

2 Algorithmische Grundkonzepte

Der Begriff des Algorithmus entstammt ursprünglich den Bemühungen, mathematische Berechnungsvorschriften eindeutig zu beschreiben und zu dokumentieren. In der Informatik geht es speziell darum, »*durch Rechner bearbeitbare Aufgaben*« zu beschreiben – Algorithmen sind somit ein abstrakteres Konzept für auf konkreten Rechnern ausführbare Programme.

In diesem Abschnitt werden die Grundkonzepte von Algorithmen und derer Beschreibung vorgestellt. Wir beginnen mit einem intuitiven Zugang zu Algorithmen, aus dem wir Anforderungen an einen formal fundierten Ansatz ableiten werden. Die folgenden Abschnitte stellen einige Voraussetzungen für die Formalisierung von Algorithmen zusammen – wie legt man eine Sprache für Algorithmen fest, wie beschreibt man von Algorithmen zu verarbeitende Daten, wie notiert man elementare arithmetische Berechnungsschritte.

Dieses Kapitel präsentiert die Grundkonzepte von Algorithmen bewusst ohne Zugriff auf eine konkrete Programmiersprache, da diese universell gültig sind und nicht an eine konkrete maschinelle Interpretation gebunden sind.

2.1 Intuitiver Algorithmusbegriff

Der Begriff des Algorithmus ist zentral für die Informatik. Wir werden uns in diesem Abschnitt diesem Begriff von der intuitiven, d.h. nicht mathematisch formalisierten Sichtweise her nähern, bevor wir den Schritt der Formalisierung in den folgenden Abschnitten vollziehen.

2.1.1 Beispiele für Algorithmen

Algorithmen sind als mathematische Berechnungsvorschriften entstanden. Allgemeiner kann man Algorithmen als Vorschriften zur Ausfüh-

rung einer Tätigkeit charakterisieren, wie die folgenden Beispiele verdeutlichen.

Beispiel 2.1
Intuitiver
Algorithmenbegriff

Die folgenden Algorithmen im intuitiven Sinne begegnen uns im täglichen Leben:

- Bedienungsanleitungen,
- Bauanleitungen,
- Kochrezepte.

Diese Ausführungsvorschriften sind für ausführende Menschen gedacht, nicht etwa für Rechner, und sind nicht unbedingt in dem engeren Sinne Algorithmen, wie wir diesen Begriff später benutzen werden. \square

Aus diesen Beispielen lässt sich nun eine intuitive Begriffsbestimmung ableiten, die als erste Konkretisierung des Algorithmenbegriffs dienen wird:

Intuitive
Begriffsbestimmung

Ein *Algorithmus* ist eine präzise (d.h. in einer festgelegten Sprache abgefasste) endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer elementarer (Verarbeitungs-)Schritte.

Die Bedeutung einiger Teile dieser Festlegung wird in den folgenden Abschnitten noch klarer. So bezieht sich die Endlichkeit auf die Tatsache, dass eine Algorithmenbeschreibung eine feste Länge haben muss und nicht einfach »endlos weitergehen« darf. Mit der *festgelegten Sprache* schließen wir vorerst nur in beliebiger »Prosa« verfasste Texte aus; stattdessen gehen wir von festen Beschreibungsmustern aus, wie sie auch in den intuitiven Beispielen zum Teil genutzt werden. Später werden wir hier weiter einschränken.

Beispiel 2.2
Bekannte
Algorithmen

Eine Reihe von Algorithmen in diesem engeren Sinne sind den meisten aus der elementaren Schulmathematik und dem Organisieren von Unterlagen bekannt:

1. Die Addition zweier positiver Dezimalzahlen (mit Überträgen) ist einer der ersten Algorithmen, die in der Schule gelehrt werden:

$$\begin{array}{r}
 33 \\
 + 48 \\
 \hline
 81
 \end{array}$$

2. Ein komplexerer Algorithmus beschreibt den Test, ob eine gegebene natürliche Zahl eine Primzahl ist.
3. Auch das Sortieren einer unsortierten Kartei (etwa lexikographisch) kann als Algorithmus beschrieben werden.
4. Die Berechnung der dezimalen Darstellung der Zahl $e = 2.7182\dots$ hingegen ist ein Spezialfall, auf den wir später noch eingehen werden.

□

Bevor wir uns die Beschreibung von Algorithmen genauer anschauen, werden erst einige grundlegende Begriffe eingeführt. Der erste ist der Begriff der *Terminierung*:

Terminierung

Ein Algorithmus heißt *terminierend*, wenn er (bei jeder erlaubten Eingabe von Parameterwerten) nach endlich vielen Schritten abbricht.

Unser erster Algorithmus zur Addition zweier Dezimalzahlen bestimmt das Ergebnis auch sehr großer Zahlen in endlich vielen Schritten, da er ja jede Stelle der Zahlen (von hinten nach vorne) nur einmal betrachtet und den Übertrag sozusagen »vor sich her schiebt«.

Ein weiterer, ähnlich klingender, aber völlig anders gearteter Begriff ist der Begriff des *Determinismus*. Der Determinismus legt im gewissen Sinne die »Wahlfreiheit« bei der Ausführung eines Verfahrens fest. Er begegnet uns dabei in zwei Spielarten:

Determinismus

- Ein *deterministischer Ablauf* bedeutet, dass der Algorithmus eine eindeutige Vorgabe der Schrittfolge der auszuführenden Schritte festlegt.
- Ein *determiniertes Ergebnis* wird von Algorithmen dann geliefert, wenn bei vorgegebener Eingabe ein eindeutiges Ergebnis geliefert wird – auch bei mehrfacher Durchführung des Algorithmus (natürlich mit denselben Eingabeparametern).

Von Rechnern ausgeführte Programme erfüllen in der Regel beide Eigenschaften – man muss sich sogar Mühe geben, um nichtdeterminierte Effekte zu simulieren, zum Beispiel bei einem Programm, das ein zufälliges Ergebnis etwa beim Würfeln nachbildet. Während nichtdeterminierte Ergebnisse in der Regel unerwünscht sind, sind nichtdeterministische Abläufe gerade beim Entwurf von Algorithmen ein häufig eingesetztes Konzept.

Ein Beispiel für einen nichtdeterministischen Ablauf bildet die folgende Bearbeitungsvorschrift für das Sortieren eines Stapels von Karteikarten:

Beispiel 2.3
*Nichtdeterministischer
Ablauf*

Sortieren: Wähle zufällig eine Karte, bilde zwei Stapel (lexikographisch vor der Karte, lexikographisch nach der Karte), sortiere diese (kleineren) Stapel, füge die sortierten Stapel wieder zusammen.

Das Ergebnis ist auf jeden Fall ein korrekt sortierter Stapel, das Verfahren ist somit *determiniert*.

Das folgende Beispiel zeigt ein nichtdeterminiertes Ergebnis:

Wähle zufällig eine Karte.

□

Determinierte Algorithmen

Wir bezeichnen nichtdeterministische Algorithmen mit determiniertem Ergebnis als *determiniert*. Die folgenden Beispiele sollen diese Begriffe noch einmal verdeutlichen:

Beispiel 2.4 Nichtdeterminierter vs. determinierter Algorithmus

Die folgende Berechnungsvorschrift bildet ein Beispiel für einen nichtdeterminierten Algorithmus:

1. Nehmen Sie eine beliebige Zahl x .
2. Addieren Sie 5 hinzu und multiplizieren Sie mit 3.
3. Schreiben Sie das Ergebnis auf.

Hingegen ist die folgende Berechnungsvorschrift ein Beispiel für einen determinierten, allerdings nichtdeterministischen Algorithmus:

1. Nehmen Sie eine Zahl x ungleich 0.
2. Entweder: Addieren Sie das Dreifache von x zu x und teilen das Ergebnis durch x

$$(3x + x)/x$$

Oder: Subtrahieren Sie 4 von x und subtrahieren das Ergebnis von x

$$x - (x - 4)$$

3. Schreiben Sie das Ergebnis auf.

Nach einigem Nachdenken sieht man, dass dieses Verfahren immer das Ergebnis 4 liefert (im ersten Fall durch Heraus kürzen von x), egal welche Variante man in Schritt 2 gewählt hat. □

Diese Beispiele zeigen, dass man Nichtdeterminiertheit und Nichtdeterminismus nur durch eine im gewissen Sinne »freie Wahlmöglichkeit« bekommen kann, ein Konzept, das einem »mechanisch« arbeitenden Computer natürlich prinzipiell fremd ist.

Eine wichtige Klasse für unsere späteren Überlegungen sind daher deterministische, terminierende Algorithmen. Diese definieren jeweils eine *Ein-/Ausgabefunktion*:

$$f : \text{Eingabewerte} \rightarrow \text{Ausgabewerte}$$

Algorithmen geben eine konstruktiv ausführbare Beschreibung dieser Funktion an. Die Ein-/Ausgabefunktion bezeichnen wir als *Bedeutung* (oder *Semantik*) des Algorithmus. Es kann mehrere verschiedene Algorithmen mit der gleichen Bedeutung geben.

*Ein-/
Ausgabefunktion*

*Bedeutung /
Semantik von
Algorithmen*

Die folgende Aufzählung zeigt Funktionen zu den Algorithmen aus Beispiel 2.2:

Beispiel 2.5
*Funktionen zu den
Beispielalgorithmen*

1. Addition zweier positiver Dezimalzahlen (mit Überträgen)

$$f : \mathbb{Q} \times \mathbb{Q} \rightarrow \mathbb{Q} \text{ mit } f(p, q) = p + q$$

\mathbb{Q} seien hierbei die positiven Rationalzahlen

2. Test, ob eine gegebene natürliche Zahl eine Primzahl ist

$$f : \mathbb{N} \rightarrow \{\mathbf{ja}, \mathbf{nein}\} \text{ mit } f(n) = \begin{cases} \mathbf{ja} & \text{falls } n \text{ Primzahl} \\ \mathbf{nein} & \text{sonst} \end{cases}$$

3. Sortieren einer unsortierten Kartei (etwa lexikographisch)

K Menge von Karteikarten

S_K Menge von sortierten Karteien über K

US_K Menge von unsortierten Karteien über K

$$f : US_K \rightarrow S_K$$

4. Berechnung der Stellen der Zahl $e = 2.7182\dots$

Diese Berechnung ist, da die Darstellung aus unendlich vielen Ziffern besteht, *nicht terminierend!* Im engeren Sinne handelt es sich somit gar nicht um einen Algorithmus, wie wir ihn im Folgenden betrachten werden.

□

2.1.2 Bausteine für Algorithmen

Unserer ersten Begriffsbestimmung für Algorithmen zufolge beschreibt ein Algorithmus ein Verfahren, das einen Bearbeitungsvorgang aus elementaren Schritten zusammensetzt. Bevor wir uns konkreten Sprachen

für die Formulierung von Algorithmen zuwenden, listen wir einige gängige Bausteine für derartige Algorithmenbeschreibungen auf, die auch aus Handlungsvorschriften des täglichen Lebens bekannt sein dürften. Wir werden diese Bausteine anhand einfacher Vorschriften aus Kochrezepten verdeutlichen.

- | | |
|--------------------------------|---|
| <i>Elementare Operationen</i> | <ul style="list-style-type: none"> ❑ Die Basiselemente sind die <i>elementaren Operationen</i>, die ausgeführt werden, ohne näher aufgeschlüsselt zu werden:
»Schneide Fleisch in kleine Würfel.« |
| <i>Sequenzielle Ausführung</i> | <ul style="list-style-type: none"> ❑ Das Hintereinanderausführen von Schritten bezeichnet man als <i>sequenzielle Ausführung</i>:
»Bringe das Wasser zum Kochen, dann gib das Paket Nudeln hinein, schneide das Fleisch, dann das Gemüse.« |
| <i>Parallele Ausführung</i> | <ul style="list-style-type: none"> ❑ Die <i>parallele Ausführung</i> hingegen bedeutet das gleichzeitige Ausführen von Schritten:
»Ich schneide das Fleisch, du das Gemüse.«
Im Gegensatz zur sequenziellen Ausführung verbindet man mit der parallelen Ausführung in der Regel mehrere <i>Prozessoren</i>, also ausführende Subjekte bzw. Maschinen. |
| <i>Bedingte Ausführung</i> | <ul style="list-style-type: none"> ❑ Unter einer <i>bedingten Ausführung</i> verstehen wir einen Schritt, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt wird.
»Wenn die Soße zu dünn ist, füge Mehl hinzu.«
Eine bedingte Ausführung erfordert also einen Test einer Bedingung. Eine Variante der bedingten Ausführung erhält man, indem auch für den negativen Ausgang des Tests ein auszuführender Schritt angegeben wird. |
| <i>Schleife</i> | <ul style="list-style-type: none"> ❑ Unter einer <i>Schleife</i> versteht man die Wiederholung einer Tätigkeit, bis eine vorgegebene Endbedingung erfüllt wird:
»Rühre, bis die Soße braun ist.«
Man bewegt sich also »im Kreise«, bis man etwas fertig gestellt hat – man kann sich den Begriff gut mit der Assoziation einer »Warteschleife« eines Flugzeuges merken. |
| <i>Unterprogramm</i> | <ul style="list-style-type: none"> ❑ Bereits aus dem Sprachgebrauch der Informatik stammt das Konzept des <i>Unterprogramms</i>:
»Bereite Soße nach Rezept Seite 42.« Ein Unterprogramm beschreibt durch einen Namen (oder hier durch eine Seitennummer) eine Bearbeitungsvorschrift, die »aufgerufen« wird, um dann ausgeführt zu werden. Nach Ausführung des Unterprogramms fährt man im eigentlichen Algorithmus an der Stelle fort, an der man zum Unterprogramm gewechselt war. |
| <i>Rekursion</i> | <ul style="list-style-type: none"> ❑ Eines der Kernkonzepte der Informatik ist die <i>Rekursion</i>. Die Rekursion bedeutet die Anwendung desselben Prinzips auf in ge- |

wisser Weise »kleinere« oder »einfachere« Teilprobleme, bis diese Teilprobleme so klein sind, dass sie direkt gelöst werden können. Hier gibt es leider offenbar keine direkten Beispiele aus Kochbüchern, so dass wir uns mit dem folgenden eher künstlichen Beispiel behelfen:

»Viertel das Fleischstück in vier gleich große Teile. Falls die Stücke länger als 2 cm sind, verfare mit den einzelnen Stücken wieder genauso (bis die gewünschte Größe erreicht ist).«

Rekursion wird uns durch das ganze Buch begleiten – sollte das Prinzip bei diesem eher künstlichen Beispiel nicht klar geworden sein, verweisen wir auf die folgenden Abschnitte, in denen Rekursion intensiv behandelt wird.

Bei einer derart langen Auflistung stellt man sich schnell die Frage, ob denn alle diese Konstrukte auch notwendig sind. Diese Frage betrifft das Problem der *Ausdrucksfähigkeit* von Algorithmensprachen und wird uns noch näher beschäftigen. Hier sei nur bemerkt, dass die Konstrukte

*Ausdrucksfähigkeit
von Algorithmensprachen*

- elementare Operationen + Sequenz + Bedingung + Schleifen

ausreichen, um eine genügend ausdrucksstarke Algorithmensprache festzulegen – allerdings genügen hierfür auch andere Kombinationen, wie wir später sehen werden.

2.1.3 Pseudocode-Notation für Algorithmen

Die genannten Bausteine sind dafür geeignet, auf der intuitiven Ebene (also nicht auf der Basis mathematischer Strenge) Algorithmen zu formulieren. Ein Aspekt der Formulierung ist die Nutzung einer genormten, festgelegten Ausdrucksweise. Wir werden eine derartige Ausdrucksweise kennen lernen, die die genannten Bausteine benutzt und auf einer leicht verständlichen Ebene bleibt. Da die Formulierung der Algorithmen nahe an den Konstrukten verbreiteter Programmiersprachen ist, in denen Programme »kodiert« werden, bezeichnet man diese intuitiven Algorithmen auch als Pseudocode-Algorithmen.

*Pseudocode-
Notation für
Algorithmen*

In der Informatik werden derartige Pseudocode-Algorithmen in der Regel unter der Verwendung spezieller englischer Begriffe formuliert. Diese englischen Begriffe sind der Alltagssprache entnommen und haben eine festgelegte Bedeutung für den Ablauf eines Verfahrens und werden daher als Kontroll- oder Schlüsselwörter bezeichnet. So wird eine bedingte Anweisung mit dem englischen **if** für »wenn/falls« eingeleitet. Wir beginnen allerdings mit einer semiformalen Notation, die

Schlüsselwort

angelehnt an das Lehrbuch von Goldschlager/Lister [GL90] vorerst mit deutschen Kontrollwörtern arbeitet, um den Zugang zu den Konzepten zu erleichtern.

Die Erläuterungen erfolgen wieder anhand eines Beispiels aus dem täglichen Leben, nämlich dem *Kaffeekochen*. Die Basisoperationen haben in diesem Beispiel die Form einfacher deutscher Befehlssätze, etwa »koche Kaffee!«.

Sequenz

Notation der Sequenz Die Sequenz, also die zeitliche Abfolge von Schritten, kann man auf verschiedene Arten notieren. Eine Möglichkeit ist es, die Schritte durchnummerieren, um ihre Reihenfolge vorzugeben:

- (1) Koche Wasser
- (2) Gib Kaffeepulver in Tasse
- (3) Fülle Wasser in Tasse

Verfeinerung Diese Notation hat den Vorteil, dass sie gleichzeitig eine elegante Form der *Verfeinerung* von Schritten ermöglicht, indem ein Schritt, zum Beispiel der Schritt (2), durch eine Folge von Einzelschritten, etwa (2.1) bis (2.4), ersetzt wird, die den bisher nur grob beschriebenen Schritt detaillierter, sozusagen »feiner« darstellen. Der Schritt

- (2) Gib Kaffeepulver in Tasse

kann zum Beispiel zu folgender Sequenz verfeinert werden:

- (2.1) Öffne Kaffeeglas
- (2.2) Entnehme Löffel voll Kaffee
- (2.3) Kippe Löffel in Tasse
- (2.4) Schließe Kaffeeglas

Schrittweise Verfeinerung Auf dieser Idee der Verfeinerung beruht das Entwurfsprinzip der *schrittweisen Verfeinerung*, das oft beim Entwurf von Algorithmen eingesetzt wird.

Sequenzoperator ; Um das oft umständliche Durchnummerieren zu vermeiden und um den Aspekt der Sequenz zu verdeutlichen, wird in Pseudocode-Algorithmen der explizite *Sequenzoperator*, notiert als `;`, eingesetzt. Die Notation mit dem Semikolon wurde auch in viele Programmiersprachen übernommen. Unser Beispiel lautet nun wie folgt:

```
Koche Wasser;
Gib Kaffeepulver in Tasse;
Fülle Wasser in Tasse
```

Diese Notation erspart die Durchnummerierung und entfernt auch irrelevante Information aus den Algorithmen – so ist die Tatsache, dass es sich beim Wassereinfüllen um den dritten, und nicht etwa den vierten Schritt handelt, für die Ausführung des Algorithmus nicht relevant.

Bedingte Anweisungen

Bedingte Anweisungen wählen aufgrund einer zu testenden Bedingung einen Schritt aus. Sie werden daher auch als *Auswahl-* oder *Selektions-schritte* bezeichnet. Wie bereits erwähnt, gibt es zwei Varianten: Entweder wird nur bei positivem Test der Schritt ausgeführt (der Schritt also andernfalls übersprungen) oder es gibt eine weitere Angabe eines Schritts für den negativen Testausgang. Notiert werden diese beiden Varianten als

*Bedingte
Anweisungen:
Auswahl / Selektion*

falls Bedingung
dann Schritt

bzw. als

falls Bedingung
dann Schritt a
sonst Schritt b

Ein Beispiel, diesmal aus einem anderen Alltagsbereich, soll den Einsatz der Auswahl verdeutlichen:

falls Ampel rot oder gelb
dann stoppe
sonst fahre weiter

Der Test liefert einen Wahrheitswert »wahr« oder »falsch«, aufgrund dessen die Ausführung des ersten oder zweiten Schritts entschieden wird.

Die Schachtelung mehrerer Selektionen ineinander ermöglicht komplexe Auswahlen unter mehreren Schritten, wie das folgende Beispiel zeigt:

*Schachtelung von
Selektionen*

falls Ampel ausgefallen
dann fahre vorsichtig weiter
sonst falls Ampel rot oder gelb
dann stoppe
sonst fahre weiter

Das Konstrukt **falls ... dann ... sonst ...** entspricht in verbreiteten Programmiersprachen den folgenden Konstrukten:

```

if Bedingung then ... else ... fi
if Bedingung then ... else ... endif
if ( Bedingung ) ... else ...

```

Das letzte Beispiel gibt die Notation in Java wieder. In Programmiersprachen werden also in der Regel die entsprechenden englischen Wörter verwendet, wobei sich die Sprachen darin unterscheiden, ob sie alle Wörter explizit notieren (in Java wird das **then** durch eine Klammerung der Bedingung überflüssig) und ob sie ein spezielles Schlüsselwort (etwa **endif**) zum Abschluss der bedingten Anweisung benutzen. Die erste Variante der Bedingung (Ausführung eines Schrittes nur bei positivem Test) wird jeweils durch Weglassen des **else**-Teiles realisiert.

Schleifen / Iteration

Schleife Die Wiederholung eines Schrittes in einer Schleife wird nach dem folgenden Muster notiert:

```

wiederhole Schritte
bis Abbruchkriterium

```

Das folgende Beispiel, das bereits nahe an einer von Rechnern interpretierbaren Form ist, zeigt das Prinzip anhand einer Suche nach der nächstgrößeren Primzahl:

```

/* nächste Primzahl */
wiederhole
  Addiere 1;
  Teste auf Primzahleigenschaft
bis Zahl Primzahl ist;
  gebe Zahl aus

```

Dieses Beispiel zeigt gleichzeitig die Kombination mehrerer Bausteine mittels Schachtelung, hier eine Sequenz innerhalb einer Schleife. Die inneren Schritte einer Schleife werden als *Schleifenrumpf* bezeichnet.

Varianten der Iteration

Die bisherige Schleifenvariante hat jeweils den Schleifenrumpf ausgeführt, um danach zu testen, ob sie eine weitere Iteration durchführt oder die Schleife abbricht. Der Schleifenrumpf wird somit mindestens einmal durchlaufen. Eine andere verbreitete Notation führt diesen Test *am Beginn* jeden Durchlaufs durch:

```

solange Bedingung
führe aus Schritte

```

Nach der englischen Übersetzung des Schlüsselwortes ist diese Variante insbesondere unter dem Namen *While-Schleife* bekannt. Ein Beispiel für den Einsatz der While-Schleife zeigt folgender Algorithmus:

While-Schleife

/ Bestimmung der größten Zahl einer Liste */*

Setze erste Zahl als bislang größte Zahl;

solange Liste nicht erschöpft

führe aus

Lies nächste Zahl der Liste;

falls diese Zahl > bislang größte Zahl

dann setze diese Zahl als bislang größte Zahl;

gebe bislang größte Zahl aus

Bei While-Schleifen kann der Fall auftreten, dass der Schleifenrumpf keinmal ausgeführt wird, da die Abbruchbedingung bereits vor dem ersten Durchlauf zutrifft.

Die letzte verbreitete Variante ist die Iteration über einen festen Bereich, zum Beispiel über einen Zahlenbereich. Wir notieren diesen Fall wie folgt:

Iteration über festen Bereich

wiederhole für Bereichsangabe

Schleifenrumpf

Nach dem englischen Schlüsselwort sind derartige Schleifen als *For-Schleifen* bekannt. Typische Bereichsangaben wären z.B. »jede Zahl zwischen 1 und 100«, »jedes Wagenrad«, »jeden Hörer der Vorlesung«. Der Bereich, und somit die Zahl der Ausführungen des Schleifenrumpfs, ist – im Gegensatz zu den bisher diskutierten Varianten – bei Beginn der Schleifenausführung festgelegt.

For-Schleife

Die vorgestellten Schleifenkonstrukte entsprechen wieder jeweils Programmiersprachenkonstrukten:

wiederhole ... bis	repeat ... until ...
	do ... while not ...
solange ... führe aus	while ... do ...
	while (...) ...
wiederhole für	for each ... do ...
	for ...do ...
	for (...) ...

Bei der Umsetzung von **wiederhole... bis** mit einem **do-while**-Konstrukt ist jedoch zu beachten, dass dabei die Bedingung negiert wird, da erstere Variante eine Abbruchbedingung erwartet, während bei der letzteren die Schleife so lange ausgeführt wird, wie die Bedingung erfüllt ist.

2.1.4 Struktogramme

Struktogramme ermöglichen eine standardisierte grafische Notation für den Aufbau von Algorithmen mittels Sequenz, Bedingung und Schleife (vergleiche z.B. Duden Informatik [Lek93]).

Die Notation für Struktogramme wird in Abbildung 2.1 eingeführt (es gibt weitere Konstrukte für Mehrfachverzweigungen etc., auf die wir hier nicht eingehen werden). Elementare Aktionen entsprechen beschrifteten Rechtecken. Die Konstrukte können beliebig ineinander geschachtelt werden.

Abbildung 2.1
Notation für
Struktogramme

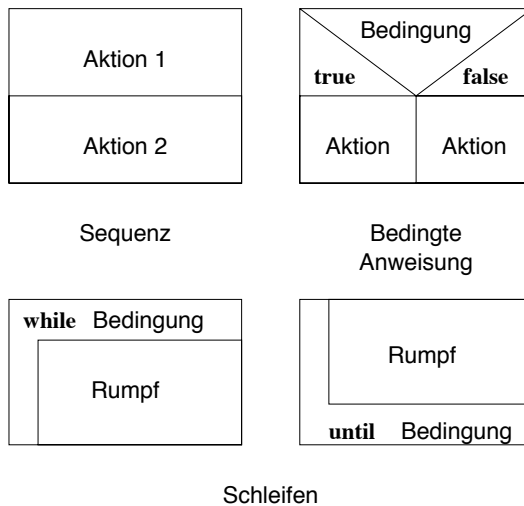
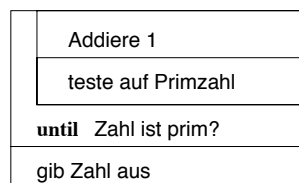


Abbildung 2.2 zeigt den bereits bekannten Algorithmus zur Bestimmung der nächstgrößeren Primzahl als Struktogramm.

Abbildung 2.2
Beispiel eines
Struktogramms



2.1.5 Rekursion

Das Thema Rekursion wird später an mehreren Stellen noch ausführlich behandelt. Da dieser Begriff zentral für den Stoff dieses Buches ist, werden wir ihn nun bereits anhand eines kurzen Beispiels erläutern. Bei

dem Beispiel geht es um die bekannten »Türme von Hanoi« (siehe auch Goldschlager/Lister [GL90] auf den Seiten 57 bis 59 in ausführlicher Beschreibung).

Türme von Hanoi

Bei den Türmen von Hanoi handelt es sich ursprünglich um eine kurze Geschichte, die erfunden wurde, um das Prinzip der Rekursion zu verdeutlichen.

In dieser Geschichte haben Mönche die Aufgabe, einen Turm von 64 unterschiedlich großen goldenen Scheiben zu bewegen. Dabei gelten folgende Regeln:

- ❑ Zu jedem Zeitpunkt können Türme von Scheiben unterschiedlichen Umfangs auf drei Plätzen stehen. Der ursprüngliche Standort wird als Quelle bezeichnet, das Ziel als Senke. Der dritte Platz dient als Arbeitsbereich (kurz AB), um Scheiben zwischenzulagern.
- ❑ Nur jeweils die oberste Scheibe eines Turmes darf einzeln bewegt werden.
- ❑ Dabei darf niemals eine größere auf einer kleineren Scheibe zu liegen kommen.

Die Aufgabe ist nun das Bewegen eines Turmes der Höhe n (etwa $n = 64$ im Originalbeispiel) von einem Standort zu einem zweiten unter Benutzung des dritten Hilfsstandorts (siehe Abbildung 2.3).

Es ist nun gar nicht so einsichtig, in welcher Reihenfolge man Scheiben von wo nach wo bewegen muss, um tatsächlich dieses Ziel zu erreichen (man probiere es selbst mit einem kleinen n , etwa $n = 4$ – statt goldener Scheiben genügen auch Teller oder unterschiedlich große Bücher).

Durch Nachdenken kommt man allerdings zu der Erkenntnis, dass man, sofern man weiß, wie man einen um eins kleineren Turm bewegen kann, auch den größeren Turm bewegen kann. Die Vorgehensweise zeigt folgender Pseudocode-Algorithmus:

Algorithmus 2.1
Türme von Hanoi
(rekursiv)

```

Modul Turmbewegung( $n$ , Quelle, Senke, AB)
  /* Bewegt einen Turm der Höhe  $n$  von Quelle
  nach Senke unter Benutzung des Arbeitsbereichs */
falls  $n = 1$ 
dann bewege Scheibe von Quelle zur Senke
sonst Turmbewegung( $n-1$ , Quelle, AB, Senke)
      bewege 1 Scheibe von Quelle zur Senke
      Turmbewegung( $n-1$ , AB, Senke, Quelle)
  
```

*Rekursives
Verfahren*

Das Prinzip besagt Folgendes: Möchte ich einen Turm der Höhe 5 von A nach B bewegen (unter Zuhilfenahme von C), kann ich das damit erreichen, dass ich einen Turm der Höhe 4 erst von A nach C bewege (jetzt unter Zuhilfenahme von B), dann die größte, fünfte Scheibe direkt nach B lege und den Turm der Höhe 4 zuletzt von C nach B auf diese Scheibe bewege. Das Verfahren heißt *rekursiv*, da sich eine Turmbewegung (der Größe n) unter anderem wiederum durch zwei Turmbewegungen (nun der Höhe $n - 1$) beschreiben lässt.

Der folgende Ablauf verdeutlicht die Vorgehensweise (genauer in Abbildung 2.3). Ziel ist eine Bewegung eines Turmes der Höhe 3 von A nach B. Die Rekursion wird durch Einrückungen verdeutlicht. Es werden nur die Aufrufe der Turmbewegungen (als Turm abgekürzt) und die Scheibenbewegungen notiert.

```
Turm(3, A, B, C)
  Turm(2, A, C, B)
    Turm(1, A, B, C)
      bewege A → B
    bewege A → C
    Turm(1, B, C, A)
      bewege B → C
  bewege A → B
  Turm(2, C, B, A)
    Turm(1, C, A, B)
      bewege C → A
    bewege C → B
    Turm(1, A, B, C)
      bewege A → B
```

Bei 64 Scheiben benötigt man übrigens ca. 600.000.000.000 Jahre, falls man jede Sekunde eine Scheibe bewegen kann (genauer: $2^{64} - 1$ Sekunden!).

*Module als
aufrufbare
Programme*

Das Beispiel zeigte noch eine weitere Besonderheit: Der beschriebene Pseudocode-Algorithmus hat sich quasi selbst als Unterprogramm aufgerufen. Das Schlüsselwort **Modul** startet die Definition eines derartig aufrufbaren Unterprogramms, indem nach ihm der Name des Unterprogramms und eventuelle Parameter für den Aufruf festgelegt werden.

sein. Mit dem Aspekt der Ausführbarkeit werden wir uns bei der Diskussion von Maschinenmodellen intensiver beschäftigen; hier geht es nun um den Aspekt der Verständlichkeit.

Verständlichkeit

Bei der Diskussion von Algorithmen betrachtet man Verstehbarkeit primär im Sinne der Festlegung einer *Sprache*, die von Rechnern interpretiert werden kann. Eine Beschreibung in natürlicher Sprache ist in der Regel mehrdeutig und entspricht somit nicht unseren Anforderungen. Unser Ziel ist es daher, spezielle Sprachen zur Festlegung von Algorithmen zu entwickeln.

Allgemein unterscheidet man bei Sprachen zwei Aspekte:

Syntax legt Muster fest

❑ Die *Syntax* gibt formale Regeln vor, welche Satzmuster gebildet werden können. Ein Beispiel aus der deutschen Sprache:

❑ »Der Elefant aß die Erdnuss.« (syntaktisch korrekt)

❑ »Der Elefant aß Erdnuss die.« (syntaktisch falsch)

Semantik legt Bedeutung fest

❑ Die *Semantik* hingegen legt die Bedeutung fest. Auch hier werden (formale oder informale) Regeln genutzt, um festzulegen, welche Sätze eine *Bedeutung* haben:

❑ »Der Elefant aß die Erdnuss.« (semantisch korrekt, sinnhaft)

❑ »Die Erdnuss aß den Elefanten.« (semantisch falsch, sinnlos)

Festlegung von Syntax und Semantik von Sprachen, insbesondere von Programmiersprachen, ist Inhalt späterer Studienabschnitte eines Informatik-Hauptstudiums und soll hier nicht vertieft werden. Wir präsentieren nur einige kurze Ausführungen, die für die restlichen Abschnitte des Buches benötigt werden. Nur als Bemerkung sei erwähnt, dass bei der Analyse natürlicher Sprache noch weitere Aspekte neben der Semantik und der Syntax, etwa die Pragmatik, untersucht werden müssen, die bei Sprachen, die für Rechner entwickelt wurden, keine besondere Rolle spielen.

2.2.1 Begriffsbildung

Für die Festlegung von Sprachen gibt es einige einfache Konzepte, die wir im Folgenden kurz vorstellen werden.

Grammatik

Eine Grammatik ist ein Regelwerk zur Beschreibung der Syntax einer Sprache. Es gibt unterschiedliche Regelwerke zur Festlegung von Grammatiken, von denen wir mit den Produktionsregeln ein einfaches benutzen werden. Eine Produktionsregel ist eine einfache Regel einer Grammatik zum Bilden von Sätzen, bei der Satzbausteine durch andere

Produktionsregel

Bausteine verfeinert werden. Ein Beispiel aus dem Bereich der natürlichen Sprache ist die folgende Regel:

Satz \mapsto Subjekt Prädikat Objekt.

Die Regeln einer Grammatik legen die so genannte generierte Sprache fest. Die generierte Sprache ist die Menge aller durch Anwendungen der Regeln einer Sprache erzeugbaren Sätze.

Generierte Sprache

Im Folgenden werden wir zwei Formalismen zur Beschreibung einfacher »Kunst«-Sprachen kennen lernen, die im weiteren Verlauf des Buches eingesetzt werden und an denen wir diese eingeführten Begriffe erläutern können.

2.2.2 Reguläre Ausdrücke

Reguläre Ausdrücke bieten einfache Operatoren, um Konstruktionsregeln für Zeichenketten festzulegen:

Reguläre Ausdrücke

- »Worte« sind die nicht weiter zerlegbaren Satzbestandteile, in unseren Beispielen etwa a , b etc.
- Die Sequenz beschreibt das »Hintereinanderschreiben«: pq
- Die Auswahl ermöglicht die Wahl zwischen zwei Alternativen:
 $p + q$
- Die Iteration ermöglicht das Wiederholen von Satzbausteinen: p^* (0-, 1- oder n -mal)
Eine oft genutzte Variante der Iteration schließt die »leere« Iteration aus: p^+ (1- oder n -mal)
- Zusätzlich besteht die Möglichkeit der Klammerung zur Strukturierung. Ansonsten gilt »Punkt- vor Strichrechnung«.

Sequenz

Auswahl

Iteration

Der Einsatz regulärer Ausdrücke soll nun anhand einiger Beispiele erläutert werden:

- Mit L beginnende Binärzahlen über L und O können wie folgt beschrieben werden:

$$L(L + O)^*$$

Eine Binärzahl beginnt also mit einem L , danach kann eine beliebig lange Folge von L und O kommen. Beispiele sind L , LO , $LOLO$, $LOOOOOOLLLL$.

Mit einer kleinen Variation generieren wir zusätzlich auch die O als einzige mit diesem Symbol startende Zahl:

$$O + L(L + O)^*$$

- Ein weiteres Beispiel sind Bezeichner einer Programmiersprache, die mit einem Buchstaben anfangen müssen, nach dem ersten Buchstaben aber auch Ziffern enthalten dürfen:

$$(a + b + \dots + z)(a + b + \dots + z + 0 + 1 + \dots 9)^*$$

Beispiele für generierte Wörter sind *abba*, *mz4*, *u2*; kein Beispiel wäre *124c4u*.

Reguläre Ausdrücke sind für mehrere Bereiche der Informatik relevant: Sie werden zur Festlegung von Datenformaten für Programmeingaben genutzt, definieren Muster zum Suchen in Texten und Suchmasken in Programmiersprachen.

Reguläre Ausdrücke sind ein einfacher Mechanismus zur Festlegung einer generierten Sprache. Im Folgenden werden wir einen komplexeren Mechanismus kennen lernen.

2.2.3 Backus-Naur-Form (BNF)

Backus-Naur-Form
BNF

Nach den einfachen Mustern der regulären Ausdrücke werden wir nun einen Formalismus vorstellen, der eine einfache Festlegung der Syntax von Kunstsprachen ermöglicht, wie sie etwa Programmiersprachen darstellen. Die Backus-Naur-Form (kurz BNF) ist benannt nach den Autoren des ursprünglichen Vorschlags und wird insbesondere zur Festlegung der Syntax von Programmiersprachen genutzt.

Eine Beschreibung einer Syntax in BNF besteht aus Ersetzungsregeln der folgenden Form:

$$\text{linkeSeite} ::= \text{rechteSeite}$$

Die *linkeSeite* ist ein Name für ein zu definierendes Konzept, also eines Satzbausteines. In einer Programmiersprachendefinition könnte ein derartiges Konzept etwa den Namen *Schleife* haben. Die *rechteSeite* gibt nun eine Definition an, die die Form einer Liste von Sequenzen aus Konstanten und anderen, ebenfalls definierten Konzepten (evtl. einschließlich dem zu definierenden!) hat. Die Listenelemente bilden Alternativen und sind durch das Symbol *|* getrennt.

Wir erläutern diese Festlegungen an einem einfachen Beispiel. Es handelt sich wieder um die Bezeichner in einer Programmiersprache, die wir ja bereits mittels regulärer Ausdrücke beschrieben hatten.

```

⟨Ziffer⟩ ::= 1|2|3|4|5|6|7|8|9|0
⟨Buchstabe⟩ ::= a|b|c|...|z
⟨Zeichenkette⟩ ::= ⟨Buchstabe⟩|⟨Ziffer⟩|
                  ⟨Buchstabe⟩⟨Zeichenkette⟩|
                  ⟨Ziffer⟩⟨Zeichenkette⟩
⟨Bezeichner⟩ ::= ⟨Buchstabe⟩|
                  ⟨Buchstabe⟩⟨Zeichenkette⟩

```

Für derartige Definitionen haben sich bestimmte Sprechweisen etabliert. Die definierten Konzepte, die in die Klammern $\langle \rangle$ eingeschlossen sind, werden als Nichtterminalsymbole bezeichnet in Abgrenzung zu den Konstanten, die Terminalsymbole genannt werden. Die Bezeichnung basiert darauf, dass bei diesen Symbolen die Ersetzung mittels Regeln endet (»terminiert«).

*Nichtterminalsymbole
und
Terminalsymbole*

Als etwas komplexeres Beispiel betrachten wir die Syntax für die eingeführten Pseudocode-Algorithmen.

Beispiel 2.6
Syntax für Pseudocode-Algorithmen

```

⟨atom⟩ ::= 'addiere 1 zu x' | ...
⟨bedingung⟩ ::= 'x=0' | ...
⟨sequenz⟩ ::= ⟨block⟩; ⟨block⟩
⟨auswahl⟩ ::= falls ⟨bedingung⟩ dann ⟨block⟩ |
              falls ⟨bedingung⟩ dann ⟨block⟩ sonst ⟨block⟩
⟨schleife⟩ ::= wiederhole ⟨block⟩ bis ⟨bedingung⟩ |
              solange ⟨bedingung⟩ führe aus ⟨block⟩
⟨block⟩ ::= ⟨atom⟩ | ⟨sequenz⟩ | ⟨auswahl⟩ | ⟨schleife⟩

```

Für Atome und Bedingungen müssen jeweils geeignete Konstanten aufgelistet werden, da die Syntax an dieser Stelle nicht festgelegt war. \square

BNF-Syntax-Festlegungen sind insbesondere relevant für die Definition der Syntax für Programmiersprachen und die Definition komplexerer Dateiformate. Die BNF bildet dabei eine spezielle Form kontextfreier Grammatiken (diese werden später im Studium genauer behandelt).

Verbreitet sind Erweiterungen der BNF (oft EBNF für *Extended BNF*). Diese integrieren Elemente regulärer Ausdrücke (optionale Teile mittels [...], Iterationen mittels {...}) in die einfache BNF-Notation (siehe etwa den Eintrag Backus-Naur-Form im Duden Informatik [Lek93]).

*EBNF und
Syntaxdiagramme*

Die verbreiteten *Syntaxdiagramme* bilden eine grafische Variante (siehe ebenfalls in [Lek93]).

2.3 Elementare Datentypen

Datentypen

Die ausführbaren elementaren Schritte eines auf Rechnern ablaufenden Algorithmus basieren meistens auf den Grundoperationen eines *Datentyps*. Während die bisher vorgestellten Bausteine einer Algorithmenbeschreibung den Bearbeitungsprozess steuern, legen Datentypen die zu bearbeitenden Informationseinheiten fest.

Die Beschreibung von Datentypen nimmt später in diesem Buch einen eigenen Abschnitt ein. Um rechnerausführbare Algorithmen beschreiben zu können, benötigt man Datentypen; um Datentypen mit ihren Operationen zu beschreiben, benötigt man wiederum eine Algorithmensprache. Wir lösen diesen Abhängigkeitszyklus dadurch auf, dass wir jetzt einige typische Datentypen vorstellen, die wir beim Leser als bekannt voraussetzen, um uns dann nach der Vorstellung von Algorithmensprachen wieder intensiv mit dem Thema Datentypen zu beschäftigen.

2.3.1 Datentypen als Algebren

Ein Algorithmus verarbeitet Daten, etwa Kontoführungsdaten oder geometrische Angaben. Ein Datentyp soll gleichartige Daten zusammenfassen und die nötigen Basisoperationen zur Verfügung stellen, wie beispielsweise eine Skalierung oder Rotation bei geometrischen Daten. Was ist nun die passende Abstraktion von derartigen Datentypen, wenn man sie mathematisch exakt definieren möchte?

Datentypen als Algebren

Eine passende mathematische Abstraktion für Datentypen sind *Algebren*. Eine Algebra ist eine Wertemenge plus Operationen auf diesen Werten. Ein typisches Beispiel für diese Konzept sind die natürlichen Zahlen \mathbb{N} mit den Operationen $+$, $-$, \cdot , \div etc. Wir betrachten nun den Zusammenhang zwischen Datentypen und Algebren etwas genauer.

Sorten eines Datentyps

Wertemengen eines Datentyps werden in der Informatik als Sorten bezeichnet. Die Operationen eines Datentyps entsprechen Funktionen und werden durch *Algorithmen* realisiert. In der Regel haben wir die Situation einer *mehrsortigen Algebra* vorliegen, also einer Algebra mit mehreren Sorten als Wertebereiche. Ein Beispiel für eine mehrsortige Algebra sind wiederum die natürlichen Zahlen plus die Wahrheitswerte mit den Operationen $+$, $-$, \cdot , \div auf den Zahlen, \neg , \wedge , \vee , ... auf den Wahrheitswerten und $=$, $<$, $>$, \leq , ... als Verbindung zwischen den beiden Sorten.

Mehrsortige Algebren

Die Informatiksichtweise eines Datentyps basiert – im Gegensatz zum auf beliebigen Wertebereichen und Funktionen basierenden mathematischen Konzept der Algebra – auf *interpretierbaren Werten* mit *ausführbaren Operationen* – genauer gesagt *durch Rechner* interpretierbare Wertebereiche und *durch Rechner* ausführbare Operationen.

In den folgenden Abschnitten werden einige Beschreibungsmethoden für Algebren kurz skizziert, eine genauere Betrachtung erfolgt in Kapitel 11.

2.3.2 Signaturen von Datentypen

Ein zentraler Begriff in der Beschreibung eines Datentyps ist der Begriff der Signatur. Eine Signatur ist eine Formalisierung der Schnittstellenbeschreibung eines Datentyps und besteht aus der Angabe der Namen der Sorten und der Operationen. Bei Operationen werden neben dem Bezeichner der Operation auch die Stelligkeit der Operationen und die Sorten der einzelnen Parameter angegeben. Die *Konstanten* eines Datentyps werden als nullstellige Operationen realisiert.

Signatur

Das Beispiel der natürlichen Zahlen verdeutlicht diese Angaben:

```
type nat
sorts nat, bool
functions
  0 :  $\rightarrow$  nat
  succ : nat  $\rightarrow$  nat
  + : nat  $\times$  nat  $\rightarrow$  nat
   $\leq$  : nat  $\times$  nat  $\rightarrow$  bool
  ...
```

Beispiel 2.7

Datentyp für natürliche Zahlen

Der Datentyp `nat` hat zwei Sorten, `nat` und `bool`. Oft ist, wie in diesem Fall, der Name des Datentyps auch der Name einer Sorte – in diesen Fällen wird in der Regel diese Sorte neu definiert, während die anderen Sorten als bereits definiert »importiert« werden.

Das Operationssymbol `succ` steht für die Nachfolgerfunktion *successor*, also den unären Operator »+1«.

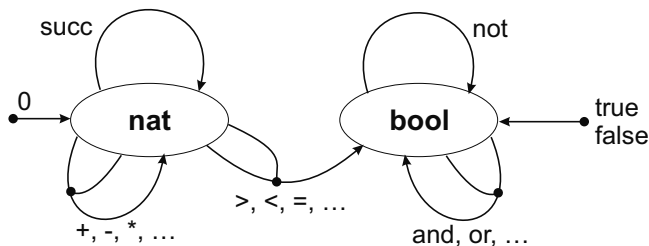
Die Operation `+` hat die Stelligkeit 2, besitzt zwei Parameter vom Typ `nat` und liefert als Ergebnis wiederum einen `nat`-Wert. Die Konstante 0 wird als nullstellige Funktion modelliert, hat also keine Parameter.

Das Beispiel ist angelehnt an die algebraische Spezifikation von Datentypen (Details hierzu in Kapitel 11). \square

Signaturgraphen

Neben der textuellen Variante ist auch die grafische Notation durch *Signaturgraphen* verbreitet. Abbildung 2.4 zeigt den Signaturgraphen für das **nat**-Beispiel. Knoten des Graphen sind die Sorten des Datentyps; Kanten beschreiben die Operationen.

Abbildung 2.4
Signaturgraph für natürliche Zahlen



Nach diesen Vorbemerkungen werden wir einige wenige Datentypen einführen, die in der Definition von Algorithmensprachen und in den Programmierbeispielen der folgenden Kapitel eingesetzt werden.

2.3.3 Der Datentyp **bool**

Datentyp der Wahrheitswerte

Der erste von uns betrachtete Datentyp **boolean**, auch kurz **bool**, ist der Datentyp der Wahrheitswerte. Er ist nach G. Boole (1815-1864) benannt, der als erster Mathematiker die Boole'sche Algebra der Wahrheitswerte formalisierte. Die einzigen zwei Werte von **bool** bezeichnen wir mit den Konstanten {**true**, **false**} für wahr und falsch. Die beiden Werte werden oft auch als 1 und 0 oder auch als *L* und *O* notiert.

Operationen auf **bool**

Die wichtigsten Operationen auf **bool**-Werten sind die folgenden:

- ❑ \neg , auch **not**: logische Negation
- ❑ \wedge , auch **and**: logisches Und
- ❑ \vee , auch **or**: logisches Oder
- ❑ \implies , auch **implies**: logische Implikation ('wenn ... dann ...')

Die Negation \neg hat einen Eingabeparameter, die anderen jeweils zwei. In Abbildung 2.4 auf Seite 36 ist die Signatur von **bool** als Teil des Gesamtgraphens abgebildet.

Da es nur zwei Werte in **bool** gibt, liegt es nahe, die Bedeutung der Operationen direkt durch *Wahrheitstabellen* zu definieren:

Negation

- ❑ Die Negation ersetzt jeden Wert durch sein Gegenstück:

	\neg	
false		true
true		false

- Das logische Und ergibt nur dann wahr, wenn beide Parameter den Wert **true** annehmen: *Logisches Und*

\wedge	false	true
false	false	false
true	false	true

- Für das logische Oder reicht es aus, wenn ein Parameter wahr wird: *Logisches Oder*

\vee	false	true
false	false	true
true	true	true

- Die Implikation $p \implies q$ ist definiert als $\neg p \vee q$:

\implies	false	true
false	true	true
true	false	true

*Logische
Implikation*

Der Datentyp **bool** spielt in der Informatik auch deshalb eine besondere Rolle, da sein Wertebereich genau einem Bit entspricht, der kleinsten möglichen Speichereinheit in einem Rechner.

2.3.4 Der Datentyp **integer**

Der Datentyp **integer**, auch **int**, stellt den Datentyp der ganzen Zahlen dar. Die Werte von **int** bilden somit die folgende (unendliche!) Menge: $\{\dots, -2, -1, 0, 1, 2, 3, 4, \dots\}$.

*Datentyp **int** der
ganzen Zahlen*

Auf **int** sind die bekannten arithmetischen Operationen $+$, $-$, \cdot , \div , **mod**, **sign**, $>$, $<$, \dots etc. definiert. Die wichtigsten werden wir nun kurz aufführen, indem wir die Signatur der Operation sowie eine informelle Erläuterung der Bedeutung angeben:

*Arithmetische
Operationen*

- $+$: **int** \times **int** \rightarrow **int**

Die Addition zweier Zahlen: Z.B. ergibt 3 plus 4 den Wert 7. Wir notieren dies wie folgt: $4 + 3 \mapsto 7$. Analog werden Multiplikation, Subtraktion und ganzzahlige Division definiert.

- **mod**: **int** \times **int** \rightarrow **int**

Der Rest bei der ganzzahligen Division wird mittels eines Operators **mod** (für modulo) definiert. Ein Beispiel ist die folgende Auswertung: $19 \text{ mod } 7 \mapsto 5$.

- **sign**: **int** \rightarrow $\{-1, 0, 1\}$

Das Vorzeichen einer Zahl gibt uns an, ob es sich um eine Zahl aus dem negativen oder dem positiven Zahlenbereich oder um die

Zahl 0 handelt. Die Vorzeichenfunktion ist ein Beispiel für eine einstellige Funktion auf **nat**. Beispiele für Auswertungen dieser Funktion sind **sign**(7) \mapsto 1, **sign**(0) \mapsto 0 und **sign**(-7) \mapsto -1.

□ **>**: **int** \times **int** \mapsto **bool**

Die Größenrelation ist ein Beispiel für eine Operation, die mehrere Sorten (hier **int** und die Wahrheitswerte) betrifft. Ein Beispiel für eine Auswertung ist $4 > 3 \mapsto$ **true**.

□ Der Datentyp **int** umfasst eine Reihe weiterer üblicher Operationen. In den unten aufgeführten Beispielen werden insbesondere folgende Operationen genutzt: **abs** zur Berechnung des Absolutbetrags einer Zahl und die einstelligen Operationen **odd** und **even**, die auf ungerade bzw. gerade Zahlen testen.

Die Operationsnamen **odd** und **even** kann man sich leicht mit folgender Merkregel verinnerlichen: **odd** hat drei Buchstaben und steht für ungerade, **even** hat vier Buchstaben und steht für gerade!

*Notation \mapsto für
Auswertung eines
Operators*

Das Symbol \mapsto bedeutet hierbei »wird ausgewertet zu«. Es wird in dieser Bedeutung auch später im Buch benutzt.

In einem Rechner sind stets nur endlich viele **integer**-Werte definiert. Dies folgt aus der Beschränktheit des Speichers jedes Rechners im Gegensatz zur unendlich großen Wertemenge der mathematischen ganzen Zahlen.

Mit der ganzzahligen Arithmetik haben wir erstmals einen Effekt, der uns später bei Algorithmen öfters beschäftigen wird: Die Operationen haben nicht für alle Auswertungen ein definiertes Ergebnis, etwa bei der Division durch 0. Wir verwenden das Zeichen \perp , wenn das Ergebnis einer Auswertung *undefiniert* ist:

*Undefinierter Wert
 \perp*

$$19 \div 0 \mapsto \perp$$

$$2 \cdot \perp \mapsto \perp$$

$$\perp + 3 \mapsto \perp$$

Wird ein undefinierter Wert mit einem beliebigen anderen Wert verknüpft, ist das Ergebnis wieder undefiniert. Mathematisch bedeutet dies, dass Operationen eines Datentyps im allgemeinen Fall *partielle Funktionen* sind.

2.3.5 Felder und Zeichenketten

Eine ganze Reihe weiterer Datentypen werden im Laufe dieses Buches behandelt. Für die praktischen Übungen, ein paar Beispiele und die Umsetzung in Java sind einige Datentypen besonders relevant, die wir nun kurz skizzieren werden:

- Der Datentyp **char** hat als Wertebereich die Zeichen in Texten $\{A, \dots, Z, a, \dots\}$ mit der einzigen Operation $=$. Als Erweiterung kann man Operationen für die lexikographische Ordnung und eine Nachfolgerfunktion definieren.

*Datentyp **char** für Zeichen*

- Der Datentyp **string** hat als Wertebereich Zeichenketten über Zeichen in **char**.

***string** für Zeichenketten*

Für Zeichenketten sind ein ganze Reihe von Operationen sinnvoll:

- Gleichheit: $=$
- Konkatenation (»Aneinanderhängen«): $+$
- Selektion des i -ten Zeichens: $s[i]$
- Länge: **length**
- jeder Wert aus **char** wird als ein **string** der Länge 1 aufgefasst, wenn er wie folgt notiert wird: "A"
- **empty** oder auch ϵ als leerer **string** der Länge 0
- weitere Operatoren, etwa **substring**

- Der Datentyp **array** erlaubt die Definition von »Feldern« mit Werten eines Datentyps als Eintrag, wobei die Ausdehnung fest vorgegeben ist. Ein Array kann eindimensional oder auch mehrdimensional sein und ist damit das Gegenstück zu den mathematischen Konstrukten Vektor und Matrix.

***array** für Felder*

Beispiele für Werte dieses Datentyps sind im Falle von Feldern über ganze Zahlen die folgenden Werte:

$$\begin{pmatrix} 3 \\ 42 \\ -7 \end{pmatrix}, \quad \begin{pmatrix} 3 & 4 & 0 \\ 42 & 424 & -1 \\ -7 & 7 & 7 \end{pmatrix}$$

In diesen Beispielen sind die Dimensionen jeweils auf 3 Werte begrenzt. Eine Definition der zugehörigen Wertebereiche könnte wie folgt aussehen:

```
array 1..3 of int;
array 1..3, 1..3 of int;
```

Mit den Angaben 1..3 werden die einzelnen Elemente selektierbar oder auch *adressierbar*. Einige Programmiersprachen, so auch Java, ziehen es vor, mit der Adressierung bei 0 anstelle von 1 zu beginnen – das Prinzip bleibt dasselbe.

Typische Operationen auf Feldern sind die folgenden:

- Gleichheit: $=$
- Selektion eines Elementes aus einem Feld A : $A[n]$ oder $A[1,2]$

- Konstruktion eines Feldes: $(1, 2, 3)$ oder $((1, 2, 3), (3, 4, 5), (6, 7, 8))$

*Datentyp-
konstruktoren*

Letzterer Datentyp ist genau genommen ein *Datentypkonstruktor*, da mit ihm Felder verschiedener Ausdehnung, Dimensionalität und verschiedener Basisdatentypen gebildet werden können, die jeweils unterschiedliche Datentypen darstellen. Diese Art von Datentypen wird später noch genauer behandelt; die bisherigen Erläuterungen reichen aber aus, um Felder in Beispielen nutzen zu können.

2.4 Terme

Wir haben bisher Datentypen und Operationen kennen gelernt; nun wollen wir mit diesen auch komplexere Rechnungen ausdrücken, in denen mehrere Operationen genutzt werden. Der Fachausdruck hierfür ist die Bildung von *Termen* und deren Auswertung.

2.4.1 Bildung von Termen

Die Frage nach komplexeren Berechnungen kann man wie folgt umformulieren: *Wie setzt man Grundoperationen zusammen?* In der Mathematik führt dies zur Bildung von Termen, etwa dem folgenden Term

$$7 + (9 + 4) \cdot 8 - 14$$

oder

$$13 - \mathbf{sign}(-17) \cdot 15$$

als Beispiele für ganzzahlige Terme. Der folgende Term zeigt, dass Terme natürlich auch für andere Datentypen, etwa **bool**, gebildet werden können:

$$\mathbf{\neg true} \vee (\mathbf{false} \vee (\mathbf{\neg false} \wedge \mathbf{true}))$$

Diese Beispiele veranschaulichen, dass bei der Termbildung Klammern und Prioritäten zur Festlegung der Auswertungsreihenfolge der Operationen genutzt werden – beim ersten Beispiel wären sonst mehrere Auswertungen (mit jeweils unterschiedlichem Ergebnis!) möglich.

Für Algorithmensprachen werden wir eine weitere Art von Termen kennen lernen, die in normaler Arithmetik nicht eingesetzt werden. *Bedingte Terme* erlauben – analog dem Auswahloperator in der Pseudocode-Notation – die Auswahl zwischen zwei Alternativen basierend auf dem Test eines Prädikats. Notiert wird ein bedingter Term wie folgt:

if b **then** t **else** u **fi**

Hierbei ist b ein boolescher Term und die beiden Terme t und u sind zwei Terme gleicher Sorte.

Die Auswertung bedingter Terme folgt einer bestimmten Regel bezüglich undefinierter Werte. Die folgenden Beispiele zeigen einige Auswertungen:

$$\begin{aligned} \mathbf{if\ true\ then\ } t \mathbf{\ else\ } u \mathbf{\ fi} &\mapsto t \\ \mathbf{if\ false\ then\ } t \mathbf{\ else\ } u \mathbf{\ fi} &\mapsto u \\ \mathbf{if\ true\ then\ } 3 \mathbf{\ else\ } \perp \mathbf{\ fi} &\mapsto 3 \\ \mathbf{if\ false\ then\ } 3 \mathbf{\ else\ } \perp \mathbf{\ fi} &\mapsto \perp \end{aligned}$$

*Auswertung
bedingter Terme*

Im Gegensatz zu allen bisherigen Operationen erzwingt in bedingten Termen ein Teilausdruck, der undefiniert ist, nicht automatisch die Undefiniertheit des Gesamtterms! Dies ist motiviert durch die Auswertungsstrategie, dass nach einem Test nur die ausgewählte Alternative weiter bearbeitet werden sollte – man weiß also gar nicht, ob die andere Alternative eventuell undefiniert ist. Eine tiefer gehende Motivation für diese Regel werden wir allerdings erst im nächsten Kapitel bei der Diskussion applikativer Algorithmen kennen lernen.

Um eine eindeutige Syntax für die Termauswertung vorzugeben, ist eine Formalisierung der Bildung und Auswertung von Termen notwendig. Wir zeigen dies für **int**-Terme in Form einer mathematischen Definition, die erst die erlaubten Konstrukte festlegt und dann alle weiteren Bildungen verbietet.

*Formalisierung von
Termen*

Definition 2.1
*Definition von
int-Termen*

1. Die **int**-Werte $\dots, -2, -1, 0, 1, \dots$ sind **int**-Terme.
2. Sind t, u **int**-Terme, so sind auch $(t + u)$, $(t - u)$, $(t \cdot u)$, $(t \div u)$, **sign**(t), **abs**(t) **int**-Terme.
3. Ist b ein **bool**-Term und sind t, u **int**-Terme, so ist auch **if** b **then** t **else** u **fi** ein **int**-Term.
4. Nur die durch diese Regeln gebildeten Zeichenketten sind **int**-Terme.

□

Eine analoge Definition ist auch für **bool**-Terme notwendig, bei denen dann auch ein Term $t < u$ basierend auf **int**-Termen ein **bool**-Term ist. Durch diese Regeln ist für jeden Ausdruck festgelegt, ob er ein korrekter Term ist und welchen Datentyp das Ergebnis seiner Auswertung hat.

Klammereinsparungsregeln

Die Regeln der Definition 2.1 ergeben vollständig geklammerte Ausdrücke und vermeiden daher jede Mehrdeutigkeit in der Auswertungsreihenfolge. Wir verwenden die üblichen aus der Schulmathematik bekannten Klammereinsparungsregeln:

- Es gelten die üblichen Vorrangregeln: Punktrechnung vor Strichrechnung, \neg vor \wedge vor \vee etc. Der **if**-Konstruktor ist schwächer als alle anderen Operatoren.
- Assoziativgesetze werden in den Fällen, in denen Klammern unnötig sind, da ein identischer Wert als Ergebnis auftritt, ausgenutzt.

Als Resultat können wir für

$$(((\mathbf{abs}((7 \cdot 9) + 7) - 6) + 8) - 17)$$

kurz

$$\mathbf{abs}(7 \cdot 9 + 7) - 6 + 8 - 17$$

schreiben. Der Multiplikationsoperator wird in der Notation oft ebenfalls eingespart, wenn es keine Verwechslung geben kann:

$$2 \cdot (2 + 3) \text{ wird kurz zu } 2(2 + 3)$$

2.4.2 Algorithmus zur TermAuswertung

TermAuswertung

Wir werden später erneut auf Algorithmen zur Auswertung von Termen kommen. Hier wird nur kurz skizziert, wie ein derartiger Algorithmus prinzipiell abläuft. Die Auswertung eines Terms geschieht von innen nach außen (der Klammerstruktur folgend). Es werden also jeweils Terme gesucht, die direkt auswertbar sind (da die Parameterwerte direkt Werte darstellen), und diese werden durch Ausführung der betreffenden Operation ausgewertet und durch das Ergebnis der Auswertung ersetzt. Wie bereits erwähnt, wird bei bedingten Termen zuerst die Bedingung ausgewertet und danach die Auswertung bei der ausgewählten Alternative fortgeführt – hier wird im Gegensatz zu anderen Operatoren also von außen nach innen vorgegangen.

Die Auswertung eines Terms verdeutlicht folgendes Beispiel:

```

1 + if true ∨ ¬false then 7 · 9 + 7 - 6 else abs(3 - 8) fi
↦ 1 + if true ∨ true then 7 · 9 + 7 - 6 else abs(3 - 8) fi
↦ 1 + if true then 7 · 9 + 7 - 6 else abs(3 - 8) fi
↦ 1 + 7 · 9 + 7 - 6
↦ 1 + 63 + 7 - 6
↦ 64 + 7 - 6
↦ 71 - 6
↦ 65

```

Der Algorithmus zur Termauswertung ist in dieser Form nicht-deterministisch, determiniert und terminierend. Als Beispiel für den Nichtdeterminismus kann man folgende Auswertung betrachten: $(7 + 9) \cdot (4 + 10)$ kann über $16 \cdot (4 + 10)$ oder über $(7 + 9) \cdot 14$ zu $16 \cdot 14$ ausgewertet werden. Man kann den Algorithmus deterministisch machen, indem z.B. bei mehreren Möglichkeiten jeweils immer der am weitesten links stehende auswertbare Teilterm ausgewertet wird.

*Eigenschaften der
Termauswertung*

2.5 Datentypen in Java

Den Begriff des Datentyps haben wir bereits in Abschnitt 2.3 als ein Konzept zur Definition von Strukturen und Wertebereichen sowie der zulässigen Operationen für Daten kennen gelernt. Datentypen spielen in Java eine wichtige Rolle, da hier eine strenge Typisierung realisiert ist. Dies bedeutet, dass jede Variable einen wohldefinierten Typ hat, der vor der ersten Verwendung der Variablen auch festgelegt werden muss. Typumwandlungen sind nur unter bestimmten Bedingungen zulässig.

In Java werden zwei Arten von Datentypen unterschieden: primitive Datentypen und Referenzdatentypen.

2.5.1 Primitive Datentypen

Die *primitiven Datentypen* sind die in der Sprache »eingebauten« Typen zur Speicherung von Werten. Die Menge dieser Typen ist dabei statisch und kann auch nicht erweitert werden. Zu den primitiven Datentypen gehören:

- ❑ die numerischen Typen für Ganz- und Gleitkommazahlen,
- ❑ der Zeichen-Datentyp,
- ❑ der boolesche Datentyp.

Ganzzahl-Datentypen

Die Ganzzahl-Datentypen dienen zur Repräsentation ganzzahliger numerischer Werte (so genannter Integerwerte) und sind vorzeichenbehaftet. Es gibt die Typen **byte**, **short**, **int** und **long**, die sich durch die Art der internen Speicherung, d.h. die Anzahl der benutzten Bits, und die sich daraus ergebenden Wertebereiche unterscheiden (Tabelle 2.1).

Gleitkomma-Datentypen

Für numerische Gleitkommawerte gibt es mit **float** und **double** zwei verschiedene Typen für unterschiedliche Wertebereiche und Genauigkeiten (Tabelle 2.1).

Tabelle 2.1
Numerische
Datentypen in Java

Datentyp	Größe	Wertebereich
byte	8 Bit	-128 ... 127
short	16 Bit	-32768 ... 32767
int	32 Bit	$-2^{31} \dots 2^{31} - 1$
long	64 Bit	$-2^{63} \dots 2^{63} - 1$
float	32 Bit	$10^{-46} \dots 10^{38}$ (6 sign. Stellen)
double	64 Bit	$10^{-324} \dots 10^{308}$ (15 sign. Stellen)

Zeichen-Datentyp

Als Zeichen-Datentyp bietet Java den Typ **char**, der einen vorzeichenlosen 16-Bit-Integer-Typ darstellt. Dieser Typ erlaubt die Repräsentation von Zeichen im so genannten Unicode-Zeichensatz, der u.a. auch chinesische und arabische Schriftzeichen unterstützt. Es ist jedoch zu beachten, dass mit einem **char**-Wert nur jeweils ein Zeichen gespeichert werden kann: Zeichenketten (Strings) werden dagegen mit Hilfe der Klasse `java.lang.String` dargestellt, die intern ein Feld von **char**-Werten verwaltet.

Boolescher Datentyp

Der boolesche Datentyp zur Repräsentation von Wahrheitswerten heißt in Java **boolean** und wird als 1-Bit-Wert gespeichert. Mögliche Werte sind dabei **true** und **false**. Der **boolean**-Typ wird insbesondere als Ergebnistyp von logischen Vergleichen angewendet.

Datentypen beschreiben Struktur und Wertebereich von Daten bzw. Werten. Die Werte selbst werden in Variablen gespeichert, die benannte Speicherbereiche (im Fall von primitiven Datentypen) bzw. benannte Verweise auf Speicherbereiche (bei Referenzdatentypen) darstellen. Jeder Variablen ist ein Typ zugeordnet, dem Wert und angewendete Operationen entsprechen müssen. Jede Variable muss in Java vor der Verwendung *deklariert* werden. Diese Deklaration umfasst

Deklaration

- ❑ die Festlegung des Typs,
- ❑ die Initialisierung (explizit durch Angabe eines Wertes oder implizit durch den Standardwert 0) und

- ❑ die Bestimmung der Sichtbarkeit (speziell bei Attributen im Rahmen einer Klassendefinition).

Eine Variable wird in folgender Notation deklariert:

```
typ bezeichner [ = init_wert ];
```

Hierbei ist die Angabe des Initialwertes optional (ausgedrückt durch die eckigen Klammern). Variablen können überall in Methoden oder Anweisungsblöcken vereinbart werden. Im Interesse einer besseren Übersichtlichkeit sollte dies jedoch vorzugsweise zu Beginn eines Blockes erfolgen.

Die Namen oder *Bezeichner* für Variablen (sowie auch Klassen und Methoden) lassen sich frei wählen und unterliegen keiner Längenbeschränkung. Sie dürfen jedoch keine Schlüsselwörter von Java sein und nicht mit einer Ziffer beginnen.

Bezeichner

Ein weiteres Konzept im Zusammenhang mit Variablen sind die *Literale* oder Konstanten. Hierbei kann unterschieden werden in

Literale

- ❑ numerische Werte, wie beispielsweise 42, 23L (für einen **long**-Wert) oder 345.7 und 7.899E+34 für Gleitkommawerte,
- ❑ Zeichen wie 'a' und '\u0012',
- ❑ Zeichenketten, z.B. "Ein String".

Bei der Schreibweise '\uxxxx' handelt es sich um eine Unicode-Escape-Sequenz, wobei xxxx für eine vierstellige Hexadezimalzahl mit dem Code des Zeichens steht. Im folgenden Beispiel ist die Deklaration von Variablen verschiedener primitiver Typen noch einmal illustriert:

```
int eineVariable = 23;
float a, b;
char c1 = 'A', c2 = 'B';
boolean einWahrheitsWert = true;
```

Für die kommende Version *Java SE 7.0* sind noch weitere Möglichkeiten zur Angabe von Literalen vorgesehen. So können Werte auch binär angegeben (z.B. 0b00100110 für den Wert 70) sowie lange Integer-Werte zur besseren Lesbarkeit mit Unterstrich separiert werden:

Java SE 7.0

```
byte einByteWert = (byte) 0b00100110;
long meineKreditkartenNummer = 9876_5432_0123_4567L;
```

2.5.2 Referenzdatentypen

Die zweite Form von Datentypen in Java sind die *Referenzdatentypen*. Variablen dieser Typen enthalten nicht die Daten selbst wie bei den primitiven Datentypen, sondern nur einen Verweis (*Referenz*) auf den Speicherort der Daten. Der Standardwert von Referenzvariablen ist **null**, der besagt, dass die Variable auf kein Objekt verweist.

Bei Referenzdatentypen lassen sich wiederum zwei Arten unterscheiden:

- Feld*
 - ❑ Ein *Feld* (Array) ist eine Datenstruktur fester Größe, die aus Elementen gleichen Typs (sowohl primitive als auch Referenzdatentypen) aufgebaut ist.
- Objektdatentyp*
 - ❑ Ein *Objektdatentyp* dient dagegen zur Repräsentation von Objekten, die wir erst später behandeln werden.

Felder werden bei der Deklaration durch das Anhängen eckiger Klammern »[]« an den Datentyp oder den Bezeichner gekennzeichnet. So werden im Folgenden zwei Referenzvariablen auf Felder von **int**-Werten vereinbart:

```
int einFeld[];
int[] auchEinFeld;
```

Allokation Felder erfordern in Java eine explizite Allokation, d.h., der benötigte Speicherplatz muss bereitgestellt werden. Dies erfolgt durch

- new-Operator*
 - ❑ Aufruf des **new**-Operators, wobei hinter dem Operator der Elementtyp und – in Klammern – die Anzahl der Elemente anzugeben sind. So wird durch die Anweisung

```
int[] feld = new int[20];
```

der Variablen `feld` ein Verweis auf einen Speicherbereich für 20 **int**-Werte zugewiesen.

- Initialisierung*
 - ❑ Initialisierung mit Literalen, die in geschweiften Klammern als Aufzählung angegeben werden. Alle Literale müssen zum Typ der Feldvariablen passen. Die Feldlänge entspricht der Anzahl der angegebenen Literale. Im folgenden Beispiel verweist `feld` daher auf einen Speicherbereich, in dem die drei angegebenen Werte abgelegt sind:

```
int[] feld = { 5, 23, 42 };
```

- Zuweisung*
 - ❑ Zuweisung, wobei hier eigentlich kein Speicherplatz bereitgestellt wird, sondern die Variable nur mit einem Verweis auf den Inhalt der rechten Seite der Zuweisung belegt wird:

```
int[] feld = einAnderesFeld;
```

Eine wichtige Eigenschaft von Referenzvariablen ist die Tatsache, dass Vergleiche (z.B. mittels »==«) und Zuweisungen (über »=«) auf den Referenzen erfolgen. So zeigt im Beispiel in Abbildung 2.5 die Variable `feld2` nach der Zuweisung von `feld1` ebenfalls auf das Feld und nicht etwa auf eine Kopie!

Referenzvariablen

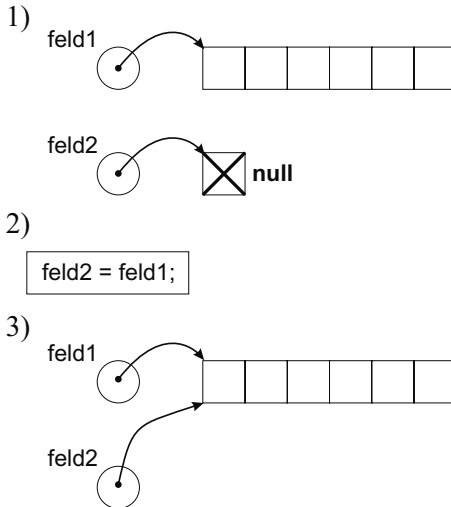


Abbildung 2.5
Zuweisung von
Referenzen

Das Kopieren des Feldes müsste daher entweder durch Anlegen eines neuen Feldes und das anschließende elementweise Kopieren erfolgen oder unter Nutzung der speziellen Methode `arraycopy`, deren Anwendung im folgenden Beispiel demonstriert wird:

*Kopieren von
Feldern*

```
int[] feld1 = { 1, 2, 3, 4, 5 };
int[] feld2 = new int[feld1.length];
int pdest = 0, psrc = 0;
// Kopiert feld1.length Elemente des Feldes feld2
// von Position psrc nach feld1 an Position pdest
System.arraycopy(feld1, psrc, feld2, pdest,
    feld1.length);
```

Der Zugriff auf ein einzelnes Element eines Feldes erfolgt über die Notation `feld[index]`. Hierbei ist zu beachten, dass das erste Element den Index 0 besitzt. Die Länge eines Feldes kann über die Eigenschaft `feld.length` bestimmt werden, so dass der gültige Bereich eines Index des Feldes `0...length-1` beträgt. Zugriffsversuche außerhalb dieses Bereiches werden vom Java-Interpreter erkannt und als Fehler signalisiert.

*Zeichenketten**Erzeugung von Strings*

Bei der Vorstellung des Zeichen-Datentyps haben wir bereits erwähnt, dass Zeichenketten in Java nicht durch einen eigenen Datentyp, sondern in Form der Klasse `java.lang.String` unterstützt werden. Strings sind damit Objekte, die man erzeugen muss und über Methoden manipulieren kann. Genau wie bei Feldern kann die Erzeugung von String-Objekten über den **new**-Operator, durch Initialisierung mit einem Zeichenkettenliteral oder durch Zuweisung eines anderen String-Objektes erfolgen.

```
String s1 = new String("Algorithmen");
String s2 = "Datenstrukturen";
String s3 = s2;
```

String-Methoden

Für die Arbeit mit Strings stehen eine Reihe von Methoden zur Verfügung, so u.a.:

- ❑ **int** `length()` liefert die aktuelle Länge des Strings (in Zeichen).
- ❑ **char** `charAt(int idx)` gibt das Zeichen an der Position `idx` als **char**-Wert zurück.
- ❑ **int** `compareTo(String other)` vergleicht das aktuelle String-Objekt mit dem Objekt `other` und liefert `-1`, wenn es lexikographisch kleiner ist, entsprechend `+1`, wenn es größer ist und `0` bei Gleichheit.
- ❑ **int** `indexOf(int ch)` liefert die Position des ersten Auftretens des Zeichens `ch` im String.
- ❑ **int** `lastIndexOf(int ch)` liefert die Position des letzten Auftretens des Zeichens `ch`.
- ❑ **int** `indexOf(String s)` liefert die Position des ersten Auftretens der Zeichenkette `s` im String.
- ❑ `String replace(char oldC, char newC)` liefert ein neues String-Objekt, das eine Kopie des ursprünglichen Strings ist, wobei alle Zeichen `oldC` durch `newC` ersetzt sind.
- ❑ `String substring(int begin)` liefert ein neues String-Objekt, das aus den Zeichen `begin...length()-1` besteht.
- ❑ `String substring(int begin, int end)` liefert ein neues String-Objekt, das aus den Zeichen `begin...end` besteht.
- ❑ **boolean** `startsWith(String prefix)` prüft, ob der String mit dem Präfix `prefix` beginnt, und liefert in diesem Fall **true**, andernfalls **false**.
- ❑ **boolean** `endsWith(String suffix)` prüft, ob der String mit der Zeichenkette `suffix` endet.

Eine Besonderheit der Klasse `String` ist, dass die Objekte nicht änderbar sind. So liefert beispielsweise die Methode `replace` ein neues Objekt mit der durchgeführten Ersetzung zurück, das ursprüngliche

Objekt bleibt aber unverändert. Allerdings lassen sich mit Hilfe des `+`-Operators sehr einfach String-Objekte aneinander hängen:

Konkatenation

```
s3 = s1 + " und " + s2;
```

Das Ergebnis dieser Anweisung ist ein Objekt `s3` mit der Zeichenkette »Algorithmen und Datenstrukturen«. Die Anwendung der verschiedenen Methoden demonstriert das folgende Beispiel anhand der Ersetzung von »und« durch »&«:

```
String und = "und";
int pos = s3.indexOf(und);
String s4 = s3.substring(0, pos) + "&" +
    s3.substring(pos + und.length());
System.out.println(s4);
```

Es sei angemerkt, dass es noch eine weitere Klasse `java.lang.StringBuffer` gibt, die ähnlich wie `String` aufgebaut ist, jedoch die Änderung der Zeichenkette (Einfügen, Anhängen von Zeichen etc.) erlaubt und in einfacher Weise von und nach `String` konvertiert werden kann.

2.5.3 Operatoren

Zu den Operatoren zählen die bekannten arithmetischen Operatoren `+`, `-`, `*` (Multiplikation), `/` (Division) und `%` (Rest der ganzzahligen Division), wobei der `+`-Operator auch auf Zeichenketten angewendet werden kann und dort das Aneinanderhängen der Operanden bewirkt. Weiterhin gibt es natürlich noch die Vergleichsoperatoren `<`, `>`, `<=`, `>=`, und `==` für Gleichheit bzw. `!=` für Ungleichheit sowie für logische Vergleiche `&&` als Konjunktion und `||` als Disjunktion. Alle diese Operatoren sind binär und werden in der gebräuchlichen Infixnotation verwendet, d.h., der Operator steht zwischen den Operanden: `a op b`.

Arithmetische Operatoren

Vergleichsoperatoren

Weiterhin gibt es Bitoperatoren, die eine bitweise Manipulation von Werten erlauben. Hierzu zählen u.a. das bitweise Verschieben nach links (`<<`), nach rechts (`>>`) sowie die bitweise Und-Verknüpfung (`&`) und die Oder-Verknüpfung (`|`). Alle diese Operatoren interpretieren die Werte der Operanden als Bitfolgen und führen dementsprechend die Manipulation durch. So liefert der Ausdruck `3 << 2` den Wert 12, da 3 binär der Bitfolge 0011 entspricht und diese um 2 Positionen nach links verschoben den Binärwert 1100 (also 12) liefert. Auf diese Weise lassen sich beispielsweise die Zweierpotenzen sehr effizient bestimmen, weil 2^n durch die Bitoperation `1 << n` berechnet werden kann. Die bitweise Verknüpfung mit `&` bzw. `|` kann u.a. genutzt werden, um bestimmte Bits zu maskieren. So liefert etwa der Ausdruck `5`

Bitoperator

& 3 den Wert 1, da 5 in Binärdarstellung 0101 ist und eine bitweise Und-Verknüpfung zwischen 0101 und 0011 den Wert 0001 liefert.

Unäre Operatoren

Zu den unären Operatoren, die nur einen Operanden erfordern, gehören die logische Negation ! sowie die Inkrement- (++) und Dekrementoperatoren (--). Letztere sind der Programmiersprache C entlehnt und erlauben das Erhöhen bzw. Verringern des Wertes des Operanden um 1. Somit ist der Ausdruck `a++` äquivalent zu `a = a + 1`. Zu beachten ist hier, dass die Stellung der Operatoren zum Operanden eine besondere Bedeutung hat. So wird bei der Präfixnotation `++a` erst der Wert von `a` inkrementiert und dann der neue Wert in den Ausdruck zur weiteren Berechnung eingesetzt, während in der Postfixnotation `a++` erst der Wert eingesetzt wird und danach inkrementiert wird. Das folgende Beispiel demonstriert diesen Unterschied. Der Variablen `b1` wird der ursprüngliche Wert von `a1` (hier: 42) zugewiesen. Erst danach erfolgt die Inkrementierung. Dagegen wird `a2` vor der Zuweisung inkrementiert, so dass `b2` den neuen Wert 43 erhält.

Präfixnotation

Postfixnotation

```
int a1 = 42, a2 = 42, b1, b2;
b1 = a1++; // b1 = 42, a1 = 43
b2 = ++a2; // b2 = 43, a2 = 43
```

Zuweisungsoperator

Schließlich gibt es noch den Zuweisungsoperator `»=«` – der nicht mit dem logischen Vergleich verwechselt werden darf – sowie erweiterte Formen davon, die die Kombination mit einem binären Operator in der Notation `op=` erlauben. Hierbei handelt es sich aber nur um eine verkürzte Schreibweise `»a op= x«`, die in der ausführlichen Form als `»a = a op x«` notiert werden kann. So sind die beiden folgenden Anweisungen äquivalent:

```
b += 5;
b = b + 5;
```

Vorrangregeln

Für die Auswertung von Ausdrücken mit diesen Operatoren gelten Vorrangregeln, die den üblichen Rechenregeln (»Punkt- vor Strichrechnung«) folgen und in Tabelle 2.2 noch einmal für die gebräuchlichsten Operatoren zusammengefasst sind. Andere Auswertereihenfolgen lassen sich natürlich durch Klammerung erzwingen.

Die Werte in der Spalte »Vorrang« geben die Reihenfolge an, in der die Operatoren ausgewertet werden. Operatoren mit kleinerem Wert werden dabei zuerst ausgewertet. Demzufolge wird in einem Ausdruck wie `»a = 3 * ++b«` zuerst die Variable `b` inkrementiert, dieser Wert dann mit 3 multipliziert und das Ergebnis schließlich der Variablen `a`

Tabelle 2.2
Wichtige
Operatoren in Java

Vorrang	Operator	Assoz.	Bedeutung
1	++	R	Prä-/Post-Inkrement
	--	R	Prä-/Post-Dekrement
	+, -	R	unäres Plus/Minus
	!	R	logische Negation
	(Typ)	R	explizite Typumwandlung
2	*, /, %	L	Multiplikation, Division, Rest
3	+, -	L	Addition, Subtraktion
	+	L	Konkatenation von Strings
4	«, »	L	Bitweises Verschieben
5	<, <=,	L	Vergleiche
	>, >=		
6	==, !=	L	Gleichheit, Ungleichheit
7	&	L	Bitweises UND
8		L	Bitweises ODER
9	&&	L	Konjunktion
10		L	Disjunktion
11	?:	R	Bedingung
12	=, op=	R	Zuweisungen

zugewiesen. Die Spalte »Assoz.« bezeichnet die Assoziativität der Operatoren, die angibt, in welcher Richtung Operatoren mit gleichem Vorrang ausgewertet werden. Im Normalfall ist dies von links nach rechts (»L«). Speziell die unären Operatoren und die Zuweisungsoperatoren sind jedoch rechts-assoziativ (»R«). Daher sind auch Ausdrücke wie z.B. »a = b = c = 0« möglich.