indem die Eingabedaten noch jeweils um die laut Spezifikation erwartete Ausgabe ergänzt werden. Wird das Programm mit Testdaten aus jeder Äquivalenzklasse (*Repräsentanten*) ausgeführt, so ist gewährleistet, dass alle spezifizierten Programmfunktionen getestet werden.

### Beispiel 10.1.1

Betrachten wir nochmals Beispiel 10.0.1. Das Problem besteht darin, zur Eingabe zweier ganzer Zahlen  $g$  und  $t$  zu prüfen, ob  $t$  ein Teiler von  $g$  ist. Ergebnis sei ein Boolean-Wert.

Zu dieser informalen Problemspezifikation versuchen wir nun, Testfälle mit Hilfe funktionaler Eingabeäquivalenzklassen zu bestimmen. Die Menge der zulässigen Eingaben, die wir in Äquivalenzklassen zerlegen müssen, ist die Menge aller Paare  $(g, t)$  ganzer Zahlen. Ein nach der Problemspezifikation offensichtliches Kriterium

1. d.h. die Äquivalenzklassen sind paarweise disjunkt, und ihre Vereinigung ergibt genau die Menge der zulässigen der Eingaben. Oder anders ausgedrückt: Jede mögliche Eingabe ist Element genau einer Eingabeäquivalenzklasse. (Vgl. „Zerlegung“ von Mengen)

zur Klassifikation von solchen Eingabepaaren ist die Teiler-Eigenschaft. Damit können wir eine erste Zerlegung in folgende zwei Eingabeäquivalenzklassen vornehmen:

Klasse 1: Alle Paare  $(g, t)$  mit  $t \mid g$

Klasse 2: Alle Paare  $(g, t)$  mit  $\neg (t \mid g)$ <sup>1</sup>

Aus dieser ersten Klassifikation können wir nur zwei grobe Testfälle direkt ableiten, das ist zu wenig. Daher betrachten wir die Relation „ist Teiler von“ etwas genauer. Offensichtlich spielt die Null als Divisor eine besondere Rolle, denn Division durch Null ist nicht erlaubt. Wie ist also die Frage „ $0 \mid g$ “ zu beantworten? Das wird in der Problemspezifikation genau definiert<sup>2</sup>:  $0 \mid g$  gilt genau dann, wenn  $g$  ein Vielfaches von 0 ist. Da Null das einzige Vielfache von Null ist, gilt  $0 \mid g$  genau für  $g = 0$ .

Neben der Null spielen weiterhin die sog. trivialen Teiler eine Sonderrolle: Jede ganze Zahl  $g$  ist durch 1, -1,  $g$  sowie  $-g$  teilbar. (Zahlen, die keine weiteren Teiler als die trivialen Teiler haben, heißen Primzahlen.)

Nach diesen Überlegungen können wir die Klasse 1 weiter zerlegen in drei Unterklassen:

Klasse 1.1: Alle Paare  $(g, t)$  mit  $t \mid g$  und  $t = 0$ ,  
also das Paar  $(0, 0)$

Klasse 1.2: Alle Paare  $(g, t)$  mit  $t \mid g$  und  $t \in \{-1, 1, -g, g\}$ ,  
das sind tatsächlich alle Paare  $(g, t)$  mit  $t \in \{-1, 1, -g, g\}$

Klasse 1.3: Alle Paare  $(g, t)$  mit  $t \mid g$  und  $t \notin \{0, -1, 1, -g, g\}$ ,  
das sind alle restlichen Paare aus Klasse 1

Klasse 2 lässt sich nach analogen Überlegungen ebenfalls weiter zerlegen. Eine Unterklasse für triviale Teiler lässt sich in dieser Klasse natürlich nicht bilden, aber der Fall  $t = 0$  kann auch in Klasse 2 eintreten:

Klasse 2.1: Alle Paare  $(g, t)$  mit  $\neg (t \mid g)$  und  $t = 0$ ,  
also alle Paare  $(g, 0)$  mit  $g \neq 0$

Klasse 2.2: Alle Paare  $(g, t)$  mit  $\neg (t \mid g)$  und  $t \neq 0$ ,  
das sind alle restlichen Paare aus Klasse 2

Zu jeder dieser fünf Eingabeäquivalenzklassen erhalten wir einen Testfall, indem wir die Eingabedaten noch um die jeweilige erwartete Ausgabe ergänzen:

- 
1. „ $\neg$ “ ist der logische Negationsoperator, „ $\neg (t \mid g)$ “ heißt also „ $t$  ist *nicht* Teiler von  $g$ “.
  2. Wenn die Definition für  $0 \mid g$  in der Spezifikation fehlte, so läge eine unpräzise Problemspezifikation vor. Denn es gibt auch eine alternative Definition der Teilbarkeit, nach der  $0 \mid g$  undefiniert ist, da die Division durch Null nicht erlaubt ist. Diese Unklarheit der Spezifikation wäre ebenfalls beim White-Box-Test aus Beispiel 10.0.1 unentdeckt geblieben, bei der hier vorgenommenen Black-Box-Betrachtung wäre sie dagegen aufgefallen.

$$\begin{aligned}
T_{1.1} &= \{ (0, 0, \text{true}) \} \\
T_{1.2} &= \{ (g, t, \text{true}) \mid t \in \{-1, 1, -g, g\} \} \\
T_{1.3} &= \{ (g, t, \text{true}) \mid (t \mid g) \text{ und } t \notin \{0, -1, 1, -g, g\} \} \\
T_{2.1} &= \{ (g, 0, \text{false}) \mid g \neq 0 \} \\
T_{2.2} &= \{ (g, t, \text{false}) \mid \neg(t \mid g) \text{ und } t \neq 0 \}
\end{aligned}$$

Als Fazit können wir festhalten, dass wir bei diesem funktionsorientierten Test dank der genaueren Betrachtung der Spezifikation den Fehler im Prüfling aufdecken, der in Beispiel 10.0.1 mit dem kontrollflussorientierten Test nicht gefunden wurde.



Analog zur Zerlegung der Menge der (laut Spezifikation) zulässigen Eingaben in Eingabeäquivalenzklassen lässt sich auch die Menge der (laut Spezifikation) zulässigen Ausgaben in *funktionale Ausgabeäquivalenzklassen* zerlegen<sup>1</sup>. Eine Ausgabeäquivalenzklasse ist dabei so definiert, dass jede mögliche Ausgabe derselben Äquivalenzklasse auch durch dieselbe Teilfunktionalität des Prüflings erzeugt wird. Eine erste Zerlegung der Ausgaben lässt sich mitunter direkt aus Fallunterscheidungen der Spezifikation ablesen.

funktionale Ausgabeäquivalenzklassen

Um zu einer Ausgabeäquivalenzklasse den korrespondierenden Testfall zu bilden, müssen noch genau alle möglichen Eingabedaten bestimmt werden, die (laut Spezifikation) zu den in der Äquivalenzklasse enthaltenen Ausgaben führen.

### Beispiel 10.1.2

Abschließend untersuchen wir nun noch unser begleitendes Fallbeispiel (vgl. Beispiel 9.1.1.3), indem wir Eingabe- und Ausgabeäquivalenzklassen für die Funktion `werteZiffernfolgeAus` bestimmen. Aus der Beschreibung der Funktion ergeben sich zunächst unmittelbar zwei *Eingabeäquivalenzklassen*:

- Klasse E1: *Alle syntaktisch korrekten Ziffernfolgen.*  
 Jede syntaktisch korrekte Ziffernfolge wird von der Funktion als `real`-Zahl interpretiert und ausgegeben.
- Klasse E2: *Alle nicht syntaktisch korrekten Ziffernfolgen.*  
 Diese führen zu einer Fehlermeldung.

Die so gewonnene Aufteilung ist viel zu grob. Die inhomogene Struktur der Klasse 1 erlaubt jedoch eine naheliegende weitere Zerlegung.

Es entstehen die Unterklassen (ZF bezeichnet dabei eine nichtleere Ziffernfolge, in der kein Punkt vorkommt):

---

1. Für obiges Beispiel 10.1.1 ist das aufgrund der nur zweielementigen Menge möglicher Ausgaben natürlich nicht sinnvoll, denn die ließe sich in maximal zwei Klassen (`true` und `false`) zerlegen und würde nur zwei triviale Testfälle liefern. Die Bildung von Ausgabeäquivalenzklassen eignet sich besser für komplexere Ausgabebereiche.

Klasse E1.1: *Klasse der ZF*

Klasse E1.2: *Klasse der ZF ‘.’*

Klasse E1.3: *Klasse der ‘.’ ZF*

Klasse E1.4: *Klasse der ZF ‘.’ ZF*

Die besondere Bedeutung des Punktes lässt einen Test der Eingaben ‘.’ und ‘..’ sowie Kombinationen dieser beiden Werte mit zulässigen Werten aus der Klasse 1 sinnvoll (wenn auch teilweise willkürlich) erscheinen:

Klasse E2.1: *Klasse der ZF ‘..’*

Klasse E2.2: *Klasse der ‘..’ ZF*

Klasse E2.3: *Klasse der ZF ‘..’ ZF*

Klasse E2.4: *Klasse der ‘.’ ZF ‘.’*

Klasse E2.5: *Klasse der ZF ‘.’ ZF ‘.’*

Klasse E2.6: *alle übrigen nicht zulässigen Zeichenfolgen.*

Wir bilden zu diesen zehn Klassen jeweils einen Testfall, aus dem Testdaten zu wählen sind. Die Testfälle beispielsweise zu den Klassen E1.1 und E1.2 sind:

$$T_{E1.1} = \{ (z, \text{konvertiere}(z)) \mid z \text{ ist eine beliebige Ziffernfolge} \}$$

$$T_{E1.2} = \{ (z + ' . ', \text{konvertiere}(z)) \mid z \text{ ist eine beliebige Ziffernfolge} \}$$

Als *Ausgabeäquivalenzklassen* der Funktion `werteZiffernfolgeAus` erhalten wir unmittelbar:

Klasse A1: *Die Funktion liefert eine positive real-Zahl zurück.*

Klasse A2: *Ein Fehlercode wird ausgegeben.*

Klasse A1 kann z.B. in der Annahme, dass ganze Zahlen anders berechnet werden als die übrigen `real`-Zahlen, weiter aufgetrennt werden:

Klasse A1.1: *Die Funktion liefert eine positive ganze real-Zahl zurück.*

Klasse A1.2: *Die Funktion liefert eine positive real-Zahl zurück, die keine ganze Zahl ist.*

Um aus den Ausgabeäquivalenzklassen Testfälle abzuleiten, sind die Mengen genau derjenigen Eingaben zu bestimmen, die zu einer in der Ausgabeäquivalenzklasse enthaltenen Ausgabe führen. Zu Ausgaben der Klasse A1.1 führen z.B. alle Eingaben von Ziffernfolgen, optional mit abschließendem Punkt. Der Testfall zu Klasse A1.1 ist demnach genau die Vereinigung der Testfälle zu den Klassen E1.1 und E1.2:

$$T_{A1.1} = T_{E1.1} \cup T_{E1.2}$$

Den Testfall zu Klasse A1.2 bildet entsprechend die Vereinigung der Testfälle zu den Klassen E1.3 und E1.4, und der Testfall zu A2 ist die Vereinigung der Testfälle zu den Klassen E2.1 bis E2.6.

In diesem Beispiel liefern also die Eingabeäquivalenzklassen feiner aufgeteilte Testfälle als die Ausgabeäquivalenzklassen (jeder Testfall zu einer Eingabeäquivalenzklasse ist eine Teilmenge eines der Testfälle zu einer Ausgabeäquivalenzklasse). Daher genügt es offensichtlich, zur Wahl der Testdaten für die durchzuführenden Testläufe allein die aus den Eingabeäquivalenzklassen gebildeten Testfälle heranzuziehen. Das muss aber nicht immer so sein.



### Aufgabe 10.1.3

Gegeben seien die folgenden Typdeklarationen:

```
type
tPunkt = record
    X,
    Y : real
end;

tKreis = record
    Mitte : tPunkt;
    Radius : real
end;
```

und die Funktion PunktAbstand:

```
function PunktAbstand (A, B : tPunkt) : real;
{ berechnet den Abstand zweier Punkte in der Ebene }

begin
    PunktAbstand := sqrt (sqr (A.X - B.X) +
                           sqr (A.Y - B.Y))
end; { PunktAbstand }
```

Betrachten Sie nun die Funktion KreisAbstand, die den Abstand zweier Kreise in der Ebene berechnet. Den Abstand zweier Kreise definieren wir als den kleinsten Abstand zwischen einem beliebigen Punkt auf dem Umfang des ersten Kreises und einem beliebigen Punkt auf dem Umfang des zweiten Kreises.

```
function KreisAbstand (A, B : tKreis) : real;
{ berechnet den Abstand zweier Kreise in der Ebene }

var
    hilf : real;
```

```

begin
  hilf := PunktAbstand (A.Mitte, B.Mitte);
  if hilf < (A.Radius + B.Radius) then
    KreisAbstand := 0.0
  else
    KreisAbstand := hilf - (A.Radius + B.Radius)
  end; { KreisAbstand }

```

Ermitteln Sie für einen funktionsorientierten Test der Funktion `KreisAbstand` Testdaten anhand von Eingabeäquivalenzklassen.



## 10.2 Fehlerorientierter Test: Test spezieller Werte

In der Praxis ist häufig zu beobachten, dass bestimmte in früheren Prüflingen häufig aufgetretene Fehler mit erhöhter Wahrscheinlichkeit in dem aktuellen Prüfling bei ähnlichen Eingabekonstellationen wieder vorkommen. *Fehlerorientierte Testverfahren* machen sich diese Beobachtung zu Nutze, indem sie eine Auswahl von Testdaten basierend auf einer konkreten Erwartungshaltung bezüglich des Eintretens bestimmter Fehler treffen. Der *Test spezieller Werte* ist ein Oberbegriff für verschiedene fehlerorientierte Black-Box-Testverfahren. Ein Test spezieller Werte wird oft im Verbund mit anderen Testverfahren eingesetzt, indem z.B. zunächst die Äquivalenzklassen für ein Programm bestimmt und aus diesen dann spezielle Werte gewählt werden.

Einen ärgerlichen Laufzeitfehler, den fast jeder Programmierer kennt, verursacht die Division durch 0 (vgl. auch Beispiel 10.0.1). Aus ihm leitet sich das *zero values-Kriterium* ab, das vorschreibt, alle Variablen, die in arithmetischen Ausdrücken verwendet werden, einmal mit dem speziellen Wert 0 zu besetzen (soweit dies nach der Spezifikation zulässig ist).

Ähnlich kritische spezielle Werte sind **nil**-Zeiger in dynamischen Datenstrukturen.

### Beispiel 10.2.1

Spezielle Eingabewerte der Funktion `werteZiffernfolgeAus` (vgl. Beispiel 9.1.1.3) sind die leere Ziffernfolge sowie die Zeichenketten `'.'`, `'0'`, `'0.'` und `'0.0'`.



Ein wichtiges Verfahren zum Ermitteln spezieller zu testender Werte ist weiterhin die *Grenzwertanalyse*. Programme sind erfahrungsgemäß gegen Eingabewerte aus Grenzbereichen der zulässigen Eingaben besonders fehleranfällig. Die Grenzwertanalyse ist besonders für die Auswahl von Testdaten aus (Testfällen zu) funktionalen Äquivalenzklassen geeignet, da hier die Grenzen der Klassen meist klar definiert sind. Das Verfahren sieht nun zum einen den Test der Grenzen selbst vor, konzentriert sich aber auch auf Eingabewerte, die geringfügig „neben“ den Grenzen

fehlerorientierte  
Testverfahren

Test spezieller Werte

zero values-  
Kriterium

Grenzwertanalyse

liegen. Ein generelles Konzept für die Wahl von Testdaten kann jedoch nicht gegeben werden, da diese stark vom jeweiligen Problem abhängen.

### Beispiel 10.2.2

Wir untersuchen die Ausgabeäquivalenzklasse A1 der Funktion `werteZiffernfolgeAus` (vgl. Beispiel 10.1.2) auf ihre Grenzwerte:

Die Klasse der positiven `real`-Zahlen besitzt nur zwei zu überprüfende Grenzen: Die Zahl 0.0 und eine (implementierungsabhängige) sehr große Zahl wie z.B. bei Turbo-Pascal  $1.7 * 10^{38}$ . Für die übrigen positiven `real`-Zahlen wählen wir mit  $2.9 * 10^{-39}$  einen sehr kleinen Wert und mit  $1.7 * 10^{38} - 2.9 * 10^{-39}$  einen sehr großen Wert. Zu den gewählten Ausgabewerten müssen nun noch entsprechende Werte für den Eingabeparameter `inZiffernString` bestimmt werden, um komplette Testdaten für eine Testdurchführung zu erhalten.



### Beispiel 10.2.3

Wir hatten bereits in Beispiel 8.1.4 bei der Wahl von Testdaten zur Fakultätsfunktion die obere und untere Grenze des ersten Testfalls berücksichtigt. Eine Grenzwertanalyse sollte aber auch Werte „nahe“ der Grenzen berücksichtigen, bei ganzzahligen Wertebereichen wären das insbesondere um eins größere bzw. kleinere Werte. Die Testdaten zu Testfall 1 würden wir demnach noch um ein Testdatum mit der Eingabe 11 ergänzen (da 12 die obere Grenze ist). Ein Wert oberhalb der Grenze läge hier außerhalb des möglichen Eingabereichs, muss daher nicht getestet werden. Das Testdatum (2, 2), dessen Eingabe um 1 über der unteren Grenze liegt, ist im Testfall 1 bereits enthalten.



## 10.3 Zufallstest

Beim *Zufallstest* werden aus der Menge der (laut Spezifikation) zulässigen Eingabedaten zufällig Eingaben für Testläufe ausgewählt. Der Zufallstest ist somit weder ein funktionsorientierter noch ein fehlerorientierter Black-Box-Test.

Zufallstest

Ein Vorteil der zufälligen Wahl ist der geringe Aufwand: Zufällige Eingabedaten lassen sich automatisch generieren – für die erwarteten Ergebnisse gilt dies jedoch im Allgemeinen nicht (denn dazu bräuchte man ein Programm, das zu den Eingaben die Ausgaben berechnet, also das leistet, was gerade der Prüfling leisten soll). Wenn die Ausgaben manuell validiert werden müssen und für eine gute Testabdeckung möglichst viele Testläufe stattfinden sollen, relativiert das die erste Aufwandserparnis teilweise wieder<sup>1</sup>. Ein anderer Vorteil der zufälligen Testdatenauswahl ist

---

1. Es gibt allerdings auch Probleme, bei denen es nicht notwendig ist, das erwartete Ergebnis zu bestimmen, sondern es einfacher ist, zu entscheiden, ob das tatsächliche Ergebnis korrekt ist. Ein Beispiel wäre ein Sortierverfahren: Man kann leicht prüfen, ob eine Ergebnisliste sortiert ist, ohne im Voraus das erwartete Ergebnis (die sortierte Liste) selbst herleiten zu müssen. Solche Probleme eignen sich vergleichsweise gut für einen vollautomatischen Zufallstest.

ihre „Objektivität“: Sie wird jedes mögliche Testdatum mit gleicher Wahrscheinlichkeit zum Test heranziehen, während ein Mensch möglicherweise unbewusst gewisse Eingabemöglichkeiten nicht in Betracht zieht und niemals testet. Gerade solche (möglicherweise abwegig erscheinenden) Eingabemöglichkeiten könnten auch vom Programmierer übersehen worden sein, so dass der Prüfling auf ihre Eingabe fehlerhaft reagiert.

Problematisch an der nicht-deterministischen Wahl von Eingabedaten ist dagegen die fehlende Reproduzierbarkeit. Damit solche Testläufe – z.B. nach dem Versuch, einen Fehler zu beheben – wiederholt werden können, sind also alle generierten Testdaten zu protokollieren (vgl. auch Kapitel 11).

Offensichtlich stellt der Zufallstest kein systematisches Testverfahren dar, anders als z.B. der Test spezieller Werte, die funktionale Äquivalenzklassenbildung oder die strukturorientierten Verfahren. Ein ausschließlicher Zufallstest ist also im Allgemeinen als nicht adäquat anzusehen. Insofern ist es sinnvoll, den Zufallstest mit anderen Testverfahren zu verbinden, indem man zufällig Testdaten aus zuvor mit systematischen (Black-Box- oder White-Box-)Testverfahren ermittelten Testfällen auswählt.



## 11. Regressionstest

Bisher haben wir verschiedene dynamische Testverfahren betrachtet, uns dabei jedoch immer auf die Tätigkeiten zur Vorbereitung einer erstmaligen Testausführung konzentriert, also insbesondere auf die Testfallkonstruktion und die Testdatenauswahl. Nun bleibt es aber in aller Regel nicht bei einer einmaligen Testausführung, sondern unabhängig vom eingesetzten Verfahren müssen Tests in vielen Fällen wiederholt werden.

Nur äußerst selten absolviert ein Prüfling alle Testläufe (bis zum Erreichen des Testendekriteriums) ohne Fehlverhalten, d.h. er „besteht“ den Test. Werden dagegen – und das ist der Normalfall – Fehler gefunden, schließt sich das Debuggen an, nach dem der Prüfling erneut getestet werden sollte, um zu kontrollieren, ob alle Fehler behoben wurden – und keine neuen hinzugekommen sind. Aber nicht nur für ehemals fehlerhafte Prüflinge müssen Testläufe wiederholt werden: Einerseits könnte ein Prüfling, obwohl er seine Aufgabe schon erfüllt, noch modifiziert werden (um sie z.B. effizienter zu erfüllen). Andererseits stellt ein Prüfling (z.B. eine Prozedur oder Funktion) meist nur einen Baustein einer komplexeren Software dar. Greift ein schon fehlerfrei funktionierender, unveränderter Prüfling seinerseits auf andere Bausteine zurück, so können sich auch Änderungen an den verwendeten Bausteinen auf seine korrekte Funktion auswirken. Daher sollten nach einer Softwaremodifikation nicht nur die modifizierten, sondern alle potentiell von der Modifikation betroffenen Teile erneut getestet werden. Dieses selektive (Neu-)Testen von Softwareteilen, um sicherzustellen, dass die Modifikationen keine unerwünschten Nebenwirkungen hatten, bezeichnet man als *Regressionstesten*. Der (nicht immer trivialen) Selektion der erneut zu testenden Teile kommt dabei gerade bei umfangreicheren Softwareprojekten eine große Bedeutung zu, denn nach jeder kleinen Modifikation *alle* Tests des gesamten Systems zu wiederholen, würde i.d.R. einen unangemessen hohen Aufwand verursachen.

Zur optimalen Vergleichbarkeit sollten bei einer Testwiederholung die Testläufe auch mit denselben Testdaten ausgeführt werden, die schon im vorhergehenden Test verwendet wurden. Diese sind typischerweise bereits in einem Testprotokoll (zusammen mit Soll- und Ist-Ergebnissen) festgehalten. Es ist sinnvoll, sie auch in computerlesbarer Form gespeichert zu haben, um eine Automatisierung der Testwiederholung zu ermöglichen. Eine solche Automatisierung ist wichtig, weil die Arbeit, einen immer gleichen Test unter Umständen sehr oft zu wiederholen, für Menschen eher lästig ist und insbesondere menschliche Fehler provoziert werden.

Wir hatten eingangs in Abschnitt 8.1 bereits erwähnt, dass die eigentliche Testdurchführung, wenn die Testdaten erst einmal ermittelt sind, automatisierbar ist. Beim Regressionstest müssen die Vorbereitungsarbeiten einschließlich Testfallkonstruktion und Testdatenauswahl im Optimalfall nicht wiederholt werden – es sei denn, an einem Prüfling wurden so große Modifikationen vorgenommen, dass z.B. Überdeckungskriterien wie die vollständige Zweigüberdeckung durch die alten Testdaten nicht mehr sichergestellt werden. Dann müsste man zusätzlich zu den Testdaten des letzten Testlaufs noch neue ermitteln, um das Überdeckungskriterium

wieder zu erfüllen. Die Einhaltung solcher Überdeckungskriterien für White-Box-Tests sollte daher für geänderte Prüflinge (entweder vor der Testwiederholung manuell oder durch Messung der bei der Testausführung erreichten Überdeckungsgrade) überprüft werden. Black-Box-Tests sind dagegen in dieser Hinsicht unproblematisch, da die Testfälle ohnehin unabhängig von der Realisierung des Prüflings konstruiert werden und stabil bleiben, so lange die Spezifikation nicht geändert wird.

Während bei einem ersten Testlauf ausschließlich die tatsächlichen mit den laut Spezifikation erwarteten Ausgaben verglichen werden können, finden wir bei einer Testwiederholung im Testprotokoll noch eine weitere Vergleichsgröße: die Ausgaben des vorherigen Testlaufs<sup>1</sup>. Vergleicht man diese mit den Ausgaben des wiederholten Testlaufs, so ist direkt zu erkennen, ob sich das Verhalten des Prüflings unter dem betrachteten Testdatum geändert hat. Führt man beide Vergleiche durch, kann man also z.B. die bei der Testwiederholung gefundenen Fehler nach „neu hinzugekommen“, „unverändert fehlerhaft“ und „anderes, aber immer noch fehlerhaftes Verhalten“ klassifizieren oder auch messen, wieviele Fehler behoben wurden. Wesentliche Voraussetzung dafür ist jedoch die *Reproduzierbarkeit* der Testläufe: Der Prüfling muss deterministisch arbeiten und, so lange er unverändert bleibt, in jedem Testlauf mit denselben Eingabedaten auch dieselben Ausgaben produzieren.

---

1. Für Prüflinge, die den vorherigen Test bereits bestanden hatten, stimmen die Ausgaben des letzten Testlaufs natürlich mit den erwarteten überein.

## 12. Abschließende Bemerkungen zu dynamischen Tests

Nachdem wir in den vorangehenden Kapiteln einige Vertreter von White-Box- und Black-Box-Tests behandelt haben, wollen wir diese nun in ihren Vor- und Nachteilen gegenüberstellen und ihren sinnvollen Einsatz diskutieren. Wir beginnen mit kontrollflussorientierten Tests.

Als Vorteile kontrollflussorientierter Tests halten wir fest:

- Die Konstruktion der Testfälle erfolgt im positiven Sinn schematisch, so dass keine hohen Anforderungen an konstruktive Überlegungen und Kreativität gestellt werden. Die Testvorbereitungsaufgaben lassen sich zumindest teilweise auch automatisieren, z.B. die Konstruktion des Kontrollflussgraphen oder das Ermitteln von Pfaden, die eine bestimmte Überdeckung erreichen.
- Mit ihren Überdeckungsgraden geben kontrollflussorientierte Verfahren klar definierte *Testendekriterien* vor. Diese können problemadäquat abgeschwächt werden, indem z.B. statt vollständiger Zweigüberdeckung nur Zweigüberdeckung von 90% gefordert wird. Der Testkomplettierungsgrad ist exakt messbar.
- Ein wichtiger Vorteil des kontrollflussorientierten Tests besteht darin, dass er den Tester zwingt, sich zur Testfallkonstruktion intensiv mit dem Testobjekt auseinanderzusetzen. In dem Bestreben, Testfälle zu definieren, die z.B. alle Anweisungen, Zweige oder bestimmte Pfade ausführen, stößt er unweigerlich auf logische Ungereimtheiten und Lücken im Code. So werden – wie in verschiedenen Beispielen demonstriert – so manche Fehler schon vor einem Testlauf gefunden. Anders als bei einem während des Testlaufs gefundenen Fehler wird in solchen Fällen die Fehlerursache meist gleich mit gefunden.

Als Nachteile kontrollflussorientierter Testverfahren stellen wir fest:

- Die Wirksamkeit der Verfahren ist begrenzt. Empirische Untersuchungen lassen eine Fehlererkennungsrate von bis zu 50% selbst bei hoher Überdeckungsrate eher als optimistisch erscheinen.
- Nur bestimmte Fehlertypen sind durch den kontrollflussorientierten Strukturtest *zuverlässig* zu entlarven, nämlich
  - grobe Abbruchfehler,
  - unerreichbare Zweige,
  - Irrpfade,
  - endlose Schleifen und
  - unvollständige bzw. inkonsistente Bedingungen.

Simple Tippfehler, Schnittstellenprobleme, logische oder datenabhängige Fehler werden nicht systematisch<sup>1</sup> gesucht. Zu den letztgenannten Fehlertypen gehören etwa die Benutzung einer falschen Variablen oder einer Variablen, der vorher ein falscher oder gar kein Wert zugewiesen wurde. Mit der gezielten Suche nach Fehlern dieser Art beschäftigen sich datenflussorientierte Verfahren (vgl. Exkurs 9.2).

Vorteile kontroll-  
flussorientierter  
Verfahren

Testendekriterium

Nachteile kontroll-  
flussorientierter  
Verfahren

## Vorteile von Black-Box-Testverfahren

- Die Testfälle werden in erster Linie auf Basis der Programmstruktur erstellt. Dagegen wird die Spezifikation dessen, was der Prüfling zu leisten hat, nur „am Rande“ herangezogen, um zulässige Eingabedaten zu bestimmen und diesen die erwarteten Ausgaben zuzuordnen. Dadurch kann es passieren, dass eine zwar spezifizierte, jedoch nicht implementierte Funktionalität gar nicht getestet wird (vgl. Beispiel 10.0.1).

Zusammenfassend lässt sich sagen, dass der kontrollflussorientierte Test für sich allein kein ausreichender Test ist. Betrachten wir demgegenüber Black-Box-Tests, so können wir insbesondere folgende Vorteile feststellen, die sich daraus ableiten, dass die Testfälle allein auf Basis der Spezifikation konstruiert werden:

- Die Testfälle können bereits konstruiert werden, bevor ein lauffähiger Prüfling fertiggestellt ist. Sie sind unabhängig von der Implementierung des Prüflings und können daher für verschiedene Prüflinge mit derselben Spezifikation Anwendung finden<sup>1</sup>.
- Insbesondere können einmal konstruierte Testfälle auch nach Modifikationen am Prüfling unverändert für Regressionstests eingesetzt werden (vgl. Kapitel 11).
- Funktionsorientierte Verfahren haben zum Ziel, die spezifizierte Soll-Funktionalität möglichst vollständig zu überprüfen. Wie Beispiel 10.1.1 demonstriert, können gründliche funktionsorientierte Tests daher bestimmte Fehler aufdecken, die bei einem strukturorientierten Test verborgen bleiben.
- Fehlerorientierte Verfahren ermöglichen die besondere Berücksichtigung bekannter Fehlerquellen.

## Nachteile von Black-Box-Testverfahren

Nachteile von Black-Box-Testverfahren sind demgegenüber insbesondere:

- Sie stellen keine der White-Box-Überdeckungsmaße sicher, weshalb es prinzipiell vorkommen kann, dass z.B. implementierte Codeteile nie getestet werden.
- Die Testfallkonstruktion ist bei Verfahren wie z.B. der funktionalen Äquivalenzklassenbildung ein nicht sehr schematischer, sondern vielmehr konstruktiver Prozess, dessen Resultat willkürlicher als bei strukturorientierten Tests ausfallen kann: Für viele Probleme sind verschiedene Kriterien für eine Zerlegung der Mengen gültiger Ein- bzw. Ausgaben denkbar.
- Entsprechend gibt es auch meist keine klar vorgegebenen Testendekriterien und kein Maß für die Güte eines Tests, wie es z.B. die Überdeckungsgrade der kontrollflussorientierten Tests bieten.

---

1. Natürlich können strukturorientierte Tests auch deartige Fehler aufdecken, falls nämlich nach einem Testlauf ein anderes Ergebnis als erwartet ausgegeben wird. Aber solche Treffer sind meist keine unmittelbare Folge der Systematik der Testfallbildung auf Basis der Kontrollstruktur.

1. Diese Eigenschaft von Black-Box-Tests ermöglicht z.B. die automatischen Tests von Einsendungen zu den Einsendeaufgaben dieses Kurses.

Bei einem Vergleich dieser Vor- und Nachteile beider Gruppen von Verfahren fällt auf, dass es wenig erfolgversprechend ist, sich exklusiv für White-Box- oder Black-Box-Testverfahren zu entscheiden. Vielmehr sollte man auf Verfahren beider Klassen in einer Weise zurückzugreifen, dass sie sich sinnvoll ergänzen. Es ist ohnehin unvorteilhaft, sich auf ein einzelnes Testverfahren zu beschränken. Je länger und ausführlicher man damit testet, umso weniger weitere Fehler wird man bei gleichbleibendem Aufwand noch finden können (abnehmender Grenznutzen). Viel erfolgversprechender ist es daher, die Tests mit einem bestimmten Testverfahren ab einem gewissen Punkt abzubrechen und mit einem anderen Verfahren fortzufahren.

Es empfiehlt sich, die Testaktivitäten mit Black-Box-, insbesondere funktionsorientierten Tests zu beginnen und erst nach der Behebung der damit gefundenen Fehler zu einem strukturorientierten Test überzugehen, zumal die Erfüllung der Spezifikation das wichtigste Gütesiegel eines Programms darstellt. Außerdem können sorgfältig durchgeführte funktionale Tests bereits hohe Knoten-, Zweig- und Bedingungsüberdeckungsgrade erreichen. In nachgeschaltete strukturorientierte Tests muss dann oft nur noch wenig Arbeit investiert werden. Beginnt man statt dessen mit strukturorientierten Tests, so kann es passieren, dass man nach hohem Vorbereitungsaufwand in den ersten Testläufen grobe funktionale Fehler aufdeckt, die ein Black-Box-Test mit deutlich geringerem Aufwand gefunden hätte. Hinzu kommt, dass die aufwändig konstruierten Testfälle des strukturorientierten Tests nach der Korrektur des Prüflings in vielen Fällen nicht mehr für einen Regressions-test wiederverwendet werden können.

Wenn strukturorientierte Tests also eher von „nachrangiger“ Bedeutung sind, warum haben wir sie dann überhaupt in dieser Ausführlichkeit besprochen? Die Praxis sieht leider selten so „wohldefiniert“ aus, wie es die Theorie voraussetzt. Wir finden nur selten präzise (d.h. eindeutige, vollständige und widerspruchsfreie) Spezifikationen, und die aus ihnen abgeleiteten Testfälle und Testdaten für funktionsorientierte Tests enthalten dementsprechend oft selbst wieder Fehler. Hier sind ergänzende strukturorientierte Tests, welche die Fehlersuche ja grundsätzlich anders angehen als funktionsorientierte Tests, sehr wichtig, da sie oft bis dato unentdeckte Fehler aufspüren. Außerdem sind sie quantitativ beurteilbar, was Tester beruhigt und Manager beeindruckt.

Die nicht funktionsorientierten Black-Box-Verfahren wie der Zufallstest oder der Test spezieller Werte schließlich sollten nicht als alleinstehende Tests eingesetzt werden, sondern sind besonders sinnvoll als Auswahlstrategien zur Auswahl von Testdaten aus Testfällen im Rahmen sowohl funktionsorientierter als auch strukturorientierter Tests.

Abschließend möchten wir noch erwähnen, dass gerade kleine Programmeinheiten wie Prozeduren/Funktionen inzwischen häufig zunächst vom Programmierer selbst getestet werden. Egal auf welche Weise er seine Testfälle konstruiert, ob er also White-Box- oder Black-Box-Verfahren anwendet, kann er heute meist auf in die Entwicklungsumgebung integrierte Werkzeuge<sup>1</sup> zurückgreifen, die automatische

---

1. Ein Beispiel ist z.B. „JUnit“ für Modultests in Java.

Testläufe ermöglichen. Insbesondere die Wiederholung der schon vorbereiteten Testläufe nach jeder Änderung am Prüfling oder einem von ihm verwendeten Baustein (Regressionstest) kostet den Programmierer dank solcher Werkzeuge wenig Zeit und Mühe. Ein Vorteil der durch den Programmierer selbst durchgeführten Tests besteht darin, dass er unmittelbar nach der Implementierung Fehler schnell korrigieren kann, weil er mit dem Code noch sehr gut vertraut ist. Ein Nachteil ist dagegen, dass solche Tests einer „Trial-And-Error“-Programmierung förderlich sein können, bei der ein Programmierer mitunter viele Dinge einfach ausprobiert, statt vorher gründlich nach einem korrekten Lösungsweg zu suchen. Der so entstehende Code zeichnet sich meist durch schlechte Struktur mit vielen „Flickstellen“ aus und ist in aller Regel schlechter wartbar und fehleranfälliger als gründlich durchdachter Code. Müsste der Programmierer seinen Code dagegen an eine Testabteilung geben ohne die Möglichkeit, ihn vorher selbst zu testen, würde er wahrscheinlich mehr Aufwand in eine konstruktive Qualitätssicherung investieren, mit dem Ziel, von vornherein möglichst fehlerfrei zu entwickeln.

Natürlich sollte es nicht beim Test durch den Programmierer bleiben, wie wir auch bereits in einer Faustregel in Abschnitt 8.1 festgehalten haben. Spezialisierte Personen, die später das Produkt ganz oder in Teilen testen, bevorzugen übrigens in der Regel Black-Box-Tests, nicht nur aus den oben genannten fachlichen Gründen, sondern insbesondere auch deswegen, weil sie im Gegensatz zu White-Box-Tests keine Einarbeitung in die Implementierung des Prüflings erfordern.

## 13. Statische Testverfahren

### 13.1 Überblick

Nachdem wir uns in den letzten Kapiteln den dynamischen Testverfahren gewidmet haben, betrachten wir nun abschließend *statische Testverfahren*. Wie wir bereits in Abschnitt 8.2 festgehalten haben, ist ein wesentlicher Vorteil statischer Verfahren, dass sie nicht nur auf ausführbare Dokumente, sondern im Grunde auf alle Arten von Softwaredokumenten angewendet werden können. Außerdem sind statische Verfahren im Vergleich zu dynamischen Verfahren meist weniger aufwändig sowie bei sorgfältigem Einsatz sehr effektiv.

statische Verfahren

Mit statischen Verfahren verbindet man in erster Linie *Reviews*, d.h. das strukturierte Lesen von Softwaredokumenten. Einige der in Reviews entdeckten Fehler können auch durch automatische Werkzeuge wie *statische Programmanalysatoren* erkannt werden, welche die Programmqualität schnell und einfach verbessern. Statische Programmanalysatoren kann man als Ergänzung des Compilers betrachten, der ja nur die syntaktische Fehlerfreiheit des Programms sicherstellt. Auf solche Werkzeuge werden wir im Rahmen eines kurzen Exkurses (Abschnitt 13.3) noch eingehen.

statische Programm-  
analysatoren

### 13.2 Reviews

Ein Programmierer wird gewöhnlich (oft schon frühzeitig, wenn der Prüfling u.U. noch unfertig und noch nicht ausführbar ist) seinen Code aufmerksam zur Kontrolle lesen. Anders als beim dynamischen Testen kann er beim Identifizieren eines Fehlers oft schon die Fehlerursache direkt erkennen. Ähnlich wie beim dynamischen Testen ist zu beobachten, dass an der Erstellung des Dokuments nicht beteiligte Personen meist mehr Fehler finden als der Autor selbst. Solche Überlegungen, ebenso wie der Wunsch nach einem systematischen Vorgehen, führen zu Review-Techniken. Aus der freiwilligen, unstrukturierten Betrachtung von (Code-)Dokumenten wird in Reviews ein präzise definierter Vorgang, bei dem der zeitliche und inhaltliche Ablauf sowie die Anzahl und Qualifikationen der beteiligten Personen genau festgelegt sind und der nicht auf den Test des Quelltextes beschränkt bleibt.

Die bekanntesten *Arten von Reviews* sind die Inspektion und der (strukturierte) Walk-Through.

Review

Eine *Inspektionssitzung* läuft wie folgt ab (angelehnt an [My01]):

Inspektion

Das Inspektionsteam besteht gewöhnlich aus vier bis acht Personen. Eine von ihnen ist der *Moderator*, der ein erfahrener Softwareentwickler sein sollte. Er ist weder Autor des Dokuments noch muss er Details des Dokuments kennen. Die Pflichten des Moderators beinhalten die Verteilung der Unterlagen und die Zeitplanung für die Inspektionssitzung, die Leitung der Sitzung und die Protokollierung aller gefundenen Fehler. Außerdem muss er sicherstellen, dass anschließend alle Fehler behoben werden.

Moderator

## Walk-Through

Die weiteren Teilnehmer sind der Autor des Dokuments, der Programmdesigner (wenn es sich um ein Codedokument handelt) und ein Testspezialist. Beim Test der Produktspezifikation ist zusätzlich der Auftraggeber oder ein kompetenter Anwender anwesend.

Der Moderator sorgt dafür, dass rechtzeitig vor der Inspektionssitzung das zu prüfende Dokument und sämtliche Vorgängerdokumente, die bindende und damit zu testende Vorgaben enthalten (z.B. die Spezifikation des codierten Moduls), zur Verfügung stehen. Es wird erwartet, dass sich die Teilnehmer mit dem Stoff vor der Sitzung auseinandergesetzt haben. In der Sitzung erläutert der Autor sein Dokument Zeile für Zeile. Alle in der Diskussion erkannten Fehler werden aufgelistet, jedoch nicht korrigiert. Als Hilfsmittel dient dabei eine Checkliste möglicher Fehler.<sup>1</sup> Die Sitzung sollte nicht länger als 120 Minuten dauern, da danach die Konzentrationsfähigkeit der Teilnehmer und damit die Produktivität zu stark nachlässt. Im Mittel können in dieser Zeit ca. 300 Anweisungen Quelltext inspiziert werden.

Der (strukturierte) *Walk-Through* unterscheidet sich geringfügig von der Inspektion. Organisation und personelle Zusammensetzung sind im wesentlichen identisch und nur das inhaltliche Vorgehen während der Sitzung weicht voneinander ab. Beim Walk-Through hat der Testspezialist die Aufgabe, für die Sitzung einige nicht zu komplizierte Testfälle vorzubereiten, die dann von den Teilnehmern durchgespielt werden. Das Team simuliert den Computer.

Die Testfälle selbst spielen dabei keine kritische Rolle; sie dienen eher als Anreiz und um den Entwickler gezielt zu Entscheidungen (z.B. zur Programmlogik) zu befragen. In den meisten Fällen werden mehr Fehler in der Diskussion mit dem Entwickler als durch die Testfälle selbst entdeckt. Walk-Throughs sind damit weniger systematisch als Inspektionen (nach [My01]).

Entscheidend ist bei beiden Verfahren die disziplinierte Haltung der Teilnehmer. Kritik oder Kommentare müssen konstruktiv sein. Empfindet der Autor die Diskussion als persönliche Kritik, so wirkt dies kontraproduktiv, da er dann dazu neigt, die Entdeckung von Fehlern zu verhindern. Dies stellt den Erfolg der Verfahren in Frage.

Die besonderen Stärken von Reviews sind ihre Flexibilität und Universalität. In Reviews kann prinzipiell jeder Fehlertyp erkannt werden. Besonderes schwerwiegende Fehlertypen, wie z.B. logische Fehler, Fehler in den Spezifikationsdokumenten und Fehlentscheidungen in früheren Entwicklungsphasen, die durch andere Verfahren nur unzureichend (z.B. zufällig bei der Fehlersuche) oder überhaupt nicht aufgedeckt werden, kommen in Reviews eher ans Licht. Außerdem werden für Reviews keine Rechner und keine teure Testsoftware

---

1. Ausführliche Checklisten insbesondere für die Inspektion von Codedokumenten finden sich z.B. in [My01]. Beispiele sind Fragen wie: „Wird lesend auf nicht initialisierte Variable zugegriffen?“, „Hat der Zeiger noch einen gültigen Wert?“, „Ist die Division mit ganzzahligen Werten korrekt?“, „Sind alle Variablen auch deklariert?“, „Werden Ausdrücke verschiedenen Typs verglichen?“, „Ist die Priorität der Operatoren richtig verstanden worden?“ u.s.w.



benötigt. Der Erfolg hängt einzig von dem analytischen Denkvermögen, der Einsatzbereitschaft und der Disziplin der Teammitglieder ab.

Zum Abschluss merken wir noch einen anderen positiven Aspekt von Reviews an. Softwaredokumente werden im Laufe der Lebensdauer des Produkts wieder und wieder von Menschen gelesen – sei es bei Änderungen oder zur Wiederverwendung. Es ist deshalb sinnvoll, auf die Lesbarkeit von Dokumenten zu achten. Genau dies geschieht indirekt durch die Reviews, die Verständlichkeit und Lesbarkeit des Prüflings voraussetzen. Weiß ein Autor, dass andere Personen sein Dokument lesen und er es persönlich in einem Review verteidigen muss, so wird er nicht nur insgesamt sorgfältiger arbeiten, sondern auch besonders auf die Lesbarkeit achten.

### 13.3 Exkurs: Statische Analysatoren

Die wichtigsten automatischen Werkzeuge zur statischen Analyse von Dokumenten sind *statische Analysatoren*. Sie erlauben die rechnergestützte Überprüfung von Dokumenten, die nach einem festgelegten Schema aufgebaut sind.

statische Analysatoren

Man unterscheidet zwischen Analysatoren, die lexikalische, syntaktische und semantische Informationen erzeugen. Bei der statischen Programmanalyse können Statistiken über die Programmlänge und die verwendeten Programmkonstrukte (lexikalische Informationen) angelegt, statische Kontrollfluss-, Aufruf- und Importgraphen generiert (syntaktische Informationen) sowie Kontrollfluss- und Datenflussanalysen und Lebendigkeitsanalysen (semantische Informationen) durchgeführt werden. Statische Programmanalysatoren ergänzen damit die Arbeit des Compilers. Besonders bei schwach getypten Programmiersprachen werden viele semantische Fehler, wie z.B. die falsche Benutzung von Variablen, bei der Übersetzung übersehen, so dass die Programmqualität durch statische Analysatoren einfach und schnell verbessert werden kann.



## Lösungen zu den Aufgaben

### Aufgabe 4.1.3.2

```

program kleinsteZahl (input, output);
{ gibt die kleinste Zahl unter den integer-Eingabezahlen
  aus }

  var
    Zahl,
    Minimum : integer;

begin
  writeln ('Geben Sie die integer-Zahlen ein. ',
           '0 beendet die Eingabe. ');
  readln (Zahl);
  if Zahl = 0 then
    writeln ('Es wurde keine Zahl <> 0 eingegeben')
  else
    begin
      Minimum := Zahl;
      repeat
        if Zahl < Minimum then
          Minimum := Zahl;
        readln (Zahl)
      until Zahl = 0;
      writeln ('Die kleinste Zahl lautet ', Minimum, '.')
    end
  end. { kleinsteZahl }

```



### Aufgabe 4.2.1.3

Das Programm ist analog zu dem Programm `FolgenMaximum` (Beispiel 4.2.1.2). Es wird lediglich eine weitere Variable benötigt, um die Position des Minimums festzuhalten.

```

program FeldMinimum (input, output);
{ bestimmt das Minimum und seine Position
  in einem Feld von 20 integer-Zahlen }

  const
    FELDGROESSE = 20;

  type
    tIndex = 1..FELDGROESSE;
    tFeld = array [tIndex] of integer;

```

```

var
Feld : tFeld;
Minimum : integer;
i,
MinPos : tIndex;

begin
  { Einlesen des Feldes }
  writeln ('Geben Sie ', FELDGROESSE, ' Werte ein:');
  for i := 1 to FELDGROESSE do
    readln (Feld[i]);

  {Bestimmen des Minimums: }
  Minimum := Feld[1];
  MinPos := 1;
  for i := 2 to FELDGROESSE do
    if Feld[i] < Minimum then
      begin
        Minimum := Feld[i];
        MinPos := i
      end;

  {Ausgabe des Minimums und der Position }
  writeln ('Die kleinste Zahl ist ', Minimum, ',');
  writeln ('Sie steht an Position ', MinPos, '.')
end. { FeldMinimum }

```



### Aufgabe 4.2.1.5

Das folgende Programm löst das "Seminar"-Problem:

```

program Seminar1 (input, output);
{ Programmvariante 1 fuer das Seminarproblem }

const
MAXTEILNEHMER = 12;

type
tNatZahlPlus = 1..maxint;
tNatZahl = 0..maxint;
tStatus = (aktiv, passiv);
tIndex = 1..MAXTEILNEHMER;
tMatrNrFeld = array [tIndex] of tNatZahlPlus;
tStatusFeld = array [tIndex] of tStatus;

var
MatrNrFeld : tMatrNrFeld;

```

```

StatusFeld : tStatusFeld;
AnzStud : tNatZahl;
i : tIndex;
Status : char; { Zeichen zum Einlesen des Studenten-
                 status. Muss vom Typ char sein, um
                 eingelesen werden zu koennen.
                 'a' entspricht aktiv,
                 'p' entspricht passiv }

begin
  write ('Wie viele Studenten nahmen am Seminar teil? ');
  readln (AnzStud);
  if AnzStud > MAXTEILNEHMER then
    begin
      writeln ('Bitte hoechstens ',
               MAXTEILNEHMER, ' Eingaben!');
      AnzStud := MAXTEILNEHMER
    end;
  writeln ('Geben Sie Matr.Nr. und Aktivitaet der ',
           AnzStud, ' Teilnehmer ein:');

  for i := 1 to AnzStud do
    begin
      write ('Matr.Nr. ');
      readln (MatrNrFeld[i]);
      write ('a - aktiv, p - passiv: ');
      readln (Status);
      if Status = 'a' then
        StatusFeld[i] := aktiv
      else
        StatusFeld[i] := passiv
    end;

    { Scheine ausgeben }
    writeln;
    for i := 1 to AnzStud do
      begin
        if StatusFeld[i] = aktiv then
          begin
            write ('Der Student mit der Matrikel-Nr. ');
            writeln (MatrNrFeld[i]);
            writeln ('hat mit Erfolg am Seminar ',
                    'teilgenommen. ');
            writeln
          end
        end
      end;

    writeln ('Liste aller aktiven Seminar-Teilnehmer');
  end;

```

```

writeln ('-----');
for i := 1 to AnzStud do
  if StatusFeld[i] = aktiv then
    writeln (MatrNrFeld[i]);
writeln;

writeln ('Liste aller Zuhoerer im Seminar');
writeln ('-----');
for i := 1 to AnzStud do
  if StatusFeld[i] = passiv then
    writeln (MatrNrFeld[i])
end. { Seminar1 }

```

*Wie viele Studenten nahmen am Seminar teil? 3  
Geben Sie Matr.Nr. und Aktivitaet der 3 Teilnehmer  
ein:*

*Matr.Nr. 11  
a - aktiv, p - passiv: a  
Matr.Nr. 10  
a - aktiv, p - passiv: p  
Matr.Nr. 1  
a - aktiv, p - passiv: a*

*Der Student mit der Matrikel-Nr. 11  
hat mit Erfolg am Seminar teilgenommen.*

*Der Student mit der Matrikel-Nr. 1  
hat mit Erfolg am Seminar teilgenommen.*

*Liste aller aktiven Seminar-Teilnehmer  
-----  
11  
1*

*Liste aller Zuhoerer im Seminar  
-----  
10*



### Aufgabe 4.3.3

```

function istPrimzahl (p : tNatZahlPlus): boolean;
{ liefert true, falls p Primzahl, und false sonst }

```

```

var
  q : tNatZahlPlus;

```

```

begin
  if p < 2 then
    istPrimzahl := false
  else
    begin { p >= 2 }
      q := 2;
      while (p mod q <> 0) do
        q := q + 1;
      if q = p then
        { die Abbruchbedingung ist das erste Mal erfuehlt
          für q = p, also ist p Primzahl }
        istPrimzahl := true
      else { die Abbruchbedingung ist fuer q < p erfuehlt,
        also ist p durch q mit 2 <= q < p teilbar }
        istPrimzahl := false
      end
    end; { istPrimzahl }

```



### Aufgabe 5.1.1.2

Da FeldMinPosUndWert zwei Werte liefern soll, hat die Prozedur zwei out-Parameter MinPos und MinWert.

```

procedure FeldMinPosUndWert (
  in  Feld : tFeld;
  in  von,
      bis : tIndex;
  out MinPos : tIndex;
  out MinWert : integer);
{ bestimmt Position und Wert des Minimums
  im Feld zwischen von und bis }

var
  ind : tIndex;

begin
  MinPos := von;
  MinWert := Feld[von];
  for ind := von + 1 to bis do
    begin
      if Feld[ind] < Feld[MinPos] then
        begin
          MinPos := ind;
          MinWert := Feld[ind]
        end
      end

```

```

    end
end; { FeldMinPosUndWert }

```

Ein Aufruf dieser Prozedur könnte so aussehen:

```

FeldMinPosUndWert (SortierFeld, i, FELDGROESSE,
                  MinimumPos, MinimumWert);

```

Die aktuellen Parameter MinimumPos bzw. MinimumWert sind als Variablen vom Typ tIndex bzw. integer deklariert.



### Aufgabe 5.1.2.1

Die Prozedur FeldSortieren erhält einen weiteren Value-Parameter inSortLaenge. Die Feldeingabe wird entsprechend angepaßt.

```

program FeldSort4 (input, output);
{ sortiert ein Feld von bis zu 100 integer-Zahlen.
  Feldlaenge und Feldelemente werden eingelesen }

```

```

const
MAX = 100;

```

```

type
tIndex = 1..MAX;
tFeld = array[tIndex] of integer;

```

```

var
Groesse : integer; { aktuelle Feldgroesse }
EingabeFeld : tFeld;
idx : tIndex;

```

```

procedure FeldSortieren (
    inSortLaenge : tIndex;
    var ioSortFeld: tFeld);
{ sortiert ioSortFeld aufsteigend }

```

```

var
MinPos,
i : tIndex;

```

```

function FeldMinimumPos (
    var inFeld : tFeld;
    inVon,
    inBis : tIndex): tIndex;

```



```

{ bestimmt die Position des Minimums
  im inFeld zwischen inVon und inBis }

var
  MinimumPos,
  j : tIndex;

begin { FeldMinimumPos }
  MinimumPos := inVon;
  for j := inVon + 1 to inBis do
    if inFeld[j] < inFeld[MinimumPos] then
      MinimumPos := j;
  FeldMinimumPos := MinimumPos
end; { FeldMinimumPos }

procedure vertauschen (
  var ioHin,
    ioHer : integer);
{ vertauscht die Parameterwerte }

var
  Tausch : integer;

begin
  Tausch := ioHin;
  ioHin := ioHer;
  ioHer := Tausch
end; { vertauschen }

begin
  if inSortLaenge > 1 then
    for i := 1 to inSortLaenge - 1 do
      begin
        MinPos := FeldMinimumPos (ioSortFeld, i,
                                   inSortLaenge);
        { Minimum gefunden, jetzt mit dem
          Element auf Position i vertauschen }
        vertauschen (ioSortFeld[MinPos], ioSortFeld[i])
      end
    end; { FeldSortieren }

begin
  { einlesen der Feldgroesse solange, bis eine
    Feldgroesse zwischen 1 und MAX eingegeben wird }
  repeat
    write ('Geben Sie eine Feldgroesse zwischen 1 und ',
           MAX, ' ein: ');

```

```

    readln (Groesse)
until (1 <= Groesse) and (Groesse <= MAX);

    { einlesen des Feldes }
    writeln ('Geben Sie ', Groesse, ' Werte ein: ');
    for idx := 1 to Groesse do
        readln(EingabeFeld[idx]);

    { sortieren }
    FeldSortieren (Groesse, EingabeFeld);

    { ausgeben des sortierten Feldes }
    for idx := 1 to Groesse do
        write (EingabeFeld[idx]:6);
    writeln
end. { FeldSort4 }

```

```

Geben Sie eine Feldgroesse zwischen 1 und 100 ein: 5
Geben Sie 5 Werte ein:
5
3
9
7
1
      1      3      5      7      9

```



### Aufgabe 5.4.2.7

Wir geben die Lösung zu Teil a) und b) der Aufgabe zusammen an.

```

program ZahlenlisteEinAus (input,output);
{ liest integer-Zahlen von der Tastatur ein und gibt
  sie nach Eingabe einer 0 in der Reihenfolge der Eingabe
  wieder auf dem Bildschirm aus }

type
tRefListe = ^tListe;
tListe = record
    info : integer;
    next : tRefListe
end;

var

```

```

RefAnfang,
RefEnde,
Zeiger : tRefListe;
Zahl : integer;

procedure ElementAnhaengen (
    inZahl : integer;
    var ioRefAnfang, ioRefEnde : tRefListe);
{ haengt ein neues Element an die Liste an, auf deren
  erstes Element ioRefAnfang und auf deren letztes
  Element ioRefEnde zeigt; der Rekordkomponenten info
  des neuen Elementes wird der Wert inZahl zugewiesen }

var
  RefNeu : tRefListe;

begin
  { neues Element erzeugen und initialisieren }
  new (RefNeu);
  RefNeu^.info := inZahl;
  RefNeu^.next := nil;
  { Element an Liste anhaengen }
  if ioRefAnfang = nil then
    begin
      ioRefAnfang := RefNeu;
      ioRefEnde := RefNeu
    end
  else
    begin
      ioRefEnde^.next := RefNeu;
      ioRefEnde := RefNeu
    end;
  end; { ElementAnhaengen }

begin {ZahlenlisteEinAus }
  { Initialisierung der Liste }
  RefAnfang := nil;
  RefEnde := nil;
  { Zahlen einlesen und Liste aufbauen }
  writeln('Bitte natuerliche Zahlen eingeben,',
    '0=Ende!');
  readln (Zahl);
  while Zahl <> 0 do
    begin
      ElementAnhaengen (Zahl, RefAnfang, RefEnde);
      readln (Zahl)
    end;

```

```

    { Listenelemente ausgeben }
    Zeiger := RefAnfang;
    while Zeiger <> nil do
    begin
        write (Zeiger^.info, ', ');
        Zeiger := Zeiger^.next
    end
end. { ZahlenlisteEinAus }

```



### Aufgabe 6.1.4

Offensichtlich ist der gesuchte Knoten  $q$  der Knoten mit dem kleinsten Wert in dem rechten Teilbaum des Ausgangsknotens  $p$ . Dieser Knoten  $q$  befindet sich auf „ganz linker Position“ in dem rechten Teilbaum. Wir gehen daher im ersten Schritt von dem Knoten  $p$  zu seinem rechten Nachfolger (der Wurzel des rechten Teilbaumes) über. Hat dieser keinen linken Nachfolger, ist der gesuchte Knoten  $q$  gefunden. Anderfalls "hangeln" wir solange links den Teilbaum herunter, bis es (links) nicht mehr weitergeht.

```

function SortierNachfolger (
    inZeiger : tRefBinBaum) : tRefBinBaum;
{ liefert einen Zeiger auf den Nachfolger in der
  Sortierreihenfolge fuer den Knoten mit zwei
  Nachfolgern, auf den inZeiger zeigt, zurueck }

    var
        HilfZeig : tRefBinBaum;

    begin
        HilfZeig := inZeiger^.rechts;
        { jetzt haben wir die Wurzel des rechten Teilbaumes }
        while HilfZeig^.links <> nil do
            HilfZeig := HilfZeig^.links;
        { jetzt hat der Knoten, auf den HilfZeig zeigt,
          keinen linken Nachfolger mehr und wir sind am Ziel.
          Falls bereits die Wurzel des rechten Teilbaumes
          keinen linken Nachfolger hat, wird die while-
          Schleife gar nicht erst durchlaufen }
        SortierNachfolger := HilfZeig
    end; { SortierNachfolger }

```



### Aufgabe 6.2.3

Analog zu Beispiel 6.2.2 können wir die mathematisch notierte Funktionsdefinition praktisch eins zu eins in eine Pascal-Funktion `fibonacciRek` übersetzen. Wir benötigen eine Fallunterscheidung, die für `inZahl = 0` das Ergebnis 0 ausgibt, für `inZahl = 1` das Ergebnis 1 und für `inZahl ≥ 2` die Summe der durch rekursive Aufrufe ermittelten Werte `fibonacciRek(inZahl-1)` und `fibonacciRek(inZahl-2)`.

Eine kleine Vereinfachung nehmen wir noch vor: Da für `inZahl = 0` und `inZahl = 1` das Ergebnis jeweils mit `inZahl` übereinstimmt, können wir diese beiden Fälle zusammenfassen.

**type**

```
tNatZahl = 0..maxint;
```

```
function fibonacciRek(inZahl: tNatZahl): tNatZahl;
{ Berechnet rekursiv die inZahl-te Fibonacci-Zahl }
begin
  if inZahl < 2 then
    fibonacciRek := inZahl
  else
    fibonacciRek := fibonacciRek(inZahl - 1)
                  + fibonacciRek(inZahl - 2)
end;
```

Eine iterative Lösung ist nicht ganz so einfach wie für die Fakultätsfunktion zu finden: Zur Berechnung der  $n$ -ten Fibonacci-Zahl für  $n > 2$  müssen zuvor ihre beiden Vorgänger in der Fibonacci-Folge berechnet worden sein. Wir benötigen daher zwei Variablen zur Aufnahme dieser beiden Vorgänger, sowie eine dritte Variable zur Aufnahme ihrer Summe. Die erste Fallunterscheidung übernehmen wir unverändert aus der rekursiven Version:

```
function fibonacciIt(inZahl: tNatZahl): tNatZahl;
{ Berechnet iterativ die inZahl-te Fibonacci-Zahl }

  var letzte, vorletzte, aktuelle, lauf: tNatZahl;

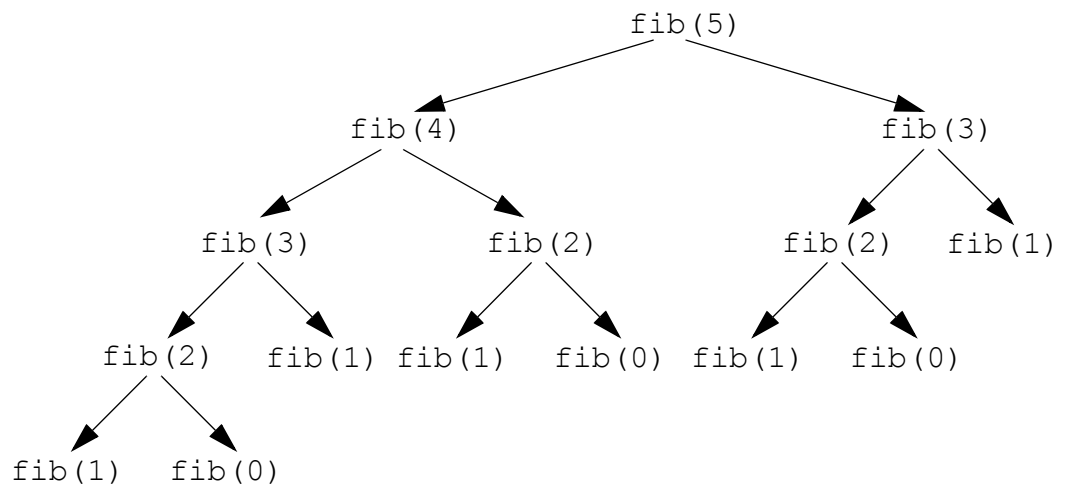
begin
  if inZahl < 2 then
    fibonacciIt := inZahl
  else
    begin
      { Initialisierung der Vorgänger (f(0) und f(1))
        für die Berechnung von f(2) }
      vorletzte := 0;
      letzte := 1;
      { Berechnung von f(2) bis f(inZahl) }
```

```

for lauf := 2 to inZahl do
begin
    aktuelle := vorletzte + letzte;
    vorletzte := letzte;
    letzte := aktuelle;
end;
fibonacciIt := aktuelle
end
end;

```

Der direkte Vergleich zeigt deutlich, dass hier – anders als bei der Fakultät – die Programmkomplexität der iterativen Lösung deutlich höher als die der rekursiven ist. Dafür ist sie aber *wesentlich* effizienter: Sie benötigt zum Einen immer gleich viel Speicherplatz für ihre vier Variablen, unabhängig von der Größe von `inZahl`. (D.h. sie kommt ohne dynamische Hilfsdatenstruktur und ohne Laufzeitstack aus.) Zum Anderen berechnet sie zur Ermittlung der  $n$ -ten Fibonacci-Zahl die vorangehenden  $n-1$  Fibonacci-Zahlen jeweils nur einmal. Im Gegensatz dazu belegt die rekursive Variante mit jedem rekursiven Aufruf Speicher auf dem Laufzeitstack (maximale Rekursionstiefe ist `inZahl-1`, d.h. der Laufzeitstack wächst linear). Die Anzahl der insgesamt abzuarbeitenden rekursiven Aufrufe ist sogar noch deutlich höher (exponentielles Wachstum), da die benötigten Zwischenergebnisse mehrfach neu berechnet werden. Die folgende Abbildung demonstriert diesen Sachverhalt am Beispiel des Aufrufs von `fibonacciRek` mit `inZahl = 5`. Die schwarzen Pfeile stellen dabei rekursive Aufrufe dar. Den Funktionsnamen haben wir aus Platzgründen in der Abbildung zu `fib` abgekürzt.



### Aufgabe 6.2.7

- a) Die Rekursion bricht ab, wenn das (rekursiv) zu untersuchende Restfeld nur noch aus einem einzigen Element besteht, dessen Maximum ja gerade das Element selbst ist.

```

function rekFeldMax (
    inMaxPos : tIndex;
    var inFeld : tFeld) : tIndex;
{ bestimmt rekursiv in dem integer-Feld inFeld ab
  Position inMaxPos die Position des Maximums }

var
  lokMaxPos : tIndex; { Hilfsvariable }

begin
  if inMaxPos = FELDGROESSE then
    { Rekursionsabbruch }
    rekFeldMax := inMaxPos
  else
    begin
      { rekursiv das Maximum unter den
        Elementen inFeld[inMaxPos + 1] bis
        inFeld[FELDGROESSE] bestimmen }
      lokMaxPos := rekFeldMax (inMaxPos + 1, inFeld);
      { dieses Maximum mit dem "ersten"
        Element inFeld[inMaxPos] vergleichen }
      if inFeld[inMaxPos] < inFeld[lokMaxPos] then
        rekFeldMax := lokMaxPos
      else
        rekFeldMax := inMaxPos
    end
  end; { rekFeldMax }

```

- b) Es handelt sich um keine sinnvolle Anwendung der Rekursion, denn die iterative Alternative besteht aus einer simplen Schleife mit konstantem Speicherbedarf. Die Anzahl der rekursiven Aufrufe und die erforderliche Größe des Laufzeitstack der rekursiven Funktion entsprechen dagegen der Anzahl der Feldelemente.



### Aufgabe 6.2.10

- a) Eine mögliche rekursive Lösung lautet:

```

function BlattSuchenRek (
    inRefWurzel : tRefBinBaum) : tRefBinBaum;
{ ermittelt rekursiv das erste Blatt von links
  des binaeren Baumes, auf dessen Wurzel inRefWurzel
  zeigt }

var
  RefElement : tRefBinBaum;

begin

```

```

RefElement := inRefWurzel;
if RefElement <> nil then
begin
  if RefElement^.links <> nil then
    RefElement :=
      BlattSuchenRek (RefElement^.links)
  else
    if RefElement^.rechts <> nil then
      RefElement :=
        BlattSuchenRek (RefElement^.rechts)
  end;
  BlattSuchenRek := RefElement
end; { BlattSuchenRek }

```

b) Eine mögliche iterative Lösung lautet:

```

function BlattSuchenIt (
  inRefWurzel : tRefBinBaum): tRefBinBaum;
{ ermittelt iterativ das erste Blatt von links
  des binären Baumes, auf dessen Wurzel inRefWurzel
  zeigt }

var
  RefElement : tRefBinBaum;

begin
  RefElement := inRefWurzel;
  if RefElement <> nil then
    { solange noch kein Blatt erreicht wurde, .. }
    while (RefElement^.links <> nil) or
      (RefElement^.rechts <> nil) do
      begin
        { .. gehe soweit möglich nach links .. }
        while RefElement^.links <> nil do
          RefElement := RefElement^.links;
        { .. und dann, falls noch ein rechter Sohn
          existiert, einmal nach rechts }
        if RefElement^.rechts <> nil then
          RefElement := RefElement^.rechts
        end;
      BlattSuchenIt := RefElement
  end; { BlattSuchenIt }

```

c) Die Rekursionstiefe und damit die Größe des Laufzeitstack von BlattSuchenRek wächst im besten Fall logarithmisch (für balancierte Bäume), im schlimmsten Fall jedoch linear (bei einem degenerierten Baum) mit der Anzahl der Knoten des Baumes. BlattSuchenIt kommt dagegen mit konstantem Speicherbedarf unabhängig von der Baumgröße aus und das bei noch

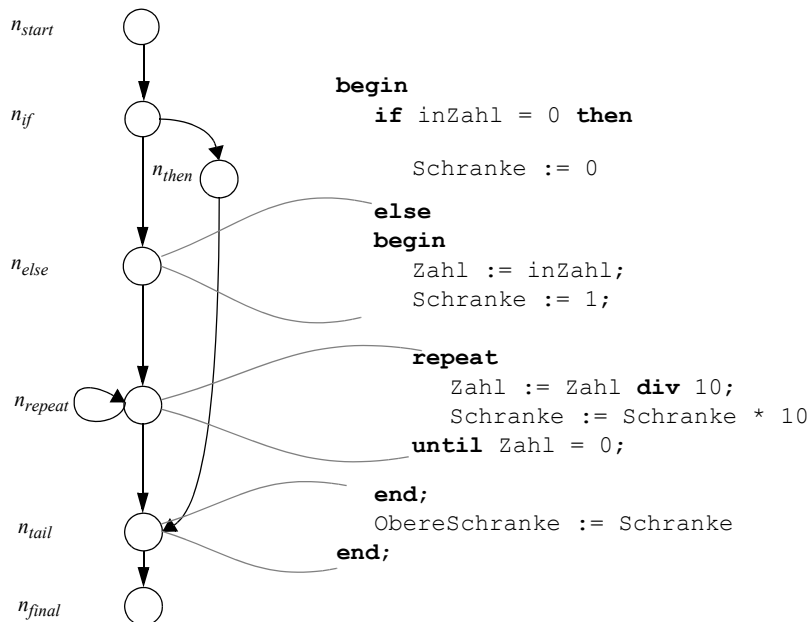


nicht einmal wesentlich größerer Programmkomplexität. Hier ist also die Rekursion nicht sinnvoll, sondern die Iteration vorzuziehen.



### Aufgabe 9.1.3.2

a) Die folgende Abbildung zeigt den kompakten Kontrollflussgraphen:

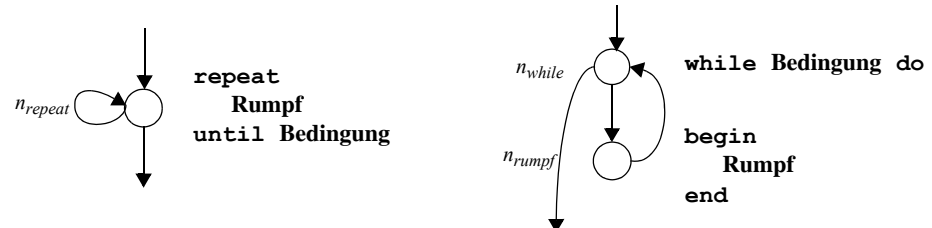


Noch einige Anmerkungen, die Ihnen die Bewertung einer abweichenden Lösung vereinfachen sollen:

Der Graph ist weitgehend eindeutig, Wahlfreiheit besteht lediglich bei der Benennung der inneren Knoten (also aller Knoten außer  $n_{start}$  und  $n_{final}$ ) sowie bei der Zuordnung der Schlüsselwortzeilen „**else begin**“, die wahlweise auch  $n_{then}$  statt  $n_{else}$  zugeordnet werden dürfen, sowie der auf die **repeat**-Schleife folgenden Schlüsselwortzeile „**end**“, die wahlweise auch  $n_{repeat}$  statt  $n_{tail}$  mit zugeordnet werden darf. (Diese Schlüsselwortzeilen enthalten keine Anweisungen, sondern nur zur Strukturierung des Quelltextes.) Den Knoten  $n_{start}$  und  $n_{final}$  werden definitionsgemäß keine Programmzeilen zugeordnet, auch nicht das einleitende **begin** oder das abschließende **end**. Da der Graphen den Kontrollfluss der Funktion darstellen soll, werden ihm auch nur diejenigen Programmzeilen zugeordnet, die den Kontrollfluss der Funktion definieren, d.h. nur die Zeilen des Anweisungsteils, nicht etwa der Funktionskopf oder der Deklarationsteil.

Beachten Sie weiterhin, dass sich die **repeat**-Schleife auf einen einzigen Knoten abbilden lässt, der sein eigener Nachfolger ist: Der gesamte Block einschließlich der Prüfung der Abbruchbedingung wird immer zusammenhängend („en bloc“) ausgeführt, anschließend erfolgt die Entscheidung, ob er ein weiteres Mal ausgeführt werden soll oder die Schleife abbricht. Um Missverständnissen vorzubeugen, weisen wir an dieser Stelle jedoch darauf hin, dass bei einer **while**-Schleife dagegen immer *zwei* Knoten benötigt werden:

einer, der die Programmzeile mit der **while**-Anweisung aufnimmt und in dem die Entscheidung stattfindet, ob als nächstes der Rumpf betreten wird oder nicht (und der folglich zwei Nachfolgerknoten hat), und ein zweiter Knoten für den besagten Schleifenrumpf, der nur den **while**-Knoten als Nachfolger hat, da nach der Schleifenausführung wieder die Abbruchbedingung überprüft werden muss. Die folgende Abbildung stellt **repeat**- und **while**-Schleife allgemein gegenüber. Diese Abbildungen gelten so natürlich nur für Schleifen, in deren Rumpf keine Verzweigungen (z.B. weitere Schleifen oder eine **if**-Anweisung) enthalten sind:



- b) Für eine vollständige Zweigüberdeckung benötigen wir mindestens zwei Pfade, von denen einer den **then**-Zweig ( $n_{if}, n_{then}$ ) und einer den **else**-Zweig ( $n_{if}, n_{else}$ ) passiert. Der Pfad durch den **else**-Zweig durchläuft mindestens einmal die **repeat**-Schleife. Bereits bei einem einzigen Durchlauf der **repeat**-Schleife wird zwar (mit beiden Pfaden zusammen) eine vollständige Anweisungsüberdeckung erreicht, jedoch bliebe der „Wiederholungszweig“ ( $n_{repeat}, n_{repeat}$ ) unüberdeckt, d.h. für eine vollständige Zweigüberdeckung muss unser Pfad durch den **else**-Zweig die Schleife mindestens zweimal durchlaufen. Wir wählen daher folgende zwei Pfade:

Durchlaufen des **then**-Zweigs:

Pfad<sub>1</sub> = ( $n_{start}, n_{if}, n_{then}, n_{tail}, n_{final}$ )

Durchlaufen des **else**-Zweigs bei genau zwei Schleifendurchläufen:

Pfad<sub>2</sub> = ( $n_{start}, n_{if}, n_{else}, n_{repeat}, n_{repeat}, n_{tail}, n_{final}$ )

- c) Der **then**-Zweig wird genau bei Eingabe `inZahl = 0` durchlaufen, der assoziierte Testfall zu Pfad<sub>1</sub> enthält also ein einziges Testdatum mit Eingabe 0 und erwarteter Ausgabe 0:

$T_1 = \{ (0, 0) \}$

Zum Durchlaufen von Pfad<sub>2</sub> muss Folgendes gelten:

$inZahl > 0$  (damit der **else**-Zweig betreten wird)  
 und  $inZahl \text{ div } 10 > 0$  (damit die Schleife wiederholt wird)  
 und  $(inZahl \text{ div } 10) \text{ div } 10 = 0$  (für Schleifenabbruch nach 2. Durchl.)

Das ist genau für alle zweistelligen Eingaben der Fall. Das erwartete Ergebnis lautet 10 für die Eingabe `inZahl = 10` sowie 100 für jede andere zweistellige Eingabe. Der assoziierte Testfall zu Pfad<sub>2</sub> lautet also:

$T_2 = \{ (10, 10), (z, 100) \mid z \in \mathbb{N}, 11 \leq z \leq 99 \}$

- d) Das einzige Testdatum aus  $T_1$  ist (0, 0). Bei Eingabe von 0 wird die Ausgabe 0 geliefert, der Prüfling reagiert hier also korrekt.

Aus Testfall  $T_2$  wählen wir zunächst ein repräsentatives Testdatum mit erwartetem Ergebnis 100, z.B. (49, 100). Bei Eingabe von 49 wird auch tatsächlich der Wert 100 ausgegeben.

Das Testdatum (10, 10) nimmt in  $T_2$  eine Sonderstellung ein, da es als einziges ein anderes erwartetes Ergebnis aufweist. Dieses sollten wir also unbedingt auch testen und stellen fest, dass auch für die Eingabe von 10 das Ergebnis 100 ermittelt wird. Hier haben wir also einen Fehler gefunden.

Übrigens hätte man das auch ohne Testdurchführung schon bei der Testfallbildung erkennen können, denn die Ausgabe ist bei  $n$  Schleifendurchläufen immer  $10^n$ , und da jedes Testdatum aus  $T_2$  zu genau zwei Schleifendurchläufen führt, wird auch für jedes Testdatum aus  $T_2$  die Zahl 100 ausgegeben.

### Aufgabe 9.1.4.2

Tabelle zu den Teilen a) und b):

Prädikate		inFeld																								
		1	2	3	3	3	3	2	1	0	0	1	2	3	4	0	1	2	2	2	3	1	2	3	4	5
A	i < FELDMAX	T, F																								
B	inFeld[i-1] <= inFeld[i]	T					F																			
C	A and B (while)	T, F																								
D	inFeld[i-1] = inFeld[i] (1. if)	T, F																								
E	i < FELDMAX	F					T																			
F	i = FELDMAX	T					F																			
G	inFeld[i-1] > inFeld[i]	F					T																			
H	F and G	F					F					T														
I	E or H (2. if)	F					T																			
J	inFeld[i-1] < inFeld[i]	F					—					—					T									
K	not dupl	F					—					—					F					?				
L	J and K (3. if)	F					—					—					F					?				
erwartete Ausgabe		sortiert					unsortiert					unsortiert					sortiert					sortiertDuplikatfrei				
tatsächliche Ausgabe		sortiert					unsortiert					unsortiert					sortiert					?				

- a) Zu jeder Entscheidung (hier: **if**- bzw. **while**-Anweisung) ist zunächst das darin enthaltene Prädikat zu überdecken. Ist es zusammengesetzt aus Teilprädikaten, sind auch diese zu überdecken. Ist ein solches Teilprädikat wiederum

zusammengesetzt, sind auch dessen Teile zu überdecken u.s.w. Die sich so ergebenden Prädikate haben wir in oben stehende Tabelle eingetragen.

- b) Wir betrachten zunächst das erste Prädikat A. Dieses wird offensichtlich bei jedem Testlauf mindestens zu `true` ausgewertet, also wählen wir ein Testdatum, unter dem es *auch* zu `false` ausgewertet wird, d.h. unter dem die Schleife das Feld bis zum Ende durchläuft. Das ist unter anderem (nicht nur) für aufsteigend sortierte Felder der Fall. Wir wählen das Testdatum `([1,2,3,3,3], sortiert)`. Wie gewünscht wird Prädikat A dadurch überdeckt, entsprechend auch Prädikat C, sogar auch Prädikat D (da wir ein Duplikat eingefügt haben), Prädikat B jedoch nicht (wird immer zu `true` ausgewertet).

Als nächstes wollen wir Prädikat B überdecken, also zu `false` auswerten. Dazu benötigen wir ein Array, dessen ersten vier Elemente nicht sortiert sind. Wir wählen das Testdatum `([3,2,1,0,0], unsortiert)`. Damit werden bereits die Prädikate A bis G und I vollständig abgedeckt. Bemerkenswert ist hierbei, dass wir sowohl die zusammengesetzte 2. **if**-Bedingung (I) als auch alle ihre Atome (E, F, und G) bereits überdeckt haben, also (bezogen nur auf diese **if**-Anweisung, nicht auf die gesamte Funktion) eine einfache Bedingungsüberdeckung und eine Zweigüberdeckung bereits erreicht haben, nicht jedoch die minimale Mehrfach-Bedingungsüberdeckung, weil das Teilprädikat H noch nicht überdeckt wurde. (Dies zeigt die Schwäche der beiden vorhergenannten Maße.)

Um nun auch H zu **true** auszuwerten, benötigen wir eine Eingabe, unter der das Feld einerseits vollständig durchlaufen wird (um F zu erfüllen) und für die das letzte Feldelement kleiner als das vorletzte ist (um zusätzlich G zu erfüllen), d.h. ein Array, das mit Ausnahme des letzten Elementes aufsteigend sortiert ist. Wir wählen das Testdatum `([1,2,3,4,0], unsortiert)`.

Nun verbleiben nur noch die Prädikate J bis L. Um J zu `true` auszuwerten, wählen wir ein aufsteigend sortiertes Feld als Eingabe, dessen letzter Wert das Maximum ist: `([1,2,2,2,3], sortiert)`. Prädikat J ist damit vollständig überdeckt.

Abschließend müssen wir noch Prädikate K und L überdecken, also ein duplikatfreies sortiertes Feld eingeben. Wir wählen als Testdatum `([1,2,3,4,5], sortiertDuplikatfrei)`. Für dieses Testdatum zeigt sich ein Fehler: Variable `dupl` und mit ihr in diesem Fall beide Prädikate K und L sind undefiniert (in der Tabelle durch „?“ notiert), da Variable `dupl` nicht initialisiert wird. Damit können wir nicht deterministisch die tatsächliche Ausgabe vorhersagen, ein Testlauf könnte zufällig die korrekte Ausgabe `sortiertDuplikatfrei` oder auch die (falsche) Ausgabe `sortiert` liefern. Eine vollständige Überdeckung ist ohne Korrektur dieses Fehlers nicht möglich. (Hätten wir die fehlende Initialisierung von `dupl` nicht bemerkt, sondern wären wir davon ausgegangen, dass `dupl` in diesem Fall den Wert `false` habe und daher die Prädikate K und L zu `true` ausgewertet würden, dann hätten wir jetzt „auf dem Papier“ eine vollständige minimale Mehrfach-Bedingungsüberdeckung erreicht und würden den Fehler vielleicht bei einer dynamischen Testausführung mit dem letzten Testdatum feststellen.)

Es sei noch angemerkt, dass die Nicht-Initialisierung von `dupl` sich nicht in jedem Fall in einem undefinierten tatsächlichen Ergebnis niederschlägt: Für das Testdatum `([1,2,3,4,4],sortiert)` z.B. bliebe zwar die Variable `dupl` undefiniert und mit ihr auch Prädikat `K`, die Konjunktion `J and K` jedoch würde unabhängig vom Wert von `K` (allein wegen `J = false`) zu `false` ausgewertet und die tatsächliche Ausgabe wäre somit `sortiert`.

- c) Wir haben in unseren Ausführungen zu Teil b) bereits die Fehlerursache bemerkt: Die Variable `dupl` wird nicht initialisiert. Hätten wir dies jedoch übersehen, und hätte ein Testlauf mit dem letzten Testdatum (zufällig) die Ausgabe `sortiert` geliefert, so wäre uns zumindest das Fehlersymptom aufgefallen und wir müssten die Fehlerursache suchen. Zur Fehlerkorrektur ist vor der **while**-Schleife die Anweisung `dupl := false` einzufügen.

### Aufgabe 9.1.5.3

Die drei Pfadklassen des Boundary-Interior-Tests sind:

1. Schleife wird nicht durchlaufen:

Diese Klasse ist leer (d.h. sie enthält keine Pfade), da eine **repeat**-Schleife immer mindestens einmal durchlaufen wird.

2. Schleife wird genau einmal durchlaufen:

Diese Klasse enthält (da es in der Schleife keine Verzweigungen gibt) genau einen Pfad:

$(n_{start}, n_{if}, n_{else}, n_{repeat}, n_{tail}, n_{final})$

Der assoziierte Testfall zu diesem Pfad lautet:

$\{ (z, 10) \mid z \in \mathbb{IN}, 1 \leq z \leq 9 \}$

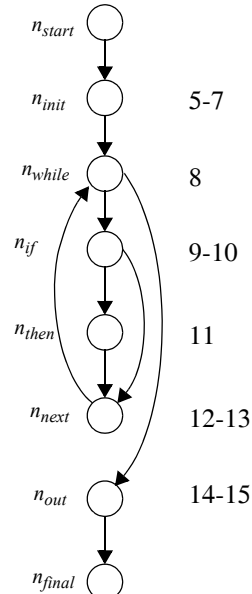
(Anmerkung: In der Lösung zu Aufgabe 9.1.3.2 haben wir bereits festgehalten, dass bei  $n$  Durchläufen immer  $10^n$  ausgegeben wird. Bei genau einem Durchlauf wird also das Ergebnis immer 10 lauten, was laut obigem Testfall dem erwarteten Ergebnis entspricht. Somit wird kein Testdatum aus diesem Testfall einen Fehler aufdecken.)

3. Schleife wird genau  $n$ -mal durchlaufen, wobei wir  $n=2$  wählen:

Auch diese Klasse enthält genau einen Pfad, den wir bereits beim Zweigüberdeckungstest in Aufgabe 9.1.3.2 betrachtet haben.

### Aufgabe 9.1.5.4

- a) Die folgende Abbildung zeigt den kompakten Kontrollflussgraphen (unter Verwendung der Zeilennummern):



Auch hier ist die Zuordnung zweier reiner Schlüsselwortzeilen wieder variabel: Zeile 9 kann wahlweise auch Knoten  $n_{while}$ , Zeile 13 auch Knoten  $n_{out}$  zugeordnet werden.

- b) Die Klasse der die Schleife nicht durchlaufenden Pfade enthält hier genau einen Pfad:  $(n_{start}, n_{init}, n_{while}, n_{out}, n_{final})$ .

Da zwei verschiedene Teilpfade durch die Schleife führen – einmal über den **then**- und einmal über den **else**-Zweig von  $n_{if}$  –, gibt es genau zwei Pfade, die die Schleife genau einmal durchlaufen. Beide sind ausführbar, daher umfasst die Boundary-Klasse genau folgende zwei Pfade:

$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{next}, n_{while}, n_{out}, n_{final})$  und  
 $(n_{start}, n_{init}, n_{while}, n_{if}, n_{next}, n_{while}, n_{out}, n_{final})$ .

Um möglichst wenige Pfade (und entsprechend Testfälle) zu erhalten, wählen wir für den Interior-Test  $n=2$ . Da pro Schleifendurchlauf zwei Teilpfade durch die Schleife führen, ergeben sich nämlich für  $n$  Schleifendurchläufe  $2^n$  zu testende Pfade (und entsprechend auch  $2^n$  assoziierte Testfälle). Für  $n=2$  erhalten wir demnach vier mögliche Pfade, welche die Interior-Klasse bilden:

$(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{next}, n_{while}, n_{if}, n_{then}, n_{next}, n_{while}, n_{out}, n_{final})$ ,  
 $(n_{start}, n_{init}, n_{while}, n_{if}, n_{then}, n_{next}, n_{while}, n_{if}, n_{next}, n_{while}, n_{out}, n_{final})$ ,  
 $(n_{start}, n_{init}, n_{while}, n_{if}, n_{next}, n_{while}, n_{if}, n_{then}, n_{next}, n_{while}, n_{out}, n_{final})$ ,  
 $(n_{start}, n_{init}, n_{while}, n_{if}, n_{next}, n_{while}, n_{if}, n_{next}, n_{while}, n_{out}, n_{final})$ .

- c) Abweisender Test (kein Schleifendurchlauf): (für einelementige Listen)  
 Assoziierter Testfall:  $T_1 = \{ ([x], x) \}$

Boundary-Test: (für zweielementige Listen)

Assoziierter Testfall zum 1. Pfad:  $T_2 = \{ ([x, y], y) \mid x < y \}$

Assoziierter Testfall zum 2. Pfad:  $T_3 = \{ ([x, y], x) \mid x \geq y \}$

Interior-Test: (für dreielementige Listen)

Assoziierter Testfall zum 1. Pfad:  $T_4 = \{ ([x, y, z], z) \mid x < y < z \}$

Assoziierter Testfall zum 2. Pfad:  $T_5 = \{ ([x, y, z], y) \mid x < y \geq z \}$

Assoziierter Testfall zum 3. Pfad:  $T_6 = \{ ([x, y, z], z) \mid x \geq y, x < z \}$

Assoziierter Testfall zum 4. Pfad:  $T_7 = \{ ([x, y, z], x) \mid x \geq y, x \geq z \}$

- d) Im Folgenden wählen wir zu den Testfällen  $T_1$  bis  $T_7$  jeweils exemplarisch zwei Testdaten:

zu  $T_1$ :  $([-100], -100), ([0], 0)$

zu  $T_2$ :  $([-10, 10], 10), ([5, 6], 6)$

zu  $T_3$ :  $([5, 5], 5), ([5, -10], 5)$

zu  $T_4$ :  $([1, 2, 3], 3), ([-100, -10, -5], -5)$

zu  $T_5$ :  $([5, 6, 6], 6), ([10, 11, -5], 11)$

zu  $T_6$ :  $([5, 5, 6], 6), ([10, -5, 11], 11)$

zu  $T_7$ :  $([5, 5, 5], 5), ([-1, -3, -2], -1)$

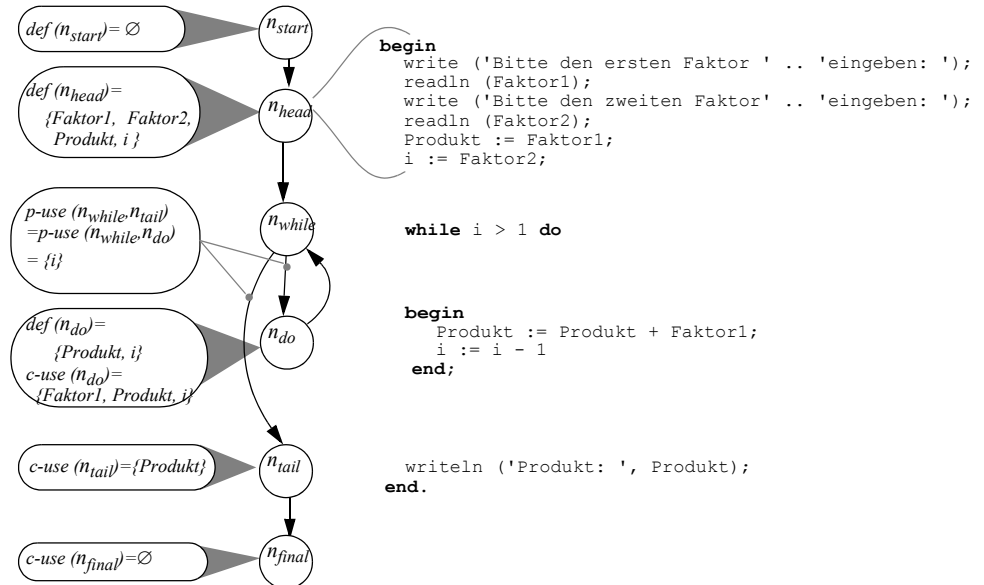
- e) Zu e1): Beim Bilden der Testfälle müsste in diesem Fall jeweils eine um ein Element längere Liste als Eingabe für die gleiche Zahl an Schleifendurchläufen gewählt werden. Zum Beispiel würde der die Schleife nicht durchlaufende Pfad würde in diesem Fall für beliebige zweielementige Listen durchlaufen, der assoziierte Testfall lautet  $\{ ([x, y], m) \mid m = \max(x, y) \}$ . Hier könnte dem Prüfer auffallen, dass die Bestimmung des Maximums von zwei Werten kaum mit dem Pfad möglich sein kann, der kein einziges Mal die Schleife durchläuft. Wahrscheinlich würde dem Tester sogar schon bei der Ermittlung der Eingabedaten für diesen Testfall auffallen, dass tatsächlich ein zweites Listenelement existieren muss, damit die While-Bedingung fehlerfrei (zu false) auswertbar ist – weil es bei einer einelementigen Liste zu einer NIL-Dereferenzierung kommt. Es ist also sehr wahrscheinlich, jedoch nicht sicher, dass der Fehler schon ohne Testdurchführung gefunden wird.

Zu e2): Hier sollte der Fehler spätestens beim Bilden der assoziierten Testfälle zu den Pfaden für Boundary- und Interior-Klasse auffallen, da die Schleife, wenn sie denn einmal betreten wird, niemals abbricht, sich also keine Eingabedaten für diese Testfälle finden lassen. Der Tester kann allerdings selbst einen Fehler machen und nicht-leere Testfälle bestimmen (die dann natürlich nicht die assoziierten Testfälle zu den Pfaden darstellen). In diesem Fall wird die Enschlossschleife aber bei der Testausführung gefunden.

Zu e3): Hier berechnet der Prüfling das Maximum zunächst korrekt, lediglich die Rückgabe des berechneten Wertes wurde vergessen. Es ergeben sich daher exakt dieselben Testfälle wie oben, und der Programmierfehler fällt bei der Testfallbildung gar nicht (oder nur zufällig) auf. Bei der dynamischen Testausführung jedoch wird der Fehler aufgedeckt (und zwar für nahezu beliebige Testdaten, d.h. dieser Fehler würde in der Regel sogar durch einen unsystematischen Zufallstestlauf gefunden).

### Aufgabe 9.2.2.2

a) Der zugehörige Kontrollflußgraph in Datenflußdarstellung ist gegeben durch



b) Man erhält folgende *du*-Mengen:

$$\begin{aligned}
 du(Faktor1, n_{head}) &= \{n_{do}\} \\
 du(Faktor2, n_{head}) &= \emptyset \\
 du(Produkt, n_{head}) &= \{n_{do}, n_{tail}\} \\
 du(i, n_{head}) &= \{(n_{while}, n_{do}), (n_{while}, n_{tail}), n_{do}\} \\
 du(Produkt, n_{do}) &= \{n_{do}, n_{tail}\} \\
 du(i, n_{do}) &= \{(n_{while}, n_{do}), (n_{while}, n_{tail}), n_{do}\}
 \end{aligned}$$

Das Ergebnis  $du(Faktor2, n_{head}) = \emptyset$  besagt, daß die Variable Faktor2 nicht global benutzt wird. Tatsächlich wird sie nur lokal benutzt.

c) Mit Hilfe der *du*-Mengen aus Aufgabenteil b) bestimmen wir nun die zu testenden Teilpfade. Gleiche Teilpfade für verschiedene Variablen können wir zusammenfassen.

$$n_{head} \leftarrow \begin{array}{l} n_{do}: (n_{head}, n_{while}, n_{do}) \end{array} \quad (1)$$

$$n_{tail}: (n_{head}, n_{while}, n_{tail}) \quad (2)$$

$$(n_{while}, n_{do}): (n_{head}, n_{while}, n_{do}) \quad (3)$$

$$(n_{while}, n_{tail}): (n_{head}, n_{while}, n_{tail}) \quad (4)$$

$$n_{do} \leftarrow \begin{array}{l} n_{do}: (n_{do}, n_{while}, n_{do}) \end{array} \quad (5)$$

$$n_{tail}: (n_{do}, n_{while}, n_{tail}) \quad (6)$$

$$(n_{while}, n_{do}): (n_{do}, n_{while}, n_{do}) \quad (7)$$

$$(n_{while}, n_{tail}): (n_{do}, n_{while}, n_{tail}) \quad (8)$$



Als nächstes suchen wir Pfade, in denen die angegebenen Teilpfade enthalten sind:

Im Pfad

$$(n_{start}, n_{head}, n_{while}, n_{tail}, n_{final})$$

sind die Teilpfade (2) und (4) enthalten. Der Pfad

$$(n_{start}, n_{head}, n_{while}, n_{do}, n_{while}, n_{tail}, n_{final})$$

enthält die Teilpfade (1), (3), (6) und (8). Die restlichen Teilpfade (5) und (7) sind im Pfad

$$(n_{start}, n_{head}, n_{while}, n_{do}, n_{while}, n_{do}, n_{while}, n_{tail}, n_{final})$$

enthalten.

Dazu gehören folgende assoziierte Testfälle:

$$T_1 := \{((a, b), \text{'Produkt: ' } c) \mid a, b, c \in \mathbb{N}, a \geq 1, b = 1, c = a \cdot b\}$$

$$T_2 := \{((a, b), \text{'Produkt: ' } c) \mid a, b, c \in \mathbb{N}, a \geq 1, b = 2, c = a \cdot b\}$$

$$T_3 := \{((a, b), \text{'Produkt: ' } c) \mid a, b, c \in \mathbb{N}, a \geq 1, b = 3, c = a \cdot b\}$$

Wählen wir nun Eingabedaten aus, beispielsweise (3, 1) für  $T_1$ , (7, 2) für  $T_2$  und (8, 3) für  $T_3$ , so erhalten wir die korrekten Ergebnisse 'Produkt: '3, 'Produkt: '14 und 'Produkt: '24.



### Aufgabe 10.1.3

Wir erhalten Eingabeäquivalenzklassen, indem wir folgende Fälle betrachten (rotations- und spiegelsymmetrische Anordnungen fassen wir zusammen):

E1: Die Kreise haben keinen gemeinsamen Punkt.

E2: Die Kreise berühren sich in einem Punkt.

E3: Die Kreise überlappen sich.

Die Klasse E1 können wir weiter aufteilen in

E1.1: Kreis 1 ist echt in Kreis 2 enthalten.

E1.2: Kreis 2 ist echt in Kreis 1 enthalten.

E1.3: Die von den Kreisen umschlossenen Flächen haben keinen gemeinsamen Punkt.

E3 kann weiter aufgeteilt werden in

E3.1: Die Kreise sind gleich.

E3.2: Die Kreise überlappen sich teilweise.

Als entsprechende Testdaten wählen wir

	X1	Y1	R1	X2	Y2	R2	erwarteter Abstand
zu E1.1	0.0	0.0	1.0	0.0	0.0	2.0	1.0
zu E1.2	0.0	0.0	2.0	0.0	0.0	1.0	1.0
zu E1.3	0.0	0.0	1.0	3.0	0.0	1.0	1.0
zu E2	0.0	0.0	1.0	2.0	0.0	1.0	0.0
zu E3.1	0.0	0.0	1.0	0.0	0.0	1.0	0.0
zu E3.2	0.0	0.0	1.0	0.0	1.0	1.0	0.0

Führen wir die Funktion `Kreisabstand` z.B. mit den Testdaten zu E1.1 aus, so wird der fehlerhafte Funktionswert 0.0 zurückgegeben. Die Funktion ist also nicht korrekt.



## Quellen- und Literaturangaben

Beispiel 2.1.1 und Definition 2.1.3 mit Anmerkungen sind weitgehend [Kl90] entnommen.

Abschnitt 2.5.1 ist weitgehend [ApL00] entnommen.

- [ApL00] H.-J. Appelrath, J. Ludewig: Skriptum Informatik - eine konventionelle Einführung, Teubner Verlag, 2000
- [Kl90] H. Klaeren: Vom Problem zum Programm, Teubner Verlag, 1990
- [Lig02] P. Liggesmeyer: Software-Qualität, Spektrum Akademischer Verlag, 2002
- [My01] G.J. Myers: Methodisches Testen von Programmen, 7. Auflage (unveränd. Nachdruck d. 3. Auflage), R. Oldenbourg Verlag, München 2001
- [OtW04] Th. Ottmann, P. Widmayer: Programmierung mit Pascal, Vieweg+Teubner 2004



## Pascal-Referenzindex

Symbole	
-	40, 42, 53
↑, ^	164, 165
:=	51
*	40, 42, 53
/	42, 53
+	40, 42, 53
<	45, 53
<=	45, 53
<>	45, 53
=	45, 53
>	45, 53
>=	45, 53
<b>A</b>	
abs	41
and	44, 53
arctan	42
array	95, 105
<b>B</b>	
begin	32, 50, 55
boolean	44
<b>C</b>	
char	43
chr	43
const	46
cos	42
<b>D</b>	
dispose	168
div	40, 53
downto	82
<b>E</b>	
else	61
end	32, 50, 55, 109
eof	57
exp	42
<b>F</b>	
false	44
for	81
function	117
<b>I</b>	
if	61
input	48
integer	36, 40
<b>L</b>	
ln	42
<b>M</b>	
maxint	36
mod	40, 53
<b>N</b>	
new	167
not	44, 53
<b>O</b>	
or	44, 53
ord	43
<b>P</b>	
pred	42
procedure	138
program	31
<b>R</b>	
read	56
readln	56
real	36
record	109
repeat	89
<b>S</b>	
sin	42
sqr	41
sqrt	42
string	45, 107
succ	42
<b>T</b>	
then	61
to	82
true	44
type	68
<b>V</b>	
var	49, 150
<b>W</b>	
while	85
write	56
writeln	56



## Stichwortverzeichnis

Symbole	
$\lceil x \rceil$ .....	203
IN .....	37
IN* .....	87
IR .....	37
IR <sub>-</sub> .....	37
IR <sub>+</sub> .....	37
$\uparrow, \wedge$ .....	164, 165
$:=$ .....	51

### A

Abbruchbedingung .....	205
Abnahmetest .....	267
abs .....	41, 42
Abstraktion .....	242
Acht-Damen-Problem .....	243
Adresse .....	150
Algorithmus .....	2, 16
Anweisung .....	2
Ausführung .....	2
Beschreibung .....	2
Determiniertheit .....	13
Effektivität .....	13
Effizienz .....	13
Finitheit .....	13
Terminierung .....	13
all uses-Kriterium .....	300
Diskussion .....	301
Analyse .....	
statische .....	265, 319, 321
and .....	44
Anfangsausdruck .....	81
Anweisung .....	
bedingte .....	60
Eingabe- .....	296
spezielle .....	56, 60
zusammengesetzte .....	55
Anweisungsteil .....	32, 50
Anweisungsüberdeckung .....	
vollständige .....	276
Anweisungsüberdeckungs- grad .....	276
Anwendersoftware .....	26

### Äquivalenzklasse

Ausgabe- .....	307, 308
Eingabe- .....	305
funktionale .....	305
arctan .....	42
array .....	95
ASCII .....	43
Assembler .....	15
Assemblersprache .....	15
Atom .....	282
Aufruf .....	
rekursiver .....	205
Ausdruck .....	51
Ausgabe .....	56
Auswahlstrategie .....	305
Auswertung .....	52

### B

Backtracking-Verfahren .....	245
Baum .....	196
binärer .....	197
Eigenschaften .....	196
leerer .....	199
Operationen .....	200
Bedingungsüberdeckung .....	282
Bedingungs-/Entscheidungsüberdeckung. 282 .....	282
einfache .....	282
Mehrfach- .....	282
minimale Mehrfach- .....	283
Begrenzersymbol .....	34
Betriebsmittel .....	
Zuteilung von .....	27
Betriebssystem .....	26, 27
Bezeichner .....	32, 33
Gültigkeitsbereich .....	152
Bibliothek .....	163
Binärbaum .....	197
balancierter .....	204
binärer Suchbaum .....	197
Bit .....	23
Black-Box-Test .....	266
Blatt .....	196
Block .....	31, 118, 269, 296
Schachtelung .....	153

Blockstruktur	
dynamische .....	158
statische .....	155
boolean .....	44
Boundary interior-Pfadtest .....	287
Byte .....	23

## C

C <sub>0</sub> -Test .....	276
C <sub>1</sub> -Test .....	278
C <sub>2</sub> -Test .....	282
C <sub>3</sub> -Test .....	282
C <sub>4</sub> -Test .....	287
call-by-reference .....	149
call-by-value .....	149
case	
average .....	195
best .....	195
worst .....	195
char .....	43
Coderedundanz .....	162
Compiler .....	15, 321
Computer .....	2
Geschwindigkeit .....	3
Speicher .....	3
Universalität .....	4
Zuverlässigkeit .....	3
cos .....	42
Cursor .....	57, 59
c-use .....	296

## D

Darstellung	
kompakte .....	197
Dateisystem .....	27
Datenfluss-	
darstellung .....	296
Datenstruktur	
dynamische .....	164, 195
rekursive .....	213
statische .....	195

Datentyp .....	40
Aufzählungstyp .....	69
Ausschnittstyp .....	70
Grundtyp .....	70
selbstdefinierter .....	68
Standard- .....	40
strukturierter .....	69
unstrukturierter .....	69
vordefinierter .....	46
Debuggen .....	261
def .....	296
Deklarationsteil .....	32, 148
Dereferenzierung .....	164
dispose .....	168
div .....	40
Division	
durch 0 .....	310
Dokument .....	321
Dokumentation .....	18
Dreieck .....	66
dynamische Blockstruktur .....	158

## E

Editor .....	30
syntaxgesteuerter .....	77, 131, 188
Effektivität .....	13
Effizienz .....	13, 210
Einfügen .....	195
Eingabe .....	56
Eingaben, zulässige .....	261
else .....	60, 63
dangling .....	63
geschachtelt .....	64
Endausdruck .....	81
Endlosschleife .....	81, 89
Entfernen .....	195
Entscheidung .....	280
eof .....	57, 91
EVA-Prinzip .....	241
Existenz .....	158
exp .....	42

## F

Fakultät .....	83, 205, 262
----------------	--------------





Iteration .....	237	Maschinenprogramm .....	15, 16
<b>K</b>		Maschinensprache .....	14
Kanten eines Baumes .....	196	Matrix .....	105
Kinder .....	196	maxint .....	36
Knoten		Median .....	103
innerer .....	196	Mehrfachverwendung .....	162
Tiefe eines .....	198	mod .....	40
Knoten eines Baumes .....	196	Moderator .....	319
Kommentar .....	35, 232	Modul	
kompakte Darstellung .....	197	-konzept .....	163
Konstanten		modulare Programmentwicklung .....	241
Großschreibung .....	74	Modularisierung .....	19
Lebensdauer .....	158	Modultest .....	267
Konstanten-		Muß-Regeln	
definition .....	39, 46	Zusammenfassung .....	238
Kontrollflussgraph .....	269, 321	<b>N</b>	
kompakter .....	270	Nachfolger .....	196
Pfad durch .....	269	Namenskonflikt .....	155
Kontrollvariable .....	81	Namenswahl .....	228
Namenswahl .....	82	von Kontrollvariablen .....	82
Korrektheit .....	17, 260	natürlicher Suchbaum .....	203
Korrektheitsbeweis .....	17	new .....	167
<b>L</b>		Nichtterminal .....	31
label .....	237	nil .....	166
Laufanweisung .....	81	nil .....	166
Laufvariable .....	81	<b>nil</b> .....	166
Laufzeitstack .....	209, 213	not .....	44
Layout .....	64	<b>O</b>	
Lebensdauer .....	158	Objekt .....	19
Lesbarkeit .....	321	Objektprogramm .....	30
Lineare Listen .....	170	odd .....	45
Anfangszeiger .....	171	Operation .....	53
Anker .....	171	Priorität einer .....	53
Aufbau von .....	172	Prioritätsklassen .....	53
Muster zur Bearbeitung .....	176	Operator	
Operationen .....	172	boolescher .....	44
sortierte .....	175	vergleich .....	44
ln .....	42	or .....	44
Lösungsalgorithmus .....	9	ord .....	43
<b>M</b>		out-Parameter .....	139
Manipulation		output .....	32
versteckte .....	162		
Marke .....	237		

## P

Paradigma .....	18	Programmiersprache .....	3
deklarative Programmierung .....	19	höhere .....	15
imperative Programmierung .....	18	problemorientierte .....	15
Parameter .....	9, 56, 296	typisierte .....	40
aktueller .....	116, 119	Programmierstil .....	72, 127, 183, 227
Änderungs- .....	144	Abkürzungen .....	74
Ausgangs- .....	139	Einrückungen .....	75
Eingangs- .....	139	Kann-Regeln .....	73
formale .....	149	Kommentare .....	77, 131, 188
formaler .....	117	Leerzeilen .....	76
in- .....	139	Merkregeln .....	78
inout- .....	144	Muß-Regeln .....	73
out- .....	139	Namenswahl .....	73, 128, 183
Referenz- .....	149	Programmtext-Formatierer .....	131
Regeln für aktuelle .....	120	Programmtext-Layout .....	75, 129, 185
var- .....	150	Seiteneffekte .....	189
Wert- .....	149	Programmierung .....	
-übergabe und Zeigervariable .....	181	deklarative .....	19
ParaPASCAL .....	139	funktionale .....	20
Pfad .....	269	imperative .....	18
Teil- .....	269, 288, 300	logische .....	20
-überdeckung .....	287	objektorientierte .....	19
Pfadtest .....		prozedurale .....	19
boundary interior- .....	287	strukturierte .....	236
Prädikat .....	281	Programmkomplexität .....	210
atomares .....	282	Programmlauf .....	16
pred .....	42	Programmtext-Formatierer .....	77, 187, 232
preorder .....	219	Programmtext-Layout .....	229
Prettyprinter .....	77, 131, 187, 232	Prozedur .....	19, 138
Problem- .....		Änderungsparameter .....	144
beschreibung .....	9, 16	Ausgangsparameter .....	139
klasse .....	9	Eingangsparameter .....	139
spezifikation .....	9, 16	Parameterübergabe .....	139
Problemspezifikation .....	260	rekursive .....	205
procedure .....	138	Prozedur- .....	
Produktspezifikation .....	320	aufruf .....	139
Programm .....	3	deklaration .....	138, 149
-kopf .....	31	kopf .....	149
-layout .....	64	rumpf .....	149
-zerlegung .....	162	prozedurale Zerlegung .....	241
Programm- .....		Beispiel .....	245
analysator .....	265, 319, 321	Prüfling .....	261
pfad .....	269	Pseudo-Code .....	246
Programmentwicklung .....		p-use .....	296
modulare .....	241		

<b>Q</b>		Review . . . . .	265, 319–??
Qualitätssicherung		Rundungsfehler. . . . .	38
analytische. . . . .	259, 264	<b>S</b>	
konstruktive. . . . .	259	Schleife . . . . .	81
Quellprogramm . . . . .	30	Abbruchbedingung . . . . .	81
<b>R</b>		Abbruchkriterium . . . . .	81
RAM . . . . .	24	Auswahl einer geeigneten. . . . .	91
read. . . . .	56	Ersetzen durch andere. . . . .	90
readln . . . . .	48, 56, 57	for. . . . .	81, 85, 91
real. . . . .	36	repeat . . . . .	89, 90, 91
Realweltproblem . . . . .	9, 16	while . . . . .	85, 90, 91
Rechner . . . . .	21	Schleifen-	
Rechnerarchitektur . . . . .	20	bedingung . . . . .	81, 89
Adresse . . . . .	24	durchlauf. . . . .	81
Akkumulator . . . . .	23	rumpf . . . . .	81
ALU . . . . .	22	Schlüsselwort . . . . .	33
Befehlsregister . . . . .	21	Schnittstelle . . . . .	162
Befehlszählregister . . . . .	22	schrittweise Verfeinerung. . . . .	242
CPU. . . . .	22	Seiteneffekt. . . . .	234
Mikroprogrammeinheit . . . . .	22	Seiteneffekte . . . . .	189
Prozessor . . . . .	22	Selektion . . . . .	237
Rechenwerk. . . . .	22	Semantik. . . . .	14
SISD-Architekturen . . . . .	25	Sequenz. . . . .	55, 237
Steuerwerk . . . . .	21	sin. . . . .	42
Zentraleinheit . . . . .	22	Software . . . . .	26
Rechnersystem . . . . .	26	Software Engineering . . . . .	241, 259
record . . . . .	109	Söhne . . . . .	196
selector . . . . .	110	Spaghetti-Programm. . . . .	237
Referenz-		Speicher . . . . .	23
konzept . . . . .	164	sequentieller Zugriff . . . . .	24
übergabe . . . . .	149, 150	wahlfreier/direkter Zugriff . . . . .	24
Reihenfolge		Zugriffsart. . . . .	24
Haupt- . . . . .	219	Zugriffszeit . . . . .	24
symmetrische . . . . .	220	zyklischer Zugriff . . . . .	24
Rekursion . . . . .	205	Speicher-	
Merkregel zur Verwendung . . . . .	226	adresse . . . . .	3, 164
Tiefe der . . . . .	209	kapazität . . . . .	24
rekursive Funktion . . . . .	205	typ. . . . .	24
rekursive Prozedur . . . . .	205	wort. . . . .	24
rekursiver Aufruf. . . . .	205	zelle. . . . .	24
repeat-Anweisung. . . . .	81	zugriff . . . . .	24
repeat-Schleife . . . . .	89, 90, 91	Spezialsymbole. . . . .	33
Abbruchkriterium . . . . .	89	Spezifikation. . . . .	261
Repräsentant . . . . .	262, 305	Problemspezifikation . . . . .	260
Reproduzierbarkeit . . . . .	314	Sprunganweisung . . . . .	237



U	
Überdeckung	
Anweisungs- .....	271, 276
Bedingungs- .....	271
einfache Bedingungs- .....	282
Mehrfach-Bedingungs- .....	282
minimale Mehrfach-Bedingungs- ....	283
Pfad- .....	271
vollständige Pfad- .....	287
Zweig- .....	271, 278, 301
Übersetzer .....	15
V	
var .....	49, 149, 150
Variable .....	18, 296
globale .....	152
indizierte .....	96
Lebensdauer .....	158
lokale .....	121, 152
selektierte .....	110
Variablen-	
benutzung .....	295
berechnende .....	295
globale .....	296
lokale .....	296
prädikative .....	296
definition .....	295
globale .....	296
lokale .....	296
deklaration .....	39, 49
Vater .....	196
Vektor .....	105
Verbund .....	109
Verbundtyp	
als Feld-Komponente .....	113
Vereinbarungen .....	49
Existenz .....	121
Gültigkeitsbereich .....	118
Lebensdauer .....	121
lokale .....	118, 152
Verfahren der analytischen Qualitätssicherung	
264, 266	
analysierende .....	264
Black-Box- .....	266, 303
Nachteile .....	316
Vorteile .....	316
datenflussorientierte .....	266, 295
dynamische .....	264, 265
fehlerorientierte .....	310
funktionsorientierte .....	304
kontrollflussorientierte .....	266, 268
Nachteile .....	315
Vorteile .....	315
statische .....	265, 319
statisches .....	321
strukturorientierte .....	266
verifizierende .....	264
White-Box- .....	266
Verfeinerung .....	241
Beispiel .....	245
datenstrukturorientierte .....	242
funktionsorientierte .....	242
schrittweise .....	242
Verifikation .....	260
versteckte Manipulation .....	162
Verweis .....	150, 164
von-Neumann	
Architektur .....	21
Prinzipien .....	25
Rechner .....	20
Vorgänger .....	196
Vorwärtsdefinition .....	170
W	
Walk-Through .....	320
Werkzeug .....	321
Wertübergabe .....	149
Wertzuweisung .....	51
while-Anweisung .....	81
while-Schleife .....	85, 90, 91
Schleifenbedingung .....	86, 88
White-Box-Test .....	266
Wiederholungsanweisung .....	81
Wiederverwendung .....	163
write .....	57
writeln .....	32, 59
Wurzel .....	196
Z	
Zahlen .....	35

---

Zeiger .....	310
-konzept. ....	164
leerer .....	166
-typ .....	164
-variable .....	164
-variable und Parameterübergabe ....	181
Vergleich von .....	168
Zeigerkonzept .....	164
Zerlegung	
prozedurale .....	241
zero values-Kriterium .....	310
Zufallstest .....	311
Zugriff .....	24
direkter .....	24
sequentieller .....	24
wahlfreier .....	24
zyklischer .....	24
Zuweisung .....	51
Zweig .....	271
Zweigüberdeckung	
vollständige .....	278
Zweigüberdeckungs-	
grad .....	278
test .....	278
zyklomatische Zahl .....	264