1 Grundlagen

Programmieren ist eine kreative Tätigkeit. Ein Programm ist ein Artefakt wie ein Haus, ein Bild oder eine Maschine, mit dem einzigen Unterschied, dass es immateriell ist. Seine Bausteine sind nicht Holz, Metall oder Farben, sondern Daten und Befehle. Trotzdem kann das Erstellen eines Programms genauso spannend und befriedigend sein, wie das Malen eines Bildes oder das Bauen eines Modellflugzeugs.

Programmieren bedeutet, einen Plan zur Lösung eines Problems zu entwerfen, und zwar so vollständig und detailliert, dass ihn nicht nur ein Mensch, sondern auch ein Computer ausführen kann. Computer sind ziemlich einfältige Geschöpfe, die zwar schnell rechnen, aber nicht denken können. Sie tun nur das, was wir ihnen ausdrücklich sagen und befolgen diese Befehle dafür auf Punkt und Komma genau. Man muss sich als Programmierer also angewöhnen, genau zu sein, an alle Eventualitäten und Fehlerfälle zu denken und nichts dem Zufall zu überlassen.

Die intellektuelle Herausforderung des Programmierens wird oft unterschätzt. Manche Leute bezeichnen das Programmieren abfällig als stumpfe Codierarbeit oder als Routinetätigkeit. Es ist aber alles andere als einfach oder gar langweilig, besonders wenn man elegante und effiziente Programme schreiben will. Programmieren ist eine höchst anspruchsvolle und kreative Tätigkeit, die nur wenige Leute meisterhaft beherrschen. Ein einfaches Programm zu schreiben ist zwar schnell erlernt, so wie jeder Grundschüler schnell Lesen und Schreiben lernt. Gute Software zu erstellen ist aber eher mit der Tätigkeit eines Schriftstellers zu vergleichen. Jeder Mensch kann schreiben, aber nur ganz wenige Menschen können gut schreiben.

Manche Informatik-Studenten fragen sich, wozu sie eigentlich Programmieren lernen sollen. Ihr Berufsziel ist vielleicht das eines IT-Managers oder eines Software-Beraters. Wozu muss man in diesen Berufen programmieren können? Die Antwort ist einfach: Auch wer später nicht selbst programmiert, muss verstehen, wie Software arbeitet. Anders wird er nicht in der Lage sein, Programmiererteams zu leiten und die Qualität von Software zu beurteilen. So wie ein Architekt über Baustoffe und Verfahren Bescheid wissen muss, so muss auch ein Informatiker das Programmieren als sein Grundhandwerk beherrschen. Ehrlich gesagt, macht Programmieren aber auch Spaß, und das ist nicht zuletzt ein wichtiger Grund, warum die meisten Informatiker gerne programmieren.

1.1 Daten und Befehle

Woraus bestehen eigentlich Programme? Jedes Stück Software besteht aus zwei Grundelementen, nämlich aus Daten und Befehlen.

Programm = Daten + Befehle

Die Daten sind jene Elemente, die das Programm verarbeitet. Das können Zahlen, Texte, aber auch Bilder oder Videos sein. Die Befehle sind die Operationen, die mit den Daten ausgeführt werden. Zum Beispiel gibt es Befehle, um Zahlen zu addieren, Texte zu lesen oder Bilder zu drucken.

Computer beherrschen nur einen sehr eingeschränkten Satz von Daten und Befehlen. Aber aus diesen einfachen Grundelementen lassen sich trotzdem fast unbegrenzt komplexe Anwendungen zusammenbauen. Sehen wir uns einmal die Daten und Befehle in einem Rechner genauer an.

Daten. Die Daten werden im Speicher eines Rechners abgelegt. Ein Speicher besteht aus Zellen, die wir uns wie kleine Schachteln vorstellen können. Jede Zelle enthält ein Datenelement, zum Beispiel eine Zahl. Damit wir die Zellen (von denen es Millionen gibt) einzeln ansprechen können, haben sie eine *Adresse*, die wir uns wie einen *Namen* vorstellen können. Ein Speicher besteht also aus benannten Zellen, die Werte enthalten (siehe Abb. 1.1).

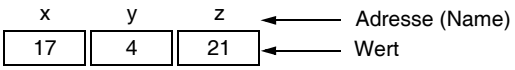


Abb. 1.1 *Speicherzellen*

Die Zelle mit der Adresse `x` enthält in Abb. 1.1 den Wert 17. Die Zelle mit der Adresse `y` enthält den Wert 4. Der Wert einer Zelle kann mit Hilfe von Befehlen geändert werden.

Die Werte in den Speicherzellen sind binär codiert, das heißt, sie bestehen aus Folgen von Nullen und Einsen. Die folgende Tabelle zeigt die Binärdarstellung der ersten 16 natürlichen Zahlen:

0 = 0000	4 = 0100	8 = 1000	12 = 1100
1 = 0001	5 = 0101	9 = 1001	13 = 1101
2 = 0010	6 = 0110	10 = 1010	14 = 1110
3 = 0011	7 = 0111	11 = 1011	15 = 1111

Die Binärdarstellung ist universell, das heißt, es lassen sich damit beliebige Informationen codieren. Neben positiven und negativen Zahlen kann man auch Texte, Bilder, Töne oder Videos binär codieren. Wir gehen hier nicht näher darauf ein; als Programmierer muss man es auch nicht wissen. Das Binärformat ist Sache des Computers. Der Programmierer denkt in höheren Begriffen.

Es sei noch erwähnt, dass eine Binärziffer (0 oder 1) als *Bit* bezeichnet wird. 8 Bits werden zu einem *Byte* zusammengefasst. Je nach Rechnertyp werden 2 oder 4 Bytes als *Wort* bezeichnet und 2 Worte als *Doppelwort*. Ein Rechner arbeitet also intern mit Bits, Bytes, Worten und Doppelworten. Wie wir sehen werden, denkt man als Programmierer aber in anderen Größen, nämlich in Variablen und Objekten.

Befehle. Ein Rechner besitzt eine Hand voll sehr einfacher Befehle, mit denen er die Datenzellen manipulieren kann. Ein einfaches Maschinenprogramm könnte zum Beispiel folgendermaßen aussehen:

ACC ← x	Lade den Wert der Zelle x in ein Rechenregister ACC (Accumulator)
ACC ← ACC + y	Addiere den Wert der Zelle y zu ACC
z ← ACC	Speichere den Wert aus ACC in Zelle z ab (d.h., ersetze den Wert der Zelle z durch den Wert von ACC)

Auch Befehle sind binär codiert, bestehen also aus Nullen und Einsen. Das zeigt, wie universell die Binärcodierung ist. Befehle werden wie Daten im Speicher eines Rechners abgelegt. Das ist bemerkenswert. Ein Programm kann die Befehle eines anderen Programms (ja sogar seine eigenen Befehle) als Daten betrachten. Es kann Programme erzeugen, inspizieren und sogar modifizieren.

Wie bei den Daten gibt es auch bei Befehlen verschiedene Abstraktionsebenen. Ein Programmierer arbeitet nur selten auf der Ebene von Maschinenbefehlen. Er benutzt mächtigere Befehle (so genannte *Anweisungen*) einer Programmiersprache wie *Java*, *C* oder *Pascal*. Die Anweisungen werden aber schlussendlich auf Maschinenbefehle zurückgeführt, denn ein Rechner kann nur Maschinenbefehle verstehen. Die Umsetzung von Anweisungen in Maschinenbefehle wird durch ein Übersetzungsprogramm vorgenommen, das man *Compiler* nennt. Abb. 1.2 zeigt die Schritte, die bei der Entstehung eines Programms von der Spezifikation bis zum Maschinenprogramm durchlaufen werden.

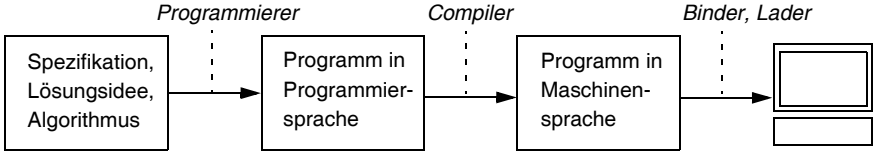


Abb. 1.2 *Entstehungsschritte eines Programms*

Alles beginnt mit einem Problem, das man lösen will, und mit seiner *Spezifikation*, d.h. mit einer genauen Beschreibung dessen, was eigentlich gesucht ist. Aus der Spezifikation ergibt sich eine erste *Lösungsidee* und aus dieser wiederum ein Lösungsverfahren für das Problem, ein so genannter *Algorithmus*. All das sind Vorarbeiten für das Programmieren.

Ein Programmierer erstellt schließlich aus einem oder mehreren Algorithmen ein Programm, das in einer Programmiersprache (z.B. in Java) geschrieben wird. Java-Programme sind zwar für den Menschen gut lesbar, nicht jedoch für einen Rechner, deshalb muss das Java-Programm von einem Compiler in ein Maschinenprogramm übersetzt werden, das aus Nullen und Einsen besteht. Dieses kann nun mit anderen Programmteilen *gebunden*, in den Speicher eines Rechners *geladen* und dort ausgeführt werden.

1.2 Algorithmen

Jedem Programm liegen *Algorithmen* zugrunde. In der Literatur finden sich verschiedene Definitionen dieses Begriffs. Wir verwenden eine sehr einfache, aber für unsere Zwecke völlig ausreichende Definition:

> *Ein Algorithmus ist ein schrittweises, präzises Verfahren zur Lösung eines Problems.*

Algorithmen sind noch keine Programme. Sie können sogar in Umgangssprache beschrieben werden. Ein Kochrezept ist zum Beispiel ein Algorithmus zur Zubereitung eines Gerichts. Eine Wegbeschreibung ist ein Algorithmus, der sagt, wie man von einem Ort zu einem anderen gelangt. In der Informatik stellen wir an Algorithmen jedoch die Forderung, dass sie schrittweise und präzise sein müssen.

Schrittweise bedeutet, dass ein Algorithmus aus einzelnen Schritten besteht, die in genau festgelegter Reihenfolge ausgeführt werden müssen. Dabei darf kein auch noch so nebensächlicher Schritt unerwähnt bleiben. Bedenken Sie: Ein Rechner ist nicht intelligent. Er kann nicht mitdenken.

Aus diesem Grund müssen Algorithmen in der Informatik auch präzise und eindeutig sein. In einem Kochrezept (das für Menschen geschrieben wurde) reicht die Anweisung »gut umrühren«. Für einen Rechner wäre das aber zu wenig präzise. Was heißt »gut«? In der Informatik muss jeder Schritt eines Algorithmus so klar beschrieben sein, dass ein Rechner ihn in eindeutiger Weise ausführen kann.

Nun wird es aber Zeit für ein Beispiel. Nehmen wir an, wir wollen die Summe der natürlichen Zahlen von 1 bis zu einer Obergrenze max berechnen. Ein Algorithmus dafür könnte folgendermaßen aussehen:

Summiere Zahlen (↓max, ↑sum)
1. Setze sum ← 0
2. Setze zahl ← 1
3. Wiederhole Schritt 3, solange zahl ≤ max
   3.1 Setze sum ← sum + zahl
   3.2 Setze zahl ← zahl + 1

Ein Algorithmus besteht aus drei Teilen:

1. Er hat einen *Namen* (Summiere Zahlen), über den man sich auf ihn beziehen kann.
2. Er kann *Eingangswerte* (max) und *Ausgangswerte* (sum) haben. Die Eingangswerte werden von außen (z.B. von einem anderen Algorithmus, dem so genannten *Rufer*) zur Verfügung gestellt und zur Berechnung von Ergebnissen verwendet. Nach Ablauf des Algorithmus kann sich der Rufer die Ergebnisse in den Ausgangswerten abholen. Die Flussrichtung der Parameter deuten wir durch Pfeile an.
3. Ein Algorithmus besteht aus einer Folge von Schritten, die Operationen mit den Eingangswerten und anderen Daten ausführen. In unserem Algorithmus sind die Schritte nummeriert und müssen in der Reihenfolge der Nummern ausgeführt werden. Schritt 3 besteht aus mehreren Teilschritten, die wiederholt ausgeführt werden, bis eine bestimmte Bedingung (hier zahl ≤ max) zutrifft. Dann wird die Wiederholung abgebrochen. Unser Algorithmus würde mit Schritt 4 fortfahren, wenn es ihn gäbe. Da es ihn nicht gibt, ist unser Algorithmus hier zu Ende.

Wir können diesen Algorithmus mit Papier und Bleistift durchspielen und uns für jeden durchlaufenen Schritt die Werte von sum und zahl notieren. Nichts anderes macht ein Computer. Am Ende des Algorithmus steht sein Ergebnis in sum bereit und wird als Ausgangswert an den Benutzer des Algorithmus geliefert.

Programmieren beginnt also damit, dass wir uns für ein gegebenes Problem einen Lösungsalgorithmus überlegen. Aus einem Algorithmus wird ein Programm, indem wir den Algorithmus in einer bestimmten Programmiersprache codieren.

> *Ein Programm ist die Beschreibung eines Algorithmus in einer bestimmten Programmiersprache.*

Ein Algorithmus kann in verschiedenen Programmiersprachen codiert werden, z.B. in Java oder in Pascal. Der Algorithmus ist also das universellere Konstrukt, ein Programm ist nur eine von vielen möglichen Implementierungen davon.

1.3 Variablen

Die Daten eines Programms werden in *Variablen* gespeichert. Eine Variable ist ein benannter »Behälter« für einen Wert. Abb. 1.3 zeigt zwei Variablen x und y mit den Werten 99 und 3.

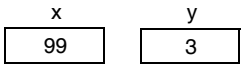


Abb. 1.3 *Variablen als benannte Behälter*

Der Begriff der Variablen ist Ihnen wahrscheinlich aus der Mathematik bekannt. Man sagt, dass die Gleichung

```
x + y = 5
```

zwei Variablen x und y enthält. Aber Vorsicht: In der Mathematik bezeichnen Variablen *Werte*. In der obigen Gleichung stehen x und y für alle Werte, die diese Gleichung erfüllen. In der Informatik hingegen ist eine Variable ein *Behälter* für einen Wert. Einer Variablen x kann man im Laufe eines Programms nacheinander zum Beispiel die Werte 3, 25 und 100 zuweisen. Der Inhalt einer Variablen ist also in der Informatik veränderlich.

Variablen haben nicht nur einen Namen, sondern auch einen *Datentyp*. Der Datentyp legt die Art der Werte fest, die man in einer Variablen speichern kann. In der einen Variablen möchte man zum Beispiel Zahlen speichern, in einer anderen Buchstaben eines Textes. Bildlich kann man sich den Datentyp wie die »Form« des Behälters vorstellen. Da Werte ebenfalls einen Datentyp (und somit eine Form) haben, passen nur jene Werte in eine Variable, deren Typ dem Typ der Variablen entspricht.

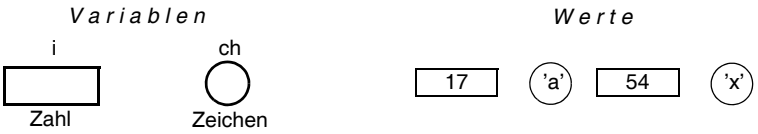


Abb. 1.4 *Variablen und Werte haben einen Datentyp (Form)*

In Abb. 1.4 ist die Variable i ein Behälter für Zahlen, ausgedrückt durch die eckige Form. Es passen z.B. die Werte 17 und 54 hinein, die in unserem Bild ebenfalls eckige Form haben. Die Variable ch ist hingegen ein Behälter für Zeichen, ausgedrückt durch ihre runde Form. Es passen z.B. die Zeichenwerte 'a' und 'x' hinein, die ebenfalls eine runde Form haben.

1.4 Anweisungen

Anweisungen greifen auf Werte von Variablen zu und führen damit die gewünschten Berechnungen durch. Programmiersprachen bieten zwar unterschiedliche Anweisungen an, aber eigentlich gibt es nur sehr wenige Grundmuster, die in den einzelnen Sprachen lediglich abgewandelt werden. Wir sehen uns nun diese Grundmuster an. Dabei verwenden wir keine Programmiersprache, sondern eine grafische Notation, ein so genanntes *Ablaufdiagramm*.

1.4.1 Wertzuweisung

Die häufigste Art einer Anweisung ist eine *Wertzuweisung*. Sie berechnet den Wert eines Ausdrucks und legt ihn in einer Variablen ab. Die Zuweisung

```
y ← x + 1
```

berechnet den Ausdruck x + 1, indem sie den Wert der Variablen x nimmt, 1 dazuzählt und das Ergebnis in der Variablen y abspeichert. Beachten Sie, dass sich der Wert von x dabei nicht ändert. Der alte Wert von y wird hingegen ersetzt durch den Wert des Ausdrucks x + 1. Man liest die Zuweisung als »y wird zu x + 1«.

Auf der linken Seite des Zuweisungssymbols ← muss immer eine Variable stehen, auf der rechten Seite ein Ausdruck aus Variablen oder Konstanten. Folgende Beispiele verdeutlichen das nochmals:

x ← 2	x enthält nun den Wert 2
y ← x + 1	y enthält nun den Wert 3, x hat noch immer den Wert 2
x ← x * 2 + y	x enthält nun den Wert 7 (2 * 2 + 3), y behält den Wert 3 (Der Operator * bedeutet eine Multiplikation)

Folgende Zuweisungen sind hingegen falsch:

3 ← x	auf der linken Seite muss eine Variable stehen (keine Zahl)
x + y ← x + 1	auf der linken Seite muss eine Variable stehen (kein Ausdruck)

1.4.2 Folge (Sequenz)

Man kann mehrere Anweisungen hintereinander schreiben. Sie werden dann in sequenzieller Reihenfolge ausgeführt. Das obige Beispiel zeigte bereits drei in Folge ausgeführte Zuweisungen:

```
x ← 2
y ← x + 1
x ← x * 2 + y
```

Oft deutet man den Programmablauf (den *Steuerfluss*) durch einen Pfeil an, der die Leserichtung vorgibt. Man erhält dadurch ein *Ablaufdiagramm* (siehe Abb. 1.5).

Das Ablaufdiagramm in Abb. 1.5 zeigt noch ein weiteres Beschreibungselement, nämlich eine *Assertion* (Zusicherung). Eine Assertion ist eine Aussage über den Zustand eines Algorithmus oder eines Programms an einer bestimmten Stelle. Sie wird vom Rechner nicht wie eine Zuweisung ausgeführt, sondern dient lediglich als Erläuterung für den Leser. Die Assertion in Abb. 1.5 sagt dem Leser zum Beispiel, dass nach Ausführung der drei Zuweisungen x den Wert 3, y den Wert 4 und z den Wert 12 hat. Die gestrichelte Linie zeigt an, an welcher Stelle die Assertion gilt.

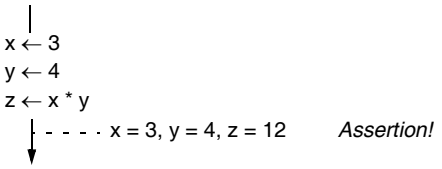


Abb. 1.5 *Ablaufdiagramm einer Sequenz*

1.4.3 Verzweigung (Selektion, Auswahl)

Anweisungen können nicht nur sequenziell hintereinander geschaltet werden, sondern man kann auch ausdrücken, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt werden soll. Wir zeigen das wieder anhand eines Ablaufdiagramms (siehe Abb. 1.6):

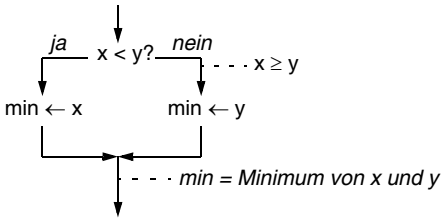


Abb. 1.6 *Ablaufdiagramm einer Verzweigung*

Der Steuerfluss erreicht in Abb. 1.6 zunächst die Abfrage x < y?, die prüft, ob x kleiner ist als y. Das Ergebnis kann »ja« oder »nein« lauten. Je nachdem geht man im Ablaufdiagramm nach links oder nach rechts. Der Steuerfluss verzweigt sich also an dieser Stelle. Ist x < y, werden die Anweisungen im linken Zweig ausgeführt (min ← x), andernfalls die Anweisungen im rechten Zweig (min ← y). Anschließend kommen die beiden Zweige wieder zusammen und der Steuerfluss geht wieder in einem einzigen Zweig weiter.

Was macht der Algorithmus aus Abb. 1.6 eigentlich? Er speichert in der Variablen min das Minimum von x und y, also den kleineren Wert der beiden Variablen. Ist x kleiner als y, geht man nach links und min bekommt den Wert von x, andernfalls geht man nach rechts und min erhält den Wert von y. Die Assertion am Ende des Diagramms gibt nochmals explizit an, was zum Schluss in min gespeichert ist.

Beachten Sie auch die Assertion x ≥ y am Beginn des nein-Zweiges. Da x hier nicht kleiner als y ist, muss gelten, dass es größer oder gleich y ist. Diese Assertion hilft uns beim Verstehen des Algorithmus, denn sie macht sofort deutlich, dass in diesem Zweig y das Minimum der beiden Zahlen ist.

1.4.4 Schleife (Iteration, Wiederholung)

Schleifen erlauben uns auszudrücken, dass eine Folge von Anweisungen mehrmals ausgeführt werden soll, bis eine bestimmte *Abbruchbedingung* eintritt. Abb. 1.7 zeigt, wie eine Schleife als Ablaufdiagramm dargestellt wird.

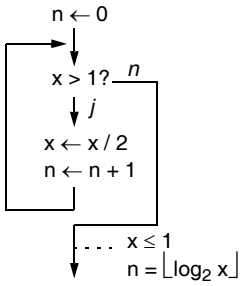


Abb. 1.7 *Ablaufdiagramm einer Schleife*

Nehmen wir an, dass x zu Beginn den Wert 4 enthält. Vor der Schleife wird n auf 0 gesetzt. Die Bedingung x > 1 trifft zu (denn x = 4), also geht man im Diagramm nach unten. x wird ersetzt durch x / 2 (x dividiert durch 2, also 4 / 2 = 2), und n wird zu n + 1, also zu 1.

Nun sehen wir, dass der Pfeil zurückführt und eine Schleife bildet. Wir gelangen wieder zur Abfrage x > 1, die wieder »ja« ergibt, weil x ja nun den Wert 2 hat, also gehen wir im Diagramm wieder nach unten. Die Abfrage und die Anweisungen der Schleife werden also mehrmals durchlaufen. Die Schleife bricht ab, wenn die Bedingung x > 1 nicht mehr zutrifft. In diesem Fall gehen wir im Diagramm nach rechts und verlassen die Schleife. Folgende Tabelle zeigt jeweils die Werte von x und n unmittelbar vor der Abfrage x > 1:

	x	n
1. Besuch	4	0
2. Besuch	2	1
3. Besuch	1	2

Beim dritten Besuch hat x den Wert 1, die Bedingung x > 1 trifft also nicht mehr zu, und die Schleife wird verlassen. n hat den Wert 2. Eine Assertion zeigt den Zustand des Algorithmus am Ende der Schleife. Es gilt hier, dass $x \leq 1$ ist, was sich durch Negation der Schleifenbedingung ergibt. Bei etwas Nachdenken können wir auch angeben, was für n gilt: n ist nämlich der ganzzahlige Logarithmus von x zur Basis 2. Somit erkennen wir auch den Zweck der Schleife: Sie berechnet den ganzzahligen Zweierlogarithmus von x.

Für Schleifen gibt es in Ablaufdiagrammen noch eine andere (kompaktere) Schreibweise (siehe Abb. 1.8). Die Abbruchbedingung wird dabei in ein langgestrecktes Sechseck eingeschlossen. Trifft die Bedingung zu, wird die Schleife betre-

ten. Trifft sie nicht zu, wird sie verlassen, das heißt hinter dem kleinen Kreis an ihrem Ende fortgesetzt. Vom Schleifenende führt ein Weg zurück zur Schleifenbedingung.

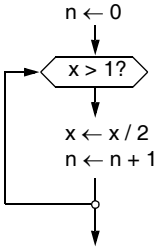


Abb. 1.8 *Schleife als Ablaufdiagramm (andere Darstellungsform)*

Die Schleifen in Abb. 1.7 und Abb. 1.8 sind völlig identisch. Die Schreibweise in Abb. 1.8 ist aber etwas kompakter und drückt besser aus, dass eine Schleife genau 1 Eingang und 1 Ausgang hat.

Schleifen sind für Programmieranfänger sicher die schwierigste Anweisungsart. Man sollte sich ihre Funktionsweise klar machen, indem man einige Schleifen mit konkreten Variablenwerten durchspielt, wie wir das oben getan haben.

Jede Software – von kleinen Beispielprogrammen bis zu komplexen Systemen wie z.B. einer Flugzeugsteuerung – besteht im Wesentlichen nur aus diesen vier Arten von Anweisungen: Zuweisungen, Anweisungsfolgen, Verzweigungen und Schleifen. Natürlich können sie miteinander kombiniert werden (eine Verzweigung kann eine Schleife enthalten, die wieder eine Verzweigung enthält usw.) und natürlich gibt es in den einzelnen Programmiersprachen noch verschiedene Varianten dieser Anweisungsarten. Aber im Wesentlichen haben Sie auf den letzten paar Seiten die Grundelemente des Programmierens kennen gelernt.

1.5 Beispiele für Algorithmen

Wir wollen nun einige Beispiele für Algorithmen betrachten, in denen Zuweisungen, Verzweigungen und Schleifen vorkommen.

1.5.1 Vertauschen zweier Variableninhalte

Gegeben seien zwei Variablen `x` und `y`. Gesucht ist ein Algorithmus, der die Werte der beiden Variablen vertauscht. Wenn `x` also zu Beginn den Wert `3` und `y` den Wert `2` enthält, soll `x` am Ende den Wert `2` und `y` den Wert `3` enthalten. Abb. 1.9 zeigt den Algorithmus als Ablaufdiagramm.

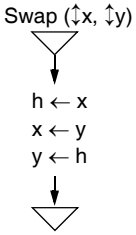


Abb. 1.9 *Algorithmus Swap*

Wir sehen hier ein neues Element eines Ablaufdiagramms: Ein kleines Dreieck deutet den Beginn und das Ende des Algorithmus an. Da es sich in Abb. 1.9 um einen vollständigen Algorithmus handelt, zeigen wir auch seinen Namen (Swap) sowie seine *Parameter*, das heißt die Liste der Eingangs- und Ausgangswerte, die vom Benutzer des Algorithmus übergeben werden und auch wieder an diesen zurückgelangen. Die Doppelpfeile deuten an, dass x und y sowohl Eingangs- als auch Ausgangswerte sind. Sie sind *Übergangswerte*.

Wie funktioniert nun das Vertauschen zweier Variablen. Wir benötigen dazu eine Hilfsvariable h, in die wir den Wert von x retten, so dass x durch den Wert von y ersetzt werden kann. Anschließend speichern wir den geretteten Wert nach y.

Ein *Schreibtischtest* hilft uns, den Algorithmus zu verstehen. Wir legen auf einem Blatt Papier eine kleine Tabelle an, die für jede Variable eine Spalte besitzt. Die erste Zeile der Tabelle füllen wir mit den Anfangswerten der Variablen. Nach jeder Zuweisung an eine Variable tragen wir an das Ende der entsprechenden Spalte den neuen Wert der Variablen ein. Auf diese Weise erhalten wir einen kleinen Papiercomputer, der zwar langsam rechnet, mit dem wir aber jeden Algorithmus durchsimulieren können.

Anfangstabelle

x	y	h
3	2	

h ← x

x	y	h
3	2	3

x ← y

x	y	h
~~3~~ 2	2	3

y ← h

x	y	h
~~3~~ 2	~~2~~ 3	3

1.5.2 Maximum dreier Zahlen berechnen

Gegeben seien drei Zahlen a, b und c. Gesucht ist das Maximum dieser Zahlen, das in der Variablen max gespeichert werden soll. Der Algorithmus ist in Abb. 1.10 dargestellt:

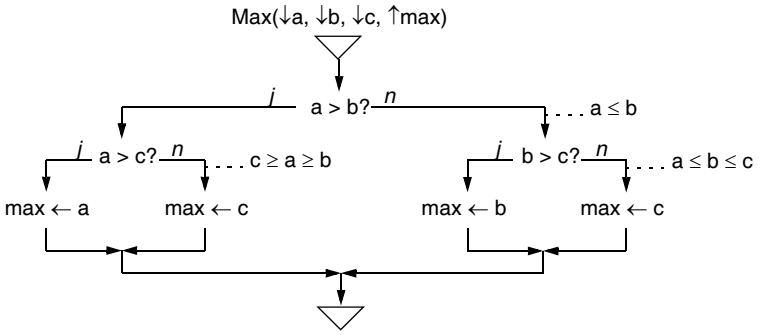


Abb. 1.10 *Algorithmus Max*

Wir sehen hier geschachtelte Verzweigungen. Zuerst wird geprüft, ob a größer als b ist. Wenn ja, wird geprüft, ob a auch größer als c ist. In diesem Fall ist a das Maximum, in anderen Fällen sind weitere Abfragen nötig. Der Algorithmus hat vier Zweige, die alle wieder zusammenkommen und in einem einzigen Zweig an das Ende des Algorithmus fließen.

Wir sehen in Abb. 1.10 auch nochmals den sinnvollen Einsatz von Assertionen. Wenn die Abfrage $a > b$ den Wert »falsch« ergibt, gilt offenbar $a \leq b$. Das ist eine wichtige Erkenntnis und wir schreiben sie als Assertion in das Diagramm. Sie hilft uns, die Logik des Algorithmus zu verstehen. Wenn anschließend auch $b > c$ den Wert »falsch« ergibt, wissen wir, dass $b \leq c$ sein muss. Wir wissen aber auch, dass $a \leq b$ ist, denn das wurde vor Ausführung der inneren Abfrage festgestellt. Daher können wir diese beiden Assertionen kombinieren und erhalten $a \leq b \leq c$. Aus dieser Assertion sehen wir sofort, dass das Maximum der drei Zahlen c ist. Die Assertionen haben uns geholfen, den Algorithmus zu formulieren.

Sie sollten sich angewöhnen, regelmäßig mit Assertionen zu arbeiten. Mit der Zeit wird es für Sie ganz selbstverständlich werden, dass im nein-Zweig einer Abfrage die Negation der Abfragebedingung gilt. Sie werden daher vielleicht eine so einfache Assertion nicht mehr anschreiben, aber Sie sollten sie im Kopf behalten, wenn Sie einen Algorithmus oder ein Programm lesen.

1.5.3 Anzahl der Ziffern einer Zahl bestimmen

Gegeben sei eine positive ganze Zahl n. Gesucht ist die Anzahl ihrer Ziffern. Die Zahl 17 hat z.B. zwei Ziffern, die Zahl 2006 hat vier Ziffern.

Um die Anzahl der Ziffern einer Zahl zu bestimmen, bedienen wir uns eines einfachen Tricks. Wenn wir die Zahl durch 10 dividieren, wird sie um eine Ziffer kürzer. Wir müssen also nur mitzählen, wie oft wir durch 10 dividieren können, bis die Zahl nur noch eine einzige Ziffer enthält, also kleiner als 10 ist. Abb. 1.11 zeigt diesen Algorithmus.

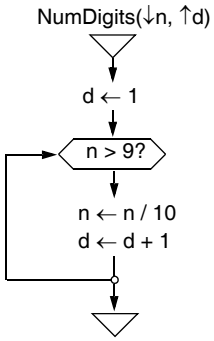


Abb. 1.11 *Algorithmus NumDigits*

Ein Schreibtischtest mit n = 123 zeigt uns, dass der Algorithmus wirklich d = 3 zurückgibt.

n	d
123	1

zu Beginn

n	d
~~123~~	~~1~~
12	2

nach dem ersten Schleifendurchlauf

n	d
~~123~~	~~1~~
~~12~~	~~2~~
1	3

nach dem zweiten Schleifendurchlauf

1.5.4 Größter gemeinsamer Teiler zweier Zahlen

Der folgende Algorithmus ist 2300 Jahre alt. Er wurde vom griechischen Mathematiker *Euklid* um 300 v. Chr. fomuliert. Natürlich verwendete Euklid dazu keine Ablaufdiagramme, sondern eine textuelle Beschreibung. Und natürlich dachte Euklid nicht an eine Umsetzung des Algorithmus in ein Computerprogramm. Aber er definierte das Lösungsverfahren, um für zwei positive ganze Zahlen x und y den größten gemeinsamen Teiler zu berechnen. Dies zeigt, dass Algorithmen »ewige Werte« darstellen können, während Programme oft nur wenige Jahre halten (manchmal nur bis zur nächsten Version der Programmiersprache, in der sie formuliert sind).

Abb. 1.12 zeigt den euklidischen Algorithmus als Ablaufdiagramm. Er berechnet zunächst den Rest der Division von x durch y. Ist dieser Rest 0, so ist y der größte gemeinsame Teiler. Ist der Rest nicht 0, so wird x durch y und y durch den Rest ersetzt. Anschließend wird erneut der Rest der Division x durch y berechnet.

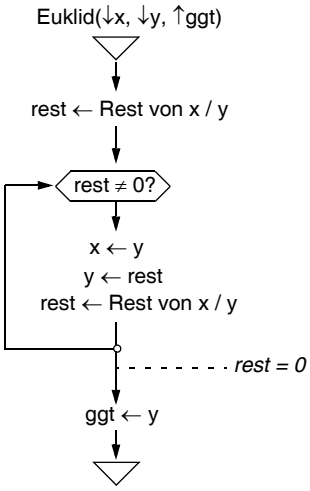


Abb. 1.12 *Euklidischer Algorithmus*

Versichern wir uns zuerst durch einen Schreibtischtest, ob der Algorithmus für x = 28 und y = 20 das richtige Ergebnis, nämlich ggt = 4, liefert, wobei die folgende Tabelle die Werte der Variablen vor Prüfung der Schleifenbedingung zeigt.

x	y	rest
28	20	8

x	y	rest
~~28~~	~~20~~	~~8~~
20	8	4

x	y	rest
~~28~~	~~20~~	~~8~~
~~20~~	~~8~~	~~4~~
8	4	0

Beweist eigentlich der Schreibtischtest, dass unser Algorithmus korrekt ist? Nein, denn er simuliert nur einen von vielen Fällen, in denen x und y ganz unterschiedliche Werte annehmen können. Genau genommen gibt es sogar unendlich viele Kombinationen von x und y. Ein Test kann also (außer in sehr einfachen Fällen) niemals die Korrektheit eines Algorithmus beweisen, allenfalls seine Fehlerhaftigkeit, nämlich dann, wenn der Test ein falsches Ergebnis liefert. Trotzdem sind Tests sinnvoll. Sie fördern das Verständnis des Algorithmus. Außerdem steigt mit zunehmender Anzahl von Testfällen die Wahrscheinlichkeit, dass der Algorithmus korrekt ist.

Wenn wir jedoch die Korrektheit eines Algorithmus *beweisen* wollen, dann müssen wir mathematische Überlegungen anstellen. Wir suchen also eine Zahl ggt mit der Eigenschaft

(ggt teilt x) und (ggt teilt y)

Für diese Zahl muss offenbar auch gelten, dass

ggt teilt (x - y)

denn wenn gilt, dass ggt teilt x, so kann x dargestellt werden als x = i * ggt für irgendein i. In ähnlicher Weise gilt y = j * ggt für irgendein j. Aus x - y = i * ggt - j * ggt = (i - j) * ggt sieht man, dass x - y ebenfalls ein Vielfaches von ggt sein muss.

Wenn wir y von x abziehen können und das Ergebnis immer noch von ggt geteilt wird, so können wir y auch mehrmals von x abziehen, z.B.

ggt teilt (x - q * y)

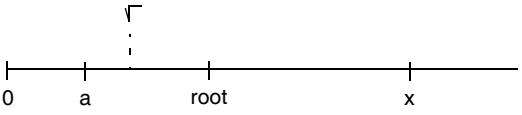
Der Ausdruck x - q * y ist aber nichts anderes als der Rest der Division x / y, wobei q der Quotient ist. Wir haben also jetzt bewiesen, dass ggt auch den Rest der Division x / y teilt. Anders ausgedrückt ist der größte gemeinsame Teiler von x und y auch der größte gemeinsame Teiler von y und dem Rest von x / y. Diese Überlegung rechtfertigt die Ersetzungen, die wir in der Schleife vorgenommen haben.

Anders als ein Test zeigt der Beweis, dass der Algorithmus Euklid für *alle* möglichen Werte von x und y das korrekte Ergebnis liefert. Falls Sie den Beweis schwierig finden und nicht von selbst darauf gekommen wären, so ist das kein Grund zur Beunruhigung. Solche Beweise sind tatsächlich schwierig, besonders für größere Programme. Mit der Zeit bekommt man zwar etwas Übung, aber es macht nichts, wenn man solche Beweise nicht von Anfang an selbst führen kann.

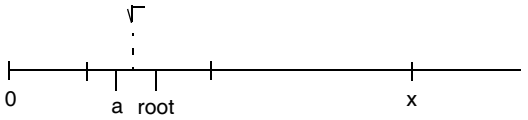
1.5.5 Quadratwurzel von x berechnen

Bis jetzt haben wir immer mit ganzen Zahlen gerechnet. Ein Computer kann aber auch mit Kommazahlen wie 2.75 rechnen. Dabei ist allerdings die Genauigkeit begrenzt. Wie Sie wissen, gibt es reelle Zahlen, deren genauen Wert man nur mit einer unendlich großen Zahl von Kommastellen beschreiben kann. Ein Rechner hat aber für jede Zahl nur eine endliche Anzahl von Kommastellen zur Verfügung, so dass sich beim Rechnen mit Kommazahlen kleine Fehler einschleichen können (mehr darüber in Kapitel 5).

Nun aber zu unserem Problem: Wir haben eine ganze positive Zahl x, z.B. 10, und suchen ihre Quadratwurzel. Ein mögliches Lösungsverfahren ist das folgende: Wir nehmen als ersten Näherungswert an, dass die Wurzel root den Wert x / 2 hat (was noch nicht stimmt). Anschließend berechnen wir eine Zahl a = x / root. Wäre root wirklich die Wurzel von x, so wäre a = root. root ist aber etwas größer als die Wurzel und a etwas kleiner.



Wir müssen daher weiterrechnen. Wir können nun als neuen Näherungswert die Mitte zwischen root und a berechnen, also root = (root + a) / 2. Der neue Wert von a ergibt sich wieder als a = x / root.



Wir sehen, dass root noch immer nicht genau der Wurzel von x entspricht. Es ist wieder etwas zu groß und a ist wieder etwas zu klein. Das Intervall zwischen a und root ist aber kleiner geworden und irgendwo dazwischen liegt der gesuchte Wert. Wir können dieses Verfahren nun fortsetzen, bis das Intervall so klein geworden ist, dass wir mit hinreichender Genauigkeit sagen können, dass a = root ist. In diesem Fall entspricht root auch mit hinreichender Genauigkeit der Wurzel von x.

Abb. 1.13 zeigt den Algorithmus SquareRoot, der die Wurzel von x nach diesem Näherungsverfahren berechnet.

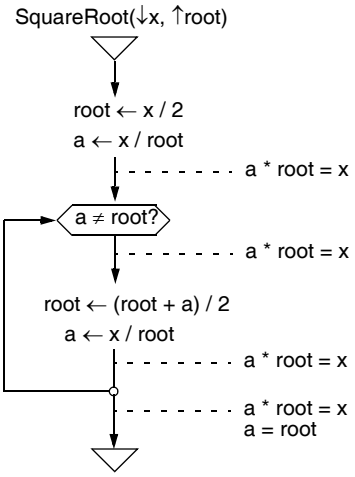


Abb. 1.13 *Algorithmus SquareRoot*

Aus den Assertionen sieht man, dass an den gekennzeichneten Stellen a * root = x gilt. Am Ende des Algorithmus gilt zusätzlich noch a = root, denn sonst wäre die Schleife nicht verlassen worden. Daraus folgt, dass root * root = x ist und root also tatsächlich die Wurzel von x.

Wie bereits gesagt, wird das Intervall zwischen a und root immer kleiner, aber nie 0. Daher sollte man die Schleifenbedingung besser als |a - root| > 0.00000001 schreiben, wenn man die Wurzel auf 8 Kommastellen genau berechnen will.

Ein Schreibtischtest überzeugt uns wieder, dass der Algorithmus terminiert und das Intervall zwischen root und a tatsächlich kleiner wird.

x	root	a
10	5	2
	3.5	2.85714
	3.17857	3.14607
	3.16232	3.16223
	3.16228	3.16228

1.6 Beschreibung von Programmiersprachen

Wir kommen nun zu einem ganz anderen Thema, nämlich zur Frage, wie man Programmiersprachen beschreiben kann. Wir wollen ja die Programmiersprache Java erlernen und daher ist es wichtig, eine Notation zu haben, die uns sagt, wie Java-Programme aussehen oder anders gesagt, welche Texte korrekte Java-Programme sind.

Eine Programmiersprache ist in gewisser Hinsicht ähnlich aufgebaut wie eine natürliche Sprache (z.B. wie Deutsch) und kann durch eine *Grammatik* beschrieben werden. Wie bei natürlichen Sprachen unterscheiden wir zwischen *Syntax* und *Semantik* der Sprache.

1.6.1 Syntax

Die Syntax einer Sprache gibt Regeln an, wie die Sätze dieser Sprache gebaut sein müssen. Im Deutschen besteht ein gewöhnlicher Hauptsatz zum Beispiel aus einem Subjekt, einem Prädikat und einem Objekt. Ähnlich kann man für eine Programmiersprache definieren, dass eine Zuweisung aus einer Variablen, einem Zuweisungssymbol und einem Ausdruck besteht. Man schreibt dann

```
Zuweisung = Variable "←" Ausdruck.
```

Diese Syntaxregel besteht aus einer linken und einer rechten Seite, die durch ein Gleichheitszeichen getrennt sind. Die linke Seite besagt, welches Sprachkonstrukt die Regel beschreibt, die rechte Seite gibt an, wie dieses Sprachkonstrukt aufgebaut ist. Jede Regel wird durch einen Punkt abgeschlossen.

1.6.2 Semantik

Um eine Sprache zu beschreiben, genügt es nicht, ihre Syntax zu definieren. Man muss auch sagen, was die Sätze der Sprache bedeuten. Dies bezeichnet man als Semantik. Die Semantik von

```
Zuweisung = Variable "←" Ausdruck.
```

ist: Werte den Ausdruck aus und weise ihn der Variablen zu. Ohne diese Erklärung könnte der Satz Beliebiges bedeuten. Die Semantik von Programmiersprachen lässt sich leider formal nur sehr kompliziert beschreiben. Daher geben wir sie hier immer textuell an.

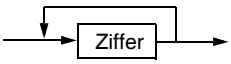
1.6.3 Grammatik

Eine Grammatik ist eine Menge von Syntaxregeln, die sämtliche Konstrukte einer Sprache beschreiben. Zum Beispiel kann man die Grammatik der Dezimalzahlen wie folgt formulieren:

```
Ziffer = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "6" | "7" | "8" | "9".
Zahl = Ziffer {Ziffer}.
```

Der senkrechte Strich ("|") bedeutet dabei *oder*. Er trennt Alternativen voneinander. Eine Ziffer ist also entweder "0" oder "1" oder ... oder "9"; die Hochkommas besagen, dass die eingeschlossenen Texte genau so im Satz vorkommen müssen, wie sie geschrieben sind. Die zweite Grammatikregel enthält geschweifte Klammern. Diese bedeuten eine null-, ein- oder mehrmalige Wiederholung des geklammerten Konstrukts. Eine Zahl besteht also immer aus einer Ziffer, auf die null, eine oder mehrere weitere Ziffern folgen können. Damit können wir beliebige Zahlen wie 1, 57, 38462 usw. beschreiben.

Grammatikregeln werden manchmal auch als *Syntaxdiagramme* angegeben. Die Regel für Zahl sieht dann folgendermaßen aus:



Indem man den Pfeilen entlangfährt, erhält man eine korrekte Folge von Symbolen. Da Syntaxdiagramme aber in der Regel mehr Platz einnehmen als eine Grammatikregel, werden wir in Zukunft Grammatikregeln bevorzugen.

Die oben gezeigte Grammatikschreibweise mit dem Gleichheitszeichen, dem Punkt, den Alternativenstrichen und den geschweiften Klammern nennt man *EBNF* (Erweiterte Backus-Naur-Form, nach den Informatikern *John Backus* und *Peter Naur*, die an der Entwicklung einflußreicher Sprachen wie *Fortran* und *Algol* beteiligt waren). Folgende Tabelle zeigt sämtliche EBNF-Metazeichen (d.h., die zur Grammatikschreibweise gehörenden Zeichen) und ihre Bedeutung.

Metazeichen	*Bedeutung*	*Beispiel*	*beschreibt*
=	trennt Regelseiten		
.	schließt Regel ab		
|	trennt Alternativen	x | y	x, y
()	klammert Alternativen	(x | y) z	xz, yz
[]	wahlweises Vorkommen	[x] y	xy, y
{}	0..n-maliges Vorkommen	{x} y	y, xy, xxy, xxxy, …

Das folgende Beispiel einer EBNF-Regel zeigt eine Definition von Gleitkommazahlen (z.B. 0.314E+1, was so viel bedeutet wie $0.314 * 10^1$).

```
Gleitkommazahl = Zahl "." Zahl ["E" ["+"|"-"] Zahl].
```

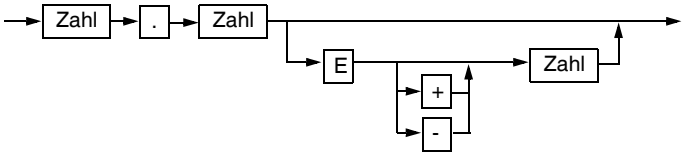
Diese Regel benutzt zwei Arten von Symbolen. Die einfachen Symbole wie ".", "E", "+" oder "-" werden *Terminalsymbole* genannt, weil sie sich selbst bedeuten, d.h. nicht mehr weiter zerlegt werden können. Das Symbol Zahl ist hingegen ein *Non-*

terminalsymbol, weil man es weiter zerlegen kann (siehe die weiter oben stehende Grammatikregel für Zahl). Auf der linken Seite einer Grammatikregel steht immer ein Nonterminalsymbol. Die rechte Regelseite besteht aus Terminalsymbolen, Nonterminalsymbolen und Metazeichen wie "|" oder "{".

Die Regel für Gleitkommazahl enthält eckige Klammern als Metazeichen. Eine eckige Klammer bedeutet, dass das geklammerte Konstrukt auch fehlen kann. Gültige Gleitkommazahlen sind also zum Beispiel

```
3.14
3.14E0
31.4E-1
```

Das folgende Syntaxdiagramm stellt die Grammatikregel nochmals grafisch dar.



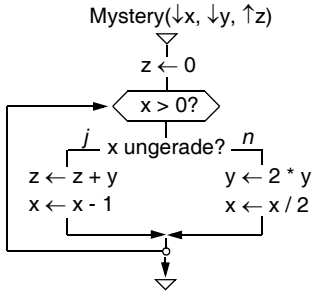
Wir werden in Zukunft die Sprachkonstrukte von Java durch eine Grammatik beschreiben und dadurch ausdrücken, wie Java-Programme aussehen dürfen.

Übungsaufgaben[1]

1. Liegt jedem Programm ein Algorithmus zugrunde? Kann man jeden Algorithmus als Programm formulieren?
2. Erklären Sie den Unterschied zwischen einer Variablen und einem Wert.
3. Warum gibt man Variablen einen Datentyp? Diskutieren Sie Vor- und Nachteile.
4. *Sortieren dreier Zahlen*. Gegeben seien drei Zahlenvariablen a, b und c. Schreiben Sie einen Algorithmus Sort, der die Variableninhalte so umordnet, dass $a \leq b \leq c$ ist. Formulieren Sie den Algorithmus als Ablaufdiagramm und versuchen Sie, seine Korrektheit durch Assertionen zu untermauern.
5. *Median dreier Zahlen bestimmen*. Gegeben seien drei Zahlen a, b und c. Schreiben Sie einen Algorithmus GetMedian, der den Median (also den mittleren Wert) der drei Zahlen ermittelt und zurückgibt. Formulieren Sie den Algorithmus als Ablaufdiagramm und geben Sie Assertionen dazu an. Versuchen Sie, mit möglichst wenigen Abfragen auszukommen.

1. Musterlösungen unter http://ssw.jku.at/JavaBuch/

6. *Schreibtischtest*. Versuchen Sie durch einen Schreibtischtest herauszufinden, was folgender Algorithmus leistet. Die Variablen enthalten ganze Zahlen. Die Division x/y schneidet Nachkommastellen ab.



7. *Schreibtischtest*. Simulieren Sie in einem Schreibtischtest die Funktionsweise des euklidischen Algorithmus für x = 96 und y = 36 sowie für x = 53 und y = 12.

8. *Erzeugen einer Multiplikationstabelle*. Schreiben Sie einen Algorithmus PrintMulTab, der für einen beliebigen Wert n eine Multiplikationstabelle der Größe n mal n ausgibt, in der das Element in Zeile i und Spalte j den Wert i * j hat. Für n = 5 soll z.B. folgende Tabelle erzeugt werden:

```
1   2   3   4   5
2   4   6   8  10
3   6   9  12  15
4   8  12  16  20
5  10  15  20  25
```

   Verwenden Sie für die Ausgabe einer Zahl x die Operation print(x) und für einen Zeilenumbruch die Operation println(). Auf die Formatierung der Tabelle brauchen Sie nicht zu achten.

9. *Grammatiken*. Gegeben seien die Terminalsymbole x, y und z. Welche Symbolfolgen können durch die folgenden Grammatikregeln erzeugt werden:

```
Sequence1 = x (y | z) x.
Sequence2 = [x | y] z {z}.
Sequence3 = x {y z | [x] y} z.
```

10. *Grammatiken*. Geben Sie eine Grammatik an, die die üblichen Datumsformate beschreibt, also z.B.

```
1. Mai 2006
1. 5. 06
2006-05-01
```

11. *Grammatiken*. Begründen Sie, warum in folgender Grammatik für arithmetische Ausdrücke die Multiplikation stärker bindet als die Addition.

```
Expression = Term { "+" Term}.
Term = number { "*" number}.
```

12. *Syntaxdiagramm*. Zeichnen Sie ein Syntaxdiagramm für die Grammatikregel

```
A = x {y z | [x] y} z.
```