3 Entwurf von Programmen

Bevor man mit einer Programmiersprache umzugehen lernt, muss man wissen, was ein Programm prinzipiell ist und wie man Programme entwirft. Damit wird sich unter anderem dieses Kapitel befassen.

3.1 Vom Problem zum Programm

Der Begriff Programm ist eng mit dem Begriff Algorithmus verbunden. Algorithmen werden in Programmen umgesetzt.

> **Algorithmen** sind Vorschriften für die Lösung eines Problems, welche die Handlungen und ihre Abfolge – also die Handlungsweise – beschreiben.

Im Alltag begegnet man Algorithmen in Form von Bastelanleitungen, Kochrezepten und Gebrauchsanweisungen.

Abstrakt kann man sagen, dass die folgenden Bestandteile und Eigenschaften zu einem Algorithmus gehören:

- Eine Menge von **Objekten**, die durch den Algorithmus bearbeitet werden.
- Eine Menge von **Operationen**, die auf den Objekten ausgeführt werden.
- Ein definierter **Anfangszustand**, in dem sich die Objekte zu Beginn befinden.
- Ein gewünschter **Endzustand**, in dem sich die Objekte nach der Lösung des Problems befinden sollen.

Dies sei am Beispiel Kochrezept erläutert:

Objekte:	Zutaten, Geschirr, Herd, ...
Operationen:	Waschen, anbraten, schälen, passieren, ...
Anfangszustand:	Zutaten im „Rohzustand“, Teller leer, Herd kalt, ...
Endzustand:	Fantastische Mahlzeit auf dem Teller

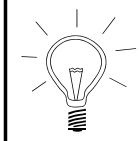
Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann https://doi.org/10.1007/978-3-658-45209-4_3.

J. Goll und T. Stamm, *C als erste Programmiersprache*,
https://doi.org/10.1007/978-3-658-45209-4_3

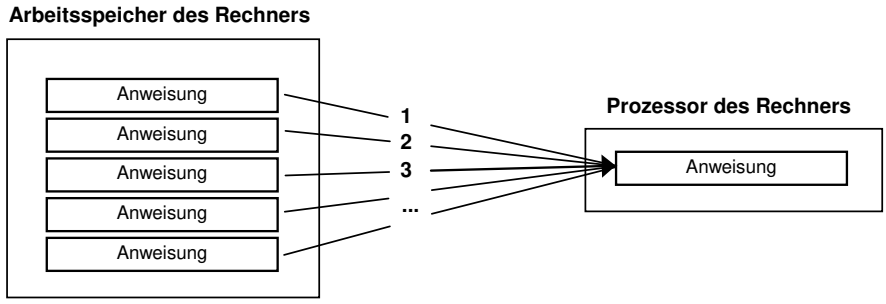
Was dann noch neben der Anleitung oder dem Rezept zur Lösung eines Problems gebraucht wird, ist jemand, der es macht, im angeführten Beispiel etwa ein Koch. Bei Programmiersprachen wird die „Anleitung“ von einem Prozessor umgesetzt.

Während bei einem Kochrezept viele Dinge gar nicht explizit gesagt werden müssen, sondern dem Koch aufgrund seiner Erfahrung implizit klar sind – beispielsweise das Rausholen des Kuchens aus dem Backofen, bevor er schwarz wird – , muss einem Prozessor alles explizit und eindeutig durch ein Programm, das aus Anweisungen einer Programmiersprache besteht, gesagt werden.

Ein Algorithmus in einer imperativen Programmiersprache sagt einem Prozessor präzise, was dieser tun soll. Ein **Programm** besteht aus **Anweisungen**, die von einem Prozessor ausgeführt werden können.



Das folgende Bild zeigt Anweisungen, die im Arbeitsspeicher des Rechners abgelegt sind und nacheinander durch den Prozessor des Rechners abgearbeitet werden:



3.1.1 Der euklidische Algorithmus als Beispiel

Als Beispiel wird der Algorithmus betrachtet, der von Euklid ca. 300 v. Chr. zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen aufgestellt wurde. Der größte gemeinsame Teiler wird zum Kürzen von Brüchen benötigt:

$$\frac{x}{y} = \frac{x/\mathsf{ggT}(x,y)}{y/\mathsf{ggT}(x,y)} = \frac{a}{b}$$

Hierbei ist ggT(x, y) der größte gemeinsame Teiler der beiden Zahlen x und y. Die Zahlen a und b stellen die gekürzten Zahlen dar. Beispielsweise:

$$\frac{24}{9} = \frac{24/\text{ggT}(24,9)}{9/\text{ggT}(24,9)} = \frac{24/3}{9/3} = \frac{8}{3}$$

Der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers zwischen zwei natürlichen Zahlen x und y definiert folgende Schritte:

Solange x ungleich y ist, wiederhole:
    Wenn x größer als y ist, dann:
        Ziehe y von x ab und weise das Ergebnis x zu.
    Andernfalls:
        Ziehe x von y ab und weise das Ergebnis y zu.
Wenn x gleich y ist, dann:
    x (wie auch y) ist der gesuchte größte gemeinsame Teiler.

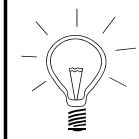
Es gibt in diesem Beispiel also die Objekte x und y, welche am Anfang beliebig vorgegebene Werte enthalten und sodann mit Vergleichen, Abziehen und Zuweisen so bearbeitet werden, dass sie am Schluss den größten gemeinsamen Teiler enthalten. Diese Objekte werden „Variablen“ genannt und im folgenden Kapitel → 3.1.2 genauer besprochen.

Des Weiteren ist ersichtlich, dass die Anweisungen nicht immer strikt der Reihe nach hintereinander ausgeführt werden, es sich somit nicht um eine rein sequenzielle Folge von Anweisungen handelt. Es gibt somit auch bestimmte Konstrukte, welche die einfache sequenzielle Folge gezielt verändern: Eine Auswahl zwischen Alternativen (Selektion) und eine Wiederholung von Anweisungen (Iteration). Diese Konstrukte werden „Kontrollstrukturen“ genannt und im Kapitel → 3.1.3 genauer behandelt.

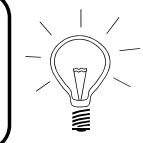
3.1.2 Variablen und Zuweisungen

Die durch den Algorithmus von Euklid behandelten Objekte sind natürliche Zahlen. Diese Zahlen sollen jedoch nicht von vornherein festgelegt werden, sondern der Algorithmus soll für die Bestimmung des größten gemeinsamen Teilers beliebiger natürlicher Zahlen verwendbar sein.

Anstelle fest vorgegebener Zahlen werden im Programm Bezeichner verwendet, die einem Speicherplatz im Arbeitsspeicher oder einem Register zugeordnet sind und als variable Größen oder kurz „**Variablen**“ bezeichnet werden. Den Variablen werden im Verlauf des Algorithmus konkrete Werte zugewiesen.



Die Werteänderung von Variablen erfolgt durch sogenannte „**Zuweisungen**“. Als Zuweisungssymbol wird das Gleichheitszeichen = benutzt.



Für die Anweisung „Ziehe y von x ab und weise das Ergebnis x zu.“ wird in C folgende Anweisung geschrieben:

```
x = x - y;
```

Die Schreibweise `x = x - y` ist zunächst etwas verwirrend. Diese Schreibweise ist nicht als mathematische Gleichung zu sehen, sondern meint etwas ganz anderes: Auf der rechten Seite des Gleichheitszeichens steht ein arithmetischer Ausdruck, dessen Wert zuerst berechnet werden soll. Dieser so berechnete Wert wird dann in einem zweiten Schritt der Variablen zugewiesen, deren Name auf der linken Seite steht. Im Beispiel also:

- Nimm den aktuellen Wert von x. Nimm den aktuellen Wert von y. Ziehe den Wert von y vom Wert von x ab.
- Der neue Wert von x sei die soeben ermittelte Differenz von x und y.

Bei einer Zuweisung wird zuerst der Ausdruck rechts vom Zuweisungs-Operator berechnet und der Wert dieses Ausdrucks dem Speicherplatz auf der linken Seite des Zuweisungs-Operators zugewiesen.

Eine Zuweisung verändert den Wert der Variablen auf der linken Seite, oder anders gesagt, deren Zustand. Daher werden solche Transformationen auch als „Zustandstransformationen“ bezeichnet. Eine solche Wertzuweisung ist eine

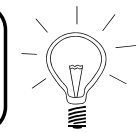
der grundlegenden Operationen, die ein Prozessor ausführen können muss. Auf Variablen wird noch ausführlicher in Kapitel → 7 eingegangen.

Der im obigen Beispiel beschriebene Algorithmus kann auch von einem menschlichen „Prozessor" ausgeführt werden – andere Möglichkeiten hatten die Griechen in der damaligen Zeit nicht. Als Hilfsmittel braucht man dazu Papier und Bleistift, um die Zustände der Variablen – im obigen Beispiel die Zustände der Variablen x und y – zwischen den Verarbeitungsschritten festzuhalten. Man erhält dann eine Tabelle, die für die Zahlen x gleich 24 und y gleich 9 das folgende Aussehen hat:

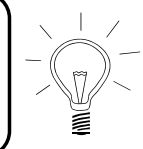
Verarbeitungsschritt	Variable x	Variable y
Initialisierung	**24**	**9**
x = x - y	15	9
x = x - y	6	9
y = y - x	6	3
x = x - y	3	3

Diese Tabelle zeigt sehr deutlich die Funktion der Variablen auf:

Variablen speichern Werte. Sie repräsentieren über den Verlauf des Algorithmus hinweg ganz unterschiedliche Werte.



Zu Beginn eines Algorithmus werden den Variablen definierte Anfangs- oder Startwerte zugewiesen. Diesen Vorgang bezeichnet man als **Initialisierung** der Variablen.



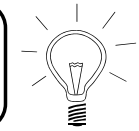
Wird derselbe Algorithmus zweimal durchlaufen, wobei die Variablen am Anfang unterschiedlich initialisiert werden, dann erhält man in aller Regel auch unterschiedliche Abläufe. Sie folgen aber ein und demselben Verhaltensmuster, das durch den Algorithmus beschrieben ist.

Für eine andere Ausgangssituation sieht die Tabelle beispielsweise so aus:

Verarbeitungsschritt	Variable x	Variable y
Initialisierung	**5**	**3**
x = x - y	2	3
y = y - x	2	1
x = x - y	1	1

3.1.3 Kontrollstrukturen: Beschreibung sequenzieller Abläufe

Die Abarbeitungsreihenfolge von Anweisungen wird auch als „**Kontrollfluss**“ bezeichnet.



Den Prozessor stört es überhaupt nicht, wenn eine Anweisung einen Sprungbefehl zu einer anderen Anweisung enthält. Solche Sprungbefehle werden in manchen Programmiersprachen beispielsweise mit dem Befehl `GOTO` und Zeilenmarken wie beispielsweise `40` realisiert:

```
10 IF(a > b) GOTO 40
20 Anweisung Y
30 GOTO 50
40 Anweisung X
50 Anweisung Z
```

In Worten lauten diese Anweisungen an den Prozessor: „Vergleiche die Werte von `a` und `b`. Wenn `a` größer als `b` ist, springe in die Zeile mit der Marke `40` und führe die dort stehende Anweisung `X` aus. Fahre dann mit der Anweisung `Z` in der nächsten Zeile fort. Ist aber die Bedingung `a > b` nicht erfüllt, so arbeite die Anweisung `Y` der nächsten Zeile ab. Springe dann zur Zeile mit der Marke `50` und fahre mit der Ausführung der Anweisung `Z` fort.“

Will man ein solches Programm jedoch lesen, so verliert man durch die Sprünge mit `GOTO` sehr leicht den Zusammenhang und damit das Verständnis.

Während in den sechziger Jahren somit typische Programme noch zahlreiche Sprünge enthielten, bemüht sich die Programmiergemeinschaft seit Dijkstras grundlegendem Artikel „Go To Statement Considered Harmful“ [Dij68], möglichst einen Kontrollfluss ohne Sprünge zu entwerfen.

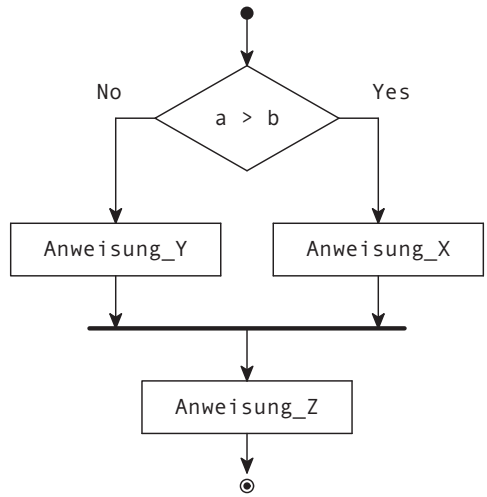
Die Ideen von Dijkstra und anderen fanden ihren Niederschlag in den Regeln für die „**Strukturierte Programmierung**“. Die Strukturierte Programmierung ist ein programmiersprachenunabhängiges Konzept. Es umfasst die Zerlegung eines Programms in Teilprogramme. Gestartet wird mit einem Hauptprogramm, welches dann Unterprogramme aufrufen kann, welche wiederum Unterprogramme aufrufen können.

Des Weiteren werden für sequenzielle Abläufe die drei grundlegenden Kontrollstrukturen Sequenz, Selektion und Iteration definiert. Eine reine Sprunganweisung wie `GOTO` ist in der Strukturierten Programmierung nicht zulässig.

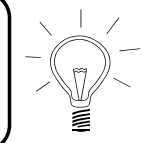
Der oben mit `GOTO` beschriebene Ablauf kann mittels der Strukturierten Programmierung in C folgendermaßen realisiert werden:

```
if (a > b)
  Anweisung_X
else
  Anweisung_Y
AnweisungZ
```

Hierbei ist `if (a > b)` die Abfrage, ob `a` größer als `b` ist. Ist dies der Fall, so wird die Anweisung `X` ausgeführt. Ist die Bedingung `a > b` nicht wahr, also nicht erfüllt, so wird die Anweisung `Y` des `else`-Zweiges durchgeführt. Damit ist die Fallunterscheidung zu Ende. Unabhängig davon, welcher der Zweige der Fallunterscheidung abgearbeitet wurde, wird nun die Anweisung `Z` ausgeführt. Dieser Ablauf ist im folgenden Bild grafisch veranschaulicht:



Für das menschliche Gehirn ist es am einfachsten, wenn ein Programm einen einfachen und überschaubaren Kontrollfluss hat.



Entsprechend wird im folgenden Kapitel das sogenannte „Flussdiagramm“ vorgestellt.

3.2 Entwurf mit Flussdiagrammen nach UML

Zur Visualisierung des Kontrollflusses von Programmen – das heißt, zur grafischen Veranschaulichung ihres Ablaufs – werden seit Jahren sogenannte **Flussdiagramme** (auf Englisch „**flow chart**“) verwendet. Sie werden in UML auch als „**Aktivitäts-Diagramme**“ bezeichnet.

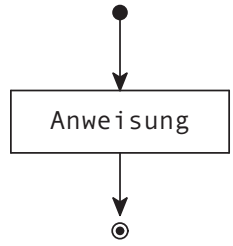
Grundsätzlich geht es bei Flussdiagrammen darum, den Kontrollfluss eines Programmes mittels Kästchen und Pfeilen so darzustellen, dass möglichst einfach ersichtlich wird, wann und in welcher Reihenfolge in einem Programm was passiert. Wie genau diese Diagramme aussehen, spielt normalerweise nicht so eine große Rolle. Manche Flussdiagramme nutzen abgerundete Kästchen, manche haben Titelleisten für bestimmte Kästchen, nutzen unterschiedliche Formen, sind farbig, manche schreiben Variablen hinzu, manche haben verschiedene Pfeilspitzen, etc.

Hier in diesem Buch wird versucht, ein sehr einfaches Set zur Verfügung zu stellen, welches durch die **UML**, die sogenannte „**Unified Modeling Language**“, motiviert ist.

Im Folgenden werden die Diagramme für die Sequenz, Selektion und Iteration in abstrakter Form, also ohne Notation in einer speziellen Programmiersprache, vorgestellt.

3.2.1 Start, Ende und einfache Anweisung

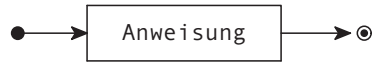
Folgendes Diagramm zeigt das fundamentale Prinzip eines Flussdiagramms:



Gestartet wird beim ausgefüllten Punkt. Dies ist der Startpunkt, von welchem aus das Programm sukzessive entlang der Pfeile abgearbeitet wird. Hier wird eine einfache Anweisung ausgeführt, was mit einem Kästchen angezeigt wird. Nach dieser Anweisung kommt das Flussdiagramm bereits zu einem Ende, markiert durch den umrandeten Punkt.

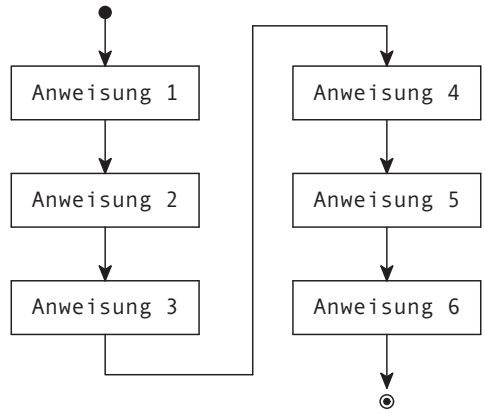
Der Begin und das Ende werden manchmal zusätzlich beschriftet. Beispielsweise werden die initialen Werte von Variablen angegeben.

Des Weiteren spielt es keine Rolle, ob ein Flussdiagramm vertikal ausgerichtet ist oder horizontal. Der Fluss folgt stets der Pfeilrichtung:



3.2.2 Anweisungen in Sequenz

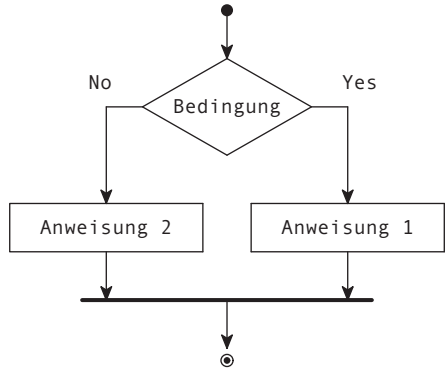
Wenn mehrere Anweisungen sequenziell hintereinander ausgeführt werden sollen, so werden sie auch im Flussdiagramm mittels Pfeile hintereinandergereiht:



Wie man sieht, kann dies auch in horizontal und vertikal gemischter Anordnung passieren. Hauptsache, die Pfeile zeigen genau an, wo der Fluss durchgeht.

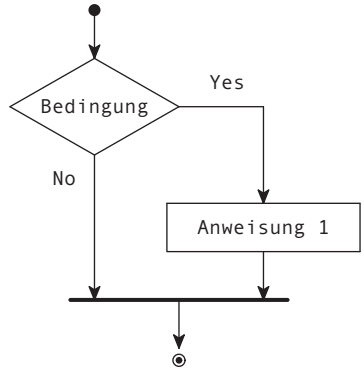
3.2.3 Einfache Selektion, einfache Alternative

Bei einer **einfachen Selektion** (auch „einfache Alternative“ genannt) wird aufgrund einer **Bedingung** der Kontrollfluss in zwei Pfade aufgeteilt. Die Bedingung wird in einen Rhombus gezeichnet, aus welchem die beiden Pfade als zwei Pfeile hervorkommen, beschriftet mit dem Ergebnis der Bedingung:



Aufgrund der Auswertung der Bedingung wird entweder Anweisung 1 oder 2 ausgeführt. Am Ende der Selektion kommen die beiden Pfade wieder zusammen, was durch eine etwas dickere Linie angedeutet wird. Danach wird wieder auf einem einzelnen Pfad fortgefahren.

Üblicherweise wird der Yes-Pfad auf der rechten Seite dargestellt, das ist aber nicht zwingend. Falls eine einfache Selektion keinen No-Pfad besitzt, so handelt es sich um eine sogenannt „**bedingte Anweisung**“:

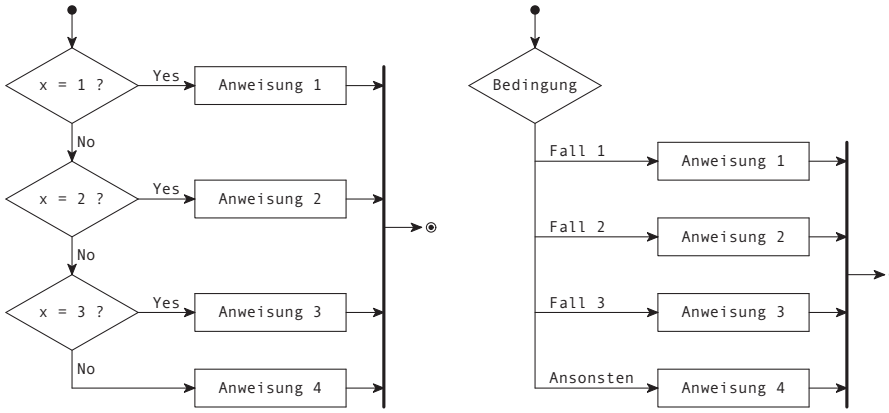


Die Anweisung wird nur ausgeführt, falls die gegebene Bedingung erfüllt ist.

3.2.4 Mehrfache Selektion, mehrfache Alternative

Eine **mehrfache Selektion** (auch „mehrfache Alternative“ genannt) tritt auf, wenn eine Bedingung geprüft wird, aufgrund derer mehr Möglichkeiten als nur `Yes` und `No` auftreten können. In C ist dies mittels der `switch` Kontrollstruktur möglich, welche in Kapitel → 10.3 behandelt werden wird.

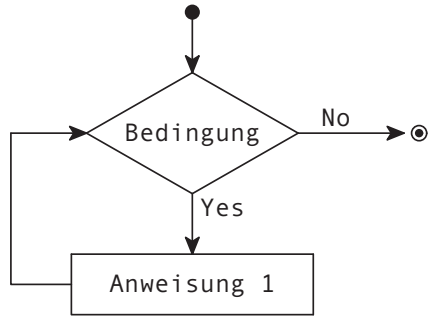
Für eine Darstellung einer Mehrfachentscheidung gibt es zwei Möglichkeiten:



Die linke Variante entspricht mehr den Prinzipien von UML, benötigt jedoch mehr Schreibarbeit. Entsprechend wird häufig die rechte Variante gewählt, auch deshalb, weil sie die Schreibweise der entsprechenden Kontrollstruktur mittels `switch` eins-zu-eins abbildet.

3.2.5 Schleifen, Iteration

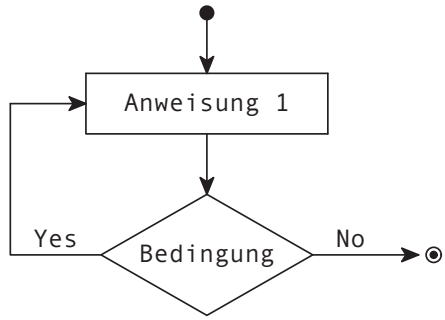
Um einen Pfad mehrere Male auszuführen, können die bisherigen Elemente einfach mittels einer Bedingung zu einem Kreis zusammengeschlossen werden:



Dies wird als eine **Schleife** bezeichnet. Diese einfache Schleife wird solange ausgeführt, wie die Bedingung erfüllt ist. Natürlich sollte die Anweisung irgendwann etwas am Zustand des Programmes verändern, ansonsten wird diese Schleife unendlich lange ausgeführt.

Diese Schleife wird als eine sogenannt „**abweisende Schleife**“ bezeichnet, da die Bedingung vor der Ausführung der Anweisung geprüft wird. Dadurch kann es sein, dass die Bedingung bereits vor dem ersten Durchlauf nicht erfüllt wird, und die Schleife somit komplett „abgewiesen“ wird.

Demgegenüber gibt es auch „**annehmende Schleifen**“. Diese sehen als Flussdiagramm folgendermaßen aus:



Hier wird die Anweisung auf jeden Fall mindestens einmal ausgeführt und erst dann mittels der Bedingung geprüft, ob sie wiederholt werden soll.

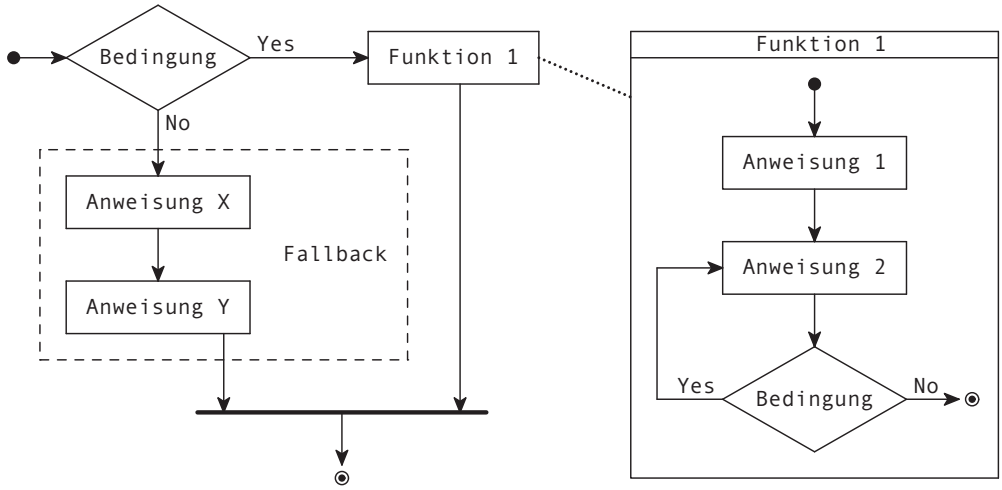
In der Programmiersprache C gibt es zwei abweisende Schleifen `for` und `while` und eine annehmende Schleife `do while`. Diese werden in Kapitel → 10 genauer behandelt.

Wird innerhalb einer Schleife eine Menge an Elementen durchschritten, beispielsweise alle Zahlen von 1 bis 100 oder alle Elemente eines Arrays, dann wird diese Schleife auch als eine „**Iteration**“ bezeichnet.

3.2.6 Gruppierungen, Blöcke, Funktionen

Häufig bilden mehrere Anweisungen und Selektionen zusammen eine neue Gesamt-Funktionalität, welche im tatsächlichen Code dann als **Anweisungsblock** oder **Funktion** ausprogrammiert wird. In einem Flussdiagramm ist man relativ frei, solche Gruppierungen vorzunehmen und sie zu benennen.

Beispielsweise könnte ein Programm so aussehen:

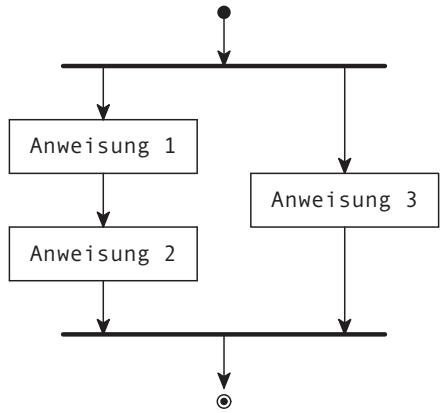


Hier wird ersichtlich, dass die beiden Anweisungen `X` und `Y` zusammen eine Gruppe namens „Fallback“ bilden. Zudem wird eine Funktion aufgerufen, deren Definition im rechten Kasten zu finden ist.

Die Verwendung der gestrichelten und gepunkteten Linien ist hier frei wählbar. Wichtig ist, dass es visuell klar strukturiert ist.

3.2.7 Parallele Anweisungen

Nebst rein sequenziellen Algorithmen gibt es auch Algorithmen zur Beschreibung von parallelen Aktivitäten, die zum gleichen Zeitpunkt nebeneinander ausgeführt werden. Diese Algorithmen werden unter anderem bei Betriebssystemen oder in der Prozessdatenverarbeitung benötigt. Ein entsprechendes Diagram würde so aussehen:

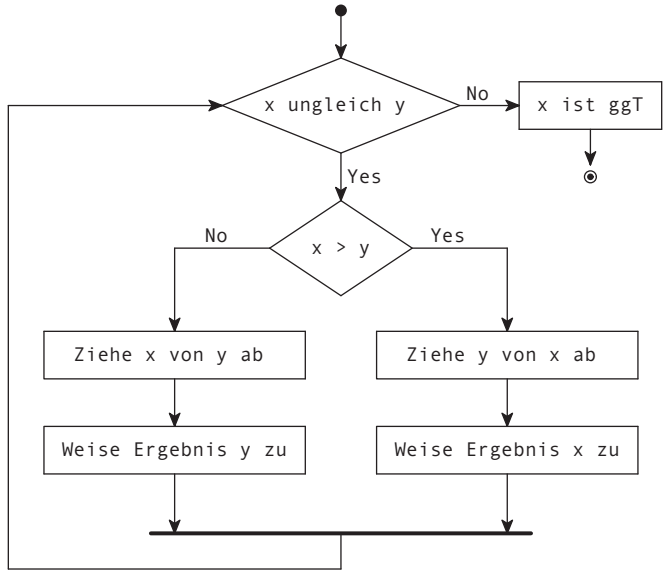


In diesem Beispiel werden zwei parallele Stränge erzeugt, wovon der eine die Anweisungen 1 und 2 abarbeitet und der andere die Anweisung 3. Die etwas dickeren Linien bezeichnen Synchronisationzeitpunkte, zu welchen die parallele Verarbeitung startet, beziehungsweise der Kontrollfluss abwartet, bis sämtliche parallelen Anweisungen abgearbeitet wurden, auf dass danach wieder rein sequenziell fortgefahren werden kann.

Parallele Abläufe werden erst in Kapitel → 23 behandelt. Dieses kleine Kapitel diente nur zur Veranschaulichung, wie mit Flussdiagrammen auch erweiterte Algorithmen dargestellt werden können. Fortan werden nur noch sequenzielle Abläufe behandelt, bei denen zum selben Zeitpunkt nur eine einzige Operation durchgeführt wird.

3.2.8 Der Euklidische Algorithmus als Flussdiagramm

Mit den Mitteln der Flussdiagramme kann nun der Algorithmus von Euklid, der in Kapitel → 3.1.1 eingeführt wurde, in grafischer Form dargestellt werden:

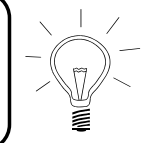


Diese abstrakte Darstellung gilt es nun, in ein tatsächlich lauffähiges Programm umzusetzen. Personen, welche bereits viel Programmiererfahrung haben, können ein solch einfaches Beispielprogramm direkt umsetzen. Hier jedoch wird noch ein weiteres Hilfsmittel vorgestellt, um solch abstrakte Diagramme sukzessive in lauffähigen Code umzuwandeln: Mittels sogenanntem „Pseudocode“.

3.3 Pseudocode

Wenn ein Programm entworfen, also quasi „erfunden“ werden soll, kann man grafische Mittel wie Flussdiagramme nutzen oder man greift auf eine „halbformale“ Sprache, einen sogenannten freien Pseudocode, zurück, der es erlaubt, zuerst die Struktur eines Programms textuell festzulegen und die Details auf später aufzuschieben.

Ein **Pseudocode** ist eine Sprache, die dazu dient, Programme zu entwerfen. Pseudocode kann von einem freien Pseudocode bis zu einem formalen Pseudocode reichen.



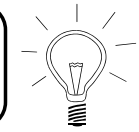
Um zu verstehen, was freier und formaler Pseudocode ist, muss zunächst definiert werden, was eine sogenannt „formale“ Sprache ist.

Generell kann man bei Sprachen zwischen „natürlichen Sprachen“ wie der Umgangssprache oder den Fachsprachen einzelner Berufsgruppen und „formalen Sprachen“ unterscheiden.

Formale Sprachen sind beispielsweise die Notenschrift in der Musik, die Formelschrift in der Mathematik oder Programmiersprachen beim Computer. Nur das, was durch eine formale Sprache – hier die Programmiersprache – festgelegt ist, ist für den Übersetzer verständlich.

Hat man ein Problem zu lösen, dann ist es empfehlenswert, nicht gleich auf der Ebene einer formalen Programmiersprache zu beginnen. Würde man das tun, dann müsste man sich sofort mit den vielen formalen Details der Programmiersprache befassen, statt erst nachzudenken, wie der Lösungsweg denn aussehen soll.

Freie Pseudocodes sind für eine grobe Spezifikation vollkommen ausreichend.

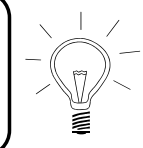


Das Beispiel in Kapitel → 3.1.1 war in einem sogenannt „freien Pseudocode“ formuliert. Es wurden deutsche Schlüsselwörter wie „solange“, „wenn“ oder „andernfalls“ benutzt. Meist lehnt man sich aber bei einem Pseudocode an eine bekannte Programmiersprache an, sodass dann englische Begriffe dominierend sind.

Bei einem freien Pseudocode formuliert man Schlüsselwörter für die Iteration, Selektion und Blockbegrenzer und fügt in diesen Kontrollfluss Verarbeitungsschritte ein, die in der Umgangssprache beschrieben werden.

Demgegenüber gibt es jedoch auch sogenannt „formalen Pseudocode“, welcher durch vorgegebene Regeln definiert, wie ein Programm abgebildet werden soll. Entsprechend gibt es Programme, welche aus der Formulierung in einer solchen formalen Pseudo-Sprache automatisch Code generieren können.

Ein formaler Pseudocode, der alle Elemente enthält, die auch in einer Programmiersprache enthalten sind, ermöglicht eine automatische Codegenerierung für diese Zielsprache.



Grundsätzlich ist jedoch das eigentliche Ziel eines Pseudocodes, eine Spezifikation zu unterstützen.

Im Folgenden wird nun der Algorithmus von Euklid mithilfe eines Pseudocodes formuliert, welcher schon sehr ähnlich ist zum finalen Programm:

```
Euklidischer Algorithmus.
{
   Initialisiere x und y.
   solange (x ungleich y)
   {
      falls (x kleiner als y)
         y = y - x
      ansonsten
         x = x - y
   }
   x ist der groesste gemeinsame Teiler.
}
```

Dieser Pseudocode und die grafische Darstellung als Flussdiagramm sind äquivalent.

3.4 Das finale Programm von Euklid in C

Der Schritt zu einem lauffähigen Programm ist nun überhaupt nicht mehr groß: Das, was soeben noch umgangssprachlich formuliert wurde, muss jetzt in C ausgedrückt werden. Die wesentlichen Punkte betreffen hier die Kommunikation des Programms mit der Person, welche es bedient: Welche Werte sollen `x` und `y` bekommen und in welcher Form soll das Ergebnis dargestellt werden? Da die Details der Ein- und Ausgabe von Daten in C-Programmen an dieser Stelle nicht behandelt werden sollen, arbeitet das folgende C-Programm für den euklidischen Algorithmus mit fest vorgegebenen Werten für `x` und `y`:

euklid.c

```
#include <stdio.h>

int main(void) {
  int x = 24;
  int y = 9;

  while (x != y) {
    if (x < y)
      y = y - x;
    else
      x = x - y;
  }

  printf("Der groesste gemeinsame Teiler ist: %d\n", x);
  return 0;
}
```

Die Ausgabe des Programmes ist:

```
Der groesste gemeinsame Teiler ist: 3
```

Viele Elemente dieses Programms sind bereits durch die vorherigen Kapitel bekannt. Es werden zwei Variablen `x` und `y` vereinbart, die Integers speichern können und jeweils mit einem vorgegebenen Wert initialisiert werden.

Die Abfragen, ob Werte ungleich sind beziehungsweise ob ein Wert kleiner als ein anderer ist, werden in C durch die Operatoren `!=` beziehungsweise `<` ausgedrückt. Die Schleife wird mittels des Schlüsselwortes `while` gemacht und die einfache Bedingung mittels dem `if`-Schlüsselwort. Die Ausgabe des Resultates erfolgt mittels Aufruf der `printf()` Funktion.

3.5 Übungsaufgaben

Aufgabe 1: Quadratzahlen

Zeichnen Sie das Flussdiagramm für ein Programm, welches in einer (äußeren) Schleife bei jedem Durchgang einen Integer-Wert in eine Variable `n` einliest. Die Reaktion des Programms soll davon abhängen, ob der in die Variable eingelesene Wert positiv, negativ oder gleich null ist. Treffen Sie die folgende Fallunterscheidung:

- Ist die eingelesene Zahl `n` größer als null, so soll in einer inneren Schleife alle Quadratzahlen von `1` bis `n` ausgegeben werden, also beispielsweise bei der Eingabe 6:
  `1 4 9 16 25 36`
- Ist die eingelesene Zahl `n` kleiner als null, so soll ausgegeben werden: „Negative Zahl“
- Ist die eingegebene Zahl `n` gleich null, so soll das Programm (die äußere Schleife) abbrechen.

Hinweise:

- Überlegen Sie sich, ob eine annehmende oder abweisende Schleife besser geeignet ist.
- Machen Sie sich keine Gedanken darüber, wie genau eine Zahl eingelesen oder ausgegeben wird. Nehmen Sie einfach an, dass es irgendwo eine Funktionalität gibt, die das kann.